# MONGODB

MongoDB (<http://www.mongodb.org>) es una base de datos orientada a documentos.

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

* Dentro de una instancia de MongoDB podemos tener 0 o más **bases de datos**.
* Una base de datos tendrá 0 o más **colecciones**. Una colección es muy similar a lo que entendemos como tabla dentro de un SGBD. 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**.
* 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.
* MongoDB soporta **índices**, para acelerar la búsqueda de datos.
* Al realizar cualquier consulta, se **devuelve un cursor**, con el cual podemos hacer cosas tales como contar, ordenar, limitar o saltar documentos.

Cuadrado

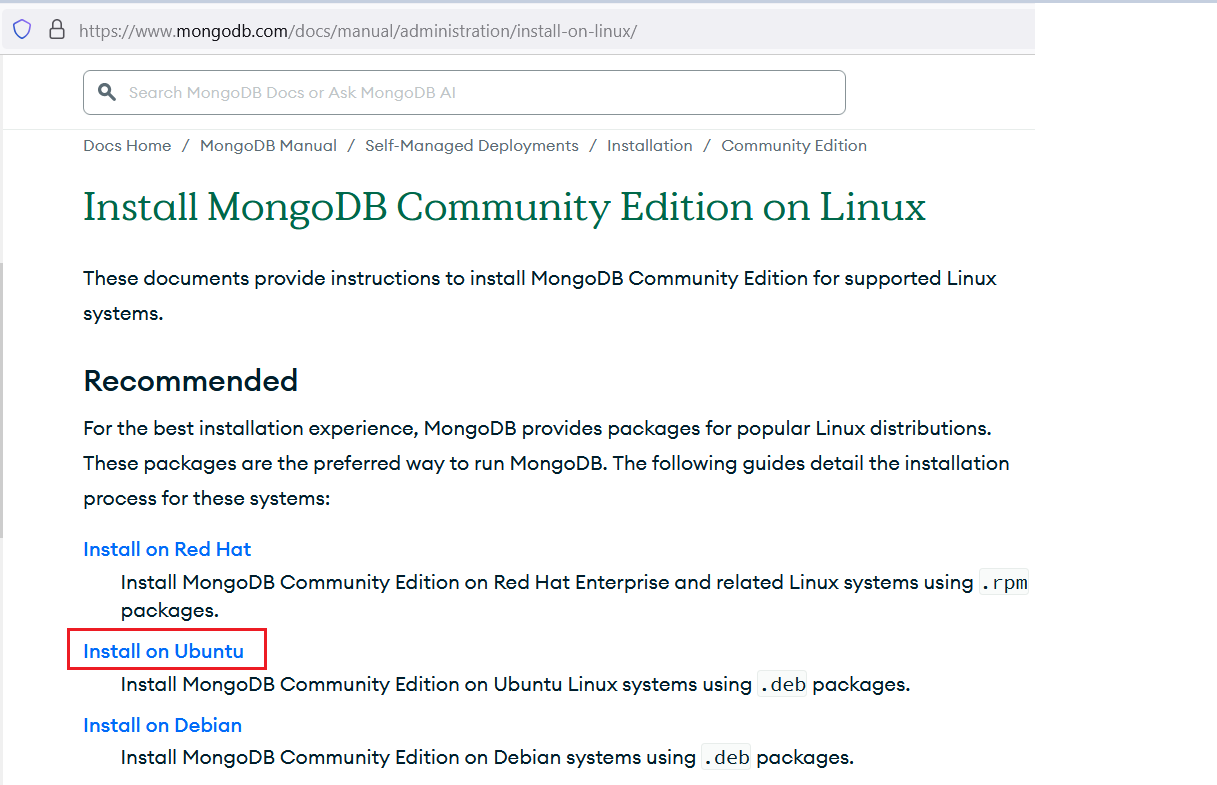
Descripción generada automáticamente con confianza media

Algunos ejemplos de uso de MongoDB son análisis en tiempo real, registro y búsqueda de texto completo, donde MongoDB tiene un buen rendimiento en cuanto a escalabilidad y concurrencia (<http://www.mongodb.com/use-cases>)

## INSTALAR MONGODB

Pasos a seguir para instalar MongoDB en Linux Mint Mate:

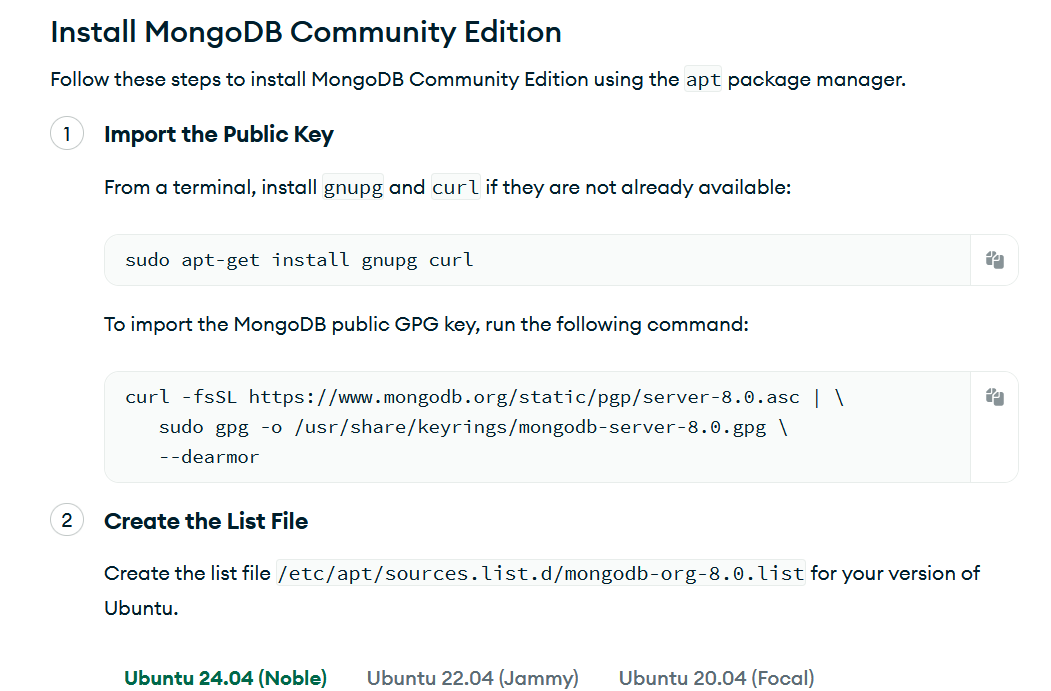
* Nos conectamos a la página oficial de Mongo DB: <https://www.mongodb.com/docs/manual/administration/install-on-linux/>
* Elegimos la distribución en función del sistema operativo de nuestra máquina:



* Para comprobar si cumplimos los requisitos ejecutamos el comando:

*cat /etc/lsb-release*

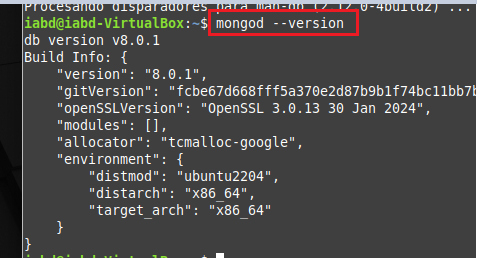
* Comenzamos la instalación siguiendo los pasos indicados:



