MongoDB

MongoDB es una de las bases de datos NoSQL más conocidas y empleadas. Sigue un modelo de datos documental, donde los documentos se basan en el formato JSON.

MongoDB destaca porque:

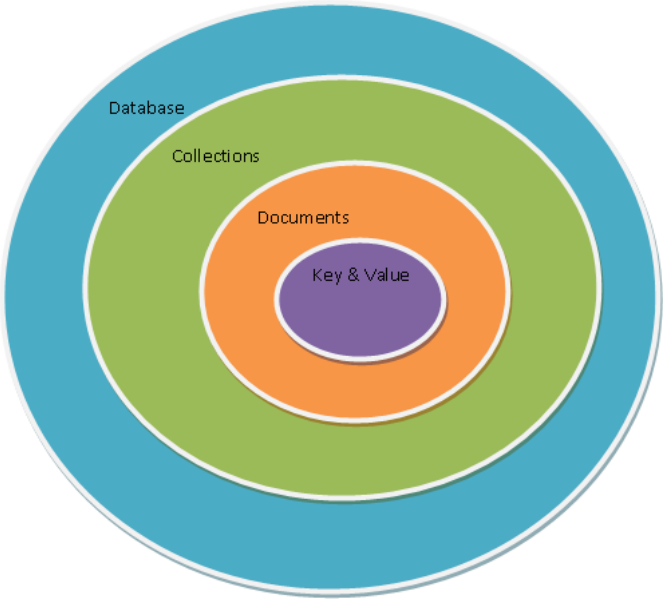
- Soporta esquemas dinámicos: diferentes documentos de una misma colección pueden tener atributos diferentes.
- Aunque inicialmente tenía un soporte limitado de *joins*, desde la versión 5.2 se pueden realizar incluso entre colecciones particionadas.
- Soporte de transacciones sólo a nivel de aplicación. Lo que en un SGBD relacional puede suponer múltiples operaciones, con *MongoDB* se puede hacer en una sola operación al insertar/actualizar todo un documento de una sola vez, pero si queremos crear una transacción entre dos documentos, la gestión la debe realizar el driver.

huMONGOus

Como curiosidad, su nombre viene de la palabra inglesa *humongous*, que significa gigantesco/enorme

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

- *MongoDB* tienen el mismo concepto de **base de datos** que un SGBD relacional. Dentro de una instancia de *MongoDB* podemos tener 0 o más bases de datos, actuando cada una como un contenedor de alto nivel.
- Una base de datos tendrá 0 o más colecciones. Una **colección** es muy similar a lo que entendemos como tabla dentro de una base de datos relacional. *MongoDB* ofrece diferentes tipos de colecciones, desde las normales cuyo tamaño crece conforme lo hace el número de documentos, como las colecciones *capped*, las cuales tienen un tamaño predefinido y que pueden contener una cierta cantidad de información que se sustituirá por nueva cuando se llene.
- Las colecciones contienen 0 o más **documentos**, por lo que es similar a una fila o registro de una tabla relacional.
- Cada documento contiene 0 o más atributos, compuestos de **parejas clave/valor**. Cada uno de estos documentos no sigue ningún esquema, por lo que dos documentos de una misma colección pueden contener todos los atributos diferentes entre sí. Todo documento contiene un campo `_id` que hace la función de atributo identificador del documento.



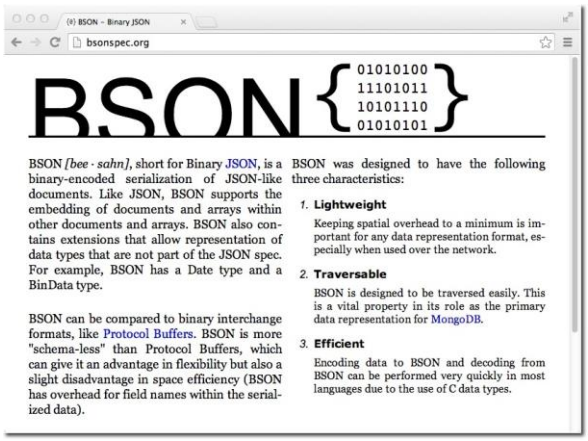
Elementos de MongoDB

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

Además, *MongoDB* soporta **índices**, igual que cualquier RDMS, para acelerar la búsqueda de datos. Al realizar cualquier consulta, se devuelve un **cursor**, con el cual podemos realizar operaciones como contar, ordenar, limitar o saltar documentos.

BSON

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



Especificación BSON

BSON representa un *superset* de JSON ya que:

- Permite almacenar datos en binario
- Incluye un conjunto de tipos de datos no incluidos en JSON, como pueden ser `ObjectId`, `Date` o `BinData`.
- Diseñado para ser más eficiente en espacio y realizar más búsquedas de forma más eficiente.

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

Un ejemplo de un objeto BSON podría ser:

```
var yo = {
  "nombre": "Manuel",
  "apellidos": "Ruiz",
  "hobbies": ["programación", "videojuegos", "baloncesto"],
  "casado": true,
  "hijos": 2,
  "contacto": {
    "bsky": "@ManuelRuiz.bsky.social",
    "email": "a.Ruiz@edu.educastur.es"
  }
}
```

Los documentos BSON tienen las siguientes restricciones:

- No pueden tener un tamaño superior a 16 MB.
- El atributo `_id` queda reservado para la clave primaria.

- Desde *MongoDB* 5.0 los nombres de los campos pueden empezar por $ y/o contener el `.`, aunque en la medida de lo posible, es recomendable evitar su uso.

