Operaciones CRUD en MongoDB

Las operaciones CRUD son las operaciones básicas que se pueden realizar sobre una base de datos. CRUD es el acrónimo de *Create, Read, Update, Delete*.

En este documento veremos cómo se realizan estas operaciones en MongoDB desde la *shell* de mongo (`mongosh`). Empezaremos por la creación de documentos. A continuación veremos lecturas / consultas para ver cómo se definen y usan los selectores, ya que se usarán los selectores en el resto de operaciones. Y finalmente iremos viendo el resto de operaciones: modificar (*update*) y borrar (*delete*).

Creación de documentos *Create* / Insertar

Para crear documentos en una colección usaremos los comandos `insertOne` o `insertMany`.

`db.<nombre de la colección>.insertOne(<json del documento>)`

`db.<nombre de la colección>.insertMany(<array con los documentos json>)`

Los métodos `insertOne` e `insertAny` también aceptan un argumento con opciones.

El campo `_id`

Si el documento no especifica un campo `_id`, entonces MongoDB añade el campo `_id` y le asigna un `ObjectId()` único para el documento. La mayoría de los drivers crean un `ObjectId` y lo asignarán al atributo `_id`, pero el `mongod` creará y rellenará el campo `_id` si el driver o la aplicación no lo hace.

Búsqueda de documentos: *Read* / Consultar

Este función sirve para realizar consultas sobre las colecciones de la base de datos. Para *filtrar* o *seleccionar* los datos que queremos obtener habrá que pasar como argumento cierta información. En MongoDB esta información se pasa en forma de documento JSON.

Vayamos viendo estos selectores poco a poco, ya que también se usan en las operaciones de modificación y borrado.

Para consultar los documentos de una colección que cumplan una determinada condición se usa el método `find()` y, en caso de que sólo nos interese el primer documento que cumpla la condición podemos usar `findOne()`.

```
db.<nombre de la colección>.find(<filtro>)
```

Cursores

El método `find()` no devuelve los documentos buscados si no que devuelve un **cursor** a los misos. Un cursor es un objeto que permite recorrer los documentos o lo que es lo mismo, *iterar sobre el cursor* para obtener los documentos resultado de la consulta.

Cuando hacemos una consulta en la *shell* de mongo y **no asignamos** el resultado a una variable, la consola de mongo itera por su cuenta sobre el cursor y muestra los resultados.

Nota: No hay una forma simple de conocer el número de resultados que devuelve un cursor **sin consumir** el mismo. Tanto el método `count()` (no recomendado y no disponible en algunos cursores) como el método `explain` **consumirán el cursor**.

Nota: Para conocer el número de resultados que, probablemente, vamos a obtener con una consulta hemos de usar el método `countDocuments(<query>, <options>)`.

Existen varias formas para iterar sobre un cursor en la shell de mongo:

Método `next()`

```
var cursor = db.airbnb_bar.find()

while (cursor.hasNext()) {
    printjson(cursor.next());
}
```

Método `forEach()`

```
db.arirbnb_bar.find().forEach(function(doc) {
    printjson(doc);
});
```

o lo que es lo mismo:

```
db.airbnb_bar.find().forEach(printjson)
```

Método `toArray()`

Podemos convertir un cursor a un array con el método `toArray()`. Hay que tener en cuenta que este método puede consumir mucha memoria si el cursor contiene muchos documentos ya que carga en la RAM **todos los documentos del cursor**.

```
var cursor = db.airbnb_bar.find()
let array = cursor.toArray()

printjson(array[6])
```

Consultas con una condición

Para hacer un filtro con una única condición simplemente se incluye un JSON con el **atributos** y el **valor** del mismo que queremos seleccionar. Si queremos seleccionar los documentos de la colección **usuarios** que están activos podríamos de filtro:

```
{
    'activo': true
}
```

pasándole este filtro a la función `find()` quedaría:

```
db.usuarios.find({activo: true})
```

(`find()` / `find({})` sin filtro (o con un filtro *vacío*) devolverá todos los documentos de la colección).

```
db.<nombre de la colección>.find( {<campo>: <valor>} )
```

Por ejemplo, para obtener los documentos de la colección `airbnb_bar` cuyo campo `host_neighbourhood` sea `Dreta de l'Eixample`, escribiríamos:

```
db.det_listings.find( { host_neighbourhood: "Dreta de l'Eixample"} )
```

Antes de continuar con selectores más complejos veamos un par de métodos que nos pueden ser útiles para depurar nuestras consultas.

