

## Operaciones CRUD

### 1. Inserción de documentos

Para insertar un documento en una colección se usa el método insert:

```
> db.prueba.insert<<"titulo":"El Quijote">>
WriteResult<< "nInserted" : 1 >>
> _
```

Esta acción añadirá al documento el campos “\_id” en caso de no existir en el documento, y almacenará el mismo en MongoDB.

Cuando es necesario insertar un conjunto de documentos, se puede pasar como parámetro un array con el conjunto de documentos que deben ser insertados:

```
> db.prueba.insert([{"_id" : 0}, {"_id" : 1}, {"_id" : 2}])
BulkWriteResult<<
  "writeErrors" : [ ],
  "writeConcernErrors" : [ ],
  "nInserted" : 3,
  "nUpserted" : 0,
  "nMatched" : 0,
  "nModified" : 0,
  "nRemoved" : 0,
  "upserted" : [ ]
>>
> _
```

Se pueden insertar mediante un array multiples documentos siempre que se vayan almacenar en una única colección, en caso de varias colecciones no es posible y habría que usar otras herramientas tales como mongoimport.

Observar:

- Cuando se inserta usando un array, si se produce algún fallo en algún documento, se insertan todos los documentos anteriores al que tuvo el fallo, y los que hay a continuación no se insertan. Este comportamiento se puede cambiar usando la opción “continueOnError” que en caso de encontrarse un error en un documento lo salta, y continua insertando el resto de documentos. Esta opción no está disponible directamente en la shell, pero si en los drivers de los lenguajes de programación.
- Actualmente existe un límite de longitud de 48 MB para las inserciones realizadas usando un array de documentos.

Cuando se inserta un documento MongoDB realizan una serie de operaciones con el objetivo de evitar inconsistencias tales como:

- Se añade el campo “\_id” en caso de no tenerlo.
- Se comprueba la estructura básica. En particular se comprueba el tamaño del documento(debe ser más pequeño de 16 Mb). Para saber el tamaño de un documento se puede usar el comando `Object.bsonsize(doc)`.
- Existencia de caracteres no válidos.

## 2. Borrado de documentos

El método `remove` elimina todos los documentos de una colección, pero no elimina la colección ni la metainformación acerca de la colección:

```
> db.prueba.find(<)<br>< "_id" : 0 ><br>< "_id" : 1 ><br>< "_id" : 2 ><br>> db.prueba.remove(< >><br>WriteResult<< "nRemoved" : 3 >><br>>
```

El método permite opcionalmente tomar una condición de búsqueda, de forma que eliminará solo aquellos documentos que encajen con la condición dada. Por ejemplo si se quisiera eliminar todos los documentos de la colección `correo.lista` dónde el valor para el campo `"salida"` es cierto entonces se usaría el siguiente comando:

**`db.correo.lista.remove({"salida":true})`**

Una vez que se ha realizado el borrado no se puede dar revertir y se pierden todos los documentos borrados.

A veces si se van a borrar todos los documentos es más rápido eliminar toda la colección en vez los documentos. Para ello se usa el método `drop`:

**`db.prueba.drop()`**

## 3. Actualización de documentos

Para modificar un documento almacenado se usa el método `update` que toma 2 parámetros:

- Una condición de búsqueda que localiza el documento a actualizar.
- Un conjunto de cambios a realizar sobre el documento.

Las actualizaciones son atómicas de manera que si se quieren realizar dos a la vez, la primera que llegue es la primera en realizarse y a continuación se hará la siguiente.

### • Reemplazamiento de documentos

El tipo de actualización más simple consiste en reemplazar un documento por otro. Por ejemplo que se quiere cambiar el siguiente documento:

```
{<br>  "nombre" : "Juan",<br>  "amigos" : 32,<br>  "enemigos" : 2<br>}
```

Y se quiere crear un campo `"relaciones"` que englobe a los campos `"amigos"` y `"enemigos"` como subdocumentos. Esta operación se puede llevar a cabo con un `update`:

```

C:\mongodb\bin\mongo.exe
> var juan = db.prueba.findOne({"nombre" : "Juan"});
> juan.relaciones= {"amigos":juan.amigos, "enemigos":juan.enemigos};
> {"amigos" : 32, "enemigos" : 2 }
> juan.PrimerNombre=juan.nombre
Juan
> delete juan.amigos
true
> delete juan.enemigos
true
> delete juan.nombre
true
> db.prueba.update({"nombre":"Juan"},juan);
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.prueba.findOne()
{
  "_id" : ObjectId("54d892c60bc8bbed882d4153"),
  "relaciones" : {
    "amigos" : 32,
    "enemigos" : 2
  },
  "PrimerNombre" : "Juan"
}

```

Un error común es cuando encaja más de un documento con el criterio de búsqueda y se crea un campo duplicado “\_id” con el segundo parámetro. En este caso la base de dato genera un error y ningún documento es actualizado. Por ejemplo supóngase que se crean varios documentos con el mismo valor para el campo “nombre”, sea “Juan”, y se quiere actualizar el valor del campo edad de uno de ellos (se quiere aumentar el valor de la edad del segundo “Juan”):

```

C:\mongodb\bin\mongo.exe
>>>
> db.prueba.find()
{ "_id" : ObjectId("54d894f00bc8bbed882d4154"), "nombre" : "Juan", "edad" : 32 }
{ "_id" : ObjectId("54d894f00bc8bbed882d4155"), "nombre" : "Juan", "edad" : 33 }
{ "_id" : ObjectId("54d894f00bc8bbed882d4156"), "nombre" : "Juan", "edad" : 45 }
> juan=db.prueba.findOne({"nombre":"Juan","edad":33})
{
  "_id" : ObjectId("54d894f00bc8bbed882d4155"),
  "nombre" : "Juan",
  "edad" : 33
}
> juan.edad++;
33
> db.prueba.update({"nombre":"Juan"},juan);
WriteResult({
  "nMatched" : 0,
  "nUpserted" : 0,
  "nModified" : 0,
  "writeError" : {
    "code" : 16837,
    "errmsg" : "The _id field cannot be changed from { '_id': ObjectId('54d894f00bc8bbed882d4154') } to { '_id': ObjectId('54d894f00bc8bbed882d4155') } ."
  }
})
>>

```

Se produce un error dado que el método update busca un documento que encaje con la condición de búsqueda y el primero que encuentra es el referido al “Juan” que tiene 32 años. Intenta cambiar ese documento por el actualizado, y se encuentra que si hace el cambio habría dos documentos con el mismo “\_id”, y eso no es posible (el “\_id” debe ser único). Para evitar estas situaciones lo mejor es usar el método update con el campo “\_id” que es único. En el ejemplo anterior se podría hacer la actualización si se hiciera de la siguiente forma:

```
> db.prueba.update(<<"_id":ObjectId<"54d894f00bc8bbed882d4155">>,juan>;
WriteResult<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>
>
```

Otra ventaja de usar el campo “\_id” es que el documento está indexado por este campo.