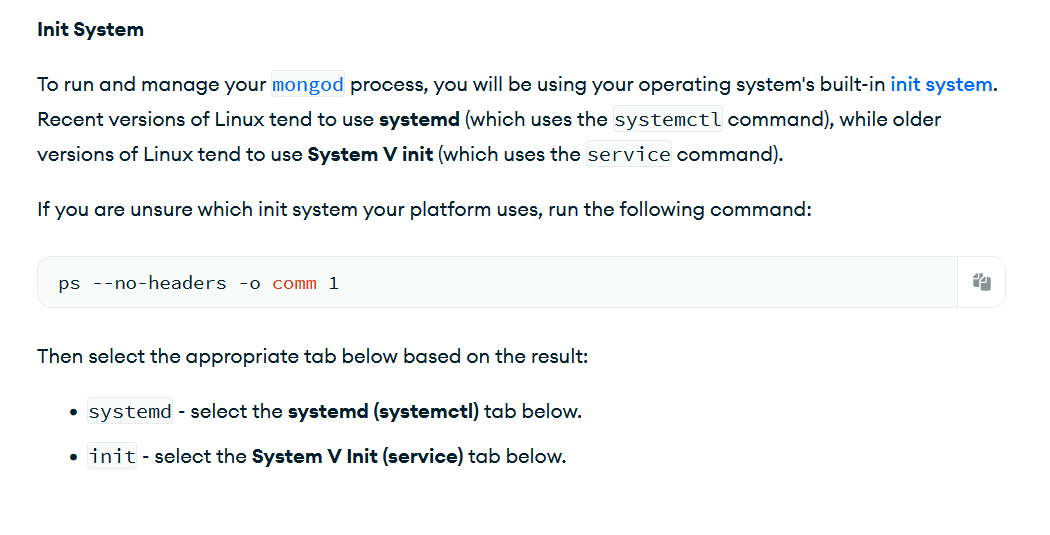
En el caso de tener Linux Mint 22 elegimos Ubuntu 22.04 (Jammy)

Y vamos llevando a cabo el resto de pasos.

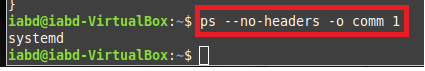
Comprobamos la versión de MongoDB instalada:



Una vez instalado en la web donde estamos nos indican los pasos para arrancar/interactuar con MongoDB. Los comandos para esta interacción dependen del sistema que tenemos instalado, como se explica en el siguiente pantallazo:



Para saber como interactuar con el servicio Mongo DB necesitamos saber el tipo de nuestro sistema:





Para un sistema ctl los comandos para interactuar con el servicio MongoDB son:

|  |  |
| --- | --- |
| **Acción** | **Comando** |
| Arrancar MongoDB | sudo systemctl start mongod |
| Comprobar si MongoDB está arrancado | sudo systemctl status mongod |
| Parar MongoDB | sudo systemctl stop mongod |
| Rearrancar MongoDB | sudo systemctl restart mongod |
| Arrancar mongosh | mongosh |

Una vez instalado MongoDB podemos usar MongoDB Compass para interactuar con la base de datos de forma gráfica y mongosh para interactuar con la base de datos desde la terminal.

### MONgODB COMPASS

Para instalar MongoDB Compass seguimos los pasos indicados en la página web:

<https://www.mongodb.com/docs/compass/current/install/>

Para Linux Mint elegiremos descargar los paquetes .deb.

Paso1: Descargamos mongoDB Compass

wget <https://downloads.mongodb.com/compass/mongodb-compass_1.43.5_amd64.deb>

Paso2: Instalamos mongoDB Compass

sudo apt install ./mongodb-compass\_1.43.5\_amd64.deb

Paso 2.1: Si nos da algún problema con los permisos:

Damos permisos sobre el archivo descargado:

sudo chmod 644 /home/iabd/mongodb-compass\_1.43.5\_amd64.deb

y ejecutamos:

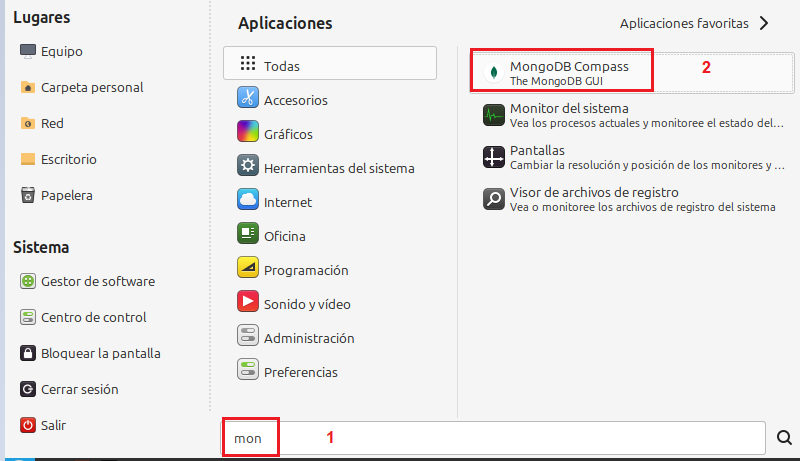
sudo dpkg -i /home/iabd/mongodb-compass\_1.43.5\_amd64.deb

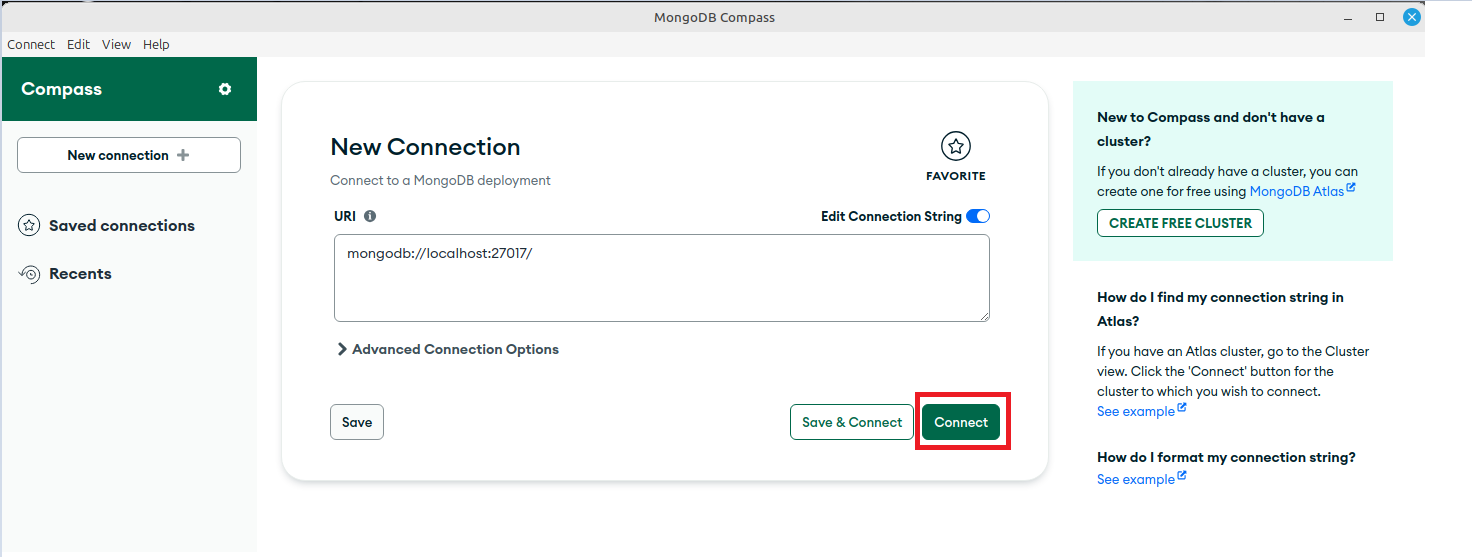
sudo apt-get install -f # This installs required compass dependencies

Paso3: Arrancamos mongoDB Compass

mongodb-compass

Como mongoDB Compass en una aplicación con interfaz gráfica la podemos arrancar desde la interfaz gráfica de nuestro equipo, así que no es necesario arrancarl por comando.





