

MONGO DB

Introducción I

- Base de datos NoSQL orientada a documentos.
 - El hecho de que sea una BD implica que los datos se almacenan y organizan de una manera estructurada.
 - NoSQL significa que la BD no usa tablas (filas y columnas) para organizar los datos.
- Se basa en el concepto de **agregado de información**, que en MongoDB recibe el nombre de **documento**. Se llama así porque un documento suele estar compuesto de pedacitos de información (agregados o subdocumentos), que se suelen repetir en otros documentos.
- Internamente, almacena la información en documentos en el formato binario BSON, basado en el formato textual JSON.
- Permite importar directamente documentos JSON y CSV.

Introducción II

- MongoDB tiene un lenguaje de consultas, MongoDB Query Language (MQL), que permite realizar operaciones semejantes a las realizadas con SQL en las BDRs.

Resumen de características de MongoDB

- Orientada a documentos
- Alto rendimiento
- Consultas específicas
- Indexado
- Sin esquema (*schemaless*)
- Agregación
- *Sharding* (fragmentación)
- Replicación
- GridFS



Conceptos

- **Documento**, forma de organizar y almacenar datos como un conjunto de pares {clave, valor}.
- **Campo**, identificador único de un punto de datos.
- **Valor**, dato relacionado con un identificador.
- **Colección**, un almacén organizado de documentos, generalmente (aunque no necesariamente) con campos comunes entre los documentos. Puede haber muchas colecciones por base de datos y muchos documentos por colección.

Documento I

- Es la unidad básica de organización de la información en MongoDB. Es semejante a un objeto en POO o a una fila en las BDRs.
- La información se almacena en formato JSON.
- Un documento es un conjunto ordenado de claves que tienen asociados valores. Se corresponden con estructuras de datos de lenguajes de programación (tablas hash o diccionarios).
- En general, los documentos contendrán múltiples pares clave-valor (equivalentes a atributos de un objeto o a campos de una tabla en una BDR) como, por ejemplo, { "nombre":"Pepe", "edad" : 35 }.
- Las claves son cadenas que deben cumplir el no contener el carácter nulo (\0), ni el punto (.), ni el dólar (\$).

Documento II

- Tanto claves como valores son sensibles a mayúsculas/minúsculas.
- Los documentos no pueden tener claves duplicadas:
Ej. { "nombre" : Pepe, ~~"nombre" : "Manolo"~~ }
- El orden de los campos es significativo.
Ej. { "usuario" : "alfa", "clave" : "beta" } ~~=~~ { "clave" : "beta", "usuario" : "alfa" }
- Todo documento tiene un campo `_id`:
 - Es una cadena alfanumérica.
 - Su valor debe ser único dentro de una colección. Es el equivalente a una clave primaria.
 - Se puede dejar a MongoDB que le asigne un valor automáticamente (al estilo de los campos auto-incrementales de las BDRs).
 - Se puede fijar manualmente un valor, siempre y cuando no esté en uso.

Colección I

- Es un almacén organizado de documentos que suelen compartir campos entre sí, pero esto no es obligatorio.
- Como se explicó anteriormente, todo documento debe tener un campo `_id` cuyo valor sea único dentro de una colección.
- Por defecto, no existe ninguna otra restricción respecto a los campos y sus valores. En una colección podría haber N documentos que fueran exactamente iguales, difiriendo únicamente en su `campo_id`.
- De igual manera, en una colección podría haber N documentos que no tuvieran absolutamente ningún campo en común.

Colección II

- Se identifica por su nombre, que es una cadena con las siguientes restricciones
 - La cadena vacía no es un nombre válido para una colección.
 - Los nombres de las colecciones no pueden contener el carácter nulo, \0, pues este símbolo se usa para indicar el fin del nombre de una colección.
 - No se deben crear colecciones cuyo nombre empiece por «system», puesto que es un prefijo reservado para colecciones de sistema.
 - Las colecciones creadas por los usuarios no deben contener el carácter reservado \$ en su nombre.

Reglas sintácticas básicas de JSON

- Los **documentos** empiezan por { y terminan por } (llaves).
- En cada par, la **clave y el valor** se separan por : (dos puntos).
- Cada **par clave-valor** se separa del siguiente por , (coma).
- El **identificador** de cada clave debe estar rodeado por " " (comillas dobles).
- Las claves también se llaman campos.
- Los **arrays** empiezan por [y terminan por] (corchetes), estando sus elementos separados por , (coma).

Análisis de JSON

VENTAJAS	INCONVENIENTES
Amigable	En texto plano. El análisis (<i>parsing</i>) de texto plano es muy lento.
Legible para el ojo humano	Es poco eficiente para el almacenamiento
Familiar para los desarrolladores	Tiene un número limitado de tipos de datos


BSON: Binary JSON

- Se basa en JSON, pero no es un formato de texto plano. Es una **representación binaria de JSON**.
- Optimizado en:
 - Velocidad
 - Espacio
 - Flexibilidad
- Propósito:
 - Eficiencia / alto rendimiento
 - Propósito general

JSON vs. BSON

	JSON	BSON
Codificación	UTF-8	Binario
Tipos de datos	String, booleano, numérico, array	String, booleano, numérico (int, long, float, decimal...), null, array, Date, binario puro (<i>raw binary</i>)
Legibilidad	Humano y máquina	Sólo máquina

Ejemplo de equivalencia de código en JSON y BSON

JSON	BSON	Significado
<pre>{ "hello": "world" }</pre>  <p>16 caracteres entre { y }</p>	<pre>\x16\x00\x00\x00 \x02 hello\x00 \x06\x00\x00\x00world\x00 \x00</pre>	Tamaño del documento (bytes) 0x02 = Tipo String Nombre del campo Valor del campo 0x00 = EOO (<i>End Of Object</i>)

- En varios lugares del documento aparece el delimitador `\x00` (End of Object).

Base de datos I

- Está compuesta por un conjunto de colecciones.
- Una instancia de MongoDB puede gestionar varias BDs, cada una conteniendo varias colecciones.
- Cada BD tiene sus propios permisos de acceso y se almacena en disco en archivos independientes.
- Una regla del pulgar es almacenar todos los datos de una aplicación en una misma BD, organizados en varias colecciones.

Base de datos II

- Se identifican mediante nombres que son cadenas con las siguientes restricciones
 - La cadena vacía no es un nombre válido para una base de datos
 - El nombre de una base de datos no puede contener ninguno de los siguientes caracteres `\\,/,,,/,...,"*,<,>,,,*,<,>,:,,|,?,,,|,?,$$,espacio` o `\0` (valor nulo)
 - Los nombres de las bases de datos son sensibles a mayúsculas y minúsculas incluso cuando el servidor está montado sobre un sistemas de archivos que no lo sean
 - Una regla práctica es usar siempre nombres en minúscula.
 - Los nombres están limitados a un máximo de 64 bytes

Bases de datos pre-existentes

- **admin:** Es el nombre de la BD «root» en términos de autenticación. Si un usuario es añadido a esta BD, entonces el usuario hereda los permisos para todas las BDs.
- **local:** Esta BD nunca será replicada y sirve para almacenar cualquier colección que debería ser local a un servidor.
- **configura:** Cuando en MongoDB se usa una configuración con *sharding*, se usa esta base de datos para almacenar información acerca de los fragmentos o *shards* que se crean.

Conceptos

- **Cluster:** Conjunto de servidores que almacenan datos relacionados.
- **Conjunto de réplicas (*Replica Set*):** Conjunto de (una pocas) instancias de MongoDB conectadas que almacenan los mismos datos. Si una instancia queda inutilizada, las demás siguen ofreciendo los datos.
- **Instancia:** Equipo que ejecuta un software (MongoDB), localmente o en la nube.
- **Cursor:** Variable en memoria que representa el resultado de una consulta. También se llama **Conjunto de resultados (*Result Set*)**.

Algunas órdenes de la línea de comandos

COMANDO	USO
<code>cls</code>	Borra la pantalla
<code>db</code>	Muestra la base de datos actualmente seleccionada.
<code>help</code>	Ayuda sobre los comandos de línea de comandos.
<code>it</code>	Iteración sobre un cursor, generado habitualmente como resultado de una consulta.
<code>show collections</code>	Muestra la colecciones de la BD en uso.
<code>show dbs</code>	Muestra las bases de datos existentes.
<code>show roles</code>	Muestra los roles asignados a la BD en uso.
<code>use <base_datos></code>	Selecciona una BD existente; la crea si no existe.
<code>mongodb+srv://<usuario>:<password>@clusterURI.mongodb.net/<bd></code>	

Tipos de datos de un campo I

TIPO DE DATOS	SIGNIFICADO	EJEMPLO
Array	Array de elementos polimórficos, accesible por índice numérico.	"orden": ["uno", 2, new Date()]
Binary	Cadena de bytes que no puede ser manipulada desde la <i>shell</i> y sirve para representar cadenas de caracteres no UTF8.	
Boolean	Booleano	"revisado" : true
Code	Código JavaScript	"esPar" : function(numero) {return numero % 2 == 0}

Tipos de datos de un campo II

TIPO DE DATOS	SIGNIFICADO	EJEMPLO
Date	Fecha y hora	"fecha": ISODate() "fecha": ISODate("2012-12-19T06:01:17.171Z") "fechaDescubrimiento" : new Date ("1492, 10, 12")
Decimal128	Número real con 128 bits de precisión.	"real": Decimal128("1") "pi" : 3.1416
Double	Tipo numérico por defecto.	"doble": 1.01
Int32	Número entero con 32 bits de precisión.	Int32("1")
Int64 / Long	Número entero con 64 bits de precisión.	Long("1")

Tipos de datos de un campo III

TIPO DE DATOS	SIGNIFICADO	EJEMPLO
MaxKey	Representa el valor máximo de una clave (campo).	MaxKey()
MinKey	Representa el valor mínimo de una clave (campo).	MinKey()
Null	Valor nulo.	"comisión" : null
Object	Un agregado dentro de una documento, un documento embebido en otro.	<pre>{"apellido" : "Smith" , "edad" : 76, "direccion" : { "calle" : "Avenida de la naturaleza", "numero" : 15, "localidad" : "Villanueva del castillo" } }</pre>

Tipos de datos de un campo IV

TIPO DE DATOS	SIGNIFICADO	EJEMPLO
ObjectId	Identificador alfanumérico de un documento que puede generar MongoDB (equivalente al tipo auto-incremental en BDR)	ObjectId('63626fa90642d258a2ec1944')
BSONRegExp	Expresión regular.	\$regex : "^B"
String	Cadena de texto	"departamento" "Ventas"
BSONSymbol	Cadenas de datos binarios.	Symbol('63626fa90642d258a2ec1944')
Timestamp	Fecha y hora con más precisión que Date	Timestamp() Timestamp(1627811580)

Expresiones regulares: operador **\$regex** |

- Permite buscar patrones dentro de cadenas de texto.