- Modificadores

En muchas ocasiones el tipo de actualización que se quiere realizar consiste en añadir, modificar o eliminar claves, manipular arrays y documentos embebidos,... Para estos casos se van a usar un conjunto de operadores de modificación.

\$inc

Este operador permite cambiar el valor numérico de una clave que ya existe incrementando su valor por el especificado junto al operador, o bien puede crear una clave que no existía inicializándola al valor dado.

Por ejemplo supóngase que se mantienen los datos estadísticos de un sitio web en una colección de manera que se incrementa un contador cada vez alguien visita una página. Para ello se tiene un documento que almacena la URL y el número de visitas de la página:

```
> db.prueba.findOne()
{
  "_id" : ObjectId<"54d89c5f0bc8bbed882d4158">,
  "URL" : "www.ejemplo.es",
  "visitas" : 34
}
```

Cada vez que alguien visita una página se busca la página a partir de su URL y se incrementa el campo de “visitas” con el modificador “\$inc” que incrementa el campo dado en el valor descrito:

```
> db.prueba.update(<<"URL":"www.ejemplo.es">>,<<"$inc": {"visitas":1}>>)
WriteResult<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>
> db.prueba.findOne()
{
  "_id" : ObjectId<"54d89c5f0bc8bbed882d4158">,
  "URL" : "www.ejemplo.es",
  "visitas" : 35
}
```

También sería posible incrementar el valor por un valor mayor que 1:

```
> db.prueba.findOne()
{
  "_id" : ObjectId<"54d8a9e00bc8bbed882d415d">,
  "URL" : "www.ejemplo.es",
  "visitas" : 34
}
> db.prueba.update(<<"URL":"www.ejemplo.es">>,<<"$inc": {"visitas":30}>>)
WriteResult<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>
> db.prueba.findOne()
{
  "_id" : ObjectId<"54d8a9e00bc8bbed882d415d">,
  "URL" : "www.ejemplo.es",
  "visitas" : 64
}
```

Y de la misma forma se podría decrementar usando números negativos:

```
> db.prueba.update(<<"URL":"www.ejemplo.es">>,<<"$inc": {"visitas":-37}>>)
WriteResult(<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>)
> db.prueba.findOne()
{
  "_id" : ObjectId<"54d8a9e00bc8bbbed882d415d">,
  "URL" : "www.ejemplo.es",
  "visitas" : 27
}
```

Y por ejemplo se podría añadir un nuevo campo para indicar el número de enlaces de la página:

```
> db.prueba.update(<<"URL":"www.ejemplo.es">>,<<"$inc": {"enlaces":20}>>)
WriteResult(<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>)
> db.prueba.findOne()
{
  "_id" : ObjectId<"54d8a9e00bc8bbbed882d415d">,
  "URL" : "www.ejemplo.es",
  "visitas" : 27,
  "enlaces" : 20
}
```

Observar:

- Cuando se usan operadores de modificación el valor del campo “\_id” no puede ser cambiado (en cambio cuando se reemplaza un documento entero si es posible cambiar el campo “\_id”). Sin embargo los valores para cualquier otra clave incluyendo claves indexadas únicas si pueden ser modificadas.
- Este operador solo puede ser usado con números enteros, enteros largos o double, de manera que si se usa con otro tipo de valores (incluido los tipos que algunos lenguajes tratan como números tales como booleanos, cadenas de números, nulos,...) producirá un fallo. En el ejemplo se intenta incrementar un campo con un valor no numérico

```
> db.prueba.update(<<"URL":"www.ejemplo.es">>,<<"$inc": {"enlaces":"20"}>>)
WriteResult(<<
  "nMatched" : 0,
  "nUpserted" : 0,
  "nModified" : 0,
  "writeError" : {
    "code" : 14,
    "errmsg" : "Cannot increment with non-numeric argument: {enlaces
: \"20\"}"
  }
>>
>
```

### “\$set” y “\$unset”

Este operador establece un valor para un campo dado, y si el campo dado no existe entonces lo crea. En este sentido es útil para modificar el esquema de un documento o añadir claves definidas por el usuario. Por ejemplo supóngase que se tiene el perfil de usuario almacenado en un documento:

```
> db.prueba.findOne()
{
  "_id" : ObjectId<"54d8a03e0bc8bbbed882d415a">,
  "nombre" : "Juan",
  "edad" : 34,
  "sexo" : "Varon",
  "localizacion" : "Madrid"
}
```

Si el usuario desea añadir un campo sobre su libro favorito, se podría hacer usando el modificador “\$set”:

```
> db.prueba.update({"_id":ObjectId("54d8a03e0bc8bbed882d415a")},
... {"$set": {"libroFavorito": "Guerra y Paz"}})
WriteResult< { "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 } >
> db.prueba.findOne()
{
  "_id" : ObjectId("54d8a03e0bc8bbed882d415a"),
  "nombre" : "Juan",
  "edad" : 34,
  "sexo" : "Uaron",
  "localizacion" : "Madrid",
  "libroFavorito" : "Guerra y Paz"
}
```

Nuevamente si el usuario desea cambiar el valor del campo sobre su libro favorito por otro valor, se podría hacer usando también el modificador “\$set”:

```
> db.prueba.update({"_id":ObjectId("54d8a03e0bc8bbed882d415a")}, {"$set": {"libroFavorito": "El Quijote"}})
WriteResult< { "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 } >
> db.prueba.findOne()
{
  "_id" : ObjectId("54d8a03e0bc8bbed882d415a"),
  "nombre" : "Juan",
  "edad" : 34,
  "sexo" : "Uaron",
  "localizacion" : "Madrid",
  "libroFavorito" : "El Quijote"
}
```

También con el modificador “\$set” es posible cambiar el tipo de un campo que se modifica. Por ejemplo si se quiere que el campo “libroFavorito” sea un array en vez de un valor único también puede usarse el modificador “\$set”:

```
> db.prueba.update({"_id":ObjectId("54d8a03e0bc8bbed882d415a")}, {"$set": {"libroFavorito": ["Guerra y Paz","El Quijote"]}})
WriteResult< { "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 } >
> db.prueba.findOne()
{
  "_id" : ObjectId("54d8a03e0bc8bbed882d415a"),
  "nombre" : "Juan",
  "edad" : 34,
  "sexo" : "Uaron",
  "localizacion" : "Madrid",
  "libroFavorito" : [
    "Guerra y Paz",
    "El Quijote"
  ]
}
```

También mediante el operador “\$set” es posible realizar cambios en documentos embebidos, para lo cual sólo es necesario indicar el campo en el que se encuentran. Por ejemplo considerar el siguiente documento:

```
> db.prueba.findOne()
{
  "_id" : ObjectId("54d8a48f0bc8bbed882d415b"),
  "titulo" : "Post de un blog",
  "contenido" : "el contenido",
  "autor" : {
    "nombre" : "Isabel",
    "email" : "isa@prueba.com"
  }
}
```