Además, *MongoDB*:

- No asegura que el orden de los campos se respete.
- Es sensible a los tipos de los datos
- Es sensible a las mayúsculas.

Por lo que estos documentos son distintos:

```
{"edad": "18"}
{"edad": 18}
{"Edad": 18}
```

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

Puesta en marcha

En la actualidad, *MongoDB* se comercializa mediante tres productos:

- Mongo Atlas, como plataforma cloud, con una opción gratuita mediante un clúster de 512MB.
- MongoDB Community Edition, versión gratuita para trabajar *on-premise*, con versiones para Windows, MacOS y Linux.
- MongoDB Enterprise Advanced, versión de pago con soporte, herramientas avanzadas de monitorización y seguridad, y administración automatizada.

En nuestro caso, por comodidad, trabajaremos con la versión *cloud* de *Mongo Atlas*.

Uso del shell

Una vez ya tenemos en marcha *MongoDB*, llega el momento de conectarnos. Aunque utilizaremos *MongoDBCompas*, durante esta unidad vamos a practicar los diferentes comandos haciendo uso del cliente `mongosh` (en versiones anteriores el comando utilizado era `mongo`). También en el botón Connect del *clúster* está la información de cómo descargar y conectarse usando *mongsh*.

El *shell* de *MongoDB* utiliza *JavaScript* como lenguaje de interacción con la base de datos, y como buen *shell*, mediante la fecha hacia arriba visualizaremos el último comando.

Si queremos ver las bases de datos que existen ejecutaremos el comando **`show dbs`**:

```
Atlas atlas-pxc2m9-shard-0 [primary] pruebas> show dbs
pruebas             216.00 KiB
sample_airbnb        52.52 MiB
sample_analytics      8.93 MiB
sample_geospatial     1.29 MiB
sample_guides        40.00 KiB
sample_mflix         47.23 MiB
sample_restaurants    6.72 MiB
sample_supplies       1.05 MiB
sample_training      52.50 MiB
sample_weatherdata    3.32 MiB
```

```
admin               40.00 KiB
local               40.00 KiB
```

Uso externo

Si nos queremos conectar únicamente desde un terminal, podemos instalar únicamente el *shell* desde https://www.mongodb.com/try/download/shell

Hola MongoDB

Pues una vez que ya nos hemos conectado a *MongoDB* mediante `mongosh`, vamos a empezar a interactuar con los datos.

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

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

```
use drfleming
```

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

```
db.personas.insertOne({ nombre: "Manuel Ruiz", edad: 47, profesion:
"Profesor" })
```

Tipos de datos

Cuidado con los tipos, ya que no es lo mismo insertar un atributo con `edad:45` (se considera el campo como entero) que con `edad:"45"`, ya que considera el campo como texto.

Tras ejecutar el comando, veremos que nos devuelve un objeto JSON con su ACK y el identificador del documento insertado:

```
{
  acknowledged: true,
  insertedId: ObjectId('67fa9f27d152b48c68424418')
}
```

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

```
db.personas.findOne()
```

Lo que nos dará como resultado un objeto JSON que contiene un atributo `_id` con el mismo identificador mostrado anteriormente, además de los que le añadimos al insertar la persona:

```
{
  _id: ObjectId('67fa9f27d152b48c68424418'),
  nombre: 'Manuel Ruiz',
  edad: 47,
  profesion: 'Profesor'
}
```

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

ObjectId

En *MongoDB*, el atributo `_id` es único dentro de la colección, y hace la función de clave primaria. Se le asocia un `ObjectId`, el cual es un tipo BSON de 12 bytes que se crea mediante:

- el *timestamp* actual (4 bytes)
- un valor aleatorio y único por máquina y proceso (5 bytes)
- un contador inicializado a número aleatorio (3 bytes).

Este objeto lo crea el *driver* y no *MongoDB*, por lo cual no deberemos considerar que siguen un orden concreto, ya que clientes diferentes pueden tener *timestamps* desincronizados. Lo que sí que podemos obtener a partir del `ObjectId` es la fecha de creación del documento, mediante el método `getTimestamp()` del atributo `_id`.

Obteniendo la fecha de creación de un documento

```
db.personas.findOne()._id
// ObjectId('67fa9f27d152b48c68424418')
db.personas.findOne()._id.getTimestamp()
// 2025-04-12T17:13:11.000Z
```

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

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

Por lo tanto, podemos asignar un identificador al insertar:

```
db.personas.insertOne({_id:4, nombre:"Marina", edad:17 })
// { acknowledged: true, insertedIds: { '0': 4 } }
```

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

```
db.personas.insertOne({_id:{nombre:'Manuel', apellidos:'Ruiz',
bsky:'@ManuelRuiz.bsky.social'}, ciudad:'Oviedo'})
// {
//     acknowledged: true,
//     insertedId: {
//       nombre: 'Manuel',
//       apellidos: 'Ruiz',
//       bsky: '@ManuelRuiz.bsky.social'
//     }
// }
```

CRUD

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

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

`db.version()`	Muestra la versión actual del servidor

Y las operaciones básicas para realizar un CRUD, siguiendo la sintaxis `db.nombreColeccion.operacion()`, son:

- `insertOne`: permite insertar un documento
- `find` y `findOne`: recupera los documentos (o el primero) de una colección
- `countDocuments`: obtiene la cantidad de documentos de una colección