Sintaxis: { <campo>: { "\$regex": <valor> } }

{ <campo>: { \$regex: /patrón/, \$options: '<opciones>' } }

{ <campo>: { \$regex: 'patrón', \$options: '<opciones>' } }

{ <campo>: { \$regex: /patrón/<opciones> } }

{ <campo>: /patrón/<opciones> }

Ej. Personas cuyo apellido1 contenga la cadena ar:

db.personas.find("apellido1": { **"\$regex"**: **"ar"** })

Expresiones regulares: operador **\$regex** II

- Valores para <opciones>:

VALOR	DESCRIPCIÓN	RESTRICCIONES SINTÁCTICAS
i	Insensible a mayúsculas y minúsculas.	
m	<p>For patterns that include anchors (i.e. ^ for the start, \$ for the end), match at the beginning or end of each line for strings with multiline values. Without this option, these anchors match at beginning or end of the string. For an example, see Multiline Match for Lines Starting with Specified Pattern.</p> <p>If the pattern contains no anchors or if the string value has no newline characters (e.g. \n), the m option has no effect.</p>	

Expresiones regulares: operador **\$regex** III

VALOR	DESCRIPCIÓN	RESTRICCIONES SINTÁCTICAS
x	<p>"Extended" capability to ignore all white space characters in the \$regex pattern unless escaped or included in a character class.</p> <p>Additionally, it ignores characters in-between and including an un-escaped hash/pound (#) character and the next new line, so that you may include comments in complicated patterns. This only applies to data characters; white space characters may never appear within special character sequences in a pattern.</p> <p>The x option does not affect the handling of the VT character (i.e. code 11).</p>	Requires \$regex with \$options syntax
s	<p>Allows the dot character (i.e. .) to match all characters including newline characters. For an example, see Use the . Dot Character to Match New Line.</p>	Requires \$regex with \$options syntax

Expresiones regulares: operador **\$regex** IV

Ej. To include a regular expression in an \$in query expression, you can only use JavaScript regular expression objects (i.e. /pattern/). For example:

```
{ name: { $in: [ /^acme/i, /^ack/ ] } }
```

Ej. To include a regular expression in a comma-separated list of query conditions for the field, use the \$regex operator. For example:

```
{ name: { $regex: /acme.*corp/i, $nin: [ 'acmeblahcorp' ] } }
```

```
{ name: { $regex: /acme.*corp/, $options: 'i', $nin: [ 'acmeblahcorp' ] } }
```

```
{ name: { $regex: 'acme.*corp', $options: 'i', $nin: [ 'acmeblahcorp' ] } }
```

Ej. To use either the x option or s options, you must use the \$regex operator expression with the \$options operator. For example, to specify the i and the s options, you must use \$options for both:

```
{ name: { $regex: /acme.*corp/, $options: "si" } }
```

```
{ name: { $regex: 'acme.*corp', $options: "si" } }
```

Tipos de datos compuestos:
arrays y agregados (objetos)

Tipos de datos compuestos

- Hasta el momento se han presentado tipos de datos simples (cadenas de texto, números, fechas, booleanos...).
- Existen dos tipos de datos compuestos:
 - Arrays
 - Agregados u objetos o documentos embebidos.

Arrays

- Se identifican porque sus elementos se muestran separados por , (comas), y con [] (corchetes) como delimitadores.
- El índice del primer elemento es el 0.
Sintaxis: [<elemento0>, <elemento1>, <elementoN-1>]
- Son polimórficos, es decir, sus elementos pueden ser de tipos heterogéneos.
Ej. "datos": ["uno", 2, new Date()]

Operadores de consulta sobre arrays

NOMBRE	DESCRIPCIÓN
\$all	Comprueba que un array contiene, al menos, todos los elementos indicados, independientemente del orden en el que aparezcan.
\$elemMatch	Comprueba si algún elemento en un array cumple una serie de condiciones
\$size	Devuelve el tamaño del array.

Operadores de actualización sobre arrays I

NOMBRE	DESCRIPCIÓN
\$ (update)	El operador posicional \$ identifica un elemento de un array para actualizarlo sin especificar explícitamente la posición del elemento en el array.
\$	El operador posicional actúa como marcador de posición para la primera coincidencia del documento de consulta de actualización.
\$[]	El operador “todas las posiciones” actúa como marcador de posición para todos los elementos de un campo de tipo array.
\$[<identificador>]	Permite filtrar datos de un array por su posición
\$addToSet	Agrega un valor a un campo de tipo array a matriz a menos que el valor ya esté presente, en cuyo caso no hace nada.
\$pop	Elimina el primer o el último elemento de un array.
\$pull	Elimina todas las instancias de un elemento de un campo de tipo array.


Modificadores sobre operadores de actualización sobre arrays

NOMBRE	DESCRIPCIÓN
\$each	Modifica los operadores \$push y \$addToSet para añadir varios elementos en la actualización de un array.
\$position	Modifica el operador \$push para especificar la posición en el array en la que agregar elementos.
\$slice	Modifica el operador \$push para limitar el tamaño del array actualizado.
\$sort	Modifica el operador \$push para reordenar los documentos almacenados en un array.

Operaciones de actualización sobre arrays II

NOMBRE	DESCRIPCIÓN
\$push	Añade un elemento a un campo de tipo array. Añade el campo array si no existe. Convierte un campo en un campo de tipo array, si antes era de otro tipo.
\$pullAll	Elimina todos los elementos que cumplen una condición de un campo de tipo array.

Consultas sobre arrays I

- El orden en que los elementos aparecen en un array es significativo, es decir, importa.
`["antes", "después"]  ["después", "antes"]`
- Se puede consultar si un valor pertenece a un campo array como si fuera un campo de texto cualquiera.
Ej. Películas en las que actúa John Wayne:
`db.películas.find("actores": "John Wayne ")`
- Se puede consultar si un valor pertenece a un campo array y es el único elemento de ese array.
Ej. Películas en las que el único actor es Tom Hanks :
`db.películas.find("actores": ["Tom Hanks"])`
- Búsqueda de múltiples elementos, independientemente del orden:
Ej. Hoteles que tengan en la habitación al menos los servicios de "Aire acondicionado", "Mini bar" y "TV":
`db.hoteles.find("servicios": { "$all": ["Aire acondicionado", "Mini bar", "TV "] })`

Consultas sobre arrays II

- Búsqueda fijando el tamaño de un campo de tipo array.

Ej. Hoteles que tengan en la habitación al menos los servicios de "Aire acondicionado", "Mini bar" y "TV", y que tengan exactamente 5 servicios:

```
db.hoteles.find( "servicios": { "$size": 5, "$all": ["Aire acondicionado", "Mini bar", "TV "] } )
```

Operador **\$all** |

- Selecciona los documentos en los que el valor de un campo es un array que contiene todos los elementos especificados.

Sintaxis: { <campo>: { **\$all**: [<valor1> , <valor2> ...] } }

- **\$all es equivalente a la operación \$and** de los valores especificados:

Ej. Estas dos expresiones son equivalentes:

{ tags: { **\$all**: ["ssl" , "seguridad"] } }

{ **\$and**: [{ tags: "ssl" }, { tags: "seguridad" }] }

Operador **\$all** ||

- En arrays anidados (ej. [["A"]]), el operador **\$all** encuentra documentos en los que el campo contiene el array anidado como elemento (ej. [["A"], ...]), o el campo igual al array anidado (ej. ["A"]).
- Ej. Estas tres consultas son equivalentes:
db.articulos.find({ tags: { **\$all**: [["ssl", "seguridad"]] } })
db.articulos.find({ **\$and**: [{ tags: ["ssl", "seguridad"] }] })
db.articulos.find({ tags: ["ssl", "seguridad"] })
- Por lo tanto, la expresión **\$all** encuentra documentos donde el campo tags es un array que contiene el array anidado ["ssl", "seguridad"] o es un array igual al array anidado:
tags: [["ssl", "seguridad"], ...]
tags: ["ssl", "seguridad"]

Operador **\$all** III

- El operador **\$all** se usa para realizar consultas sobre arrays, pero se puede usar también para realizar consultas sobre campos que no sean arrays.

Ej. Estas dos consultas seleccionan todos los documentos de la colección inventory donde el valor del campo numérico cantidad es 50.

```
db.inventory.find( { "cantidad": { $all: [ 50 ] } } )
```

```
db.inventory.find( { "cantidad" : 50 } )
```

Operador **\$elemMatch** |

- Se utiliza con dos funciones:
 1. Recupera documentos que contengan un campo de tipo array **con al menos un elemento** que cumpla los criterios de consulta.
- Sintaxis: { <campo_array> : { "**\$elemMatch**": { <elemento> : <criterio> } } }

Ej. Muestra el campo nota del array notas cuyo valor sea mayor que 85 y que el documento tenga como id_clase el valor 431:

```
db.notas.find( {"id_clase": 431 },
```

```
  {"notas": { "$elemMatch": { "nota": { "$gt": 85 } } } }  
  )
```

Condiciones

Proyecciones

Operador **\$elemMatch** ||

2. Muestra únicamente todos los elementos del campo array en el que al menos uno de ellos cumpla los criterios de consulta.

Ej. Muestra todos los documentos en los que exista una nota cuyo tipo sea "Calificación extra" :

```
db.notas.find({ "notas": { "$elemMatch": { "tipo": "Calificación extra " } }  
  })
```

Condiciones

Ejemplo con **\$elemMatch**

```
db.empresas.find( { "relaciones": { "$elemMatch": {  
                                "es_pasado": true,  
                                "persona.nombre": "Mark" } } },  
                  { "nombre": 1 } )
```


Operadore **\$nin**

- **Selecciona** documentos en los que:
 - Ninguno de los valores indicados para el campo- existen en el array
 - El campo array no existe.

Sintaxis: { campo: { **\$nin**: [<valor1>, <valor2> ... <valorN>] } }

Ej. Establece el campo excluido a true para los documentos que no tengan el valor colegio en el array uso.

```
db.collection.updateMany( { uso: { $nin: [ "colegio" ] } }, { $set: { excluido: true } } )
```

Operador **\$size**

- Encuentra arrays que tengan el número de elementos especificados.
Ej. Devuelve los documentos que contengan un campo array datos que contenga dos elementos, es decir, que su tamaño sea 2:
`db.collection.find({ datos: { $size: 2 } })`
- **\$size** not acepta rangos of valores. Para seleccionar documentos con un campo array de diferentes tamaños, se debe crear un campo Contador que se incremete al añadir un element al array, y decremente al eliminarlo.
- Las consultas no ueden usar índices para la sección \$size de una consulta, aunque las otras secciones sí pueden (si procede).

\$size - Ejemplo

Ej. Devuelve el número de elementos del array colores de los documentos que contengan ese array. Si en algún documento no es un array, devuelve "No es un array".

```
db.inventario.aggregate( [ { $project: {  
    elemento: 1,  
    numeroDeColores: { $cond: { if: { $isArray: "$colores"  
    },  
    then: { $size: "$colores" },  
    else: "No es un array"} }  
} } ] )
```

Objetos

- Un documento se considera del tipo objeto.
- Un objeto puede contener campos simples, campos de array y otros objetos.

Ej. Dada una colección de personas, un documento podría ser:

```
{ "apellido" : "Smith" ,  
  "edad" : 76,  
  "direccion" : {  
    "calle" : "Avenida de la naturaleza",  
    "numero" : 15,  
    "localidad" : "Villanueva del castillo"  
  },  
  "Hijos" : [ "Lola", "Paco", "Charo" ]  
}
```

The diagram illustrates the structure of the JSON object with red curly braces and labels:

- Campo simple**: Points to the value "Smith" for the "apellido" field.
- Campo simple**: Points to the value 76 for the "edad" field.
- Campo simple**: Points to the value "Avenida de la naturaleza" for the "calle" field.
- Campo simple**: Points to the value 15 for the "numero" field.
- Campo simple**: Points to the value "Villanueva del castillo" for the "localidad" field.
- Campo Objeto**: A bracket groups the "calle", "numero", and "localidad" fields, indicating they are part of a nested object.
- Objeto**: A large bracket on the right groups the entire JSON structure, indicating it is a single object.

Operadores de comparación

Nombre	Equivalente matemático	Descripción
\$eq	=	Comprueba que dos valores son iguales
\$gt	>	Comprueba que un valor es mayor que otro
\$gte	≥	Comprueba que un valor es mayor o igual que otro
\$in	∈	Comprueba que un valor coincide con alguno de los existentes en un array
\$lt	<	Comprueba que un valor es menor que otro
\$lte	≤	Comprueba que un valor es menor o igual que otro
\$ne	≠	Comprueba que dos valores son distintos
\$nin	∉	Comprueba que un valor no coincide con ninguno de los existentes en un array

Ejemplos de uso de los operadores de comparación

```
db.coleccion.find( { qty: { $eq: 20 } } )
```

```
db.coleccion.find( { qty: { $ne: 20 } } )
```

```
db.coleccion.find( { qty: { $gt: 20 } } )
```

```
db.coleccion.find( { qty: { $gte: 20 } } )
```

```
db.coleccion.find( { qty: { $lt: 20 } } )
```

```
db.coleccion.find( { qty: { $lte: 20 } } )
```

```
db.coleccion.find( { quantity: { $in: [ 5, 15 ] } }, { _id: 0 } )
```

```
db.coleccion.find( { quantity: { $nin: [ 5, 15 ] } }, { _id: 0 } )
```

Operadores lógicos

NOMBRE	DESCRIPCIÓN
\$and	Vincula condiciones de una consulta con el operador AND. Es el operador por defecto cuando no se especifica un operador.
\$or	Vincula condiciones de una consulta con el operador OR
\$nor	Vincula condiciones de una consulta con el operador NOR. Es una combinación de NOY y OR.
\$not	Niega una condición.