Se quiere cambiar el valor del campo del nombre del autor del post, entonces se haría lo siguiente:

```
> db.prueba.update(<<"autor.nombre":"Isabel">>,<<"$set":{"autor.nombre":"Isabel Sanz"}}>>)
WriteResult(<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>)
> db.prueba.findOne(<>)
{
  "_id" : ObjectId("54d8a48f0bc8bbbed882d415b"),
  "titulo" : "Post de un blog",
  "contenido" : "el contenido",
  "autor" : {
    "nombre" : "Isabel Sanz",
    "email" : "isa@prueba.com"
  }
}
```

Observar:

- Existe un operador denominado “\$unset” que permite eliminar campos de un documento. En el ejemplo anterior si se quiere eliminar el campo “libroFavorito”:

```
> db.prueba.findOne(<>)
{
  "_id" : ObjectId("54d8a6230bc8bbbed882d415c"),
  "nombre" : "Juan",
  "edad" : 34,
  "sexo" : "Uaron",
  "localizacion" : "Madrid",
  "libroFavorito" : [
    "Guerra y Paz",
    "El Quijote"
  ]
}
> db.prueba.update(<<"_id" : ObjectId("54d8a6230bc8bbbed882d415c")>>,<<"$unset":{"libroFavorito":1}>>)
WriteResult(<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>)
> db.prueba.findOne(<>)
{
  "_id" : ObjectId("54d8a6230bc8bbbed882d415c"),
  "nombre" : "Juan",
  "edad" : 34,
  "sexo" : "Uaron",
  "localizacion" : "Madrid"
}
```

- Para añadir, modificar o eliminar claves se debe usar siempre los modificadores “\$”. En este sentido observar que si intentara hacer un cambio en las claves con un comando como el siguiente: `db.prueba.update(criterio, {"edad":"país"})` tendría como efecto reemplazar el documento que encaje con el criterio de búsqueda por el documento {"edad": "país"}.

#### 4. Modificadores de los arrays

- Adición de elementos

El modificador “\$push” añade elementos al final del array si existe o bien crea uno nuevo si no existe. Por ejemplo, supóngase que se almacenan posts de un blog y se quiere añadir una clave “comentarios” que contenga un array de comentarios. Esto puede hacerse usando el modificador “\$push” que en el ejemplo crea una nueva clave denominada “comentarios”

```

> db.prueba.findOne()
{
  "_id" : ObjectId("54d8c0a60bc8bbed882d415f"),
  "titulo" : "Posts de un blog",
  "contenido" : "el contenido"
}
> db.prueba.update(<<"titulo":"Posts de un blog">>,<<"$push":{"comentarios":
... <"nombre":"Juan",
... "email":"juan@ejemplo.com",
... "contenido":"buen post."}>>>)
WriteResult(<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>)
> db.prueba.findOne()
{
  "_id" : ObjectId("54d8c0a60bc8bbed882d415f"),
  "titulo" : "Posts de un blog",
  "contenido" : "el contenido",
  "comentarios" : [
    {
      "nombre" : "Juan",
      "email" : "juan@ejemplo.com",
      "contenido" : "buen post."
    }
  ]
}
1
>

```

A continuación, si se quiere añadir otro comentario se usa nuevamente el modificador "\$push":

```

> db.prueba.update(<<"titulo":"Posts de un blog">>,<<"$push":{"comentarios": <"nombr
re":"Isabel", "email":"isabel@ejemplo.com", "contenido": "buen post."}>>>)
WriteResult(<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>)
> db.prueba.findOne()
{
  "_id" : ObjectId("54d8c2d40bc8bbed882d4160"),
  "titulo" : "Posts de un blog",
  "contenido" : "el contenido",
  "comentarios" : [
    {
      "nombre" : "Juan",
      "email" : "juan@ejemplo.com",
      "contenido" : "buen post."
    },
    {
      "nombre" : "Isabel",
      "email" : "isabel@ejemplo.com",
      "contenido" : "buen post."
    }
  ]
}
1
>

```

Este modificador también puede usarse junto al modificador "\$each" para añadir múltiples valores en una sola operación. Por ejemplo si se quisiera añadir 3 valores a un array denominado horas:

```

> db.prueba.update(<<"_id" : ObjectId("54d8c58e0bc8bbed882d4162")>>, <<"$push":{"ho
ras":{"$each":[34,56,33]}>>>)
WriteResult(<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>)
> db.prueba.findOne()
{
  "_id" : ObjectId("54d8c58e0bc8bbed882d4162"),
  "horas" : [
    34,
    56,
    33
  ]
}
1
>

```

Observar que si se especifica un array con un único elemento, su comportamiento es similar a un "\$push" sin "\$each".

También es posible limitar la longitud hasta la que puede crecer un array usando el operador "\$slice" junto al operador "\$push". Por ejemplo en el siguiente array se limita el crecimiento hasta 10 elementos, de manera que si se introducen 10 elementos se guardan todos, pero si se introducen más de 10 elementos entonces solo se guardan los 10 últimos elementos.



```

> db.peliculas.update(<<"genero" : "historicas">>, <<"$push" : {"top10" : < "$each"
: [{"nombre" : "Los cañones del Navarone", "valoracion" : 6.6}, {"nombre" : "El
Cid Campeador", "valoracion" : 4.3}]>, "$slice" : -10}>>>)
WriteResult<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>
> db.peliculas.findOne()
<
  "_id" : ObjectId<"54d8d4690bc8bbbed882d4163">,
  "genero" : "historicas",
  "top10" : [
    "Los cañones del Navarone",
    "El Cid Campeador"
  ]
1
>

```

Observar que el operador “\$slice” puede tomar valores negativos(empieza a contar desde el final) o bien valores positivos(empieza a contar desde el principio) y esencialmente actúa creando una cola o una pila en el documento.

Por último el operador “\$sort” permite ordenar los elementos indicando el campo de ordenación y el criterio en forma de 1(ascendente) 0 -1(descendente). En el siguiente ejemplo se guardan los primeros 10 elementos ordenados de acuerdo al campo “valoración” en orden ascendente:

```

> db.peliculas.update(<<"genero" : "historica">>, <<"$push" : {"top10" : < "$each"
: [{"nombre" : "Los cañones del Navarone", "valoracion" : 6.6}, {"nombre" : "El
Cid campeador", "valoracion" : 4.3}]>, "$slice" : -10, "$sort" : {"valoracion" :
-1}>>>>)
WriteResult<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>
> db.peliculas.findOne()
<
  "_id" : ObjectId<"54d8d8030bc8bbbed882d4165">,
  "genero" : "historica",
  "top10" : [
    {
      "nombre" : "Los cañones del Navarone",
      "valoracion" : 6.6
    },
    {
      "nombre" : "El Cid campeador",
      "valoracion" : 4.3
    }
  ]
1
>

```

Observar que:

- Tanto “\$slice” como “\$sort” deben ir junto a un operador “\$each” y no pueden aparecer solos con un “\$push”.
- “\$sort” también puede ser usado para ordenar elementos que no son documentos, en cuyo caso no hay que indicar ningún campo. En el siguiente ejemplo se insertan dos elementos y se ordena el conjunto de manera ascendente:

```

> db.estudiantes.findOne()
< "_id" : 2, "tests" : [ 89, 70, 89, 50 ] >
> db.estudiantes.findOne()
< "_id" : 2, "tests" : [ 89, 70, 89, 50 ] >
> db.estudiantes.update( < _id: 2 >, < $push: { tests: { $each: [ 40, 60 ]
, $sort: 1 } } > )
WriteResult<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>
> db.estudiantes.findOne()
< "_id" : 2, "tests" : [ 40, 50, 60, 70, 89, 89 ] >
>

```

También es posible ordenar un array completo especificando un array vacío junto al operador “\$each”. En el siguiente ejemplo se ordena el array en orden descendente.

```

> db.estudiantes.findOne()
< "_id" : 2, "tests" : [ 40, 50, 60, 70, 89, 89 ] >
> db.estudiantes.update( < _id: 2 >, < $push: { tests: { $each: [ ], $sort
: -1 } } > )
WriteResult<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>
> db.estudiantes.findOne()
< "_id" : 2, "tests" : [ 89, 89, 70, 60, 50, 40 ] >
>

```

- Si “\$slice” se usa con un array vacío entonces se aplica a los elementos actuales del array. Por ejemplo en el siguiente ejemplo se reduce la longitud del array, y nos quedamos con el primer elemento.

```
> db.peliculas.findOne()
{
  "_id" : ObjectId("54d8d8030bc8bbed882d4165"),
  "genero" : "historica",
  "top10" : [
    {
      "nombre" : "Los cañones del Navarone",
      "valoracion" : 6.6
    },
    {
      "nombre" : "El Cid campeador",
      "valoracion" : 4.3
    }
  ]
}
> db.peliculas.update(<<"_id" : ObjectId("54d8d8030bc8bbed882d4165")>>,
... <<{"$push":{"top10":{"$each":[1,"$slice":1]}}}>>)
WriteResult(<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>)
> db.peliculas.findOne()
{
  "_id" : ObjectId("54d8d8030bc8bbed882d4165"),
  "genero" : "historica",
  "top10" : [
    {
      "nombre" : "Los cañones del Navarone",
      "valoracion" : 6.6
    }
  ]
}
>
```

- Usando arrays como conjuntos

Los arrays se pueden tratar como un conjunto añadiendo valores solo si no estaban ya. Para ello se usa el operador “\$ne” junto al operador “\$push”. Por ejemplo si se quiere añadir un autor a una lista de citas pero solo en el caso de que no estuviera, entonces se podría hacer de la siguiente manera:

```
> db.articulos.findOne()
{
  "_id" : ObjectId("54d8e24b0bc8bbed882d4166"),
  "autores citados" : [
    "Juan",
    "Pablo"
  ]
}
> db.articulos.update(<<"autores citados": <<{"$ne":"Pepe"}>>, <<{"$push":{"autores ci
tados": "Pepe"}}>>)
WriteResult(<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>)
> db.articulos.findOne()
{
  "_id" : ObjectId("54d8e24b0bc8bbed882d4166"),
  "autores citados" : [
    "Juan",
    "Pablo",
    "Pepe"
  ]
}
>
```

Alternativamente también es posible hacer la misma operación mediante el operador “\$addToSet”. Por ejemplo supóngase que se tiene un documento que representa a un usuario y se dispone de un clave que es un array de direcciones de correo electrónico, entonces si se quiere añadir una nueva dirección sin que se produzcan duplicados se puede hacer de la siguiente manera mediante “\$addToSet”:

```

> db.usuarios.findOne()
{
  "_id" : ObjectId("54d8e4aa0bc8bbbed882d4167"),
  "nombre" : "Isabel",
  "emails" : [
    "isa@ejemplo.com",
    "isabel@mail.com",
    "isabels@yahoo.es"
  ]
}
> db.usuarios.update(
...   "_id" : ObjectId("54d8e4aa0bc8bbbed882d4167"),
...   < "$addToSet": { "emails": "isa@gmail.com" } >,
WriteResult(
  < "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 > )
> db.usuarios.findOne()
{
  "_id" : ObjectId("54d8e4aa0bc8bbbed882d4167"),
  "nombre" : "Isabel",
  "emails" : [
    "isa@ejemplo.com",
    "isabel@mail.com",
    "isabels@yahoo.es",
    "isa@gmail.com"
  ]
}
>

```

Es posible utilizar “\$addToSet” junto al operador “\$each” para añadir valores múltiples únicos lo cual no puede ser hecho con la combinación “\$ne”/”\$push” . Por ejemplo si quisiéramos añadir más de una dirección de correo electrónico se podría hacer de la siguiente manera:

```

> db.usuarios.findOne()
{
  "_id" : ObjectId("54d8e4aa0bc8bbbed882d4167"),
  "nombre" : "Isabel",
  "emails" : [
    "isa@ejemplo.com",
    "isabel@mail.com",
    "isabels@yahoo.es",
    "isa@gmail.com"
  ]
}
> db.usuarios.update(
...   "_id" : ObjectId("54d8e4aa0bc8bbbed882d4167"),
...   < "$addToSet": { "emails": { "$each": [ "isa@php.com", "isa@python.com", "isa@java.com" ] } } >,
WriteResult(
  < "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 > )
> db.usuarios.findOne()
{
  "_id" : ObjectId("54d8e4aa0bc8bbbed882d4167"),
  "nombre" : "Isabel",
  "emails" : [
    "isa@ejemplo.com",
    "isabel@mail.com",
    "isabels@yahoo.es",
    "isa@gmail.com",
    "isa@php.com",
    "isa@python.com",
    "isa@java.com"
  ]
}
>

```

#### • Borrado de elementos

Existen varias formas de eliminar elementos de un array dependiendo de la forma en la que se quieran gestionar. Si se quiere gestionar como si fuera una pila o una cola entonces se puede usar el operador “\$pop” que permite eliminar elementos del final del array (si toma el valor 1) o bien del principio del array (si toma el valor -1):

```

> db.prueba.findOne()
{ "_id" : 1, "scores" : [ 8, 9, 10 ] }
> db.prueba.update(
...   < "_id" : 1 >,
...   < "$pop": { "scores": -1 } >,
WriteResult(
  < "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 > )
> db.prueba.findOne()
{ "_id" : 1, "scores" : [ 9, 10 ] }
> db.prueba.update(
...   < "_id" : 1 >,
...   < "$pop": { "scores": 1 } >,
WriteResult(
  < "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 > )
> db.prueba.findOne()
{ "_id" : 1, "scores" : [ 9 ] }
>