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

```
db.personas.insertOne(yo)
// {
//   acknowledged: true,
//   insertedId: ObjectId('67fbe737b496860bf5c3679e')
// }
db.personas.find()
// [
//   {
//     _id: ObjectId('67fa9f27d152b48c68424418'),
//     nombre: 'Manuel Ruiz',
//     edad: 47,
//     profesion: 'Profesor'
//   }
//   {
//     _id: ObjectId('67fbe737b496860bf5c3679e'),
//     nombre: 'Manuel',
//     apellidos: 'Ruiz',
//     fnac: 1977-10-02T23:00:00.000Z,
//     hobbies: [ 'programación', 'videojuegos', 'baloncesto' ],
//     casado: true,
//     hijos: 2,
//     contacto: {
//       bsky: "@ManuelRuiz.bsky.social",
//       email: 'a.Ruiz@edu.educastur.es'
//     },
//     fechaCreacion: Timestamp({ t: 1744561974, i: 5 })
//   }
// ]
yo.profesion = "Profesor"
// Profesor
db.personas.insertOne(yo)
// {
//   acknowledged: true,
//   insertedId: ObjectId('67fbe851b496860bf5c3679f')
// }
db.personas.find()
// [
//   {
//     _id: ObjectId('67fa9f27d152b48c68424418'),
//     nombre: 'Manuel Ruiz',
//     edad: 47,
```

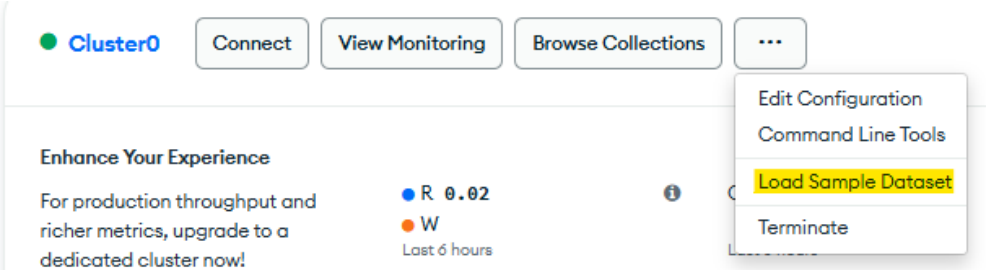
```
//     profesion: 'Profesor'
//   }
//   {
//     _id: ObjectId('67fbe737b496860bf5c3679e'),
//     nombre: 'Manuel',
//     apellidos: 'Ruiz',
//     fnac: 1977-10-02T23:00:00.000Z,
//     hobbies: [ 'programación', 'videojuegos', 'baloncesto' ],
//     casado: true,
//     hijos: 2,
//     contacto: {
//       bsky: "@ManuelRuiz.bsky.social",
//       email: 'a.Ruiz@edu.educastur.es'
//     },
//     fechaCreacion: Timestamp({ t: 1744561974, i: 5 })
//   }
//   {
//     _id: ObjectId('67fbe851b496860bf5c3679f'),
//     nombre: 'Manuel',
//     apellidos: 'Ruiz',
//     fnac: 1977-10-02T23:00:00.000Z,
//     hobbies: [ 'programación', 'videojuegos', 'baloncesto' ],
//     casado: true,
//     hijos: 2,
//     contacto: {
//       bsky: "'@ManuelRuiz.bsky.social",
//       email: 'a.Ruiz@edu.educastur.es'
//     },
//     fechaCreacion: Timestamp({ t: 1744562256, i: 4 }),
//     profesion: 'Profesor'
//   }
// ]
db.personas.countDocuments()
// 3
```

Con este ejemplo, hemos podido observar cómo los documentos de una misma colección no tienen por qué tener el mismo esquema, ni hemos necesitado definirlo explícitamente antes de insertar datos. Así pues, el esquema se irá generando y actualizando conforme se inserten documentos. Más adelante veremos que podemos definir un esquema para validar que los datos que insertamos cumplan restricciones de tipos de datos o elementos que obligatoriamente deben estar rellenados.

Consultas

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

Para los siguientes ejemplos nos vamos a centrar en la colección `zips` de la base de datos de ejemplo **`sample_trainings`** que tenemos cargada en *MongoAtlas*, la cual contiene cerca de 29.000 documentos con información de códigos postales de USA. Puedes cargarla en t *clúster* desde el enlace de la imagen:




```
use sample_training
db.zips.find()
// {
//   _id: ObjectId('5c8eccc1caa187d17ca6ed40'),
//   city: 'MORRIS',
//   zip: '35116',
//   loc: { y: 33.739172, x: 86.772551 },
//   pop: 3622,
//   state: 'AL'
// }
// ...
// Type "it" for more
```

El método `find()` sobre una colección devuelve un cursor a los datos obtenidos, el cual se queda abierto con el servidor y que se cierra automáticamente a los 30 minutos de inactividad o al finalizar su recorrido. Si hay muchos resultados, la consola nos mostrará un subconjunto de los datos (20). Si queremos seguir obteniendo resultados, solo tenemos que introducir `it`, para que continúe iterando el cursor.

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

```
db.zips.findOne()
// {
//   _id: ObjectId('5c8eccc1caa187d17ca6ed40'),
//   city: 'MORRIS',
//   zip: '35116',
//   loc: { y: 33.739172, x: 86.772551 },
//   pop: 3622,
//   state: 'AL'
// }
```

Criterios en consultas

Si queremos indicar un criterio para filtrar los datos, tanto a `find` como a `findOne` le pasaremos un documento JSON con los criterios a cumplir. El caso más sencillo es filtrar por el valor de un determinado campo.