- $\$nor = \$not (\$or)$

Operador **\$and**

- Es el operador por defecto cuando no se especifica un operador.
- Estos pares de condiciones son equivalentes:

```
{ "$and": [ {sector: "Mobile Food Vendor - 881"}, {result: "Warning"} ] }
```

```
{ sector: "Mobile Food Vendor - 881", result: "Warning" }
```

```
{ "$and": [ {"student_id": { "$gt": 25}}, {"student_id": { "$lt": 100}} ] }
```

```
{ "student_id": { "$gt": 25, "$lt": 100} }
```

Ejemplos de uso de los operadores lógicos I

```
db.coleccion.find( { $and: [
    { x: { $ne: 0 } },
    { $expr: { $eq: [ { $divide: [ 1, "$x" ] }, 3 ] } }
]
} )
```

```
db.coleccion.find( { $and: [
    { price: { $ne: 1.99 } },
    { price: { $exists: true } }
]
} )
```

```
db.coleccion.find( { price: { $ne: 1.99, $exists: true } } )
```

Ejemplos de uso de los operadores lógicos II

```
db.coleccion.find( { $and: [  
    { $or: [ { qty: { $lt : 10 } }, { qty : { $gt: 50 } } ] },  
    { $or: [ { sale: true }, { price : { $lt : 5 } } ] }  
]  
})
```

```
db.coleccion.find( { price: { $not: { $gt: 1.99 } } } )
```

```
db.coleccion.find( { item: { $not: /^p.* / } } )
```

```
db.coleccion.find( { item: { $not: { $regex: /^p.* / } } } )
```

```
db.coleccion.find( { item: { $not: { $regex: "p.*" } } } )
```

Ejemplos de uso de los operadores lógicos III

```
db.coleccion.find( { $nor: [ { price: 1.99 }, { sale: true } ] } )
```

```
db.coleccion.find( { $nor: [ { price: 1.99 }, { qty: { $lt: 20 } }, { sale: true } ] } )
```

```
db.coleccion.find( { $nor: [ { price: 1.99 }, { price: { $exists: false } },  
                             { sale: true }, { sale: { $exists: false } } ] } )
```

```
db.coleccion.find( { $or: [ { quantity: { $lt: 20 } }, { price: 10 } ] } )
```

```
db.inventory.createIndex( { quantity: 1 } )
```

```
db.inventory.createIndex( { price: 1 } )
```

Operadores consulta sobre elementos

NOMBRE	DESCRIPCIÓN
\$exists	Comprueba si un documento contiene un determinado campo
\$type	Comprueba si un campo es de un determinado tipo

Operadores de evaluación

NOMBRE	DESCRIPCIÓN
\$expr	Permite expresiones de agregación
\$jsonSchema	Valida documentos contra un esquema JSON
\$mod	Operación resto (módulo)
\$regex	Comprueba si se cumple una cierta expresión regular
\$text	Realiza búsqueda en texto
\$where	Comprueba los documentos cumplen una condición JavaScript

Operador **\$expr**

- Sirve para comparar campos de un mismo documento.

- La sintaxis es:

`{ $expr: { <expresión> } }`

Ej. Comprobar los viajes en bicicleta cuya estación de origen es la misma que la de destino. El valor del campo id estación inicio (el código de la estación), que se representa por "\$id estacion inicio" debe ser igual que el de la estación de destino "\$id estacion fin"

`db.viajes.find({ "$expr": { "$eq": ["$id estacion inicio", "$id estacion fin"] } })`

- Permite usar:

- Funciones de agregación.
- Variables y sentencias condicionales.

Operadores de actualización sobre campos

NOMBRE	DESCRIPCIÓN
\$currentDate	Establece el valor de un campo en la fecha actual, ya sea un Date o un Timestamp.
\$inc	Incrementa el valor de un campo numérico.
\$min	Sólo actualiza el campo si el valor especificado es menor que el valor del campo existente
\$max	Sólo actualiza el campo si el valor especificado es mayor que el valor del campo existente.
\$mul	Multiplica el valor del campo por la cantidad especificada.
\$rename	Cambia el nombre de un campo.
\$set	Establece el valor de un campo. Añade el campo si no existe.

Ejemplos de actualización

- db.zips.updateMany({ "city": "HUDSON" }, { "\$inc": { "pop": 10 } })
- db.zips.updateOne({ "zip": "12534" }, { "\$set": { "pop": 17630 } })
- db.zips.updateOne({ "zip": "12534" }, { "\$set": { "population": 17630 } })
- db.grades.updateOne({ "student_id": 250, "class_id": 339 },
 { "\$push": { "scores": { "type": "extra credit",
 "score": 100 } } })

Operador \$

- Indica el uso de un operador.
Ej. \$or, \$lt...
- Representa el valor de un campo

Proyección

- Seleccionar los campos de uno o más documentos que se van a mostrar se llama **proyección**.
- Es el equivalente a los campos que se indican en el SELECT en SQL.
- Después de las condiciones de búsqueda (equivalentes a la cláusula WHERE en SQL), se indican los campos que se van a mostrar, asociándoles un 1.
- La sintaxis es:

```
db.<colección>.find( { <condiciones> }, { <proyecciones> } )
```

- Si se quiere no mostrar explícitamente un campo, como el `_id`, se le asignará un 0.

Ej. Seleccionar los hoteles que tengan exactamente 5 servicios, incluyendo obligatoriamente "Wifi", "Aire acondicionado" y "TV". De los hoteles seleccionados, **se mostrará su nombre, su dirección, y se ocultará su `_id`**.

```
db.hoteles.find( { "servicios": { "$size": 5,  
                                "$all": [ "Wifi", "Aire acondicionado", "TV" ] } },  
                { "_id": 0, "nombre": 1, "dirección": 1 } )
```

Operador . |

- Permite acceder a las estructuras anidadas existentes en un documento.
Ej. Un documento, que representa a una persona, en el que su campo dirección es un "objeto" que representa una dirección.

```
{ "apellido" : "Smith" ,  
  "edad" : 76,  
  "direccion" : {  
    "calle" : "Avenida de la naturaleza",  
    "numero" : 15,  
    "localidad" : "Villanueva del castillo"  
  }  
}
```

Para consultar las personas que viven en la Calle del Pez.

```
db.personas.find( {"direccion.calle": "Calle del Pez" } )
```

Operador . II

- Permite acceder a arrays por su posición.
- Sintaxis: { <campo_array>.<posición> }

Ej. Recupera los documentos de la **colección empresas** que contengan un **campo de tipo array llamado relaciones**, cuyo **primer elemento** (índice 0) contenga un **campo persona** (que es un objeto), que a su vez contiene un **campo apellido1** cuyo valor es **García**. Se mostrará sólo el nombre de la empresa y su campo `_id`.

```
db.empresas.find({ "relaciones.0.personas.apellido1": "García" },  
                  { "nombre": 1 } )
```

\$avg

- Dados los siguientes documentos de la colección ventas:

```
{ "_id" : 1, "item" : "abc", "precio" : 10, "cantidad" : 2, "fecha" : ISODate("2014-01-01T08:00:00Z") }  
{ "_id" : 2, "item" : "jkl", "precio" : 20, "cantidad" : 1, "fecha" : ISODate("2014-02-03T09:00:00Z") }  
{ "_id" : 3, "item" : "xyz", "precio" : 5, "cantidad" : 5, "fecha" : ISODate("2014-02-03T09:05:00Z") }  
{ "_id" : 4, "item" : "abc", "precio" : 10, "cantidad" : 10, "fecha" : ISODate("2014-02-15T08:00:00Z") }  
{ "_id" : 5, "item" : "xyz", "precio" : 5, "cantidad" : 10, "fecha" : ISODate("2014-02-15T09:12:00Z") }
```

- Se realiza la consulta:

```
db.ventas.aggregate( [ { $group: {  
    _id: "$item",  
    mediaPrecio: { $avg: { $multiply: [ "$price", "$quantity" ] } },  
    avgCantidad: { $avg: "$quantity" }  
    } } ] )
```


Métodos

Métodos de base de datos I

MÉTODO	USO
<code>createCollection()</code>	Crea una colección.
<code>createView()</code>	Crea una vista.
<code>dropDatabase()</code>	Elimina una base de datos.
<code>getCollectionInfos()</code>	Información sobre las colecciones y vistas en una BD.
<code>getCollectionNames()</code>	Nombres de las colecciones y vistas en una BD.
<code>getName()</code>	Nombre de la BD en uso.
<code>help()</code>	Información sobre los métodos del objeto db .

Métodos de base de datos II

MÉTODO	USO
<code>hostInfo()</code>	Información sobre el sistema subyacente.
<code>listCommands()</code>	Información de todos los comandos de BD.
<code>serverStatus()</code>	Información sobre el estado del proceso de base de datos.
<code>shutdownServer()</code>	Detiene el proceso mongod o mongos de forma limpia y segura.
<code>stats()</code>	Estadísticas sobre la BD en uso.
<code>use <nombre_bd></code>	Crea una BD, si no existe; la selecciona, si existe.
<code>version()</code>	La versión del proceso mongod o mongos en ejecución.

Métodos de colección I

MÉTODO	USO
aggregate()	Calcula valores de agregado para los datos de una colección o una vista.
<code>bulkWrite()</code>	Escritura masiva de documentos.
count()	Cuenta los documentos que cumplen las condiciones de filtrado.
countDocuments()	Cuenta los documentos que cumplen las condiciones de filtrado. Se apoya en el método aggregate() .
createIndex()	Crea un índice.
<code>createIndexes()</code>	Crea varios índices.
dataSize()	Tamaño en bytes de la colección en uso.

Métodos de colección II

MÉTODO	USO
deleteOne()	Elimina un documento al azar. Se recomienda usar siempre el filtrado por el campo <code>_id</code> para determinar el documento afectado.
deleteMany()	Elimina todos los documentos que cumplan un filtro. Si no hay filtro, elimina todos los elementos de la colección.
distinct()	Muestra los valores distintos para un campo.
drop()	Elimina una colección o vista, incluyendo los índices que contuviera.
dropIndex()	Elimina un índice.
dropIndexes()	Elimina varios índices, según las condiciones de filtrado.
estimatedDocumentCount()	Cuenta los documentos de una colección que cumplan un determinado filtro. Se apoya en el método count() .
explain()	Devuelve el plan de ejecución para algunos métodos como find() .

Métodos de colección III

MÉTODO	USO
find()	Devuelve los documentos que cumplen las condiciones de filtrado.
<code>findAndModify()</code>	Encuentra documentos y los modifica.
findOne()	Devuelve un documento al azar que cumpla las condiciones de filtrado.
<code>findAndDelete()</code>	Encuentra documentos y los elimina.
<code>findOneAndReplace()</code>	Encuentra un documento y lo sustituye.
<code>findOneAndUpdate()</code>	Encuentra un documento y lo actualiza.
getIndexes()	Lista todos los índices de una colección.

Métodos de colección IV

MÉTODO	USO
<code>hideIndex()</code>	Oculta un índice existente al planificador de consultas, de manera que éste ya no es evalúa como parte de la selección del plan de consulta.
<code>insertOne()</code>	Inserta un único documento en una colección.
<code>insertMany()</code>	Inserta múltiples documentos en una colección.
<code>isCapped()</code>	Informa sobre si una colección es limitada.
<code>renameCollection()</code>	Renombra un a colección.
<code>replaceOne()</code>	Reemplaza un documento.
<code>stats()</code>	Devuelve estadísticas.

Métodos de colección V

MÉTODO	USO
storageSize()	La cantidad total de almacenamiento (en bytes) reservada para que una colección almacene documentos.
totalIndexSize()	Devuelve el tamaño total de los índices.
totalSize()	Devuelve el tamaño total de la colección.
unhideIndex()	Activa un índice que estaba desactivado.
updateOne()	Actualiza un documento.
updateMany()	Actualiza múltiples documentos.
validate()	Valida una colección.

Comandos de administración para consulta I

- Usados para consultar información sobre estructuras de datos (bases de datos, documentos, índices...), usuarios...