```

Otra forma alternativa de eliminar elementos es especificando un criterio en vez de una posición en el array usando el operador “\$pull”. Por ejemplo si se tiene un conjunto de tareas que se tienen que realizar en un orden dado:

```
> db.listas.findOne()
{
  "_id" : ObjectId("54d8eb3f0bc8bbed882d4168"),
  "tareas" : [
    "desayunar",
    "comer",
    "merendar",
    "cenar"
  ]
}
> db.listas.update({"_id" : ObjectId("54d8eb3f0bc8bbed882d4168")}, {"$pull" : {"tareas" : "comer"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.listas.findOne()
{
  "_id" : ObjectId("54d8eb3f0bc8bbed882d4168"),
  "tareas" : [
    "desayunar",
    "merendar",
    "cenar"
  ]
}
>
```

Observar que el operador “\$pull” elimina todas las coincidencias que encuentre en los documento no solo la primera coincidencia. Por ejemplo la eliminación del valor 1 en el siguiente array numéricos lo dejará con un solo elemento:

```
> db.lista.findOne()
{
  "_id" : ObjectId("54d8ed800bc8bbed882d4169"),
  "aciertos" : [
    1,
    2,
    1,
    1,
    1
  ]
}
> db.lista.update({"_id" : ObjectId("54d8ed800bc8bbed882d4169")}, {"$pull": {"aciertos":1}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.lista.findOne()
{
  "_id" : ObjectId("54d8ed800bc8bbed882d4169"),
  "aciertos" : [ 2 ]
}
>
```

Observar que los operadores de array sólo pueden ser usados sobre claves con array de valores, no siendo posible modificar valores escalares(para ello se usa “\$set” o “\$inc”).

- Modificaciones posicionales en un array

Las manipulaciones de un array se convierten en algo complejo cuando se tienen múltiples valores y se quieren modificar solo algunos de ellos. En este sentido existen dos caminos para manipular valores de un array:

- Mediante su posición. En este caso sus elementos son seleccionados como si se indexaran las claves de un documento. Por ejemplo supóngase que se tiene un array con documentos embebidos tales como los comentarios a un post de un blog y se quiere incrementar el número de votos del primer comentario entonces se puede hacer de la siguiente manera:

```

> db.blog.posts.findOne()
{
  "_id" : ObjectId("4b329a216cc613d5ee930192"),
  "contenido" : "el contenido",
  "comentarios" : [
    {
      "comentario" : "buen post",
      "autor" : "Juan",
      "votos" : 0
    },
    {
      "comentario" : "un poco corto",
      "autor" : "Clara",
      "votos" : 3
    },
    {
      "comentario" : "no está mal",
      "autor" : "Alicia",
      "votos" : -1
    }
  ]
}
>
> db.blog.posts.update(<<"_id" : ObjectId("4b329a216cc613d5ee930192")>>,<<"$inc": <
"comentarios.0.votos":1>>>)
WriteResult<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>
> db.blog.posts.findOne()
{
  "_id" : ObjectId("4b329a216cc613d5ee930192"),
  "contenido" : "el contenido",
  "comentarios" : [
    {
      "comentario" : "buen post",
      "autor" : "Juan",
      "votos" : 1
    },
    {
      "comentario" : "un poco corto",
      "autor" : "Clara",
      "votos" : 3
    },
    {
      "comentario" : "no está mal",
      "autor" : "Alicia",
      "votos" : -1
    }
  ]
}
>
>

```

- En algunas ocasiones no se conoce el índice del array que se quiere modificar sin antes consultar el documento. Para estos casos se dispone de un operador posicional "\$" que indica el elemento del array que encaja con la condición de búsqueda y actualiza ese elemento. Si en el ejemplo anterior se tiene un autor de un comentario que se llama "Juan" y se quiere actualizar a "Juanito" entonces se puede hacer de la siguiente manera con el operador posicional:

```

> db.blog.posts.update(<<"comentarios.autor":"Juan">>,<<"$set": <{"comentarios.$.autor":"Juanito"}>>)
WriteResult<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>
> db.blog.posts.findOne()
{
  "_id" : ObjectId("4b329a216cc613d5ee930192"),
  "contenido" : "el contenido",
  "comentarios" : [
    {
      "comentario" : "buen post",
      "autor" : "Juanito",
      "votos" : 1
    },
    {
      "comentario" : "un poco corto",
      "autor" : "Clara",
      "votos" : 3
    },
    {
      "comentario" : "no está mal",
      "autor" : "Alicia",
      "votos" : -1
    }
  ]
}
>
>

```

Observar que el operador posicional solo actualiza la primera coincidencia, de manera que si hubiera más de un comentario realizado por "Juan" solo se actualizaría el primer comentario.

## 5. Rapidez de los modificadores

Algunos modificadores son más rápidos que otros. Así por ejemplo "\$inc" modifica un documento variando unos pocos bytes por lo que es muy eficiente, sin embargo los modificadores de array pueden variar considerablemente el tamaño de un array por lo que son lentos ("\$set" puede modificar documentos de manera eficiente si no cambia el tamaño pero en caso contrario entonces tiene las mismas limitaciones que los operadores de array).

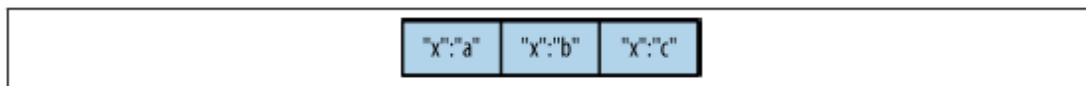
La eficiencia bien determinada por la forma en que se insertan los documentos. Cuando se empieza a insertar documentos, cada documento se pone junto a los previos en el disco, de manera que si un documento crece de tamaño entonces no podrá colocarse en el sitio original y tendrá que ser movido a otra parte de la colección. Para ver cómo funciona se puede crear una colección con un par de documentos de manera que el documento que se encuentra en la mitad se le cambia de tamaño, entonces será empujado al final de la colección.

```
> db.coll.insert<<"x" : "a">>
WriteResult<< "nInserted" : 1 >>
> db.coll.insert<<"x" : "b">>
WriteResult<< "nInserted" : 1 >>
> db.coll.insert<<"x" : "c">>
WriteResult<< "nInserted" : 1 >>
> db.coll.find()
< "_id" : ObjectId<"54d908950bc8bbbed882d416d">, "x" : "a" >
< "_id" : ObjectId<"54d908970bc8bbbed882d416e">, "x" : "b" >
< "_id" : ObjectId<"54d908980bc8bbbed882d416f">, "x" : "c" >
> db.coll.update<<"x" : "b">, <$set: <"x" : "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb">>
WriteResult<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>
> db.coll.find()
< "_id" : ObjectId<"54d908950bc8bbbed882d416d">, "x" : "a" >
< "_id" : ObjectId<"54d908980bc8bbbed882d416f">, "x" : "c" >
< "_id" : ObjectId<"54d908970bc8bbbed882d416e">, "x" : "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb" >
>
```