Consulta/Resultado

```
db.zips.find({city: "TITUS"})
{ _id: ObjectId('5c8eccc1caa187d17ca6ee50'),
 city: 'TITUS',
 zip: '36080',
 loc: {
        y: 32.690019,
        x: 86.239334 },
 pop: 2683,
```

```
 state: 'AL'
}
```

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

Para acceder a la coordenada `x` usaremos la propiedad `loc.x`, la cual obligatoriamente deberemos rodear mediante comillas:

```
db.zips.find({"loc.x": 86.239334})
```

En cambio, si queremos indicar **más de un criterio**, el documento con las condiciones contendrá tantos campos como elementos a filtrar. Así pues, para obtener aquellas poblaciones del estado `AL` dentro de la ciudad `DELTA`, haríamos:

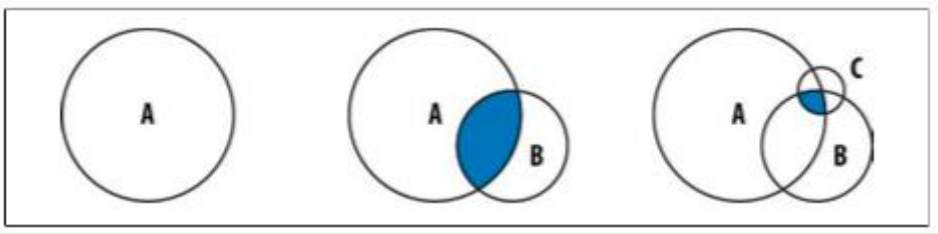
Consulta/Resultado

```
db.zips.find({city: "DELTA", state: "AL"})
{ _id: ObjectId('5c8eccc1caa187d17ca6ee6e'),
 city: 'DELTA',
 zip: '36258',
 loc: {
       y: 33.457303,
       x: 85.679279
 },
 pop: 1405,
 state: 'AL'
}
```

Consejo de Rendimiento

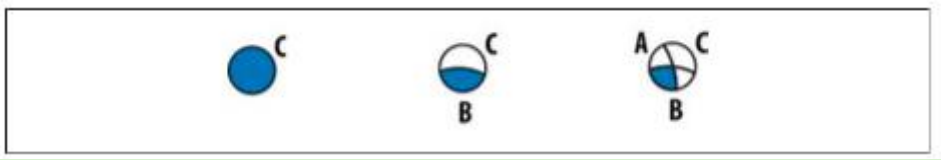
Las consultas disyuntivas, es decir, con varios criterios u operador `$and`, deben filtrar el conjunto más pequeño cuanto más pronto posible.

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



Restringiendo consultas AND

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



Restringiendo consultas AND de menor a mayor

MongoDB también ofrece operadores lógicos para los campos numéricos:

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

Estos operadores se pueden utilizar de forma simultánea sobre un mismo campo o sobre diferentes campos, sobre campos anidados o que forman parte de un array, y se colocan como un nuevo documento en el valor del campo a filtrar, compuesto del operador y del valor a comparar mediante la siguiente sintaxis:

```
db.<coleccion>.find({ <campo>: { <operador>: <valor> } })
```

Por ejemplo, para recuperar las poblaciones con menos de 1000 habitantes, o aquellas que tienen entre 2000 y 3000 personas, podemos hacer:

```
db.zips.find({pop: {$lt:1000} })
db.zips.find({pop: {$gt:2000, $lte:3000} })
```

Cuidado al repetir campos

Podemos pensar que la consulta anterior donde filtramos las poblaciones que tienen entre 2000 y 3000 personas, también la podríamos expresar mediante:

```
db.zips.find({pop: {$gt:2000}, pop: {$lte:3000}})
```

Pero no es así. Cuando repetimos el mismo campo, *MongoDB* unicamente evalúa el último, y por lo tanto, en esta consulta, recuperaría todos los documentos cuya población fuera menor o igual a 3000 habitantes (incluidos los menores de 2000), siendo equivalente a:

```
db.zips.find({pop: {$lte:3000}})
```

Para los campos de texto, además de la comparación directa, podemos usar el operador `$ne` para obtener los documentos cuyo campos no tienen un determinado valor (*not equal*). Así pues, podemos usarlo para averiguar todos las poblaciones que no pertenecen al estado de `PA`:

```
db.zips.find({state: {$ne:"PA"} })
```

Por supuesto, podemos tener diferentes operadores en campos distintos. Por ejemplo, si queremos recuperar las poblaciones del estado de `AL` con menos de 100 habitantes haríamos:

```
db.zips.find({pop: {$lt:100}, state: "AL" })
```

Case sensitive

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

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

```
use drfleming
db.personas.find({edad: {$exists:true}})
```

Polimorfismo

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

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

Por ejemplo, si queremos obtener todas las personas cuya edad no sea múltiplo de 5, podríamos hacerlo así:

```
db.personas.find({edad: {$not: {$mod: [5,0]}}})
```

Proyección de campos

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

```
db.zips.find({city: "TITUS"}, {pop: 1})
// {
//   _id: ObjectId('5c8eccc1caa187d17ca6ee50'),
//   pop: 2683
// }
```

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

```
db.zips.find({city: "TITUS"}, {pop: 1, _id:0})
// { pop: 2683 }
```

No mezcles

Al hacer una proyección, no podemos mezclar campos que se vean (`1`) con los que no (`0`). Es decir, hemos de hacer algo similar a:

```
db.<coleccion>.find({ <consulta> }, {<campo1>: 1, <campo2>: 1})
db.<coleccion>.find({ <consulta> }, {<campo1>: 0, <campo2>: 0})
```

Así pues, sólo se mezclará la visibilidad de los campos cuando queramos ocultar el `_id`.