USO	COMANDO
Documento con información sobre el SO subyacente	db.hostInfo()
Documento con información sobre el servidor de MongoDB	db.serverStatus()
Documento con información sobre el servidor de MongoDB	db.serverBuildInfo()
Documento con información sobre todos los comandos de la BD	db.listCommands()
Documento que describe el rol de la instancia MongoDB	db.hello()
Listado de comandos sobre una BD	db.help()
Versión de mongod o mongos	db.version()

Comandos de administración para consulta II

USO	COMANDO
Documento con información sobre las operaciones en curso sobre las instancia de la BD	db.currentOp()
Nombre de la BD en uso	db.getName()
Documento con información de la BD en uso	db.stats()
Documento con información por cada colección existente de la BD en uso	db.getCollectionsInfo()
Array con los nombres de las colecciones de la BD en uso	db.getCollectionNames()
Nombres de los índices definidos en una colección	db.<colección>.getIndexes()
Número de documentos en una colección	db.<colección>.countDocuments()

Comandos de administración para consulta III

USO	COMANDO
Nombres de los campos de un documento	Object.keys(db.<colección>.findOne())

Comandos de administración para acciones

USO	COMANDO
Detiene el proceso mongod o mongos actual de manera limpia y segura	db.shutdownServer()
Termina una operación usando de su ID	db.killOp()

Funciones de creación de objetos

Función **use** <**base_datos**>

- Si existe, la selecciona.
- Si no existe, la crea.

Función **dropDatabase()**

- Se utiliza para eliminar una base de datos.

Sintaxis: db.**dropDatabase()**

Función `createCollection()` |

- Se utiliza para crear una colección.

Sintaxis: `db.createCollection(<nombre_colección>, <opciones>)`

- Existen colecciones:

- **Acotada (*capped*)**. Tienen un límite en el tamaño y en el número de documentos que contienen, para evitar un crecimiento descontrolado. Al crearlas hay que definir estos límites.

Ej. Creación de una colección acotada a 5MB y 5000 documentos:

```
db.createCollection("registro", { capped : true, size : 5242880, max : 5000 } )
```

"registro", nombre de la colección.

capped : true, la colección es **acotada**.

size: 5242880, el tamaño máximo de la colección es de 5242880 bytes (5MB).

max: 5000, el número máximo de documentos en la colección es 5000.

Función `createCollection()` II

- En cluster (*clustered*).

Ej. `db.createCollection("acciones",`

`{ clusteredIndex: { "key": { _id: 1 }, "unique": true,`

`"name": "clave en cluster de acciones" } })`

- `"key": { _id: 1 }`, el campo `_id` es la clave del índice en cluster.
- `"unique": true`, la clave del índice en cluster debe ser única.
- `"name": "clave en cluster de acciones "`, el nombre del índice en cluster.

Función **renameCollection()**

- Renombra una colección.

Sintaxis: db.<colección>.**renameCollection**(<nuevo_nombre>)

Función **drop** () |

- Elimina una colección.

Sintaxis: db.<colección>.**drop**()

Función **createView()** |

- Se utiliza para crear una vista.

Sintaxis: **db.createView**(<vista>, <opciones>)

Función **createIndex()** |

- Se utiliza para crear un índice.

Sintaxis: **db.<colección>.createIndex(<índice>, <opciones>)**

Función `getIndexes ()` |

- Lista todos los índices creados en una colección.

Sintaxis: `db.<colección>.getIndexes()`

Función **dropIndex** () |

- Elimina un índice de una colección.

Sintaxis: **db.<colección>.dropIndex(<índice>, <opciones>)**

Consulta y manipulación de documentos

Operaciones de CRUD

- Create
- Read
- Update
- Delete
- Sobre documentos, arrays...

Función **find()** |

- Permite consultar documentos. El resultado de la consulta no está ordenado de ninguna manera. Es semejante a la clausula SELECT de SQL.

Sintaxis: db.<colección>.**find**(<condiciones_filtrado>, <proyección>)

Ej. Todos los documentos de una colección:

db.<colección>.find()

- Paralelismo con la sentencia SELECT de SQL.

Ej. Mostrar el nombre y la edad de las personas de género masculino.

SQL: **SELECT** nombre, edad **FROM** personas **WHERE** genero="Masculino";

MQL: db.personas.find({"genero": "Masculino"},
{"_id": 0, "nombre": 1, "edad": 1})

Función **find()** II

- Se pueden definir condiciones para la búsqueda:
Ej. Proveedores españoles:
`db.proveedores.find({ "pais": "España" })`
- Se pueden definir varias condiciones que debe cumplir un documento. Estas condiciones se vinculan con el operador lógico AND.
Ej. Proveedores españoles de Soria:
`db.proveedores.find({ "pais": "España", "ciudad": "Soria" })`
- Con **count()** se puede contar el número de resultados de una consulta:
`db.<colección>.find(<consulta>).count()`
Ej. Número de pedidos cuyo precio sea mayor que 1000.
`db.pedidos.find({ "precio" $gt 1000 }).count()`

Función **findOne()** |

- Es semejante a `find()`, solo que devuelve un único documento de una colección, elegido aleatoriamente.

Sintaxis: `db.<colección>.findOne()`

- Se pueden definir condiciones que debe cumplir el documento devuelto.

Ej. Un proveedor español:

```
db.proveedores.findOne( {"pais": "España"} )
```

Ejemplos de equivalencias con SQL

- El nombre de la colección para MQL coincide con el de la tabla para SQL.

MQL	SQL
db.<colección>.find({})	SELECT * FROM <tabla>;
db.<colección>.find({ status: "D" })	SELECT * FROM <tabla> WHERE status = "D";
db.<colección>.find({ status: { \$in: ["A", "D"] } })	SELECT * FROM <table> WHERE status IN ("A", "D")
db.<colección>.find({ status: "A", qty: { \$lt: 30 } })	SELECT * FROM <table> WHERE status = "A" AND qty < 30;

Ejemplos de equivalencias con SQL II

MQL	SQL
db.<colección>.find({ \$or: [{ status: "A" }, { qty: { \$lt: 30 } }] })	SELECT * FROM inventory WHERE status = "A" OR qty < 30
db.<colección>.find({{ status: "A", \$or: [{ qty: { \$lt: 30 } }, { item: /^p/ }] } }	SELECT * FROM inventory WHERE status = "A" AND (qty < 30 OR item LIKE "p%")

Ejemplos de consultas I

```
db.coleccion.find({etiqueta : {$in : ["Libro", "BluRay"]} },  
                  {_id : 0, descripcion : 1})
```

```
db.coleccion.find({descripcion: "Thermomix"})
```

```
db.coleccion.find({}, {precio:1, descripcion:1})
```

```
db.coleccion.find({descripcion: "Thermomix"}, {_id:0, precio:1})
```

```
db.coleccion.find({etiquetas: "consolas"}, {_id:0, descripcion:1})
```

```
db.items.find({etiquetas: "consolas", etiquetas: "wii"},  
              {_id:0, descripcion:1})
```

```
db.coleccion.find({etiquetas: {$in:["consolas", "mensaje"]}},  
                  {_id:0, descripcion:1})
```

Consulta en sub-agregados

```
db.coleccion.find({"contraofertas.oferta": {$gt: 5}})
```

```
var feb2020 = new Date(2020, 2, 1)
```

```
db.coleccion.find({"contraofertas.fecha": {$lt: feb2020}} )
```


Operaciones de agregación

- Conteo

`db.coleccion.count()`

- Ordenación

`db.coleccion.find().sort()`

- Valores distintos:

`db.coleccion.distinct()`

- Agrupación

`db.coleccion.group()`

Ejemplos de consulta

```
db.coleccion.distinct("persona.apellido")
```

```
db.coleccion.count({"direccion.ciudad": "Madrid"})
```

```
db.coleccion.find({"direccion.ciudad": "Madrid"}).count()
```

```
db.coleccion.find().sort( { "direccion.ciudad": 1 } )
```

```
db.coleccion.find({"direccion.cp": {$regex: /^08.*$/}})
```

```
    .sort( { "direccion.ciudad": 1 } )
```

```
db.coleccion.distinct("vendedor.email")
```

Métodos de cursor

- No se aplican sobre los datos almacenados en la BD, sino sobre los datos recuperados por una consulta, ubicados en un cursor.
 - `limit()`
 - `sort()`
 - `count()`
 - `pretty()`
- Cuando se usan `sort()` y `limit()` conjuntamente, siempre se debe usar antes `sort()` que `limit()`. MongoDB lo aplica automáticamente de esta manera, aunque se escriba la consulta con `limit()` primero y `sort()` después.

Método **distinct()**

- Obtiene valores únicos para un determinado campo.

Método **sort()**

- Permite ordenar los datos obtenidos por una consulta.
- Se puede ordenar:
 - Ascendente (valor 1) y descendentemente (valor -1).
 - Por uno o más criterios.

Ej. Obtener todas las localidades y ordenar la salida por población (ascendentemente = 1) y por ciudad (descendentemente = -1).

```
db.localidades.find().sort( { "poblacion": 1, "ciudad": -1 } )
```
- Valores **null** en un campo. Si hay valores **null** en un campo, al ordenar por ese campo aparecerán como los menores de todos. Por ello, cuando en un campo hay valores nulos y se va a ordenar por ese campo, se recomienda filtrar previamente para eliminar los documentos que tengan valor nulo en ese campo.

Método **limit()**

- Indica el número de resultados de una consulta que se mostrarán.
Ej. Ordena ascendentemente las localidades por el campo población y muestra la primera, es decir, la menos poblada:
`db.localidades.find().sort({ "poblacion": 1 }).limit(1)`
Ej. Ordena descendientemente las localidades por el campo población y muestra la primera, es decir, la más poblada:
`db.localidades.find().sort({ "poblacion": -1 }).limit(1)`
Ej. Ordena descendientemente las localidades por el campo población y muestra las diez primeras, es decir, las más pobladas:
`db.localidades.find().sort({ "poblacion": -1 }).limit(10)`

Método **skip()**

Inserción de documentos

Función **insertOne** () |

- Inserta un documento en una colección.

Sintaxis: db.<colección>.**insertOne**(<documento>)

- Si la colección no existiese, MongoDB la creará automáticamente sin producir ningún error.
- Devuelve un documento informativo de la operación.

Ej. db.<colección>.**insertOne**({"a": 3})

{ "acknowledged" : true,
"insertedId" : ObjectId("571a218011a82a1d94c02333") }

La operación
fue bien

Id asignado
automáticamente al
nuevo documento

Método **insertOne** () ||

- También permite insertar un array de documentos.
Sintaxis: db. <colección>.**insertOne**([<documento1>, <documento2>, <documento3>])
- Cuando se inserta un array, la inserción se realiza en el orden en el que aparecen los documentos listados en el array. Si se produce un error de clave duplicada, se suspende la operación de inserción.
- Si se quiere cambiar esto, se debe añadir la opción { "**ordered**": **false**}.

Método **insertMany** () |

- Permite insertar varios documentos en una colección.
Sintaxis: db.<colección>.**insertMany**(<array_documentos>)
- Devuelve un documento informativo del proceso de inserción.

Ej. db.<colección>.**insertMany**([{ "b": 3 }, { 'c': 4 }])
{ "acknowledged" : true,
 "insertedIds" : [ObjectId("571a22a911a82a1d94c02337"),
 ObjectId("571a22a911a82a1d94c02338")] }

Ej.

```
db.postres.insertMany( [  
  { _id: 1, categoria: "pastel", tipo: "chocolate", cantidad: 10 },  
  { _id: 2, categoria: "pastel", tipo : "fresa", cantidad: 25 },  
  { _id: 3, categoria: "tarta", tipo : "zanahoria", cantidad: 20 },  
  { _id: 4, categoria: "tarta", tipo : "frutas", cantidad: 15 }  
)
```

Inserción I

- Al insertar un documento, se realizan una serie de operaciones a fin de evitar inconsistencias posteriores:
 - Se añade el campo **_id** en caso de no tenerlo.
 - Se comprueba que el tamaño del documento no exceda un cierto tamaño. Para conocer el tamaño de un documento se puede usar el comando **Object.bsonsize (doc)**.
 - Se verifica la no existencia de caracteres no válidos
- Ej
- ```
db.ejemplo.insertOne({"titulo", "Un mundo feliz", "autor", "Aldous Huxley", "anioPublicacion" : 1932})
```
- Se pueden insertar documentos uno por uno, o bien crear un array de documentos y hacer una única inserción.
  - Cuando se inserta usando un array, si se produce algún fallo en algún documento, se insertan todos los documentos anteriores al que produjo el fallo, descartando los posteriores.
  - Este comportamiento se puede cambiar con la opción **continueOnError** que, en el caso de encontrarse un error en un documento, lo salta, y continúa insertando el resto de los documentos.