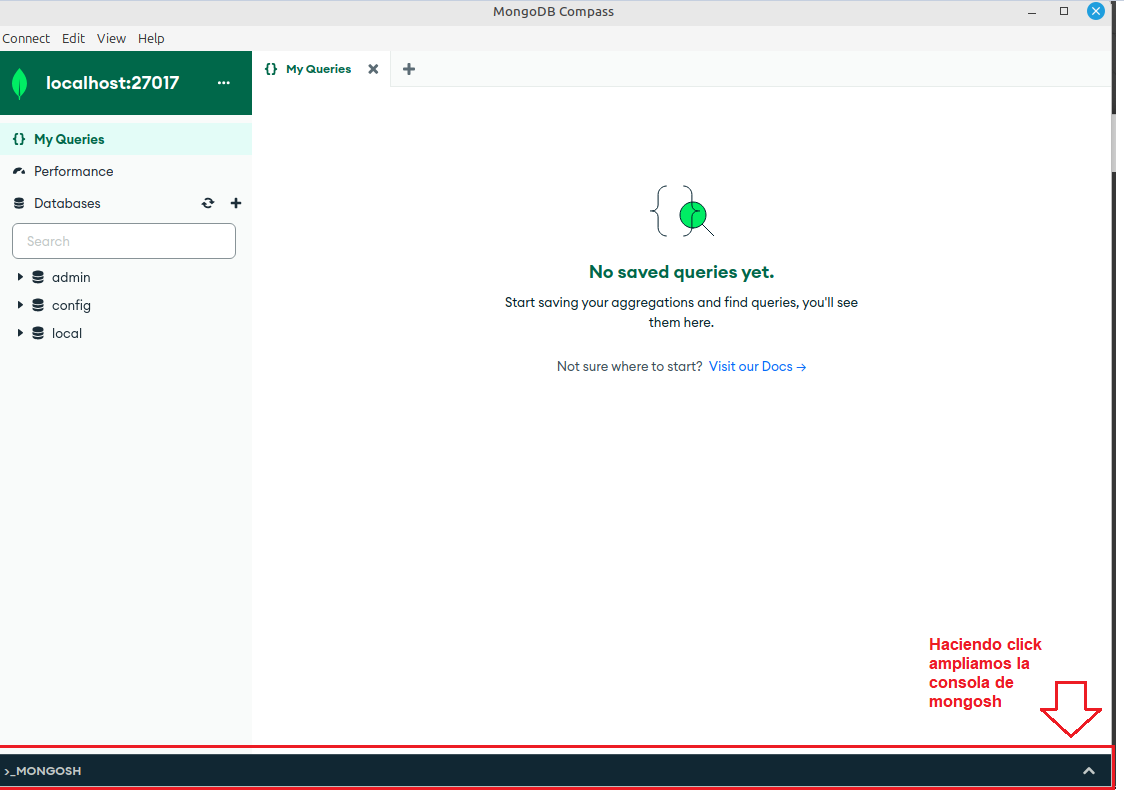
Hacemos clic en Connect

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Ya tenemos nuestra interfaz gráfica para trabajar con MongoDB.

En la parte inferior tenemos la posibilidad de trabajar con mongosh.



### MONGOSH

Desde MongoDB 5.0, **mongosh** es la shell predeterminada de MongoDB y, en la mayoría de los casos, se instala automáticamente cuando instalas MongoDB a través de los paquetes oficiales.

Para comprobar si mongosh está instalado:

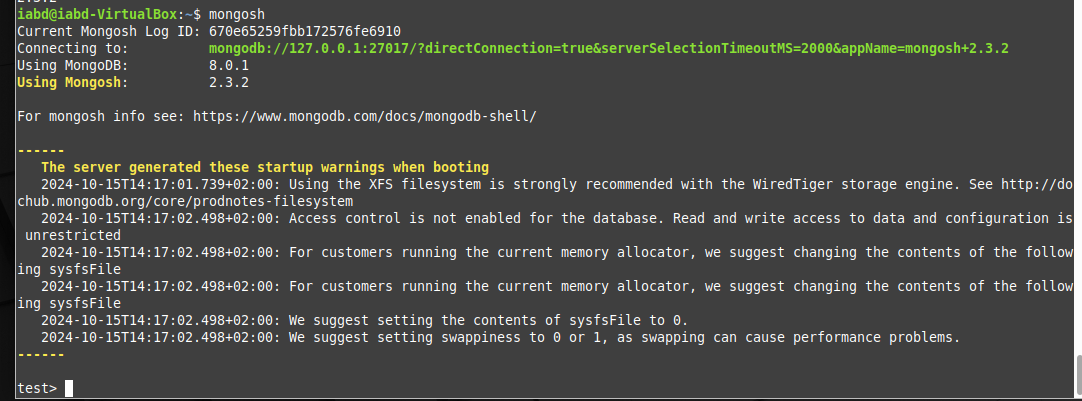
mongosh -- version

Si mogosh no está instalado para instalarlo nos conectamos a la página web y seguimos los pasos:

<https://www.mongodb.com/docs/mongodb-shell/install/>

Ya podemos abrir otra terminal y ejecutar:

mongosh



Ya podemos trabajar con MongoDB desde la terminal.

### MONGODB DATABASE TOOLS

MongoDB Database Tools es un conjunto de herramientas proporcionadas por MongoDB que facilitan la administración y el manejo de bases de datos MongoDB. Estas herramientas son especialmente útiles para realizar tareas como la importación y exportación de datos, la creación de copias de seguridad, la restauración de bases de datos y la administración de la configuración de MongoDB.

Suelen instalarse junto con MongoDB, pero también pueden instalarse por separado.

El comando para comprobar si está instalado:

dpkg -l | grep mongodb-database-tools

Si está instalado, veremos una línea que muestra el paquete y su versión.

Si no está instalado, lo instalamos:

Paso 1: Agregar el repositorio mongoDB (si no lo está)

wget -qO - https://www.mongodb.org/static/pgp/server-8.0.asc | sudo gpg --dearmor -o /usr/share/keyrings/mongodb-archive-keyring.gpg

echo "deb [signed-by=/usr/share/keyrings/mongodb-archive-keyring.gpg] https://repo.mongodb.org/apt/ubuntu jammy/mongodb-database-tools/8.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-database-tools.list

**Nota**: Ajusta jammy a tu versión de Ubuntu o Linux Mint si es necesario.

Paso2: Actualizar la lista de paquetes

sudo apt-get update

Paso3: Instalar MongoDB Database Tools

sudo apt-get install -y mongodb-database-tools

## BSON

Internamente, MongoDB almacena los documentos mediante BSON (Binary JSON).

BSON representa un superconjunto 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.

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: "Ainara",

apellidos: "Montoya",

fnac: new Date("Oct 31, 1977"),

hobbies: ["viajar", "leer", "programar"],

casado: true,

hijos: 2,

contacto: {

twitter: "@ainaramu",

email: "ainara.mu@icjardin.com"

},

fechaCreacion: new Timestamp()

}

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.

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}

{Edad: 18}

## CRUD (CREAR, LEER, ACTUALIZAR Y BORRAR)

En MongoDB, un documento es un registro compuesto por pares "campo: valor", es decir, en formato BSON.

{

nombre: "Hegoi",

apellido: "Ordoñana",

edad: 30,

grupos: ["Desarrolladores", "MongoDB"],

pais: {nombre: "United States", codigo: "USA"}

}

Las operaciones de lectura y escritura y las consultas que hacemos serán sobre una base de datos y sus colecciones:

# Visualizar las bases de datos existentes

>> show dbs

#Crear una base de datos si no existe y la seleccionarla para ser usada.

>> use usuariosdb

#Crear una colección

>> db.createCollection("usuarios")

#Obtener las colecciones de la base de datos actual

>> show collections

### DOCUMENTOS

#### INSERTAR

Para insertar documentos tenemos dos documentos

* **insertOne**: para insertar un documento
* **insterMany**: para insertar varios documentos

db.usuarios.insertOne(

{

\_id: 1,

nombre: "Hegoi",

apellido: "Ordoñana",

edad: 30,

grupos: ["Desarrolladores", "MongoDB"],

pais: { nombre: "United States", codigo: "USA"}

}

)

El índice único (clave primaria) para cada documento es guardado en el campo \_id. Si no necesitamos que el campo \_id tenga un formato especial, lo más sencillo es dejar que MongoDB lo genere automáticamente para cada documento insertado, omitiendo dicho campo a la hora de hacer el insert.

db.usuarios.insertMany([

{

nombre: "Enaitz",

apellido: "Uriarte",

edad: 30,

grupos: ["Sistemas", "MongoDB"],

pais: { nombre: "United States", codigo: "USA"}

},

{

name: "Miren",

apellido: "Ugarte",

edad: 33,

grupos: ["Ingenieros de datos", "MongoDB"],

pais: { nombre: "United States", codigo: "USA"}

}

])

#### ACTUALIZAR

Para actualizar datos, utilizaremos los siguientes comandos:

* **updateOne**: actualiza el primer documento que coincida con el criterio de búsqueda.
* **updateMany**: actualiza todos los documentos que coincidan con el criterio de búsqueda.