Contar resultados

Para contar los resultados de una consulta se usa el método `count()`.

```
db.<nombre de la colección>.find(<filtro>).count()
```

Si deseamos conocer el número de elementos de una colección podemos usar `count()` directamente sobre la colección.

```
db.<nombre de la colección>.count()
```

Limitar el número de resultados

Para limitar el número de valores que obtendremos como resultado de una consulta tenemos la función `limit()`. Recibirá como argumento un número entero en el que le indicamos cuantas respuestas nos interesan.

```
db.<nombre de la colección>.find(<filtro>).limit(<número de resultados>)
```

`limit()` se usa frecuentemente en combinación con `skip()` y `sort()`:

- `skip(<número>)`: Sirve para *saltarse* cierto número de documentos en un resultado (cursor).
- `sort(<documento>)`: Sirve para ordenar los resultados en función a uno o más campos.

Ejemplo de sort:

En el documento que se le pasa al sort se indican los campos sobre los que se quiere realizar la ordenación (numérica o lexicográfica). El valor del campo será `1` o `-1`, para indicar si queremos que la ordenación sea creciente o decreciente.

```
db.alumnos.find({aprobado: true}).sort({nota: -1, nombre: 1, apellidos:
1}).limit(3)
```

Con este comando obtendremos los tres alumnos con mejor nota (en caso de empate se ordenarán alfabéticamente por nombre y apellidos).

Consulta con múltiples condiciones

Este selector es igual al anterior pero incluyendo en el JSON todos los pares **atributo - valor** que nos interese. Es equivalente a un **and** lógico.

```
{
    'rol': 'admin',
    'activo': true
}
```

sería equivalente a:

```
{
    $and: [ { 'rol': 'admin' }, { 'activo': true } ]
}
```

Consulta con múltiples condiciones válidas (`$or`)

Incluiremos un *atributo* `$or` cuyo valor ha de ser un **array** con las condiciones que serían válidas.

```
{
    $or: [ { 'rol': 'admin' }, { 'activo': true } ]
}
```

A diferencia de en el caso anterior ahora nos devolvería todos los usuarios que sean administradores o que estén activos.

Consulta con selección por exclusión (`$not`)

Normalmente lo usaremos junto con el atributo `$eq` para significar *not equal*, es decir, distinto.

```
{
    'nombre': { $not: { $eq: "Manuel" } }
}
```

Operadores lógicos de consulta

- `$not`: Es la única que se aplica a una única expresión. En el resto de operadores lógicos se aplica a un array de expresiones.
- `$and`: Une las cláusulas de búsqueda con un **and** lógico y devolverá los resultados que cumplan todas las cláusulas.
- `$or`: Une las cláusulas de búsqueda con un **or** lógico y devolverá los resultados que cumplan alguna de las cláusulas.
- `$nor`: Une las cláusulas de búsqueda con un **nor** lógico y devolverá los resultados que no cumplan ninguna de las cláusulas.

Ejemplo de `$and`:

```
{ $and: [ { <expression1> }, { <expression2> } , ... , { <expressionN> } ] }
```

Operadores de comparación

Con estos operadores se pueden realizar consultas de comparación sobre los campos de los documentos. Toman como argumento un valor y devuelven los documentos que cumplan la condición.

- `$eq`: Igual a.
- `$ne`: Distinto de.
- `$gt`: Mayor que.
- `$gte`: Mayor o igual que.
- `$lt`: Menor que.
- `$lte`: Menor o igual que.
- `$in`: Igual a cualquiera de los valores de un array.
- `$nin`: Distinto a todos los valores de un array.

De este modo podríamos hacer una consulta para obtener los usuarios con una edad entre 18 y 65 años:

```
{
    'edad': { $gte: 18, $lte: 65 }
}
```

Consultas sobre arrays

Seleccionar por igualdad en array

```
db.alumnos.find( { modulos: ['BDA', 'SBD'] } )
```

En esta consulta hay que tener en cuenta que **importa el orden** de los elementos del array. **Estamos comparando mediante una igualdad**.

Seleccionar por contenido del array

Si lo que queremos es buscar los elementos que contienen un elemento (y posiblemente otros) lo haremos de la siguiente manera:

```
db.alumnos.find({
    modulos: 'SBD'
})
```

Nótese que no se compara con ningún array, si no con un elemento del array.

Si queremos comprobar si contiene viarios elementos a la vez (y posiblemente otros) usaremos el operador `$all`.
El operador `$all` permite consultar los documentos que contengan todos los valores de un array que especificamos.