# Inserción II

- Se puede insertar directamente un documento.

```
db.coleccion.insertOne({
 descripcion: "Cazuela",
 fecha: new Date ("2020, 5, 2"),
 precio: 15,
 etiquetas: ["cocina", "menaje"],
 vendedor: { email: "ffernandez@gmail.com",
 psw: "ffernandez" },
 localizacion: {longitude: 38.743671, latitude: -10.552276},
 estado: "disponible"
})
```

# Inserción III

- Se puede definir una variable, asignarle un documento y luego insertar la variable.

```
item1 = {
 descripcion: "Mando xBox negro",
 fecha: new Date ("2015, 3, 21"),
 precio: 10,
 etiquetas: ["consolas", "xbox", "entretenimiento"],
 vendedor: {email: "pperez@gmail.com", psw: "pperez"},
 localizacion: {longitude: 37.743671, latitude: -2.552276},
 estado: "disponible",
 contraofertas: [{
 email: "llopez@gmail.com",
 psw: "llopez",
 oferta: 8,
 fecha: new Date("2020, 4, 2")},
 {
 email: "ggomez@gmail.com",
 psw: "ggomez",
 oferta: 7,
 fecha: new Date("2020, 4, 13")}
]
}

db.items.insertOne(item1)
```

# Inserción

- Se pueden definir varias variables, asignarles un documento a cada una, añadir esas variables a un array y, finalmente, insertar el array.

```
usr1={
 nombre: "Luis López",
 email: "llopez@gmail.com",
 psw: "llopez",
 direccion: {
 via: "C/Pez", num:3, ciudad: "Albacete", cp: "28031"}},

usr2={
 nombre: "Francisco Fernández",
 email: "ffernandez@gmail.com",
 psw: "ffernandez",
 direccion: {
 via: "C/Luna Nueva", num:145, ciudad: "Murcia", cp:
"030005"}},
usr3={
 nombre: "Gema Gomez",
 email: "ggomez@gmail.com",
 psw: "ggomez",
 direccion: {
 via: "C/Sansa", num:28, ciudad:"Valencia", cp: "46015"}}

db.coleccion.insertMany([usr1, usr2, usr3])
```

Actualización de documentos



# Actualización de documentos I

- Cuando se intenta realizar una operación de actualización sobre un campo que no existe, éste se crea de manera automática.

# Método **updateOne** ( ) |

- Actualiza un documento cualquiera de la colección.

Sintaxis: db.<colección>.**updateOne**( { <filtro> }, { <actualización> } )

Ej. En el documento con valor 10 para su `_id`, añade un campo estado, con valor "D", e incrementa en 2 el valor del campo cantidad.

```
db.<colección>.updateOne({ "_id": 10 },
 { $set: { "estado": "D" }, $inc: { "cantidad": 2 } })
```

- Siempre se debe usar indicando el `_id` del documento a actualizar. Sino, se deja "en manos de MongoDB" la decisión de qué documento actualizar, lo que no es muy recomendable.

Ej. Actualiza el documento cuyo `_id` es 100.

```
db.<colección>.updateOne({ "_id": 100 }, ...)
```

# Método **updateMany** ( ) |

- Actualiza todos los documento de los que cumplen el criterio de actualización.

Sintaxis: db.<colección>.**updateMany**( { <filtro> }, { <actualización> } )

Ej. Incrementar en 100 unidades el stock de todos los productos cuya categoría sea pasta:

```
db.productos.updateMany({ "categoria": "Pasta" }, { "$inc": { "stock": 100 } })
```

# Actualización I

- La función **updateOne()** modifica sólo un documento. Si no se indica ninguna condición, modifica uno al azar.
- Para modificar varios, hay que añadir el parámetro **multi**.

Ej. Modificar uno o múltiples documentos.

```
db.coleccion.updateOne({ }, { $set : { "nivel": "Medio" } })
```

```
db.coleccion.updateOne ({ }, { $set: { "nivel ": "Medio" } }, {multi: true})
```

- También se pueden modificar varios documentos a la vez con la función **updateMany()**.

Ej. `db.coleccion.updateMany( { }, { $set: { nivel : "Alto" } } )`

## Actualización II

- Para actualizar sólo los documentos que cumplan alguna condición.

```
db.coleccion.updateOne(
 {descripcion: "Mando xBox negro"},
 {$set: {estado: "vendido",
 comprador: {email: "llopez@gmail.com",
 psw: "llopez"}}}
 })
```

# Adicción de campos

- Adicción de un campo.

Sintaxis: db.<colección>.updateMany( { }, { \$set: { <nuevo\_campo>: <valor> } } )

- Adicción de un campo usando valores de campo(s) existente(s).

Sintaxis: db.<colección>.updateMany( { }, [ { \$set: { <nuevo\_campo>: \$<campo\_existente> } } ] )

Ej. Adicción del campo nombre\_completo, a partir de los campos nombre y apellidos, ambos existentes:

```
db.<colección>.updateMany({ }, [{ $set: { "nombre_completo": { $concat: ["$nombre", " ", "$apellidos"] } } }])
```

Ej. Crear un campo dos\_pi a partir de uno existente, pi, multiplicado por 2:

```
db.<colección>.updateMany({ }, [{ $set: { dos_pi: { $multiply: [$pi, 2] } } }])
```

# Renombrado de campos

- Renombrado de un único campo.

Sintaxis: db.<colección>.**updateMany**( { }, { \$rename: { <campo\_viejo>: <campo\_nuevo> } } )

- Renombrado de múltiples campos.

Sintaxis: db.<colección>.**updateMany**( { }, { \$rename: { <campo\_viejo1>: <campo\_nuevo1>, <campo\_viejo2>: <campo\_nuevo2>, ... } } )

- Renombrar un subcampo.

Sintaxis: db.<colección>.**updateMany**( { }, { \$rename: { <campo.subcampo\_viejo1>: <campo.subcampo\_nuevo2> } } )

# Eliminación de campos

- Eliminación de un campo.

Sintaxis: db.<colección>.updateMany( { }, { \$unset: { <campo\_a\_eliminar>: 1 } } )

- Eliminación de múltiples campos.

Sintaxis: db.<colección>.updateMany( { }, { \$unset: { <campo\_a\_eliminar1>: 1, <campo\_a\_eliminar2>: 1, ... } } )



# Upsert I

- Es una combinación de las operaciones de actualización (*update*) e inserción (*insert*).  
    **upsert** = **update** + **insert**.  
    Sintaxis: db.<colección>.updateMany( <condiciones>, <actualizaciones>, { **upsert: true** } )
- Se quiere actualizar un documento de una colección pero, en caso de que no exista el documento que se quiere modificar, se añadirá combinando el criterio de búsqueda y la actualización.
- Es semejante a la función **MERGE** de Oracle.
- Se realiza partiendo del método updateOne().
- Su valor por defecto es *false*.

# Upsert II

Ej. Modificar el documento cuyo campo email vale "ffernandez@gmail.com", y su campo psw vale "ffernandez2 ". Si el documento existe, se actualiza; si no existe. En ambos casos se utilizan los valores definidos en la sección **\$set**.

```
db.coleccion.updateOne(
 { email: "ffernandez2@gmail.com", psw: "ffernandez2" },
 { $set: { email: "ffernandez@gmail.com",
 psw: "ffernandez",
 direccion: {
 via: "C/Escalona", num:8, ciudad: "Barcelona" } } },
 { upsert: true }
)
```

# Ejemplo Upsert I

- Actualizar una lectura de un sensor:
  - Añadir la nueva lectura al array de lecturas (modificarlo), si ya existen lecturas previas para ese sensor en la misma fecha y si el número de lecturas es menor o igual que 64.
  - Insertar un nuevo documento para el sensor, si no existen lecturas previas para ese sensor en la misma fecha o si el número de lecturas es mayor que 64.
- Se parte de una colección con documentos como el que sigue, que representa la información de un sensor (se id 3), del que existen dos lecturas (cada lectura tiene un v(alar) y la h(ora) a la que se tomó):

```
{ "_id": ObjectId("123456abcdef"),
 "id_sensor": 3,
 "fecha": Date("2022-03-18"),
 "contador_lecturas": 6,
 "suma_lecturas": 57,
 "lecturas": [{ "v": 25, "h": "0000" }, { "v": 32, "h": "0005" }]
}
```

# Ejemplo Upsert II

- Se recibe una nueva lectura:

```
lectura = {
 "id_sensor": 3,
 "fecha": Date("2022-03-18"),
 "valor": 28,
 "hora": "0010" }
```

**CONDICIONES DE  
FILTRADO**

- Se realiza la operación de actualización:

```
db.medidas.updateOne({ "id_sensor": lectura.id_sensor, "fecha": lectura.fecha,
 "contador_lecturas": { "$lte": 64 } },
```

## **ACTUALIZACIONES**

- Inserción de un elemento en el array de lecturas
- Incremento de los campos contador\_lecturas y suma\_lecturas

```
{ "$push": { "lecturas": { "v": lectura.valor, "t": lectura.hora } },
 "$inc": { "contador_lecturas": 1,
 "suma_lecturas": lectura.valor } },
 { "upsert": true })
```

# Ejemplo Upsert III

- Las condiciones que determinan la operación a realizar son:  
    "id\_sensor": lectura.id\_sensor,  
    "fecha": lectura.fecha,  
    "contador\_lecturas": { "\$lte": 64 }
- Si se cumplen las condiciones, se actualiza el documento ya existente.
- Si no se cumplen, se inserta un nuevo documento generado a partir de los valores de las condiciones y las actualizaciones:  
    { "\_id": ObjectId("345678cdefgh"),  
      "id\_sensor": 3,  
      "fecha": Date("2022-03-18"),  
      "contador\_lecturas": 1,  
      "suma\_lecturas": 28,  
      "lecturas": [ { "v": 28, "h": "0010" } ]  
    }

# Otros ejemplos de consultas I

```
db.coleccion.updateMany(
 {"direccion.via": "C/Escalona"},
 {$set: {"direccion.via": "C/Ancha"}})
```

```
db.coleccion.updateOne(
 {descripcion: "Thermomix"},
 {$addToSet: {
 contraofertas: {email:
 "ggomez@gmail.com",
 psw: "ggomez",
 oferta: 73,
 fecha: new Date("2020, 5,
15")}}
 })
```

## Otros ejemplos de consultas II

```
db.coleccion.updateOne(
 {descripcion: "Thermomix"},
 {$inc: {precio: -7},
 $set: {estado: "vendido",
 comprador: {email: "ggomez@gmail.com",
 psw: "ggomez"}}}
)
```

```
db.coleccion.updateMany(
 {etiquetas: "entretenimiento"},
 {$pull: {etiquetas: "entretenimiento"}})
```

Eliminación de documentos



# Función **deleteOne** ( ) |

- Elimina un documento cualquiera de una colección.

Sintaxis: db.<colección>.**deleteOne**( <filtro> )

- Siempre se debe usar indicando el `_id` del documento a eliminar. Sino, se deja “en manos de MongoDB” la decisión de qué documento eliminar, lo que no es muy recomendable.

Ej. Elimina el documento cuyo `_id` es 100.

db.<colección>.**deleteOne**( { "\_id": 100 } )

# Función **deleteMany** ( ) |

- Elimina un documento cualquiera de una colección.

Sintaxis: db.<colección>.**deleteMany**( <filtro> )

Índices

# Índices I

- Los índices se usan para **optimizar las consultas**.
- Un índice en una BD es, funcionalmente, lo mismo que en una lista de contenidos/conceptos en un libro:
  - Habitualmente aparece al final del libro.
  - Consiste en una lista de conceptos, ordenada alfabéticamente, con referencias a las páginas del libro en las que aparece el concepto.
- En una BD, un índice se usa para lo mismo. Es una estructura que almacena conceptos y punteros referenciando a la ubicación de esos contenidos en una tabla, o en una colección.
- El mantenimiento de los índices también lleva un coste, así que **no se deben crear índices indiscriminadamente**.
- Se recomienda indexar los campos sobre los que se realizan más consultas.

# Índices II

- Sobre el campo `_id` se crea un índice automáticamente.
- Una forma de categorizar los índices es por el número de campos sobre los que se crean:
  - **Simple**, el índice está creado sobre un único campo
  - **Compuestos**, el índice está creado sobre dos o más campos. Los documentos se ordenarán primero por el primer campo, después por el segundo, etc.
- Otra forma de categorizar los índices es por la existencia de valores repetidos en el campo sobre el que se indexa:
  - **Únicos**, no existen valores repetidos en el campo sobre el que se indexa (ej. dni).
  - **Múltiples**, existen valores repetidos en el campo sobre el que se indexa (ej. provincia).