Cualquiera de los dos comandos consta de varias partes:

* La consulta para filtrar los documentos a modificar
* Los elementos que se modificarán ($set para añadir o modificar atributos y $unsert para borrar atributos).

db.usuarios.updateOne(

{ \_id:1 },

{

$set:{

nombre: "Hegoitxu",

campo\_nuevo: "Un nuevo campo para las observaciones."

},

$unset:{

pais: ""

}

}

)

db.usuarios.updateMany(

{ },

{ $set:

{

pais: { nombre: "Francia", codigo: "FRA"},

fecha\_alta: new Date(),

fecha\_baja: new Date()

}

}

)

db.usuarios.updateMany(

{ },

{ $unset: { fecha\_baja: "" } }

)

#### CONSULTAR

Para realizar búsquedas tenemos los métodos:

* **find():** devuelve todos los documentos que cumplen la condición
* **findOne()**: devuelve el primer documento que cumple la condición

La mejor forma de ver las consultas es con ejemplos.

#Obtener todos los documentos de una colección

db.usuarios.find()

#Obtener todos los documentos cuyo apellido sea Ordoñana

db.usuarios.find(

{ apellido: "Ordoñana" }

)

#Obtener todos los documentos cuya edad sea 30 ordenados inversamente por apellido

db.usuarios.find( { edad: 30 } ).sort( { apellido: -1 } )

#Obtener todos los usuarios cuyo país sea Francia

db.usuarios.find(

{ "pais.nombre": "Francia"}

)

#Obteber los usuarios que tengan más de 30 años:

db.usuarios.find(

{ edad: { $gt: 30}}

)

#Obtener los usuarios que tengan 30 años o menos:

db.usuarios.find(

{ edad: { $lte: 30}}

)

#Obtener los usuarios que estén en el grupo MongoDB:

db.usuarios.find(

{ grupos: "MongoDB"}

)

#Obtener los usuarios que estén en el grupo Desarrolladores o en el grupo Sistemas

db.usuarios.find({

$or: [

{ grupos: "Desarrolladores" },

{ grupos: "Sistemas" }

]

})

#Otra forma sería

db.usuarios.find({

grupos: {

$in: ["Desarrolladores", "Sistemas"]

}

})

#Obtener los usuarios que solo tengan el grupo MongoDB

db.usuarios.find( { grupos: ["MongoDB"] } )

#Otener los usuarios que tenga más de 30 años y estén en el grupo MongoDB

db.usuarios.find(

{

$and:

[

{ edad: {$gt:30}},

{ grupos: "MongoDB" }

]

}

)

#Obtener los usuarios que tengan más de 30 años o estén en el grupo MongoDB

db.usuarios.find(

{

$or:

[

{ edad: {$gt:30}},

{ grupos: "MongoDB" }

]

}

)

Podemos consultar todos los operadores que tenemos disponibles en la documentación de MongoDB <https://www.mongodb.com/docs/manual/reference/operator/query/#query-selectors>

A la hora de hacer consultas también los métodos:

* **countDocument():** para contar el número de documentos que cumplen una condición
* **distinct():** se utiliza para encontrar los valores únicos de un campo específico en una colección

#Contar los usuarios cuya edad es mayor que 30

db.usuarios.countDocuments({edad: {$gt:30}})

#Recuperar las diferentes edades de los usuarios de francia

db.usuarios.distinct("edad", {"pais.codigo":"FRA"})

#### PROYECCIONES

En MongoDB, las proyecciones se utilizan para recuperar datos de una colección con solo los campos que son de interés, en lugar de recuperar todos los campos de los documentos.

Las proyecciones se definen en el segundo argumento de la función find()

Podemos controlar qué campos se incluyen o excluyen en los resultados utilizando el valor 1 para incluir un campo y 0 para excluirlo. Si un campo se incluye, solo se mostrará ese campo en el resultado. Si un campo se excluye, se mostrarán todos los demás campos, excepto ese.

Un caso especial es el campo \_id; a menos que especifiques explícitamente { \_id: 0 } en las proyecciones, el campo \_id se incluirá en los resultados de forma predeterminada.

db.usuarios.find({},{nombre:1,\_id:0, grupos:1, "pais.nombre":1})

#### CURSORES

Al hacer una consulta 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 |
| cur.hasNext() | true/false para saber si quedan elementos |
| cur.next() | Pasa al siguiente documento |
| cur.limit(cantidad) | Restringe el número de resultados a cantidad |
| cur.sort({campo:1}) | Ordena los datos por campo: 1 ascendente o -1 descendente |
| cur.skip(cantidad) | Permite saltar cantidad elementos con el cursor |

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.

#Ordenar por apellido

db.usuarios.find().sort({apellido:1})

#### ELIMINAR

Para eliminar datos, utilizaremos los siguientes comandos:

* **deleteOne**: elimina el primer documento que coincida con el criterio de búsqueda.
* **deleteMany**: elimina todos los documentos que coincidan con el criterio de búsqueda.

Estos comandos reciben como parámetro la consulta que se utilizará para filtrar los documentos que se borrarán. Si no especificamos ninguna consulta, se eliminarán todos los datos de la colección.

#Borrar el documento cuyo \_id es 1

db.usuarios.deleteOne({\_id:1})

Otras operaciones relacionadas con el borrado de elementos están relacionadas con borrar una colección entera y borrar una base de datos.

#Borrar colección

db.usuarios.drop()

#Borrar base de datos

db.dropDatabase()

### IMPORTAR EXPORTAR DATOS

En MongoDB, podemos importar y exportar datos utilizando varias herramientas y métodos vamos:

**mongoexport** es una herramienta de línea de comandos proporcionada por MongoDB que permite exportar datos de una colección a un archivo en formato JSON o CSV.

mongoexport --db nombre\_de\_la\_base\_de\_datos --collection nombre\_de\_la\_coleccion --out archivo\_salida.json

**mongoimport** es una herramienta de línea de comandos proporcionada por MongoDB que permite importar datos desde un archivo JSON o CSV a una colección.

mongoimport --db nombre\_de\_la\_base\_de\_datos --collection nombre\_de\_la\_coleccion --file archivo\_entrada.json --jsonArray

El argumento jsonArray es necesario cuando se está importando un array de documentos (más de un documento),

## AGREGACIONES

Las agregaciones en MongoDB son una característica poderosa que permite realizar operaciones complejas de transformación, filtrado, agrupación y cálculo en documentos almacenados en una colección.

### ETAPAS

Una agregación es una secuencia de etapas, donde cada etapa realiza una operación específica y la salida de una etapa es la entrada de la siguiente:

db.coleccion.aggregate([op1, op2, ... opN])

El resultado del pipeline es un documento y por lo tanto está sujeto a la restricción de BSON, que limita su tamaño a 16MB.

Veamos un ejemplo con dos etapas

* Primero se eligen los elementos que vamos a agrupar mediante $match.
* Después el resultado saliente se agrupa con $group, y con $sum se calcula el total de cada grupo.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

No hay límite en el número de etapas, pero **el orden importa** (por ejemplo, hacer un $match al principio reduce la cantidad de datos desde el principio)

Cada etapa viene representada por un operador y las etapas (o los operadores) se pueden repetir.