Finalmente, si queremos renombrar un campo como resultado de la consulta, a modo de alias, podemos hacerlo referenciando el campo mediante **$** (a la izquierda de los dos puntos ponemos el nuevo nombre (alias) y a la derecha va el campo original con el prefijo `$` entre comillas):

```
db.zips.find({city: "TITUS"}, {poblacion: "$pop", _id:0})
// { poblacion: 2683 }
```

Condiciones compuestas

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

```
db.zips.find({ $or:[{city: "TITUS"}, {pop: 3}] })
db.zips.find({ $or:[{city: "TITUS"}, {pop: {$lte:10}}] })
```

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

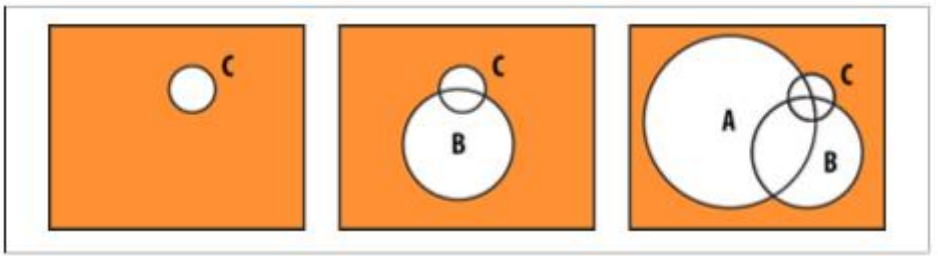
```
db.zips.find({city: "OXFORD", state: "AL"})
db.zips.find({ $and:[ {city: "OXFORD"}, {state: "AL"} ] })
```

Consejo de Rendimiento

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

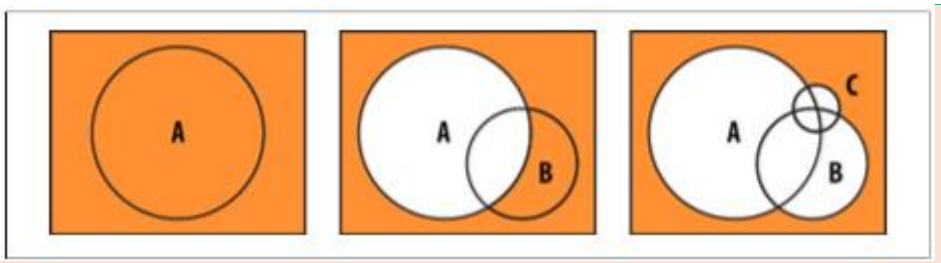
Supongamos que vamos a consultar los mismos documentos que cumplen los criterios A (40.000 documentos), B (9.000 documentos) y C (200 documentos).

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



Restringiendo consultas OR de menor a mayor

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



Restringiendo consultas OR de mayor a menor

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

Autoevaluación

¿Qué obtendríamos al ejecutar la siguiente consulta?

```
db.zips.find({ city: "OXFORD", $nor:[ {state: "AL"}, {pop: 1306} ] })
```

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

```
db.zips.find({ state: {$in: ["AL", "AR"]} })
```

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

Consultas avanzadas

Una vez sabemos realizar consultas con diferentes operadores, vamos a detallar casos concretos donde podemos realizar filtrados de datos más específicos.

Preparando los ejemplos

Para los siguientes ejemplos, vamos a utilizar una colección de 10.000 documentos sobre los viajes realizados por los usuarios de una empresa de alquiler de bicicletas, los cuales han sido extraídos de Citi Bike System Data | Citi Bike NYC | Citi Bike NYC.

Esta colección (`trips`) está cargada en los datos de ejemplo del clúster de *MongoAtlas*, dentro de la base de datos `sample_training`.

Un ejemplo de viaje sería el siguiente (puedes comprobar como para los campos que contiene espacios se indican entre comillas):

```
use sample_training
// switched to db sample_training
db.trips.findOne()
// {
//   _id: ObjectId('572bb8222b288919b68abf81'),
//   tripduration: 265,
//   'start station id': 376,
//   'start station name': 'John St & William St',
//   'end station id': 152,
```

```
//   'end station name': 'Warren St & Church St',
//   bikeid: 24119,
//   usertype: 'Subscriber',
//   'birth year': 1962,
//   'start station location': {
//     type: 'Point',
//     coordinates: [ -74.00722156, 40.70862144 ]
//   },
//   'end station location': {
//     type: 'Point',
//     coordinates: [ -74.00910627, 40.71473993
//     ]
//   },
//   'start time': 2016-01-01T00:14:26.000Z,
//   'stop time': 2016-01-01T00:18:52.000Z
// }
```

Cursores

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

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

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

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

Así pues, si quisiéramos obtener los tres viajes que más han durado (recuperando sólo el identificador de la bicicleta y la duración del viaje), podríamos hacerlo así:

```
db.trips.find({},{"bikeid":1, "tripduration":1}).sort({"tripduration":-
1}).limit(3)
// [
//   {
//     _id: ObjectId('572bb8222b288919b68ac07c'),
//     tripduration: 326222,
//     bikeid: 18591
//   },
//   {
//     _id: ObjectId('572bb8232b288919b68b0f0d'),
//     tripduration: 279620,
//     bikeid: 17547
//   },
//   {
//     _id: ObjectId('572bb8232b288919b68b0593'),
```

```
//     tripduration: 173357,
//     bikeid: 15881
//   }
// ]
```

También podemos filtrar previamente a ordenar y limitar:

```
db.trips.find({usertype:"Customer"}).sort({"tripduration":-1}).limit(3)
```

Finalmente, podemos paginar utilizando el método `skip`. Conviene destacar, que independientemente del orden en el que indiquemos las operaciones con cursores, se ejecuta así:

1. `find()`
2. `sort()`
3. `skip()`
4. `limit()`

Es decir, *MongoDB* siempre aplicará primero el filtro, luego el ordenamiento, después el salto y finalmente el límite.