# Función **createIndex ( )** |

- Crea un índice sobre una serie de campos de una colección.  
Sintaxis: db.<colección>.createIndex( <campos> )  
Sintaxis: db.<colección>.createIndex( <campos>, nombre )
- Igual que con las proyecciones, se indican los campos sobre los que se quiere indexar fijando su valor a 1 para crear un índice con los valores del campo ordenados ascendentemente.  
Ej. Crear un índice sobre el campo dni de los documentos de la colección contribuyentes:  
db.contribuyentes.createIndex( { "dni": 1 } )
- Antes de la creación del índice, una consulta sobre el campo dni provocaría un recorrido completo de la colección. Tras su creación, la consulta irá primero al índice, obtendrá el identificador del documento (o documentos) que contenga el valor, y se “saltará” directamente a ellos.

# Función **createIndex ( )** ||

Ej. Devolver el documento del contribuyente con dni 12345678L:

```
db. contribuyentes.find({ "dni": "12345678L" })
```

Ej. Devolver los contribuyentes de la provincia de Soria, ordenadas por dni.

```
db. contribuyentes.find({ "provincia": "Soria" }).sort("dni": 1)
```

Esta consulta no usa el índice para la operación de filtrado ({ "dni": "12345678L" }), pero sí para la operación de ordenado del cursor resultante del filtrado (sort("dni": 1)). De hecho, no es necesario aplicar explícitamente la función sort() ya que debido al índice existente, los resultados se devuelven ordenados por dni.

- Para que una consulta como la anterior se beneficie plenamente del uso de índices habría que crear un índice compuesto, es decir, creado sobre múltiples campo.

Ej. Creación de un índice sobre los campos provincia y dni de la colección contribuyentes:

```
db. contribuyentes.createIndex({ "provincia": 1, "dni": 1 })
```

# Función **dropIndex ( )** |

- Elimina un índice.

Sintaxis: db.<colección>.**dropIndex**( <campos> )

Sintaxis: db.<colección>.**dropIndex**( <nombre\_índice> )

Ej.



# Función **dropIndexes ( )** |

- Elimina varios índices a la vez.

Sintaxis: db.<colección>.**dropIndexes**( <nombres\_índices> )

Ej. Eliminar todos los índices de la colección (excepto el del campo \_id).

db.<colección>.**dropIndexes**()

# Ejemplos de creación de índices

```
db.coleccion.createIndex({"estado" : 1})
```

```
db.coleccion.createIndex({"estado" : 1, "calificacion" : 1})
```

```
db.coleccion.createIndex({"persona.email" : 1})
```

Administración desde línea de  
comandos

# **mongodump** - Copia de seguridad de una BD

- `mongodump --db dbName --out outFile --host "IP:PORT" --username <user> --password <pass>`

Ej.

```
mongodump --db peliculas --out
"/home/alumno/Documentos/peliculas.dump" --host "127.0.0.1:27017"
```

# **mongorestore** – Recuperación de una copia de seguridad de una BD

- `mongorestore --db dbName <ruta_archivo_backup>`

Ej.

```
mongorestore --db peliculas "/home/alumno/Documentos/peliculas.dump" --
host "127.0.0.1:27017"
```

# **mongoexport** – Exportación de colecciones enteras

- `mongoexport --db dbName --collection collectionName --out outputFile`

**Ej.** Exportación de la colección **movies**, perteneciente a la BD **peliculas**:

```
mongoexport --db peliculas --collection movies --out
"/home/alumno/Documentos/movies.json"
```

## **mongoexport** – Exportación algunos campos de una colección

- `mongoexport --db dbName --collection collectionName --out outputFile --fields fieldname`

**Ej.** Exportación del campo **title**, perteneciente a la colección **movies**, perteneciente a la BD **peliculas**:

```
mongoexport --db peliculas --collection movies --out
"/home/alumno/Documentos/movies.json" --fields title
```

# **mongoimport** – Importación de colecciones

- `mongoimport --db dbName --collection collectionName --file inputFile`

**Ej.** Importación de la colección **movies**, en la BD **peliculas**:

```
mongoimport --db peliculas --collection movies --file
"/home/alumno/Documentos/movies.json"
```



Agregados y framework de  
agregación

# Framework de agregación I

- Forma alternativa de hacer consultas.
- Es un superconjunto de MQL.

Ej. Encontrar todos los hoteles que tienen TV como uno de sus servicios de habitación. Mostrar solo el precio y la dirección en el curso resultante.

- Con MQL:

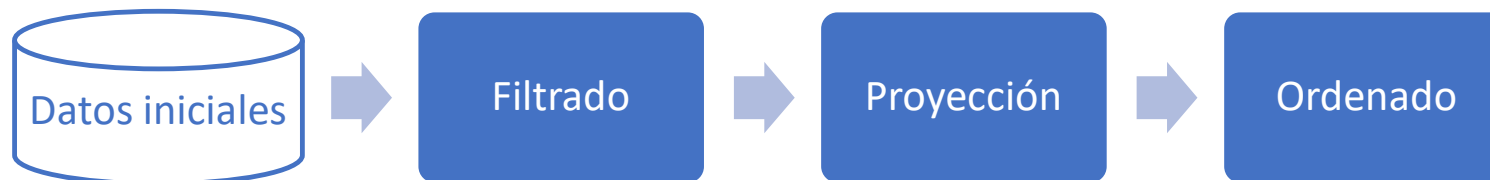
```
db.hoteles.find({ "servicios": "TV" },
 { "precio": 1, "direccion": 1, "_id": 0 })
```

- Con el framework de agregación:

```
db.hoteles.aggregate([{ "$match": { "servicios": "TV" } },
 { "$project": { "precio": 1, "direccion": 1, "_id": 0 } }])
```

# Framework de agregación III

- En el ejemplo, los elementos de la consulta forman parte de un array. En una array el orden es significativo, luego hay que indicar los elementos en el orden correcto.
- Cada elemento de la consulta (un elemento del array) representa una etapa (*stage*) de la misma.
  - En la **fase de filtrado** (**\$match**) se seleccionan documentos que cumplen alguna condición.
  - En la **fase de proyección** (**\$project**) se indica qué campos de los documentos seleccionados anteriormente se proyectarán (visualizarán).



# Framework de agregación III

- En una consulta de agregación existen una serie de etapas (*stages*):
  - **Selección** o filtrado (**\$match**), se seleccionan documentos que cumplen alguna condición.
  - **Agrupamiento** (**\$group**), se agrupan los documento por algún criterio. Es similar a la clausula GROUP BY en SQL.
  - **Ordenación** (**\$order**), se ordenan los datos por algún criterio. Es similar a la clausula ORDER BY en SQL.
  - **Proyección** (**\$project**), se indica qué campos se proyectarán (visualizarán).
- Las etapas que no filtran no modifican los datos originales, sino que operan sobre el cursor en memoria.

# Sintaxis MQL vs. Sintaxis de agregación

- MQL:

Sintaxis: { <campo>: { <operador>: <valor> } }

- Agregación:

Sintaxis: { <operador>: { <campo>, <valor> } }

# Etapas

- \$addField
- \$bucket
- \$bucketAuto
- \$collStats
- \$count
- \$facet
- \$geoNear
- \$graphLookup
- \$group
- \$indexStats
- \$limit
- \$lookup
- \$match
- \$merge
- \$out
- \$planCacheStats
- \$project
- \$redact
- \$replaceRoot
- \$replaceWith
- \$sample
- \$search
- \$set
- \$setWindowFields
- \$skip
- \$sort
- \$sortByCount
- \$unionWith
- \$unset
- \$unwind

# Etapas principales I

| ETAPA          | USO                                                                                                                                                                                                                                                                                                                                                                                     |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>\$count</b> | Devuelve el conteo del número de documentos en esta etapa de la tubería de agregación. Es distinto que el acumulador de agregación \$count.                                                                                                                                                                                                                                             |
| <b>\$group</b> | Agrupar documentos de entrada por una expresión de identificación específica y aplica expresiones de acumulación (si existen) a cada grupo. Consume todos los documentos de entrada y genera un documento de salida para cada grupo distinto. Los documentos de salida sólo contienen el campo de identificación y campos de acumulación (si se especificaron).                         |
| <b>\$match</b> | Filtra el flujo de documentos para permitir que sólo los documentos que cumplan los criterios pasen sin modificación a la siguiente etapa de la tubería. \$match usa consultas MongoDB estándar. Para cada documento de entrada, genera en la salida, bien un documento (una <b>coincidencia</b> o <b>match</b> ), bien ningún documento ( <b>sin coincidencia</b> o <b>no match</b> ). |

## Etapas principales II

| ETAPA            | USO                                                                                                                                                                                                                                 |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>\$project</b> | Reformatea cada documento en el flujo, añadiendo nuevos campos o eliminando alguno de los existentes. Para cada documento de entrada se genera uno de salida. También se puede usar <b>\$unset</b> para eliminar campos existentes. |
| <b>\$sort</b>    | Reordena el flujo de documentos por una clave de ordenación especificada. Sólo cambia el orden, los documentos no se modifican. Para cada documento de entrada se genera uno de salida.                                             |



# \$match

- Filtra el flujo de documentos para permitir que sólo los documentos que cumplan los criterios pasen sin modificación a la siguiente etapa de la tubería.
- \$match usa consultas MongoDB estándar para definir las condiciones de filtrado.
- Para cada documento de entrada, genera en la salida, bien un documento (una coincidencia o match), bien ningún documento (sin coincidencia o no match).

**Ej.** Sacar a la salida de la tubería los documentos cuyo año sea  $1990 \leq x < 2000$ :

```
db.<colección>.aggregate([{
 $match: { anio: { $gte: 1990, $lt: 2000 } } }])
```

# \$group

- Agrupa documentos de entrada por una expresión de identificación específica y aplica expresiones de acumulación (si existen) a cada grupo.
- Consume todos los documentos de entrada y genera un documento de salida para cada grupo distinto.
- Los documentos de salida sólo contienen el campo de identificación y campos de acumulación (si se especificaron).

**Ej.** Agrupación de documentos por el campo título y conteo de los mismos.

```
db.<colección>.aggregate([{
 $group: { _id: "$title", "cuantos": { $sum : 1 } }
}])
```

# \$sort

- Reordena el flujo de documentos por una clave de ordenación especificada.
- Sólo cambia el orden, los documentos no se modifican. Para cada documento de entrada se genera uno de salida.

**Ej.** Ordenar los documentos descendientemente por el campo oficio y ascendientemente por el campo apellido:

```
db.<colección>.aggregate([{
 $sort: { oficio: -1, apellido: 1 }
}])
```

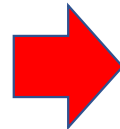
# \$project

- Pasa los documentos con los campos indicados a la siguiente etapa de la tubería (pipeline). Los campos especificados pueden ser campos existentes en los documentos de entrada o nuevos campos calculados.

Sintaxis: { \$project: { <especificaciones> } }

Ej. Seleccionar campos de los documentos de entrada para sacarlos en la salida:

```
{
 "_id" : 1,
 "titulo" : "abc123",
 "isbn" : "0001122223334",
 "autor" : { "apellido" : "Doe", "nombre" : "Jane" },
 "copias" : 5
}
```



```
db.<colección>.aggregate([{
 $project : { "titulo" : 1 , "autor" : 1 } }])
```



```
{ "_id" : 1, "titulo" : "abc123", "autor" : {
 "apellido" : "Doe", "first" : "Jane" } }
```

# \$count

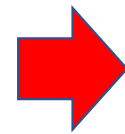
- Pasa un document a la siguiente etapa que contiene un conteo del número de documentos de entrada en la etapa.

Sintaxis: { **\$count**: <string> }

- La etapa \$count es equivalente a la secuencia \$group + \$Project.