### OPERADORES

Veamos algunos operadores:

|  |  |
| --- | --- |
| Operador | Descripción |
| $group | Para agrupar los datos. |
| $project | Proyección de campos, es decir, seleccionar las propiedades en las que estamos interesados. |
| $match | Filtrado de campos. |
| $sort | Ordenar |
| $skip | Saltar |
| $limit | Limitar los resultados |
| $sample | Seleccionar un subconjunto de datos, para hacer pruebas con menos datos |
| $unwind | Separa los datos que hay dentro de un array |
| $lookup | Relacionar documentos |

Para ver los primeros ejemplos vamos la colección productos de una tienda de electrónica con las características y precios de los mismos (ejecutar el contenido del archivo productos.js).

Un ejemplo de un producto sería:

*{*

*\_id: ObjectId("65381de61087c4cdbfccc578"),*

*nombre: 'iPad 16GB Wifi',*

*fabricante: 'Apple',*

*categoria: 'Tablets',*

*precio: 499*

*}*

#### $group

La fase group agrupa los documentos con el propósito de calcular valores agregados de una colección de documentos.

Hay que especificar un identificador (por ejemplo, un campo \_id) para el grupo y después los campos agregados que va a tener el nuevo documento.

Cuando referenciemos al valor de un campo lo haremos poniendo un $ delante del nombre del campo y entre comillas. Por ejemplo para referenciar al fabricante de un producto lo haremos mediante $fabricante.

db.productos.aggregate([

{ $group:

{

\_id: "$fabricante",

total: { $sum:1 }

}

}

])

*[*

*{ \_id: 'Samsung', total: 2 },*

*{ \_id: 'Amazon', total: 2 },*

*{ \_id: 'Google', total: 1 },*

*{ \_id: 'Sony', total: 1 },*

*{ \_id: 'Apple', total: 4 }*

*]*

Si lo que queremos es que el valor del identificador contenga un objeto, lo podemos hacer asociándolo como valor:

db.productos.aggregate([

{ $group:

{

\_id: { empresa: "$fabricante" },

total: { $sum:1 }

}

}

])

*[*

*{ \_id: { empresa: 'Google' }, total: 1 },*

*{ \_id: { empresa: 'Apple' }, total: 4 },*

*{ \_id: { empresa: 'Amazon' }, total: 2 },*

*{ \_id: { empresa: 'Samsung' }, total: 2 },*

*{ \_id: { empresa: 'Sony' }, total: 1 }*

*]*

También podemos agrupar más de un atributo, de tal modo que tengamos un \_id compuesto.

db.productos.aggregate([

{ $group:

{

\_id:

{

empresa: "$fabricante",

tipo: "$categoria"

},

total: {$sum:1}

}

}

])

*[*

*{ \_id: { empresa: 'Samsung', tipo: 'Smartphones' }, total: 1 },*

*{ \_id: { empresa: 'Apple', tipo: 'Tablets' }, total: 3 },*

*{ \_id: { empresa: 'Samsung', tipo: 'Tablets' }, total: 1 },*

*{ \_id: { empresa: 'Sony', tipo: 'Portátiles' }, total: 1 },*

*{ \_id: { empresa: 'Apple', tipo: 'Portátiles' }, total: 1 },*

*{ \_id: { empresa: 'Google', tipo: 'Tablets' }, total: 1 },*

*{ \_id: { empresa: 'Amazon', tipo: 'Tablets' }, total: 2 }*

*]*

Además de $sum existen más acumuladores entre los que podemos destacar:

|  |  |
| --- | --- |
| Nombre | Descripción |
| $sum | Devuelve la suma de todos los valores del grupo. Si el valor es 1 cuenta el número de elementos en el grupo. |
| $avg | Devuelve el promedio de todos los valores de un grupo |
| $addToSet | Devuelve un array con todos los valores únicos del campo seleccionado entre cada documento del grupo (sin repeticiones) |
| $max | Devuelve el mayor valor de un grupo |
| $min | Devuelve el menor valor de un grupo. |
| $first | Devuelve el primer valor del grupo. Se suele usar después de ordenar. |
| $last | Devuelve el último valor del grupo. Se suele usar después de ordenar. |

##### $sum

Para obtener el montante total de los productos agrupados por fabricante, haríamos:

db.productos.aggregate([{

$group: {

\_id: {

empresa: "$fabricante"

},

totalPrecio: {$sum:"$precio"}

}

}])

*[*

*{ \_id: { empresa: 'Google' }, totalPrecio: 199 },*

*{ \_id: { empresa: 'Sony' }, totalPrecio: 499 },*

*{ \_id: { empresa: 'Samsung' }, totalPrecio: 1014.98 },*

*{ \_id: { empresa: 'Apple' }, totalPrecio: 2296 },*

*{ \_id: { empresa: 'Amazon' }, totalPrecio: 328 }*

*]*

##### $avg

Para obtener el precio medio de los productos agrupados por categoría, haríamos:

db.productos.aggregate([{

$group: {

\_id: {

categoria:"$categoria"

},

precioMedio: {$avg:"$precio"}

}

}])

*[*

*{ \_id: { categoria: 'Tablets' }, precioMedio: 396.4271428571428 },*

*{ \_id: { categoria: 'Smartphones' }, precioMedio: 563.99 },*

*{ \_id: { categoria: 'Portátiles' }, precioMedio: 499 }*

*]*

##### $addToSet

Para obtener para cada empresa las categorías en las que tienen productos, haríamos:

db.productos.aggregate([{

$group: {

\_id: {

fabricante:"$fabricante"

},

categorias: {$addToSet:"$categoria"}

}

}])

*[*

*{ \_id: { fabricante: 'Sony' }, categorias: [ 'Portátiles' ] },*

*{ \_id: { fabricante: 'Apple' }, categorias: [ 'Tablets', 'Portátiles' ] },*

*{ \_id: { fabricante: 'Amazon' }, categorias: [ 'Tablets' ] },*

*{ \_id: { fabricante: 'Samsung' }, categorias: [ 'Smartphones', 'Tablets' ]},*

*{ \_id: { fabricante: 'Google' }, categorias: [ 'Tablets' ] }*

*]*

##### $max y $min

Para obtener el precio del producto más caro que tiene cada empresa haríamos:

db.productos.aggregate([{

$group: {

\_id: {

empresa:"$fabricante"

},

precioMaximo: {$max:"$precio"},

precioMinimo: {$min:"$precio"},

}

}])

*[*

*{ \_id: { empresa: 'Google' }, precioMaximo: 199, precioMinimo: 199 },*

*{ \_id: { empresa: 'Apple' }, precioMaximo: 699, precioMinimo: 499 },*

*{ \_id: { empresa: 'Amazon' }, precioMaximo: 199, precioMinimo: 129 },*

*{ \_id: { empresa: 'Samsung' }, precioMaximo: 563.99, precioMinimo: 450.99},*

*{ \_id: { empresa: 'Sony' }, precioMaximo: 499, precioMinimo: 499 }*

*]*

##### $first y $last

Para obtener para cada empresa, cual es el tipo de producto que más tiene y la cantidad de dicho tipo haríamos:

db.productos.aggregate([

{$group:

{

\_id: {

empresa: "$fabricante",

tipo: "$categoria"

},

total: {$sum:1}

}

},

{$sort: {"total":-1}},

{$group:

{

\_id:"$\_id.empresa",

producto: {$first: "$\_id.tipo"},

cantidad: {$first:"$total"}

}

}

])

*[*

*{ \_id: 'Apple', producto: 'Tablets', cantidad: 3 },*

*{ \_id: 'Amazon', producto: 'Tablets', cantidad: 2 },*

*{ \_id: 'Google', producto: 'Tablets', cantidad: 1 },*

*{ \_id: 'Sony', producto: 'Portátiles', cantidad: 1 },*

*{ \_id: 'Samsung', producto: 'Tablets', cantidad: 1 }*

*]*

##### Doble $group

Para obtener el precio medio de los precios medios de los tipos de producto por empresa haríamos:

db.productos.aggregate([

{$group:

{

\_id: {

empresa:"$fabricante",

categoria:"$categoria"

},

precioMedio: {$avg:"$precio"}

}

},

{$group:

{

\_id: "$\_id.empresa",

precioMedio: {$avg: "$precioMedio"}

}

}

])

#### $project

Si queremos realizar una proyección sobre el conjunto de resultados y quedarnos con un subconjunto de los campos usaremos el operador $project.