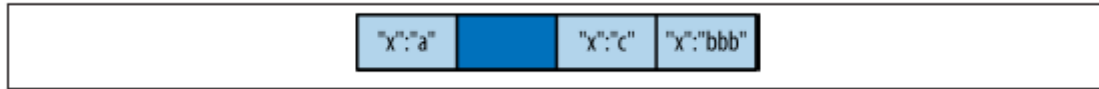
Cuando MongoDB tiene que mover un documento, aplica un factor de relleno a la colección que es la cantidad de espacio extra que deja alrededor de cada documento por si crecen en tamaño. En el ejemplo anterior se puede ver mediante el método stats():

```
> db.coll.stats()
<
  "ns" : "prueba.coll",
  "count" : 3,
  "size" : 208,
  "avgObjSize" : 69,
  "storageSize" : 8192,
  "numExtents" : 1,
  "nindexes" : 1,
  "lastExtentSize" : 8192,
  "paddingFactor" : 1,
  "systemFlags" : 1,
  "userFlags" : 1,
  "totalIndexSize" : 8176,
  "indexSizes" : <
    "_id_" : 8176
  >,
  "ok" : 1
>
```

Inicialmente el factor de relleno es 1 puesto que asigna exactamente el tamaño del documento por cada nuevo documento.



Cuando se ejecuta varias veces las actualizaciones haciendo más grandes los documentos entonces el factor de relleno crece alrededor de 1.5 dado que cada nuevo documento se le asignará la mitad del tamaño como espacio libre para crecer.



Si posteriores actualizaciones producen movimientos de documentos entonces el factor de relleno continuará creciendo aunque no tan rápidamente como en el primer caso. Si no existen más movimientos entonces el factor de relleno irá decreciendo lentamente.

El movimiento de documentos es lento dado que hay que liberar el espacio que ocupaba el documento y escribirlo en un nuevo sitio. Por tanto hay que intentar mantener el factor de relleno tan cerca de 1 como sea posible. En general no se puede modificar manualmente el factor de relleno salvo que se esté realizando una compactación de datos pero si es posible realizar un esquema que no dependa de que los documentos se hagan grandes arbitrariamente.

El siguiente programa es un ejemplo que muestra la diferencia entre las actualizaciones que no producen variación del tamaño del documento (y por tanto son eficientes) y las actualizaciones que requieren mover documentos. En el siguiente programa de ejemplo se inserta una clave y se incrementa su valor 10000 veces:

```
var timeLnc=function() {
var start=(new Date()).getTime();
for (var i=0; i<10000; i++) {
  db.testers.update({}, {"$inc": {"x": 1}});
  db.getLastError();
}
var timeDiff=(new Date()).getTime()-start;
print ("Tiempo tomado en actualizar: "+ timeDiff + " ms");
}
```

```
> db.testers.insert({"x" : 1})
WriteResult({ "nInserted" : 1 })
> var timeLnc=function() { var start=(new Date()).getTime(); for (var i=0; i<10000; i++) {
  db.testers.update({}, {"$inc": {"x": 1}}); db.getLastError(); }
var timeDiff=(new Date()).getTime()-start; print ("Tiempo tomado en actualizar: "+ timeDiff + " ms"); }
> timeLnc()
Tiempo tomado en actualizar: 50237 ms
> _
```

Como se puede observar se tarda casi 51 segundos. Sin embargo si se cambia la llamada de actualización y se usa un "\$push":

```
var timeLnc=function() {
var start=(new Date()).getTime();
for (var i=0; i<10000; i++) {
  db.testers.update({}, {"$push": {"x": 1}})
  db.getLastError();
}
var timeDiff=(new Date()).getTime()-start;
print ("Tiempo tomado en actualizar: "+ timeDiff + " ms");
}
```

```

> db.testar.insert({ "x" : 1 })
WriteResult({ "nInserted" : 1 })
> var timeInc=function() {
...   var start=(new Date()).getTime();
...   for (var i=0; i<10000; i++) {
...     db.testar.update({}, {"$push" : {"x" : 1}})
...     db.getLastError();
...   }
...   var timeDiff=(new Date()).getTime()-start;
...   print ("Tiempo tomado en actualizar: " + timeDiff + " ms");
... }
> timeInc()
Tiempo tomado en actualizar: 54720 ms

```

En este caso se tarda casi 55 segundos.

En conclusión usar “\$push” y modificadores de array con frecuencia es necesario pero es bueno tener en cuenta las ventajas y desventajas que tienen este tipo de actualización. En este sentido si “\$push” se convierte en un cuello de botella entonces puede merecer la pena colocar los documentos embebidos en una colección separada, rellenarlos manualmente o bien utilizar alguna otra técnica de optimización. Por otra parte MongoDB no es bueno reusando espacio vacío por lo que los cambios de documentos pueden dar lugar a grandes espacios vacíos de datos, y producir mensajes de advertencia avisando de tal situación para que se lleve a cabo una compactación dado que hay un alto nivel de fragmentación.

Si el esquema que se va a usar requiere muchos movimientos de documentos, inserciones y borrados se puede mejorar el uso del disco usando la opción `usePowerOf2Sizes` que se puede configurar con el comando `collMod`:

```
db.runCommand({"collMod" : collectionName, "usePowerOf2Sizes" : true})
```

Con esta opción activada, todas las asignaciones de memoria que se hagan serán en bloques de tamaño de potencia de 2, pero la asignación de espacio será menos eficiente. Por ejemplo si se utiliza esta configuración es colecciones en las que solo se inserta o se hacen actualizaciones que no producen variación del tamaños, estas operaciones se realizarán de una manera más lenta.

Observar que si se ejecuta nuevamente el comando anterior con la opción a falso se vuelve al modo normal:

```
db.runCommand({"collMod" : collectionName, "usePowerOf2Sizes" : false})
```

Así mismo los efectos de esta opción solo afectan a los registros que asignados nuevos, por lo que no existe peligro en correrlo sobre una colección existente.

## 6. Upsert

Un upsert es un tipo especial de actualización. Si no se encuentra ningún documento que coincida con el criterio de búsqueda, entonces se crea un nuevo documento que combina el criterio de búsqueda y la actualización. Si se encuentra un documento que encaja con las condiciones de búsqueda entonces será actualizado de forma normal. Este tipo de actualización puede ser muy útil para generar una colección si se tiene el mismo código para crear y actualizar documentos.