De esta forma con el siguiente filtro se devolverán los documentos que contengan los valores `matemáticas` y `física` en su array `especialidades`.

```
db.profesores.find( {especialidades: { $all: ['matemáticas', 'física'] } } )
```

O los alumnos que cursan los módulos `BDA` y `SBD` (y posiblemente otros ya que no es excluyente):

```
db.alumnos.find({
    modulos: {
        $all: ['BDA', 'SBD']
    }
})
```

Cuando utilizamos el operador `$all` no importa el orden.

Filtros compuestos

Con estos filtros lo que hacemos es poner condiciones a **algún** elemento del array. Así el siguiente código:

```
const cursor = db.collection('inventory').find({
  dim_cm: { $gt: 15, $lt: 20 }
});
```

Seleccionará los documentos cuyo array `dim_cm` contenga **algún elemento** cuyo valor sea estrictamente mayor que 15 y menor que 20.

Filtro para TODOS los elementos de un array

Supongamos que tenemos la siguiente base de datos:

```
[
    {
        'id': 1,
        'notas': [5, 6, 8]
    },
    {
        'id': 2,
        'notas': [4, 5, 7]
    }
]
```

Si queremos seleccionar todos los documentos cuyas notas (**todas**) sean mayores o iguales a 5 hemos de escribir el siguiente código:

```
db.alumnos.find({
    notas: {$not: {$lt: 5}}
})
```

Puesto que no hay un operador que exija que todos los elementos cumplan una condición, lo que habrá que hacer es exigir que no haya algún elemento que cumpla la condición opuesta.

Por ejemplo, si no queremos que ningún elemento del array de notas contenga la notas 5 y 6:

```
db.alumnos.find({
   "notas": {
      $not: {
         $all: [5, 6]
      }
   }
})
```

Consultas sobre documentos embebidos

Para realizar consultas sobre documentos embebidos se usa la notación de punto.

Si tenemos un documento con la siguiente estructura:

```
{
   'id': 1,
   'nombre': 'Manuel',
   'apellidos': 'Piñeiro',
   'dirección': {
      'calle': 'Calle de la Rosa',
      'número': 1,
      'piso': 2,
      'puerta': 'A'
   }
}
```

para hacer una consulta sobre el campo `calle` del documento embebido `dirección` escribiríamos:

```
db.alumnos.find({
    'dirección.calle': 'Calle de la Rosa'
})
```

Update / Actualizar

`updateOne` o `updateMany`

`db.<nombre de la colección>.updateOne(<json selector>, <json de actualización>)`

Atomicidad

En MongoDB una operación de escritura es atómica a nivel de un único documento. Aún cuando la operación modifique múltiples documentos embebidos en dicho documento.

Cuando una única operación de escritura modifique múltiples documentos la modificación de **cada documento** es atómica pero la operación **en su conjunto** no lo es.

Operadores a nivel de campos

Estos operadores, como el título indica, realizarán cambios en los campos de un documento.

Operador `$set`

El operador `$set` reemplaza el valor de un campo por el valor especificado. Si el campo no existe lo crea.

Sintaxis de `$set`

```
{ $set: { <campo1>: <valor1>, ... } }
```

Si queremos especificar un campo dentro de un documento embebido hemos de usar la notación con punto.

Ejemplo de `$set`

```
db.alumnos.updateMany({}, {
    $set: {
        activo: false,
        modulos: []
    }
})
```

Operador `$unset`

Este operador se utilizará para eliminar campos de un documento.

Sintaxis de `$unset`

```
{ $unset: { <campo1>: <valor>, <campo2>: <valor>} }
```

Los valores de los campos se ignoran.

Ejemplo de `$unset`

El siguiente comando cambiará el estado de los alumnos cuyo curso sea igual a `2022/2023` a `inactivo` y eliminará el campo `notas`.

```
db.alumnos.updateMany( { curso: '2022/2023'}, {
    $set: { estado: 'inactivo' },
    $unset: { notas: 0 }
} )
```

Operadores `$inc` y `$mul`

El operador `$inc` *incrementa* el valor de un campo un determinado valor.
Este operador admite valores tanto positivos como negativos.

El operador `$mul`, por su parte, multiplica el valor de un campo por un determinador valor.

Sintaxis de `$inc`

```
{
    $inc: { <campo1>: <cantidad a incrementar>, <campo2>: <cantidad a
incrementar>, ... }
    $mul: { <campo3>: <multiplicador>, <campo4>: <multiplicador>, ... }
}
```

Si el campo a modificar no existe se crea. En ambos casos se crea con el valor inicial 0 y luego se aplica la modificación.