La proyección dentro del framework de agregación es mucho más potente que dentro de las consultas normales y se emplea para:

* renombrar campos.
* introducir campos calculados en el documento resultante mediante $add, $substract, $multiply, $divide o $mod
* transformar campos a mayúsculas ($toUpper) o minúsculas ($toLower), concatenar campos ($concat) u obtener subcadenas ($substr).
* ...

db.productos.aggregate([

{$project:

{

\_id: 0,

"empresa": { "$toUpper": "$fabricante" },

"detalles": {

"categoria": "$categoria",

"precio": { "$multiply": ["$precio", 1.1] }

},

"elemento": "$nombre"

}

}

])

*[*

*{*

*empresa: 'APPLE',*

*detalles: { categoria: 'Tablets', precio: 548.9000000000001 },*

*elemento: 'iPad 16GB Wifi'*

*},*

*{*

*empresa: 'APPLE',*

*detalles: { categoria: 'Tablets', precio: 658.9000000000001 },*

*elemento: 'iPad 32GB Wifi'*

*},*

*{*

*empresa: 'APPLE',*

*detalles: { categoria: 'Tablets', precio: 768.9000000000001 },*

*elemento: 'iPad 64GB Wifi'*

*},*

*…*

*]*

#### $match

El operador $match se utiliza principalmente para filtrar los documentos que pasarán a la siguiente etapa del pipeline o a la salida final.

Para seleccionar sólo las tablets haríamos:

db.productos.aggregate([{$match:{categoria:"Tablets"}}])

Para obtener la cantidad de Tablets por empresa de menos de 500 euros haríamos:

db.productos.aggregate([

{$match:

{

categoria:"Tablets",

precio: {$lt: 500}

}

},

{$group:

{

\_id: {"empresa":"$fabricante"},

cantidad: {$sum:1}

}

}

])

[

{ \_id: { empresa: 'Google' }, cantidad: 1 },

{ \_id: { empresa: 'Samsung' }, cantidad: 1 },

{ \_id: { empresa: 'Apple' }, cantidad: 1 },

{ \_id: { empresa: 'Amazon' }, cantidad: 2 }

]

#### $sort

El operador $sort ordena los documentos recibidos por el campo, y el orden indicado por la expresión indicada al pipeline.

Para ordenar los productos por precio descendentemente haríamos:

db.productos.aggregate({$sort:{precio:-1}})

El operador $sort ordena los datos en memoria, por lo que hay que tener cuidado con el tamaño de los datos. Por ello, se emplea en las últimas fases del pipeline, cuando el conjunto de resultados es el menor posible.

Si retomamos el ejemplo anterior, y ordenamos los datos por el precio total para cada fabricante tenemos:

db.productos.aggregate([

{$match:{categoria:"Tablets"}},

{$group:

{

\_id: {empresa:"$fabricante"},

totalPrecio: {$sum:"$precio"}

}

},

{$sort:{totalPrecio:-1}}

])

*[*

*{ \_id: { empresa: 'Apple' }, totalPrecio: 1797 },*

*{ \_id: { empresa: 'Samsung' }, totalPrecio: 450.99 },*

*{ \_id: { empresa: 'Amazon' }, totalPrecio: 328 },*

*{ \_id: { empresa: 'Google' }, totalPrecio: 199 }*

*]*

#### $skip y $limit

El operador $limit únicamente limita el número de documentos que pasan a través del pipeline.

db.productos.aggregate([{$limit:3}])

El operador $skip, salta un número determinado de documentos:

db.productos.aggregate([{$skip:3}])

El orden en el que empleemos estos operadores importa:

db.productos.aggregate([{$skip:2}, {$limit:3}])

*[*

*{*

*\_id: ObjectId("65381de61087c4cdbfccc57a"),*

*nombre: 'iPad 64GB Wifi',*

*categoria: 'Tablets',*

*fabricante: 'Apple',*

*precio: 699*

*},*

*{*

*\_id: ObjectId("65381de61087c4cdbfccc57b"),*

*nombre: 'Galaxy S3',*

*categoria: 'Smartphones',*

*fabricante: 'Samsung',*

*precio: 563.99*

*},*

*{*

*\_id: ObjectId("65381de61087c4cdbfccc57c"),*

*nombre: 'Galaxy Tab 10',*

*categoria: 'Tablets',*

*fabricante: 'Samsung',*

*precio: 450.99*

*}*

*]*

db.productos.aggregate([{$limit:3}, {$skip:2}])

*[*

*{*

*\_id: ObjectId("65381de61087c4cdbfccc57a"),*

*nombre: 'iPad 64GB Wifi',*

*categoria: 'Tablets',*

*fabricante: 'Apple',*

*precio: 699*

*}*

*]*

#### $sample

Si tenemos un dataset muy grande, y queremos probar las consultas con un número reducido de documentos, podemos emplear el operador $sample y reducir la cantidad de documentos de manera aleatoria:

db.productos.aggregate([ { $sample: { size: 3 } } ])

#### $unwind

El operador $unwind se utiliza sólo con arrays. Al usarlo con un campo array de tamaño N en un documento, lo transforma en N documentos con el campo.

Vamos a usar la colección enlaces:

db.enlaces.insertOne({

titulo: 'www.google.es',

tags: ['blog', 'mapas', 'buscador', 'drive', 'blog', 'mapas']

})

Podemos observar como el campo tags contiene 6 valores dentro del array (con un valor repetido). A continuación vamos a desenrollar el array:

db.enlaces.aggregate([

{$match:{titulo:"www.google.es"}},

{$unwind:"$tags"}

])

*[*

*{ \_id: ObjectId("6538d8e8c9588eb7e530062a"), titulo: 'www.google.es', tags: 'blog' },*

*{ \_id: ObjectId("6538d8e8c9588eb7e530062a"), titulo: 'www.google.es', tags: 'mapas' },*

*{ \_id: ObjectId("6538d8e8c9588eb7e530062a"), titulo: 'www.google.es', tags: 'buscador' },*

*{ \_id: ObjectId("6538d8e8c9588eb7e530062a"), titulo: 'www.google.es', tags: 'drive' },*

*{ \_id: ObjectId("6538d8e8c9588eb7e530062a"), titulo: 'www.google.es', tags: 'blog' },*

*{ \_id: ObjectId("6538d8e8c9588eb7e530062a"), titulo: 'www.google.es', tags: 'mapas' }*

*]*

De este modo, podemos realizar consultas que sumen/cuenten los elementos del array.

Si queremos obtener las 3 etiquetas que más aparecen en todos los enlaces haríamos:

db.enlaces.aggregate([

{$unwind:"$tags"},

{$group:

{

\_id:"$tags",

total:{$sum:1}

}

},

{$sort:{"total":-1}},

{$limit: 3}

])

[

{ \_id: 'mapas', total: 2 }

{ \_id: 'blog', total: 2 }

{ \_id: 'drive', total: 1 }

]

##### Doble $unwind

Si trabajamos con documentos que tienen varios arrays, podemos necesitar desenrollar los dos arrays. Al hacer un doble unwind se crea un producto cartesiano entre los elementos de los 2 arrays.

Supongamos que tenemos los datos del siguiente inventario de ropa:

db.inventario.insertOne({'nombre':"Camiseta",'tallas':["S", "M", "L"], 'colores':['azul', 'blanco', 'naranja', 'rojo']})

db.inventario.insertOne({'nombre':"Jersey",'tallas':["S", "M", "L", "XL"], 'colores':['azul', 'negro', 'naranja', 'rojo']})

db.inventario.insertOne({'nombre':"Pantalones",'tallas':["32x32", "32x30", "36x32"], 'colores':['azul', 'blanco', 'naranja', 'negro']})

Para obtener un listado de cantidad de pares talla/color haríamos:

db.inventario.aggregate([

{$unwind: "$tallas"},

{$unwind: "$colores"},

{$group:

{

\_id: {talla: '$tallas', color: '$colores'},

total : {$sum: 1}

}

}

])

*[*

*{ \_id: { talla: 'S', color: 'azul' }, total: 2 },*

*{ \_id: { talla: '32x30', color: 'naranja' }, total: 1 },*

*{ \_id: { talla: 'L', color: 'rojo' }, total: 2 },*

*…*

*]*

#### $lookup

Si necesitamos unir los datos de dos colecciones, emplearemos el operador $lookup.