Es por ello por lo que, para mostrar viajes de 10 en 10 a partir de la tercera página, podríamos hacer algo así, obteniendo los mismos resultados:

```
db.trips.find({usertype:"Customer"}).sort({"tripduration":-
1}).limit(10).skip(20)
db.trips.find({usertype:"Customer"}).sort({"tripduration":-
1}).skip(20).limit(10)
```

Autoevaluación

A partir de la colección `trips`, escribe una consulta que recupere los viajes realizados por subscriptores ordenados descendentemente por su duración y que obtenga los documentos de 15 al 20.

Contando

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

```
db.trips.countDocuments({"birth year":1977})
// 186
db.trips.countDocuments({"birth year":1977, "tripduration":{$lt:600}})
// 116
```

count

Desde la versión 4.0, los métodos `count` a nivel de colección y de cursor están caducados (*deprecated*), y no se recomienda su utilización. Aun así, es muy común utilizarlo como método de un cursor:

```
db.trips.find({"birth year":1977, "tripduration":{$lt:600}}).count()
```

Cuando tenemos muchísimos datos, si no necesitamos exactitud, pero queremos un valor estimado el cual tarde menos en conseguirse (utiliza los metadatos de las colecciones), podemos usar `estimatedDocumentCount`, si bien no usa el filtro.

```
db.trips.estimatedDocumentCount({"birth year":1977})
// 10.000
db.trips.estimatedDocumentCount({"birth year":1977,
"tripduration":{$lt:600}})
// 10.000
```

CONJUNTO DE VALORES

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

```
db.trips.distinct('usertype')
[ 'Customer', 'Subscriber' ]
```

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

```
db.trips.distinct('usertype', { "birth year": { $gt: 1990 } } )
[ 'Subscriber' ]
```

Como el comando `distinct` no devuelve un cursor, sino un array, podemos obtener la cantidad de elementos mediante la propiedad `length`:

```
db.trips.distinct('usertype').length
// 2
```

Trabajando con fechas

Con la estructura de BSON podemos utilizar tipos de datos asociados a fechas como `Timestamp` o `Date`.

Por ejemplo, si creamos un documento con diferentes campos con fechas:

```
let fechas = {
    hoy: new Date(),
    cumple: new Date("Oct 3, 1977"),
    inicioCurso: new Date("2024-10-01"),
    ahora: new Timestamp(),
    ahoraISO: ISODate(),
    nochevieja: ISODate("2024-12-31T00:00:00.000Z")
}
```

Al mostrarlo en la consola de `mongosh` podemos ver cómo ha generado las fechas:

```
fechas
// {
//   hoy: 2025-04-23T08:53:26.186Z,
//   cumple: 1977-10-02T23:00:00.000Z,
//   inicioCurso: 2024-10-01T00:00:00.000Z,
//   ahora: Timestamp({ t: 0, i: 0 }),
//   ahoraISO: 2025-04-23T08:53:26.186Z,
//   nochevieja: 2024-12-31T00:00:00.000Z
// }
```

A la hora de realizar consultas, utilizaremos los campos de fecha de forma similar al resto de tipos de datos.

Por ejemplo, podemos ordenar por fechas:

```
db.trips.find({"bikeid": 24119}, {"start time":1,"stop time":1})
        .sort({"start time":1}).limit(3)
// [
//   {
//     _id: ObjectId('572bb8222b288919b68abf81'),
//     'start time': ISODate('2016-01-01T00:14:26.000Z'),
//     'stop time': ISODate('2016-01-01T00:18:52.000Z')
//   },
```

```
//   {
//     _id: ObjectId('572bb8222b288919b68ac20a'),
//     'start time': ISODate('2016-01-01T02:02:40.000Z'),
//     'stop time': ISODate('2016-01-01T02:15:58.000Z')
//   },
//   {
//     _id: ObjectId('572bb8222b288919b68ac3d4'),
//     'start time': ISODate('2016-01-01T03:49:38.000Z'),
//     'stop time': ISODate('2016-01-01T03:54:21.000Z')
//   }
// ]
```

O utilizar un rango:

```
db.trips.find({"bikeid": 24119,
   "start time": {$gt:ISODate("2016-01-01T00:00:00.000Z"),
                  $lte:ISODate("2016-01-01T12:00:00.000Z")}}).count()
// 5
```