Operadores `$max` y `$min`

Estos operadores **sólo** modificarán el valor de un campo **si el nuevo valor es mayor**, en el caso de `$max`, **o menor**, en el caso de `$min`.

Sintaxis de `$max`

```
{
    $max: { <campo1>: <nuevo valor>, <campo2>: <nuevo valor> },
    $min: { <campo3>: <nuevo valor>, <campo4>: <nuevo valor> }
}
```

Si el campo no existe se crea y se le asigna el valor indicado.

Operador `$rename`

Sirve para renombrar campos.

Sintaxis de `$rename`

```
{
    $rename: { <campo1>: <nuevo nombre>, ... }
}
```

Operador `$currentDate`

Este operador es algo más complejo que los anteriores. Sirve para modificar un campo al valor de la fecha actual.

Sintaxis de `$currentDate`

```
{
    $currentDate: {
        <campo1>: true | { $type: 'date' } | { $type: 'timestamp' }
    }
}
```

Si el valor del campo es `true` se actualiza al valor de `Date` del momento actual. Si indicamos `$type` `timpestamp` se creará una marca de tiempo `Timestamp`. En el caso de que el valor se `date` se creará `Date`.

Si los campos no existen se crean.

Operador `$setOnInsert`

Este operador sólo tiene sentido si indicamos la opción `upsert: true`. Esta opción indica que si el selector no selecciona ningún documento se cree uno nuevo con los valores que se indique en la modificación.

Mediante el operador `$setOnInsert` indicaremos valores *por defecto* para campos que **sólo se crearán** si estamos insertando un nuevo documento como resultado del `upsert: true`.

Sintaxis de `$setOnInsert`

```
{
    $setOnInsert: { <campo1>: <valor1>, ... }
}
```

Operadores para *arrays*

En este apartado veremos los operadores para actuar sobre *arrays* y también los modificadores.

- `$`: Actúa como un *placeholder* para actualizar **el primer elemento** del array que cumpla las condiciones del selector.
- `$[]`: Actúa como un *placeholder* para actualizar **todos los elementos** que cumplan la condición del selector.
- `$[<identificador>]`: Actúa como un *placeholder* para actualizar **todos los elementos** que cumplan la condición de los ***arrayFilters*** para los documentos que hayan sido seleccionados por el selector.
- `$addToSet`: Añade valores a un array si no se encuentran ya en el mismo.
- `$pop`: Elimina el primer elemento del array.
- `$pull`: Elimina **todos los elementos** del array que cumplan la condición del selector.
- `$push`: Añade elementos al array.
- `$pullAll`: Elimina todos los elementos de un array.

Modificadores de los operadores de actualización

- `$each`: Se aplica al operador `$push` y `$addToSet` para añadir, uno a uno, varios elementos de un array.
- `$position`: Se aplica a `$push` para indicar la posición del array donde se añadirán los elementos.
- `$slice`: Se aplica a `$push` para limitar el tamaño del array modificado.
- `$sor`: También se aplica a `$push` para reordenar los documentos almacenados en un array.

Operador `$`

Este operador hace referencia al primer elemento de un array que cumpla la condición del selector. Se utiliza de la siguiente manera:

```
db.alumnos.updateOne(
    { notas: { $lt: 5 } },
    { $set: { 'notas.$': 5} }
)
```

Es **obligatorio** que, si queremos modificar el array `notas` éste debe de aparecer en el selector.

Operador `$[]`

Este operador afectará a **todos los elementos** del array del documento que cumplan con las condiciones del selector.

Si modificamos el ejemplo anterior:

```
db.alumnos.updateOne(
    { notas: { $lt: 5 } },
    { $set: { 'notas.$[]': 5} }
)
```

Ahora, en lugar de modificar el primer elemento del array que cumpla la condición, modificará **todos los elementos**.

Operador `$[<indentificador>]`

En este caso hemos de usar un documento de **opciones** con la opción `arrayFilters`. Esta opción sirve para aplicar *filtros* etiquetados (de ahí el `identificador`) para modificar sólo los elementos que pasen el filtro.

Ejemplo del operador `$[<indentificador>]`

El siguiente código:

```
db.alumnos.updateMany(
    { },
    { $set: { 'notas.$[cond1]': 10} },
    { arrayFilters: [ { cond1: { $eq: 5 } } ] }
)
```

1. Aplicará el selector `{ }` de manera que seleccionará todos los documentos de la colección.
2. A continuación empezará a ejecutar la operación `$set` del *update document*.
3. En dicha operación se hace referencia al filtro `cond1` que implica la igualdad de un elemento del array con el valor 5.
4. Se aplicará la operación `$set` a los elementos del array `notas` que cumplan que su valor es 5.