Considérese el ejemplo anterior sobre el almacenamiento del número de visitas de cada página de un sitio web. Sin un upsert, se podría intentar encontrar la URL e incrementar el



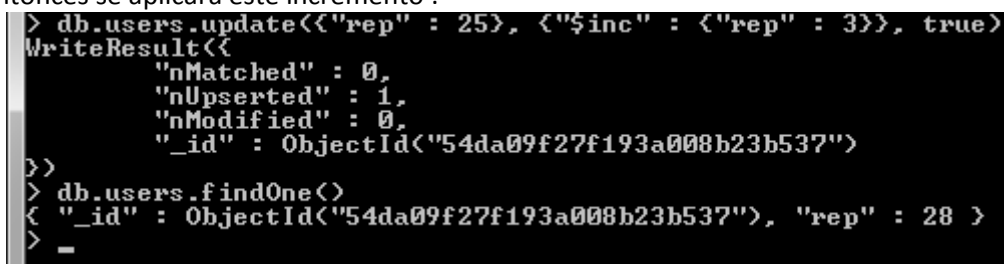
número de visitas o crear un nuevo documento si la URL no existe. Por ejemplo si se implementara mediante un programa en JavaScript sería de la siguiente forma:

```
// Se cheque si existe alguna entrada en la página
blog = db.analytics.findOne({url : "/blog"})
// Si existe, se actualiza, se añade uno y se almacena
if (blog) {
  blog.pageviews++;
  db.analytics.save(blog);
}
// En caso contrario, se crea un nuevo documento
else {
  db.analytics.save({url : "/blog", pageviews : 1})
}
```

Todo este código se podría reducir si usamos un upsert(es el tercer parámetro de un update) con las ventajas de ser una operación atómica y muchas más rápida:

```
db.analytics.update({"url" : "/blog"}, {"$inc" : {"pageviews" : 1}}, true)
```

El nuevo documento es creado usando la especificación dada del documento y aplicando las modificaciones descritas. Por ejemplo si se hace un upsert sobre la clave de un documento y tiene especificada como modificación el incremento del valor de la clave, entonces se aplicará este incremento :



```
> db.users.update({"rep" : 25}, {"$inc" : {"rep" : 3}}, true)
WriteResult<{
  "nMatched" : 0,
  "nUpserted" : 1,
  "nModified" : 0,
  "_id" : ObjectId<"54da09f27f193a008b23b537">
}>
> db.users.findOne()
< {"_id" : ObjectId<"54da09f27f193a008b23b537">, "rep" : 28 }
> _
```

El upsert crea un nuevo documento con valor 25 para la clave “rep” y a continuación lo incrementa en 3, generando un documento donde la clave “rep” toma el valor de 28. Si no se hubiera especificado la opción de upsert entonces no habría encontrado ningún documento encajando con las condiciones dadas y no habría hecho nada. Observar que si se vuelve a correr la misma actualización pasará lo mismo pues seguirá sin encontrar un documento que encaje con las condiciones de búsqueda, creando otro documento nuevamente.

A veces un campo necesita ser inicializado cuando es creado pero no cambiado en actualizaciones posteriores. Para ello se usa el modificador “\$setOnInsert” junto a upsert activado(valor a cierto). Este modificador lo que hace es que inicializa el valor de una clave cuando el documento es insertado pero si vuelve a ejecutarse no hace nada sobre la clave que ha inicializado en el documento que se ha insertado. En el ejemplo la primera vez crea un documento y actualiza el campo “createdAt”, pero si se volviera a ejecutar la misma actualización se podría ver que no se actualiza nuevamente el campo “createdAt”.

```

> db.users.update({}, {"$setOnInsert" : {"createdAt" : new Date()}}, true)
WriteResult<{
  "nMatched" : 0,
  "nUpserted" : 1,
  "nModified" : 0,
  "_id" : ObjectId<"54da0e427f193a008b23b538">
}>
> db.users.findOne()
{
  "_id" : ObjectId<"54da0e427f193a008b23b538">,
  "createdAt" : ISODate<"2015-02-10T13:57:22.092Z">
}

```

Observar que “\$setOnInsert” puede ser útil para la inicialización de contadores, colecciones que no usan “ObjectIds” o para crear rellenos en los documentos.

Relacionado con upsert está la función de la shell “save” que permite insertar un documento si no existe y actualizarlo si existe. Toma como argumento un documento. Si el documento contiene un “\_id” entonces “save” hará un upsert, y en caso contrario lo insertará. Está pensado para facilitar la modificación de documentos en la shell. Por ejemplo:

```

> db.prueba.insert({"num":4})
WriteResult<{ "nInserted" : 1 }>
> var x = db.prueba.findOne()
> x.num=42
42
> db.prueba.save(x)
WriteResult<{ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }>
> db.prueba.findOne()
{ "_id" : ObjectId<"54da11ccc794d8f74ef8b20d">, "num" : 42 }

```

Sin el “save” habría que haber hecho un update de la siguiente forma:

```

> db.prueba.update({"_id":x._id},x)

```

## 7. Actualización de múltiples documentos

Las actualizaciones por defecto solo modifican el primer documento que encaja con los criterios de búsqueda, de manera que si existen más documentos que cumplen las condiciones, entonces no se cambian. Para que los cambios se realicen en todos los documentos que coinciden, entonces hay que pasar como cuarto parámetro del update un valor true. Por ejemplo considerar el caso de que se quiere dar un regalo a cada usuario en el día de su cumpleaños, en este caso se podría realizar una actualización múltiple de la siguiente manera:

```

db.users.update({"cumpleaños" : "10/13/1978"},
... {"$set" : {"regalo" : "¡Feliz cumpleaños!"}}, false, true)

```

Con esta expresión se añadiría el mensaje de “¡Feliz cumpleaños!” a todos los documentos en los que el campo “cumpleaños” tuviera el valor de 13 de octubre de 1978.

Para ver el número de documentos que han sido actualizados mediante una actualización múltiple se puede usar el comando getLastError que retorna información acerca de la última operación que ha sido realizada. La clave “n” contiene el número de documentos que se han visto afectados por una actualización.

```

> db.count.insert(<<"x":1>>)
WriteResult<< "nInserted" : 1 >>
> db.count.update(<<x : 1>>, <<$inc : {x : 1}>>, false, true)
WriteResult<< "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 >>
> db.runCommand(<<getLastError : 1>>)
{
  "connectionId" : 4,
  "updatedExisting" : true,
  "n" : 1,
  "syncMillis" : 0,
  "writtenTo" : null,
  "err" : null,
  "ok" : 1
}

```

Se puede observar que n=1 que indica que fue actualizado 1 documento y por otra parte "updateExisting" toma el valor de "true" que indica que algún documento fue actualizado.

## 8. Retornando documentos actualizados

Mediante el comando "getLastError" se puede obtener información acerca de las actualizaciones, sin embargo no es posible recuperar los documentos que fueron actualizados. Para ello se puede usar el comando "findAndModify". Para ilustrar una situación en la que puede ser útil se va a plantear un ejemplo. Supóngase que se tiene una colección de procesos que se ejecutan en un cierto orden de forma que cada proceso es representado con un documento que tiene la siguiente forma:

```

{
  "_id" : ObjectId(),
  "estado" : state,
  "prioridad" : N
}

```

donde estado es una cadena que puede ser "Preparado", "Ejecutándose" o bien "Ejecutado".