También podemos utilizar las fechas y formatearlas al proyectar los campos (por ejemplo, utilizando el operador `$dateToString` pasamos un campo de tipo fecha a una cadena con el formato deseado):

```
db.trips.find({"bikeid": 24119,
   "start time": {$gt: ISODate("2016-01-01T00:00:00.000Z"),
                  $lte: ISODate("2016-01-01T06:00:00.000Z")}},
    {bikeid:1,
     fechaInicio: {$dateToString: {date: "$start time", format: "%Y-%m-%d %H:%M"}},
     fechaConsulta: {$dateToString: {date: new Date(), format: "%Y-%m-%d"}},
    })
// [
//   {
//     _id: ObjectId('572bb8222b288919b68abf81'),
//     bikeid: 24119,
//     fechaInicio: '2016-01-01 00:14',
//     fechaConsulta: '2025-04-23'
//   },
//   {
//     _id: ObjectId('572bb8222b288919b68ac20a'),
//     bikeid: 24119,
//     fechaInicio: '2016-01-01 02:02',
//     fechaConsulta: '2025-04-23'
//   },
//   {
//     _id: ObjectId('572bb8222b288919b68ac3d4'),
//     bikeid: 24119,
//     fechaInicio: '2016-01-01 03:49',
//     fechaConsulta: '2025-04-23'
//   }
// ]
```

Expresiones regulares

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

Por ejemplo, para buscar la cantidad de viajes que salen da alguna estación cuyo nombre contenga `Tree` podemos hacer:

```
db.zips.find({city: /Tree/}).count()
db.zips.find({city: /tree/i}).count()
db.zips.find({city: {$regex:/tree/i}}).count()
```

Búsquedas sobre textos

Si vamos a realizar búsquedas intensivas sobre texto, desde *MongoDB* han creado un producto específico dentro del ecosistema de *Mongo Atlas* el cual ofrece un mejor rendimiento y mayor funcionalidad que el uso de expresiones regulares, conocido con Mongo Atlas Search.

Si usamos una solución *on-premise*, mediante índices de texto y el operator `$text` podemos realizar búsquedas.

Operador $expr

El operador `$expr` es un operador de consulta expresiva que permite utilizar expresiones de agregación dentro de las consultas.

Permite utilizar variables y sentencias condicionales, así como comparar campos dentro de un documento. Así pues, si queremos comparar valores entre dos campos, podemos hacerlo mediante `$expr` referenciando a los campos anteponiendo un dólar (`$`) delante del campo, de manera que si queremos obtener los viajes que comienzan y finalizan en la misma estación podemos hacer:

```
db.trips.find({ "$expr": { "$eq": [ "$end station id", "$start station id"
]}})
```

Al poner el `$` delante de un campo, en vez de referenciar al campo, lo que hace es referenciar a su valor, por lo que `$end station id` está referenciando al valor del campo `end station id`.

Otros operadores

El operador `$type` permite recuperar documentos que dependan del tipo de campo que contiene.

```
db.trips.find({"start station name":{$type:"string"}})
```

El operador `$where` permite introducir una expresión *JavaScript*. No se recomienda su uso al no utilizar los índices existentes, a no ser que no haya manera de expresar un criterio mediante los operadores existentes.

Consultas sobre arrays

Preparando los ejemplos

Para los siguientes ejemplos sobre documentos anidados y arrays, vamos a utilizar una colección de 500 documentos sobre mensajes de un blog.

Esta colección (`posts`) está cargada tanto en el clúster de MongoAtlas de la base de datos `sample_training`.

Un ejemplo de mensaje sería:

```
use sample_training
// switched to db sample_training
db.posts.findOne()
// { _id: ObjectId("50ab0f8bbcf1bfe2536dc3f9"),
//   body: 'Amendment I\n<p>Congress shall make ...."\n<p>\n',
//   permalink: 'aRjNnLZkJkTyspAIoRGe',
//   author: 'machine',
//   title: 'Bill of Rights',
//   tags: [
//     'santa',
//     'xylophone',
//     'math',
//     'dream',
//     'action' ],
```

```
//   comments:
//   [ { body: 'Lorem ipsum dolor ...',
//       email: 'HvizfYVx@pKvLaagH.com',
//       author: 'Santiago Dollins' },
//     { body: 'Lorem ipsum dolor sit...',
//       email: 'WpOUCpdD@hccdxJvT.com',
//       author: 'Jaclyn Morado' },
//     { body: 'Lorem ipsum dolor sit amet...',
//       email: 'OgDzHfFN@cWsDtCtx.com',
//       author: 'Houston Valenti' }],
//   date: 2012-11-20T05:05:15.231Z }
```

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

Si hacemos una consulta sobre si un *array* contiene un determinado elemento, obtendremos todos los documentos que contengan dicho valor:

```
db.posts.find({tags:"santa"}, {tags:1})
// { _id: ObjectId('50ab0f8bbcf1bfe2536dc3f9'),
//    tags: ['watchmaker', 'santa', 'xylophone', 'math', 'handsaw', 'dream',
'undershirt', 'dolphin', 'tanker', 'action'] }
// { _id: ObjectId('50ab0f8bbcf1bfe2536dc43b'),
//    tags: ['cup', 'community', 'santa', 'height', 'peen', 'beer', 'criminal',
'cousin', 'refund', 'clover'] }
// { _id: ObjectId('50ab0f8bbcf1bfe2536dc50d'),
//    tags: ['santa', 'spear', 'opinion', 'rainbow', 'century', 'puffin', 'romanian',
'scent', 'river', 'supermarket'] }
```

Si queremos que coincida exactamente (mismos elementos en el mismo orden), debemos indicarlo pasando el propio array

```
db.posts.find({tags:["santa"]}, {tags:1})
// No encuentra ningún elemento
```

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

- `$all` para filtrar ocurrencias que tienen todos los valores especificados, es decir, los valores pasados a la consulta serán un subconjunto del resultado. Puede que devuelva los mismos, o un array con más campos (el orden no importa)
- `$in`, igual que SQL, para obtener las ocurrencias que cumple con alguno de los valores pasados (similar a usar `$or` sobre un conjunto de valores de un mismo campo). Si queremos su negación, usaremos `$nin`, para obtener los documentos que no cumplen ninguno de los valores.