Operador `$push`

El operador `$push` añade un valor a un array.

Sintaxis de `$push`

```
{ $push: { <campo1>: <valor1>, ... } }
```

Para indicar un campo en un **documento embebido** se puede usar la **notación de punto**.

Modificadores

- `$each`: Añadirá múltiples valores al array.
- `$slice`: Limita el número de elementos a añadir. Debe usarse con el modificador `$each`.
- `$sort`: Ordena los elementos del array. Se usará junto al modificador `$each`. Opciones:
  - Si insertamos elementos simples el valor `1` indicará orden ascendente y `-1` descendente.
  - Si insertamos documentos habrá que indicar el campo y luego especificar `1` o `-1` para indicar orden ascendente o descendente.
- `$position`: Especifica la posición del array en la que insertar los nuevos elementos. Requiere el uso de `$each`. Si no se usa `$position` `$push` insertará el elemento al final del array.

Ejemplo de sort:

```
{ $push: {
  <campo1>: { $each: [ { <c1>: 1, <c2>: 'a'}, { <c1>: 2, <c2>: 'b'}],
  sort: { <c1>: -1}}
}}
```

Sintaxis de los modificadores

```
{ $push: { <campo1>: { <modificador1>: <valor1>, ...}, ... } }
```

La forma de procesarse los modificadores será la siguiente:

1. Actualizar el array para añadir los elementos en la posición correcta.
2. Aplicar la ordenación si se especificó.
3. *Recortar* el array si se especificó.
4. Guardar el array.

Ejemplos de uso

Creamos la colección `estudiantes`:

```
db.estudiantes.insertOne( { _id: 1, notas: [ 44, 78, 38, 80] } )
```

Añadir un valor al array

El siguiente código añadirá el valor 89 al array `notas`:

```
db.estudiantes.updateOne( { _id: 1 }, { $push: { notas: 89 } } )
```

Añadir un valor a un array en múltiples documentos

Añadiremos el valor 89 al array `notas` de todos los estudiantes.

```
db.estudiantes.updateMany( {}, { $push: { notas: 89 } } )
```

Añadir múltiples valores a un array

```
db.estudiantes.updateOne( { _id: 1 }, { $push: {notas: {$each: [ 90, 92, 95 ] } }
} )
```

Utilizar `$push` con múltiples modificadores

Crearemos un nuevo estudiante para el ejemplo:

```
db.estudiantes.insertOne(
    {
    "_id" : 5,
        "quizzes" : [
            { "wk": 1, "score" : 10 },
            { "wk": 2, "score" : 8 },
            { "wk": 3, "score" : 5 },
            { "wk": 4, "score" : 6 }
        ]
    }
)
```

Haremos lo siguiente:

- Usaremos el operador `$each` para añadir cada `quiz` del array al array `quizzes` del documento.
- Usaremos `$sort` para ordenar dicho array `quizzes` de manera **descendiente** (argumento -1).
- Usamos `$slice` para quedarnos únicamente con los tres primeros elementos de `quizzes`.

```
db.estudiantes.updateOne(
    { _id: 5},
    {
        $push: {
            quizzes: {
                $each: [ { wk: 5, score: 8 }, { wk: 6, score: 7 }, { wk: 7,
score: 6 } ],
                $sort: { score: -1 },
                $slice: 3
            }
        }
    }
)
```

Ejemplo: Si tenemos un documento de la colección "usuarios" con el siguiente formato:

```
{
    id: 1,
    nombre: "Manuel",
    apellidos: "Piñeiro",
    fecha_nacimiento: "1977-05-07"
}
```

Para actualizar `apellidos` y `fecha_nacimiento` a "Piñeiro Mourazos" y "1977-05-15" respectivamente, usaríamos el siguiente comando:

```
db.usuarios.updateOne({ id: 1 }, { apellidos: "Piñeiro Mourazos",
fecha_nacimiento: "1977-05-07" })
```

Nota: se puede añadir atributos.

Destroy / Eliminar

`deleteOne` o `deleteMany` y usar los filtros / selectores de MongoDB. Estos selectores los veremos en el apartado de *read* / consultas pues son los mismos.

Estos selectores lo veremos en detalle en el apartado *Read* / consultas pues son los mismos.

Estos comandos tienen como **valor de retorno** un documento (JSON) con dos campos:

- ***acknowledged***: un valor booleano que será cierto si la operación se ejecutó con *write concern* o falso si no lo hizo.
- ***deleteCount***: Indica el número total de documentos borrados.