Se necesita encontrar la prioridad más alta en el estado "Preparado", ejecutar el proceso y actualizar el estado a "Ejecutado". Se podría consultar sobre los procesos preparados, ordenarlos por prioridad y actualizar el estado del proceso de más alta prioridad a "Ejecutándose". Una vez que hubiera finalizado se actualizaría a "Ejecutado". Esto se podría implementar de la siguiente manera:

```

var cursor = db.procesos.find({"estado" : "Preparado"});
ps = cursor.sort({"prioridad" : -1}).limit(1).next();
db.procesos.update({"_id" : ps._id}, {"$set" : {"estado" : "Ejecutándose"}});
do_something(ps);
db.procesos.update({"_id" : ps._id}, {"$set" : {"estado" : "Ejecutado"}});

```

Este algoritmo tiene un problema dado que tiene una condición de carrera. Supóngase que se tienen dos hilos corriendo, y uno de los hilos(sea A) recupera el documento y el otro hilo(sea B) recupera el mismo documento pero antes que A haya actualizado su estado a "Ejecutándose", entonces ambos hilos estarían ejecutando el mismo proceso. Esta situación se podría evitar consultando previamente el estado como parte de la actualización:

```

var cursor = db.procesos.find({"estado" : "Preparado"});
cursor.sort({"prioridad" : -1}).limit(1);
while ((ps = cursor.next()) != null) {
  ps.update({"_id" : ps._id, "estado" : "Preparado"},
    {"$set" : {"estado" : "Ejecutándose"}});
  var lastOp = db.runCommand({getLastError : 1});
}

```

```

if (lastOp.n == 1) {
  do_something(ps);
  db.procesos.update({"_id" : ps._id}, {"$set" : {"estado" : "Ejecutado"}})
  break;
}
cursor = db.procesos.find({"estado" : "Preparado"});
cursor.sort({"prioridad" : -1}).limit(1);
}

```

También podría ocurrir que dependiendo de la temporización, un hilo realizara todo el trabajo mientras otro hilo lo está intentado. El hilo A podría quedarse el proceso, y el B intentaría conseguir el mismo proceso, fallas y dejar que A haga todo el trabajo.

Estas situaciones se pueden evitar usando el comando `findAndModify` que retorna un documento y lo actualiza en una sola operación. Para este ejemplo:

```

> ps = db.runCommand({"findAndModify" : "procesos",
... "query" : {"estado" : "READY"},
... "sort" : {"prioridad" : -1},
... "update" : {"$set" : {"estado" : "Ejecutándose"}})
{
  "ok" : 1,
  "valor" : {
    "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
    "prioridad" : 1,
    "estado" : "Preparado"
  }
}

```

Observar que el estado está aún a “Preparado” en el documento retornado puesto que el comando retorna por defecto el documento en el estado antes de ser modificado. Sin embargo si se consulta la colección se puede ver que el estado del documento ha sido actualizado a “Ejecutándose”:

```

> db.procesos.findOne({"_id" : ps.value._id})
{
  "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
  "prioridad" : 1,
  "estado" : "Ejecutándose"
}

```

Por tanto el programa queda de la siguiente manera:

```

ps = db.runCommand({"findAndModify" : "procesos",
"query" : {"estado" : "Preparado"},
"sort" : {"prioridad" : -1},
"update" : {"$set" : {"estado" : "Ejecutándose"}}}.value
do_something(ps)
db.procesos.update({"_id" : ps._id}, {"$set" : {"estado" : "Ejecutado"}})

```

El comando puede tener un “update” clave o bien un “remove” clave. En este último caso indica que el documento que coincide con las condiciones de búsqueda debería ser eliminado de la colección. Por ejemplo si se quiere eliminar el trabajo en vez de actualizar el estado, se podría programar de la siguiente manera:

```

ps = db.runCommand({"findAndModify" : "procesos",
"query" : {"estado" : "Preparado"},
"sort" : {"prioridad" : -1},

```

**"remove" : true}).value  
do\_something(ps)**

Los campos que tiene el comando son los siguientes:

- findAndModify: Es una cadena que representa el nombre de la colección.
- query: Es una consulta sobre un documento.
- sort(optativo): Criterio de ordenación de los resultados
- update: Un modificador de documento (actualización a realizar). Si no aparece esta opción entonces aparece la opción de borrar.
- remove: Es un booleano que especifica si el documento debe ser eliminado. Si no aparece esta opción entonces aparece la opción de actualizar.
- new: Es un booleano que especifica si el documento retornado debe ser el documento actualizado o bien el documento previo a la actualización (esta es la opción por defecto).
- fields(optativo): Los campos del documento a retornar.
- upsert: Es un booleano que especifica si o no la actualización se debe comportar como un "upsert" (la opción por defecto es a false).

Observar que o bien "update" o bien "remove" deben aparecer, pero no ambos. Si ningún documento encaja, el comando devolverá un error.

## **9. Configuración de la escritura.**

Se puede configurar el nivel de seguridad en cómo se realiza la escritura de datos antes de que la aplicación continúe ejecutándose. Por defecto las inserciones, actualizaciones y borrados esperan a que la base de datos responda, y los clientes lanzarán una excepción en caso de fallo.

Existen dos formas de llevar a cabo las escrituras en la base de datos:

- Con reconocimiento. Es la opción por defecto. Se genera una respuesta que dice si la base de datos proceso con éxito o no la escritura
- Sin reconocimiento. No se retorna ninguna respuesta, de manera que no se sabe si la base de datos proceso con éxito la escritura.

En general las aplicaciones suelen usar escrituras con reconocimiento pero en determinados casos tales como logs, carga de datos,..., puede que no se quiera esperar una respuesta de la base de datos y se use escritura sin reconocimiento.

Aunque la escritura sin reconocimiento no genera errores, no elimina la necesidad de controlarlos pues se producirá una excepción en caso contrario.

Un tipo de error que a veces no se gestiona cuando se usa escritura sin reconocimiento es la inserción de datos inválidos. Por ejemplo si se intenta insertar dos documentos con el mismo "\_id" entonces la shell lanzará una excepción.

```
> db.foo.insert({"_id" : 1})
```

```
> db.foo.insert({"_id" : 1})
```

```
E11000 duplicate key error index: test.foo.$_id_ dup key: { : 1.0 }
```

Sin embargo si se hubiera escrito el segundo documento con inserción sin reconocimiento entonces no se habría generado la excepción.

La shell usa escritura sin reconocimiento y después chequea que la última operación fue exitosa, de manera que si se hacen un conjunto de operaciones sobre una colección finalizando con una operación inválida, entonces la shell no se quejará:

```
> db.foo.insert<<"_id" : 1>>; db.foo.insert<<"_id" : 1>>; db.foo.count<>
1
> _
```

Sin embargo es posible forzar a la shell a chequearlo manualmente llamando al método `getLastError`, el cual chequea errores para la última operación:

```
> db.foo.insert<<"_id" : 1>>; db.foo.insert<<"_id" : 1>>; print<
... .. db.getLastError>>; db.foo.count<>
insertDocument :: caused by :: 11000 E11000 duplicate key error index: test.foo.
$_id_ dup key: { : 1.0 }
1
> _
```