El operador $lookup utiliza cuatro parámetros:

* from: colección con la que se realiza el join.
* localField: campo de la colección origen.
* foreignField: campo en la colección destino que permite.
* as: nombre del array que contendrá los documentos enlazados.

Para los siguientes ejemplos vamos a usar las colecciones zips y states:

* zips: está en la base datos sample\_training y para cargarla tenemos que cargar el archivo sampledata.archive ( Tenemos que tener instalado mongodb database tools que suele instalarse con mongoDB y en la terminal ejecutamos:

curl https://atlas-education.s3.amazonaws.com/sampledata.archive -o sampledata.archive

mongorestore --archive=sampledata.archive)

* states: ejecutar el contenido del archivo states.js

Veamos algún ejemplo del contenido de estas colecciones:

*{*

*\_id: ObjectId("5c8eccc1caa187d17ca6ed18"),*

*city: 'ACMAR',*

*zip: '35004',*

*loc: { y: 33.584132, x: 86.51557 },*

*pop: 6055,*

*state: 'AL'*

*}*

*{*

*"name": "Alabama",*

*"abbreviation": "AL"*

*}*

Primero vamos a recuperar los tres estados más poblados:

db.zips.aggregate([

{$group:

{

\_id: "$state",

totalPoblacion: {$sum:"$pop"}

}

},

{$sort:{"totalPoblacion":-1}},

{$limit: 3}

])

*[*

*{ \_id: 'CA', totalPoblacion: 29760021 },*

*{ \_id: 'NY', totalPoblacion: 17990455 },*

*{ \_id: 'TX', totalPoblacion: 16986510 }*

*]*

Si ahora queremos recuperar el nombre de esos tres estados, añadimos una nueva etapa:

db.zips.aggregate([

{$group:

{

\_id: "$state",

totalPoblacion: {$sum:"$pop"}

}

},

{$sort:{totalPoblacion:-1}} ,

{$limit: 3},

{$lookup:

{

from: "states",

localField: "\_id",

foreignField: "abbreviation",

as: "estados"

}

}

])

Y ahora obtenemos para cada documento, un array con los documentos que coinciden (en este caso es una relación 1:1, y por eso cada array sólo contiene un elemento):

*[*

*{*

*\_id: 'CA',*

*totalPoblacion: 29760021,*

*estados: [*

*{*

*\_id: ObjectId("6538e5f6f32c013201689fdd"),*

*name: 'California',*

*abbreviation: 'CA'*

*}*

*]*

*},*

*{*

*\_id: 'NY',*

*totalPoblacion: 17990455,*

*estados: [*

*{*

*\_id: ObjectId("6538e5f6f32c013201689ffc"),*

*name: 'New York',*

*abbreviation: 'NY'*

*}*

*]*

*},*

*{*

*\_id: 'TX',*

*totalPoblacion: 16986510,*

*estados: [*

*{*

*\_id: ObjectId("6538e5f6f32c01320168a00a"),*

*name: 'Texas',*

*abbreviation: 'TX'*

*}*

*]*

*}*

*]*

Como la relación siempre va a provocar la creación de un array, mediante $unwind, lo podemos deshacer:

db.zips.aggregate([

{$group:

{

\_id: "$state",

totalPoblacion: {$sum:"$pop"}

}

},

{$sort:{totalPoblacion:-1}} ,

{$limit: 3},

{$lookup:

{

from: "states",

localField: "\_id",

foreignField: "abbreviation",

as: "estados"

}

},

{$unwind:"$estados"}

])

*[*

*{*

*\_id: 'CA',*

*totalPoblacion: 29760021,*

*estados: {*

*\_id: ObjectId("6538e5f6f32c013201689fdd"),*

*name: 'California',*

*abbreviation: 'CA'*

*}*

*},*

*{*

*\_id: 'NY',*

*totalPoblacion: 17990455,*

*estados: {*

*\_id: ObjectId("6538e5f6f32c013201689ffc"),*

*name: 'New York',*

*abbreviation: 'NY'*

*}*

*},*

*{*

*\_id: 'TX',*

*totalPoblacion: 16986510,*

*estados: {*

*\_id: ObjectId("6538e5f6f32c01320168a00a"),*

*name: 'Texas',*

*abbreviation: 'TX'*

*}*

*}*

*]*

Así pues, para finalmente obtener el nombre de cada estado, mediante $project recuperamos el campo name:

db.zips.aggregate([

{$group:

{

\_id: "$state",

totalPoblacion: {$sum:"$pop"}

}

},

{$sort:{ totalPoblacion:-1}} ,

{$limit: 3},

{$lookup:

{

from: "states",

localField: "\_id",

foreignField: "abbreviation",

as: "estados"

}

},

{$unwind:"$estados"},

{$project:

{

estado: "$estados.name",

poblacion: "$totalPoblacion"

}

}

])

*[*

*{ \_id: 'CA', estado: 'California', poblacion: 29760021 }*

*{ \_id: 'NY', estado: 'New York', poblacion: 17990455 }*

*{ \_id: 'TX', estado: 'Texas', poblacion: 16986510 }*

*]*

#### MAS INFORMACIÓN

Más información: <https://www.mongodb.com/docs/manual/reference/operator/aggregation-pipeline/>

### PERSISTIENDO LOS RESULTADOS

Una vez hemos realizado nuestras consultas mediante el framework de agregación, es muy posible que queramos almacenar el resultado en una nueva colección para poder volver a consultar el resultado sin necesidad de ejecutar todas las fases.

Para ello, podemos emplear los operadores:

* **$out** recoge los documentos de una agregación y los persiste en una colección, sobrescribiendo los datos existentes.
* **$merge** similar a $out, pero permite añadir el resultado a la misma colección.

Vamos a crear una nueva colección con la población total de todos dos estados, y la vamos a almacenar en una nueva colección denominada states\_population:

db.zips.aggregate([

{$group:

{

\_id: "$state",

totalPoblacion: {$sum:"$pop"}

}

},

{$sort:{totalPoblacion:-1}} ,

{$limit: 3},

{$lookup:

{

from: "states",

localField: "\_id",

foreignField: "abbreviation",

as: "estados"

}

},

{$unwind:"$estados"},

{$project:

{

estado: "$estados.name",

poblacion: "$totalPoblacion"

}

},

{$out: "states\_population"}

])

Tras su ejecución, podemos recuperar los datos:

db.states\_population.findOne()

*{ \_id: 'WY', estado: 'Wyoming', poblacion: 453588 }*

### AGREGACIONES CON COMPASS

MongoDB Compass nos ofrece la herramienta para crear, borrar y reorganizar fácilmente etapas en una agregación.