Ej.

```
{ "_id" : 1, "subject" : "History", "score" : 88 }
{ "_id" : 2, "subject" : "History", "score" : 92 }
{ "_id" : 3, "subject" : "History", "score" : 97 }
{ "_id" : 4, "subject" : "History", "score" : 71 }
{ "_id" : 5, "subject" : "History", "score" : 79 }
{ "_id" : 6, "subject" : "History", "score" : 83 }
```



```
db.<colección>.aggregate([{
 $match: { score: { $gt: 80 } }
},
 { $count: "passing_scores" }
])
```



```
{ "passing_scores" : 4 }
```

# Otras etapas I

| ETAPA                 | USO                                                                                                                                                                                                                                                                                                 |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>\$addField</b>     | Añade campos a un documento. Similar a \$project, \$addField reformatea cada documento en el flujo, añadiendo nuevos campos a los documentos de salida que contendrán tanto los campos previamente existentes, como los recién añadidos.                                                            |
| <b>\$bucket</b>       | Organiza los documentos entrantes en grupos, llamados <b><i>buckets</i></b> (cubos), basándose en una expresión especificada y los límites del bucket.                                                                                                                                              |
| <b>\$bucketAuto</b>   | Organiza los documentos entrantes en un número específico de grupos, llamados <b><i>buckets</i></b> (cubos), basándose en una expresión especificada. Los límites del bucket se determinan automáticamente intentando distribuir los documentos uniformemente en el número de buckets especificado. |
| <b>\$changeStream</b> | Devuelve un cursor de cambio de flujo ( <i>change stream</i> ) para la colección. Esta etapa puede aparecer una única vez en una tubería de agregación ( <i>aggregation pipeline</i> ) y debe hacerlo como la primera etapa.                                                                        |

# \$count

- La etapa \$count es equivalente a la secuencia \$group + \$project.

```
db.<colección>.aggregate([
 { $group: { _id: null, cuantos: { $sum: 1 } } },
 { $project: { _id: 0 } }
])
```

- Sintaxis: { \$count: <string> }

Ej.

```
{ "_id" : 1, "asignatura" : "Historia", "nota" : 7.8 }
{ "_id" : 2, "asignatura" : "Lengua", "nota" : 4.2 }
{ "_id" : 3, "asignatura" : "Filosofía", "nota" : 8.7 }
{ "_id" : 4, "asignatura" : "Matemáticas", "nota" : 3.1 }
{ "_id" : 5, "asignatura" : "Música", "nota" : 6.6 }
{ "_id" : 6, "asignatura" : "Física y Química", "nota" : 5.9 }
```

```
db.<colección>.aggregate([
 { $match: { nota: { $gt: 5.0 } } },
 { $count: "notas_apas" }
])
```

```
{ "notas_apas" : 4 }
```

# Operaciones de agregado

- Son semejantes a las funciones de agregado en SQL: MAX, MIN, SUM, AVG, COUNT.
- Procesan múltiples documentos y devuelven resultados calculados. Se pueden usar operaciones de agregación para:
  - Agrupar juntos valores de múltiples documentos.
  - Realizar operaciones sobre los datos agrupados para devolver un resultado simple.
  - Analizar los cambios sobre los datos con el paso del tiempo.
- Para realizar operaciones de agregación, se puede usar:
  - **Tuberías (*pipeline*) de agregación.** Método recomendado para realizar agregaciones.
  - Métodos de agregación de propósito simple. Son sencillos pero carecen de las capacidades de una tubería de agregación.



# Función `aggregate()` |

Ej. El agregado tiene **dos etapas**, filtrado y agrupado, y devuelve la cantidad total en el pedido de pizzas de tamaño mediano agrupados por el nombre de la pizza:

```
db.pedidos.aggregate([
 { $match: { tamaño: "mediano" } },
 { $group: { _id: "$nombre",
 cantidad_total: { $sum: "$cantidad" } } }
)
```

Etapa 1: filtrado los documentos de pedido de pizza por el tamaño de la pizza

Etapa 2: agrupado de los documentos restantes por el nombre de la pizza y cálculo de la cantidad total

# Función **aggregate** ( ) II

- Etapa **\$match** (filtrado):
  - Filtra los documentos de pedido de pizza para que el tamaño de la pizza se mediano.
  - Pasa los documentos restantes a la etapa **\$group**.
- Etapa **\$group** (agrupado) :
  - Agrupa los documentos restantes por el nombre de la pizza.
  - Usa la operación **\$sum** para calcular la cantidad total de pedidos por cada nombre de pizza. La cantidad total se almacena en el campo `cantidad_total`, devuelto por el **pipeline** (tubería) de agregación.

# Ejemplo de aggregate I

- Datos de **pedidos**:

```
{ _id: 1, id_cliente: "abc1", fecha: ISODate("2012-11-02T17:04:11.102Z"),
status: "A", cantidad: 50 }
```

```
{ _id: 2, id_cliente : "xyz1", fecha: ISODate("2013-10-01T17:04:11.102Z"),
status: "A", cantidad: 100 }
```

```
{ _id: 3, id_cliente: "xyz1", fecha: ISODate("2013-10-12T17:04:11.102Z"),
status: "D", cantidad: 25 }
```

```
{ _id: 4, id_cliente: "xyz1", fecha: ISODate("2013-10-11T17:04:11.102Z"),
status: "D", cantidad: 125 }
```

```
{ _id: 5, id_cliente: "abc1", fecha: ISODate("2013-11-12T17:04:11.102Z"),
status: "A", cantidad: 25 }
```

# Ejemplo de aggregate II

Ej. La consulta:

1. **Selecciona** (`$match`) documentos cuyo campo status valga "A",
2. **Agrupar** (`$group`) los documentos obtenidos por el campo id\_cliente,
3. **Calcula** un valor **total** de cada id\_cliente a partir de la **suma del campo cantidad** (`$sum`).
4. **Ordena** (`$sort`) los resultados **por** el campo calculado total en orden descendente.

```
db.pedidos.aggregate([
 { $match: { status: "A" } },
 { $group: { _id: "$id_cliente", total: { $sum: "$cantidad" } } },
 { $sort: { total: -1 } }
])
```

# Ejemplo de aggregate III

- Resultado de los pasos anteriormente citados:

1. **Extrae** con los documentos 1, 2 y 5:

```
{_id: 1, id_cliente: "abc1", fecha: ISODate("2012-11-02T17:04:11.102Z"), status: "A", cantidad: 50 }
{_id: 2, id_cliente: "xyz1", fecha: ISODate("2013-10-01T17:04:11.102Z"), status: "A", cantidad: 100 }
{_id: 5, id_cliente: "abc1", fecha: ISODate("2013-11-12T17:04:11.102Z"), status: "A", cantidad: 25 }
```

2. Hay dos id\_cliente diferentes, "abc1", que tiene dos documentos, y "xyz1", que tiene un documento. Por lo tanto, los documentos quedan “**divididos**” en dos grupos, "\_id": "abc1" y "\_id": "xyz1".

3. **Suma** el campo cantidad de todos los documentos de cada grupo.

4. **Ordena** los resultados en orden descendente.

| Grupo         | \$sort: { total: -1 } |
|---------------|-----------------------|
| "_id": "xyz1" | 100                   |
| "_id": "abc1" | 75                    |

| Grupo         | total: { \$sum: "\$cantidad" } |
|---------------|--------------------------------|
| "_id": "abc1" | 50 + 25 = 75                   |
| "_id": "xyz1" | 100                            |

# Explicación sobre la operación de agregación

```
db.orders.explain().aggregate([
 { $match: { status: "A" } },
 { $group: { _id: "$cust_id", total: { $sum: "$amount" } } },
 { $sort: { total: -1 } }
])
```

## Fase de agrupamiento (**\$group**) |

- Se realiza con la operación \$group.
- Sintaxis: { "\$group": { "\_id": <expresión>, // expression de GROUP BY  
          <campo1>: { <acumulador1>: <expresión1>},  
          <campoN>: { <acumuladorN>: <expresiónN> } }
- Al campo \_id se le asigna el nombre del campo por el que se va a agrupar.  
Ej. Mostrar únicamente el campo dirección de cada documento, entonces agrupar todos los documentos en un único documento por valor de direccion.pais:  

```
db.alojamientos.aggregate([{ "$project": { "direccion": 1, "_id": 0 } },
 { "$group": { "_id": "$direccion.pais" } }])
```

## Fase de agrupamiento (**\$group**) II

- Con la segunda parte de la fase **\$group** se pueden realizar operaciones como la suma.

Ej. Se crea un nuevo campo "**cuantos**", que contendrá la suma de documentos que hay en cada grupo creado en la fase de agrupación, es decir, cuantos alojamientos hay por país,

```
{ "$group": { "_id": "$address.country",
 "cuantos": { "$sum": 1 } } }
```



# Ejemplo de agregación

```
fecha2020 = new Date("2020, 1, 1")
```

```
db.coleccion.aggregate([
 {$match: { fecha: {$lt: fecha2020 } } },
 { $group: {_id:"$vendedor.email",
 articulos: {$push: "$descripcion" } } }
])
```

# Operadores en el framework de agregación

| OPERACIÓN            | USO                                                                                                                                                                                                           |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>\$add</b>         |                                                                                                                                                                                                               |
| <b>\$arrayElemAt</b> |                                                                                                                                                                                                               |
| <b>\$round</b>       | Redondea un número a un entero o al número de decimales especificado. Esto producir que el número redondeado sea mayor o menor que el original.                                                               |
| <b>\$truncate</b>    | Trunca un número a un entero o al número de decimales especificado. Esto significa que el número truncado siempre es menor que el original, ya que la operación “recorta” (trunca) parte del número original. |
| <b>\$sample</b>      | Devuelve una serie de documentos de una colección eligiéndolos al azar.                                                                                                                                       |
| <b>\$rand</b>        | Genera un número real aleatorio entre 1 y 0.                                                                                                                                                                  |
| <b>\$sampleRate</b>  | selecciona documentos de una colección con una cierta probabilidad.                                                                                                                                           |

# Ejemplos

- Dada la colección datos:

```
{ "_id" : 1, "dato" : 8.99 }
{ "_id" : 2, "dato" : 8.45 }
{ "_id" : 3, "dato" : 8.451 }
{ "_id" : 4, "dato" : -8.99 }
{ "_id" : 5, "dato" : -8.45 }
{ "_id" : 6, "dato" : -8.451 }
{ "_id" : 7, "dato" : 8 }
{ "_id" : 8, "dato" : 0 }
{ "_id" : 9, "dato" : 0.5 }
{ "_id" : 10, "dato" : 0.9 }
```

- Se le aplica la operación:

```
db.test.aggregate([
 { $project: { _id: 0, dato: 1,
 redondeado: { $round: ["$dato"] },
 truncado: { $trunc: ["$dato"] }
 } }
)
```

- El resultado es:

```
{ "dato " : 8.99, "redondeado" : 9, "truncado" : 8 }
{ "dato " : 8.45, "redondeado" : 8, "truncado" : 8 }
{ "dato " : 8.451, "redondeado" : 8, "truncado" : 8 }
{ "dato " : -8.99, "redondeado" : -9, "truncado" : -8 }
{ "dato " : -8.45, "redondeado" : -8, "truncado" : -8 }
{ "dato " : -8.451, "redondeado" : -8, "truncado" : -8 }
{ "dato " : 8, "redondeado" : 8, "truncado" : 8 }
{ "dato" : 0, "redondeado" : 0, "truncado" : 0 }
{ "dato " : 0.5, "redondeado" : 0, "truncado" : 0 }
{ "dato " : 0.9, "redondeado" : 1, "truncado" : 0 }
```

# Ejemplo

- Devolver tres documentos al azar de la colección mascotas:  
`db.mascotas.aggregate( [ { $sample: { size: 3 } } ] )`
- Generar aleatoriamente un valor real entre 0 y 1 y devolver aquellos documentos cuyo valor sea menor que 0.5  
`db.mascotas.aggregate( [ { $match: { $expr: { $lt: [ 0.5, { $rand: {} } ] } } } ] )`
- Selecciona documentos de la colección mascotas con una probabilidad del 50% (0.5):  
`db.mascotas.aggregate( [ { $match: { $sampleRate: 0.5 } } ] )`

# \$first

- Es un alias de { **\$arrayElemAt**: [ <campo\_array>, 0 ] }.

# \$last

- Es un alias de { **\$arrayElemAt**: [ <campo\_array>, -1 ] }.