Por ejemplo, si queremos obtener los mensajes que contenga las etiquetas `dream` y `action` tendríamos:

```
db.posts.find( {tags: {$all: ["dream", "action"]}}, {tags:1})
// { "_id": ObjectId('50ab0f8bbcf1bfe2536dc3f9'),
//   "tags": ["watchmaker", "santa", "xylophone", "math", "handsaw", "dream",
"undershirt", "dolphin", "tanker", "action"] }
```

En cambio, si queremos los mensajes que contengan alguna de esas etiquetas haríamos:

```
db.posts.find( {tags: {$in: ["dream", "mongodb"]}}, {tags:1})
// { "_id": ObjectId('50ab0f8bbcf1bfe2536dc3f9'),
//   "tags": ["watchmaker", "santa", "xylophone", "math", "handsaw", "dream",
"undershirt", "dolphin", "tanker", "action"] }
```

Si lo que nos interesa es la cantidad de elementos que contiene un array, emplearemos el operador `$size`. Por ejemplo, para obtener los mensajes que tienen 10 etiquetas haríamos:

```
db.posts.find( {tags : {$size : 10}} )
```

Finalmente, si queremos buscar por un valor en una posición específica del array, sabiendo que el primer índice es el elemento 0, podemos hacerlo mediante la notación punto:

```
db.posts.find({"tags.0": "santa"}, {tags:1})
// { _id: ObjectId('50ab0f8bbcf1bfe2536dc50d'),
//    tags: ['santa', 'spear', 'opinion', 'rainbow', 'century', 'puffin',
'romanian', 'scent', 'river', 'supermarket'] }
```

Arrays de documentos

Si el array contiene documentos y queremos filtrar la consulta sobre los campos de los documentos del *array*, tenemos que utilizar `$elemMatch`, de manera que obtengamos aquellos que al menos encuentre un elemento que cumpla el criterio. Dicho de otra manera, el operador `elemMatch` garantiza que todas las condiciones se cumplan en el mismo elemento del array.

Así pues, si queremos recuperar los mensajes que tienen un comentario cuyo autor sea `Santiago Dollins` el cual tiene como email `xnZKyvWD@jHfVKtUh.com` haríamos:

```
db.posts.find( {comments: {$elemMatch: { author: "Santiago Dollins", email:
"xnZKyvWD@jHfVKtUh.com"}}} )
```

Criterio con notación punto

En el ejemplo anterior, si sólo hubiéramos tenido un campo para el filtrado, podríamos haber utilizado la notación punto `comments.author`.

De forma similar a antes, si queremos filtrar por un elemento que ocupa una determinada posición, usaremos la notación punto sabiendo que el primer índice comienza por 0. Así pues, si queremos recuperar aquellos mensajes donde el primer mensaje tenga un valor determinado, podemos hacer:

```
db.posts.find( {"comments.0.author": "Santiago Dollins"} )
```

Proyección de *arrays*

A la hora de proyectar los datos, si no estamos interesados en todos los valores de un campo que es un *array*, podemos restringir el resultado mediante el operador `$slice`. Así pues, si quisiéramos obtener los mensajes titulados `US Constitution` y que, de esos mensajes, mostrara sólo tres etiquetas y dos comentarios, haríamos:

```
db.posts.find( {title : "US Constitution"}, {comments: {$slice:2}, tags:
{$slice:3}} )
// { _id: ObjectId("50ab0f8bbcf1bfe2536dc416"),
//   body: 'We the personas ...',
//   permalink: 'NhWDUNColpvxFjovsgqU',
//   author: 'machine',
//   title: 'US Constitution',
//   tags: [ 'engineer', 'granddaughter', 'sundial' ],
//   comments:
//    [ { body: 'Lorem ipsum dolor ...',
//        email: 'ftRlVMZN@auLhwhlj.com',
//        author: 'Leonida Lafond' },
//      { body: 'Lorem ipsum dolor sit...',
//        email: 'dsoLAdFS@VGBBuDVs.com',
//        author: 'Nobuko Linzey' } ],
//   date: 2012-11-20T05:05:15.276Z }
```

En cuanto a las proyecciones sobre subdocumentos contenidos dentro de un array, además de filtrar mediante `elemMatch` hemos de indicarlo en la proyección. Así pues, si sólo queremos los comentarios escritos por un determinado autor, haríamos:

```
db.posts.find(
  {comments: {$elemMatch: { author: "Santiago Dollins", email:
"xnZKyvWD@jHfVKtUh.com"}}},
  {comments: {$elemMatch: { author: "Santiago Dollins", email:
"xnZKyvWD@jHfVKtUh.com"}}} )

// {
//   _id: ObjectId('50ab0f8bbcf1bfe2536dc45e'),
//   comments: [
//     {
//       body: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit
esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est
laborum',
//       email: 'xnZKyvWD@jHfVKtUh.com',
//       author: 'Santiago Dollins'
//     }
//   ]
// }
```

Si sólo nos interesa el primer elemento que coincida podemos emplear el operador posicional $ (lo estudiaremos más profundamente cuando trabajemos las operaciones DML sobre arrays):

```
db.posts.find(
  {comments: {$elemMatch: { author: "Santiago Dollins", email:
"xnZKyvWD@jHfVKtUh.com"}}},
  {"comments.$": 1} )
```