# Agrupación y conteo

Sintaxis: db.<colección>.aggregate( [ { **\$group** : { \_id: \$<campo\_por\_que\_agrupar>,  
cuantos: { **\$sum**: 1 } } } ] )

**Ej.** A partir de la colección de datos de la nba, que tiene documentos de la forma:

{ equipo: "Mavs", posicion: "Escolta", puntos: 31 }

{ equipo: "Spurs", posicion: "Escolta", puntos: 22 }

{ equipo: "Rockets", posicion: "Pivot", puntos: 19 }

{ equipo: "Warriors", posicion: "Alero", puntos: 26 }

{ equipo: "Cavs", posicion: "Escolta", puntos: 33 }

**Ej.** Agrupar por el campo posición, contando cuántos documentos hay por posición

db.nba.aggregate( [ { **\$group** : { \_id: "\$posicion", cuantos: { **\$sum**: 1 } } } ] )

{ \_id: "Alero", cuantos: 1 }

{ \_id: "Escolta", cuantos: 3 }

{ \_id: "Pivot", cuantos: 1 }

**SELECT** posicion, **COUNT(\*)** cuantos  
**FROM** nba **GROUP BY** posicion;

# Agrupación, conteo y ordenación

Sintaxis: db.<colección>.aggregate( [ { **\$group** : {\_id: \$<campo\_por\_que\_agrupar>,  
cuantos: { **\$sum**: 1 } } }  
{ **\$sort**: { cuantos: <1 (asc) | -1 (desc)> } } ] )

**Ej.** Agrupar por el campo posición, contando cuántos documentos hay por posición y ordenando por el conteo:

```
db.nba.aggregate([{ $group : {_id: "$posicion", cuantos: { $sum: 1 } } }
 { $sort: { cuantos: 1 } }])
```

```
{ _id: "Alero", cuantos: 1 }
```

```
{ _id: "Pivot", cuantos: 1 }
```

```
{ _id: "Escolta", cuantos: 3 }
```

```
SELECT posicion, COUNT(*) cuantos
FROM nba GROUP BY posicion
ORDER BY cuantos;
```



# Agrupación por varios campos y conteo

Sintaxis: db.<colección>.aggregate( [ { \$group : { \_id: { campo1: \$<campo1\_por\_que\_agrupar>, campo2: \$<campo2\_por\_que\_agrupar>, ... }, cuantos: { \$sum: 1 } } } ] )

{ equipo: "Mavs", position: "Escolta", puntos: 31 }

{ equipo: "Mavs", position: "Escolta", puntos: 22 }

{ equipo: "Mavs", position: "Alero", puntos: 19 }

{ equipo: "Rockets", position: "Escolta", puntos: 26 }

{ equipo: "Rockets", position: "Alero", puntos: 33 }

**Ej.** Agrupar por los campos equipo y posición, contando las ocurrencias de cada agrupamiento:

```
db.teams.aggregate([
 { $group : { _id: { equipo: "$equipo", posicion: "$posicion" }, cuantos: { $sum: 1 } } }])
```

{ \_id: { equipo: "Rockets", posicion: "Alero" }, cuantos: 1 }

{ \_id: { equipo: "Mavs", posicion: "Escolta" }, cuantos: 2 }

{ \_id: { equipo: "Mavs", posicion: "Alero" }, cuantos: 1 }

```
SELECT posicion, COUNT(*) cuantos
FROM nba GROUP BY equipo, posicion
```

ANEXO

# Funciones obsoletas I

| OBSOLETO                            | USO                                                  | ALTERNATIVA                                                                                                                                                                  |
|-------------------------------------|------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>db.collection.copyTo()</code> | Copia de una colección.                              |                                                                                                                                                                              |
| <code>db.collection.count()</code>  | Conteo de documentos en una colección.               | <code>db.collection.countDocuments()</code><br><code>db.collection.estimatedDocumentCount(options)</code>                                                                    |
| <code>db.collection.insert()</code> | Inserción de uno o más documentos en una colección.  | <code>db.collection.insertOne()</code><br><code>db.collection.insertMany()</code><br><code>db.collection.bulkWrite()</code>                                                  |
| <code>db.collection.remove()</code> | Eliminación de uno o más documentos de una colección | <code>db.collection.deleteOne()</code><br><code>db.collection.deleteMany()</code><br><code>db.collection.findOneAndDelete()</code><br><code>db.collection.bulkWrite()</code> |

# Funciones obsoletas II

| OBSOLETO                            | USO                                                                   | ALTERNATIVA                                                                                                                                                                                                             |
|-------------------------------------|-----------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>db.collection.save()</code>   | Modificación de documentos en una colección (altas y modificaciones). | <code>db.collection.insertOne()</code><br><code>db.collection.insertMany()</code><br><code>db.collection.updateOne()</code><br><code>db.collection.updateMany()</code><br><code>db.collection.findOneAndUpdate()</code> |
| <code>db.collection.update()</code> | Modificación de documentos en una colección.                          | <code>db.collection.updateOne()</code><br><code>db.collection.updateMany()</code><br><code>db.collection.findOneAndUpdate()</code><br><code>db.collection.bulkWrite()</code>                                            |

Importación y exportación

# Comando **mongoimport** |

- Para importar en una colección documentos existentes en un archivo:  

```
mongoimport --
uri="mongodb+srv://<usuario>:<password>@<cluster>.mongodb.net/sample_s
upplies" sales.json
```
- Se puede eliminar el contenido completo de una colección antes de importar documentos en ella:  

```
mongoimport --
uri="mongodb+srv://<usuario>:<password>@<cluster>.mongodb.net/sample_s
upplies" --drop sales.json
```

# Importación y exportación

- `mongodump --uri "mongodb+srv://<your username>:<your password>@<your cluster>.mongodb.net/sample_supplies"`
- `mongoexport --uri="mongodb+srv://<your username>:<your password>@<your cluster>.mongodb.net/sample_supplies" --collection=sales --out=sales.json`
- `mongorestore --uri "mongodb+srv://<your username>:<your password>@<your cluster>.mongodb.net/sample_supplies" --drop dump`
- `mongoimport --uri="mongodb+srv://<your username>:<your password>@<your cluster>.mongodb.net/sample_supplies" --drop sales.json`

# Mensajes informativos tras ejecutar consultas

| MENSAJE                         | EXPLICACIÓN                                                          |
|---------------------------------|----------------------------------------------------------------------|
| WriteResult( {"nInserted": x} ) | Se han insertado x documentos en una operación de escritura          |
| "nUpserted": x                  | Se ha realizado una operación upsert en x documentos                 |
| "nMatched": x                   |                                                                      |
| "nModified": x                  | Se han modificado x documentos                                       |
| "nRemoved": x                   | Se han eliminado x documentos                                        |
| "writeErrors": []               |                                                                      |
| "writeConcernErrors": []        | "                                                                    |
| "upserted": []                  |                                                                      |
| "acknowledged": bool            | Indica si una operación se ha identificado correctamente             |
| "matchedCount": x               | Número de documentos que cumplen una condición                       |
| "modifiedCount": x              | Número de documentos a los que afecta una operación de actualización |



# Errores

| ERROR  | TEXTO               | EXPLICACIÓN                                                                    |
|--------|---------------------|--------------------------------------------------------------------------------|
| E11000 | Duplicate key error | Se ha intentado insertar en una colección un documento con un _id ya existente |

# Atlas: MongoDB en la nube

- Creación de clústeres
- Ejecución y mantenimiento de bases de datos desplegadas.
- Clúster gratis (Free tier cluster):
  - Conjunto de réplicas con 3 servidores.
  - 512 MB de almacenamiento.
  - No expira.

# Cientes de MongoDB

- **mongosh**: Es un cliente en línea de comandos. Es, así mismo, un intérprete de JavaScript.
- **Compass**: Es un cliente gráfico (IDE).

Compass

# Opciones sobre un documento

- Pestañas:
  - Conexión
  - Documentos
  - Esquema
  - Índices
  - Explicación del plan de ejecución
  - Validación
- Operaciones sobre documentos:
  - Editar
  - Copiar
  - Clonar
  - Eliminar

Modelado

# Modelado I

- Es la forma de organizar los campos en los documento para facilitar las consultas y optimizar su rendimiento.
- MongoDB no fuerza una manera de modelar los datos.
- Se plantea la pregunta sobre qué estructura es la más adecuada para almacenar los datos.
  - Subdocumentos
  - Arrays de valores
  - Una nueva colección
- **Regla del pulgar:** Los datos se almacenan en la forma en la que se usan, es decir, según el tipo de consultas más frecuentes que se realizan sobre ellos.

# Modelado II

- **Regla del pulgar (bis):** Los datos que se consultan juntos se almacenan juntos.
- Una pregunta a plantearse es cómo se van a consultar los datos, que muchas veces se responde sabiendo qué perfil de usuario va a consultar los datos.
- Cuando una aplicación va evolucionando/cambiando su uso con el tiempo, el tipo de consultas puede ir cambiando, y eso supondría una modificación del modelado de datos, lo que es fácil y flexible en MongoDB, frente a las limitaciones de las BDRs.



# Ejemplo de modelado I: datos médicos de pacientes

- Se quiere almacenar nombre, género, fecha de nacimiento, datos de contacto, histórico de visitas, prescripciones...

Ej.

```
{ _id: 1,
 "nombre": "Lola Pérez",
 "fechaNacimiento": "1950-07-23",
 "genero": "Femenino",
 "contacto_preferido": { "tipo": "email", "contacto": "a@b.org" },
 "info_contactos": [{ "tipo": "# trabajo", "contacto": "111111111" },
 { "tipo": "# móvil", "contacto": "222222222" }],
 "prescripciones": [127, 74, 96],
 "hist_visitas": [{ "fecha": "2021-11-23", "notas": "Dolor de cabeza..." },
 { "fecha": "2022-02-07", "notas": "Erupción cutánea..." }]
}
```

## Ejemplo de modelado II

Ej.

```
{ _id: 2,
 "nombre": "Paco García",
 "fechaNacimiento": "1985-10-08",
 "genero": "Masculino",
 "contacto_preferido": { "tipo": "email", "contacto": "c@d.org" },
 "info_contactos": [{ "tipo": "# móvil", "contacto": "333333333" }]
}
```

- Asumiendo que la aplicación la usan médicos, la información que precisan más frecuentemente es: prescripciones actuales, diagnósticos, información de contacto de paciente...
- Otra información complementaria sería las características de los medicamentos prescritos, para consultar efectos secundarios, alergias...

# Ejemplo de modelado III

- Colecciones recomendadas y estructura de sus documentos:

- Colección **pacientes**:

```
{_id:, "nombre": "...", "fechaNacimiento": "...", "genero": "...",
 "contacto_preferido": { "tipo": "...", "contacto": "..." },
 "info_contactos": [{ "tipo": "...", "contacto": "..." },
 { "tipo": "...", "contacto": "..." }, ...],
 "prescripciones": [..., ..., ...], "alergias": ["...", "...", ...],
 "hist_visitas": [{ "fecha": "...", "notas": "..." },
 { "fecha": "...", "notas": "..." }, ...],
 "sig_visita": "...", "diagnosticos": ["...", "...", ...]
}
```

# Ejemplo de modelado IV

- Colección **medicamentos**:

```
{_id:, "nombre": "...", "codigo": ..., "efectos_secundarios": ["...", "...", ...],
 "alergenos": ["...", "...", ...], "principios_activos": ["...", "...", ...]
 "interacciones": [{ "# medicamento": "...", "efecto": "..." },
 { "# medicamento": "...", "efecto": "..." }, ...],
 "uso_efermedades": ["...", "...", ...]
}
```

- Consultas frecuentes:

```
db.pacientes.find("nombre": "...")
```

```
db.pacientes.find("sig_visita": "...")
```

```
db.medicamentos.find("uso_efermedades": "...")
```

```
db.medicamentos.find("codigo": ...)
```

# Referencias I

- ObjectId  
<https://www.mongodb.com/docs/manual/reference/method/ObjectId/#objectid>
- Update Operators  
<https://www.mongodb.com/docs/manual/reference/operator/update/#id1>
- Query and Projection Operators  
<https://www.mongodb.com/docs/manual/reference/operator/query/>
- Query an Array  
<https://www.mongodb.com/docs/manual/tutorial/query-arrays/>

# Referencias II

- Operador \$regex  
<https://docs.mongodb.com/manual/reference/operator/query/regex/index.html>
- Aggregation Pipeline  
<https://www.mongodb.com/docs/manual/core/aggregation-pipeline/>
- Curso de agragación de MongoDB  
<https://university.mongodb.com/courses/M121/about>
- Schema Validation  
<https://www.mongodb.com/docs/manual/core/schema-validation/#schema-validation>

# Referencias III

- 48 MongoDB Commands and Queries to Know as Developer and DBA  
<https://geekflare.com/mongodb-queries-examples/>