Vamos a practicar con la colección de zips:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Y vamos a reproducir la consulta que acabamos de realizar:

db.zips.aggregate ([

{$group:

{

\_id: "$state",

totalPoblacion: {$sum:"$pop"}

}

},

{$sort:{totalPoblacion:-1}} ,

{$limit: 3},

{$lookup:

{

from: "states",

localField: "\_id",

foreignField: "abbreviation",

as: "estados"

}

},

{$unwind:"$estados"},

{$project:

{

estado: "$estados.name",

poblacion: "$totalPoblacion"

}

}

])

Para ello, pulsamos sobre el botón de Add Stage, seleccionamos el operador $group y escribimos la expresión de agrupación. Al hacerlo, en el panel anexo aparecerán algunos documentos de muestra con el resultado de ejecutar dicha etapa:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Paso a paso, iremos añadiendo el resto de las etapas hasta tenerlas todas, pudiendo ver los datos que va generando cada etapa:

Imagen de la pantalla de un computador

Descripción generada automáticamente con confianza baja

Una vez tenemos nuestra agregación, podemos obtener una versión en Python, Java, C# o Node.js, mediante la opción Export-to-Language.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

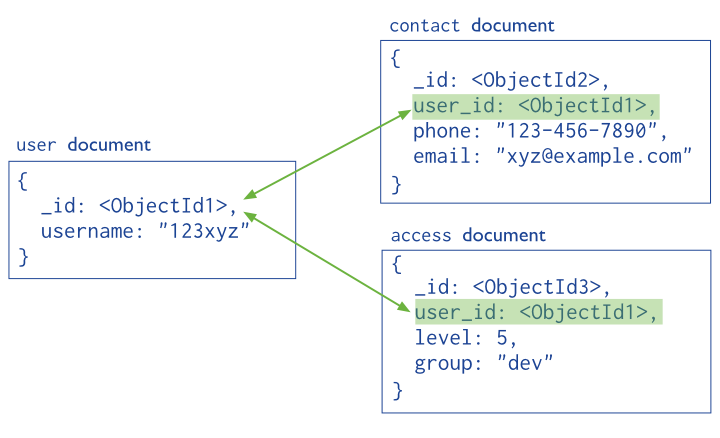
## RELACIONES

Las aplicaciones que emplean MongoDB utilizan dos técnicas para relacionar documentos:

* Crear referencias
* Embeber o incrustar documentos

### REFERENCIAS MANUALES

De manera similar a una base de datos relacional, se almacena el campo \_id de un documento en otro documento a modo de clave ajena. De este modo, la aplicación realiza una segunda consulta para obtener los datos relacionados.



Por ejemplo, si nos basamos en el gráfico anterior, podemos conseguir referenciar estos objetos del siguiente modo:

var idUsuario = ObjectId();

db.usuario.insertOne({

\_id: idUsuario,

nombre: "123xyz"

});

db.contacto.insertOne({

usuario\_id: idUsuario,

telefono: "123 456 7890",

email: "xyz@ejemplo.com"

});

Para relacionar los dos documentos, haremos uso de la operación $lookup para hacer el join, o haremos una segunda consulta para la segunda colección.

db.usuario.aggregate([ {

$lookup: {

from: "contacto",

localField: "\_id",

foreignField: "usuario\_id",

as: "contacto\_data"

}

} ])

Y como resultado obtenemos un documento con el usuario y la información del contacto dentro de un array embebido (aunque en este ejemplo sólo tenemos un contacto para el usuario)

*[*

*{*

*\_id: ObjectId("6536b48cb97061fa4b0cd4f6"),*

*nombre: '123xyz',*

*contacto\_data: [*

*{*

*\_id: ObjectId("6536b48db97061fa4b0cd4f7"),*

*usuario\_id: ObjectId("6536b48cb97061fa4b0cd4f6"),*

*telefono: '123 456 7890',*

*email: 'xyz@ejemplo.com'*

*}*

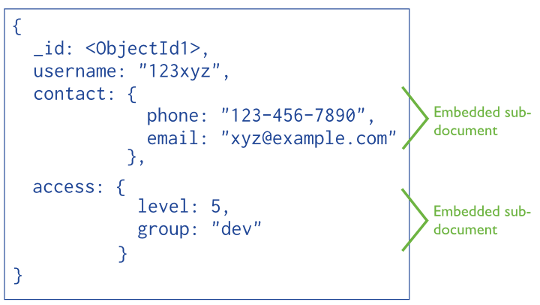
*]*

*}*

*]*

### DATOS EMBEBIDOS

En cambio, si dentro de un documento almacenamos los datos mediante subdocumentos, ya sea dentro de un atributo o un array, podremos obtener todos los datos mediante un único acceso, sin necesidad de claves ajenas ni comprobaciones de integridad referencial.



Los datos embebidos/incrustados ofrecen mejor rendimiento al permitir obtener los datos mediante una única operación, así como modificar datos relacionados en una sola operación atómica de escritura (sin necesidad de transacciones).

Un aspecto a tener en cuenta es que un documento BSON puede contener un máximo de 16MB.

### RELACIONES 1:1

Cuando existe una relación 1:1, como pueda ser entre persona y dirección, en general, hay que embeber un documento dentro del otro, como parte de un atributo:

db.persona.insertOne({

nombre: "Ainara",

edad: 25,

dirección: {

calle: "Secreta",

ciudad: "Elx"

}

})

La principal ventaja de este planteamiento es que mediante una única consulta podemos obtener tanto los detalles del usuario como su dirección.

### RELACIONES 1:N

Vamos a distinguir dos tipos:

#### 1 A POCOS (1:F)

**1 a pocos (1:F)**, como por ejemplo, dentro de un blog, la relación entre mensaje y comentario. En este caso, la mejor solución es crear un array dentro de la entidad mensaje. De este modo, mensaje contiene un array de comentarios:

db.mensaje.insertOne({

titulo: "La broma asesina",

url: "http://es.wikipedia.org/wiki/Batman:\_The\_Killing\_Joke",

texto: "La dualidad de Batman y Joker",

comentarios: [

{

autor: "Bruce Wayne",

fecha: ISODate("2022-10-11T09:31:32Z"),

comentario: "A mí me encantó"

},

{

autor: "Bruno Díaz",

fecha: ISODate("2022-10-11T10:07:28Z"),

comentario: "El mejor"

}

]

})

Hay que tener siempre en mente la restricción de los 16 MB de BSON.

#### 1 A MUCHOS (1:N)

**1 a muchos (1:N)**, como puede ser entre editorial y libro. Para este tipo de relación es mejor usar referencias entre los documentos colocando la referencia en el lado del muchos:

db.editorial.insertOne({

\_id: 1,

nombre: "O'Reilly",

pais: "EE.UU."

})

db.libro.insertMany([

{

\_id: 1234,

titulo: "MongoDB: The Definitive Guide",

autor: [ "Kristina Chodorow", "Mike Dirolf" ],

numPaginas: 216,

editorial\_id: 1,

},

{

\_id: 1235,

titulo: "50 Tips and Tricks for MongoDB Developer",

autor: "Kristina Chodorow",

numPaginas: 68,

editorial\_id: 1,

}

])

Si cada vez que recuperamos un libro queremos tener el nombre de la editorial y con una sola consulta recuperar todos los datos, en vez poner la referencia a la editorial, podemos embeber toda la información), a costa de que un futuro cambio en el nombre de la editorial conlleve modificar muchos libros:

db.libro2.insertMany([

{

\_id: 1234,

titulo: "MongoDB: The Definitive Guide",

autor: [ "Kristina Chodorow", "Mike Dirolf" ],

numPaginas: 216,

editorial: {

\_id: 1,

nombre: "O'Reilly",

pais: "EE.UU."

}

},

{

\_id: 1235,

titulo: "50 Tips and Tricks for MongoDB Developer",

autor: "Kristina Chodorow",

numPaginas: 68,

editorial: {

\_id: 1,

nombre: "O'Reilly",

pais: "EE.UU."

}

}

])

Un caso particular en las relaciones uno a muchos que se traducen en documentos embebidos es cuando la información que nos interesa tiene un valor concreto en un momento determinado. Por ejemplo, dentro de un pedido, el precio de los productos debe embeberse, ya que si en un futuro se modifica el precio de un producto determinado debido a una oferta, el pedido realizado no debe modificar su precio total.

### RELACIONES N:M

Supongamos que tenemos libros modelados de la siguiente manera:

db.libro.insertOne({

\_id: 1,

titulo: "La historia interminable",

anyo: 1979

})

Y autores con la siguiente estructura:

db.autor.insertOne({

\_id: 1,

nombre: "Michael Ende",

pais: "Alemania"

})

Podemos resolver esta relación de varias maneras:

**OPCIÓN 1**

Siguiendo un enfoque relacional, empleando un documento como la entidad que agrupa con referencias manuales a los dos documentos.

db.libro\_autor.insertOne({

\_id: 1,

autor\_id: 1,

libro\_id: 1

})

Este enfoque se desaconseja porque necesita acceder a tres colecciones para obtener toda la información.

**OPCIÓN 2**

Mediante 2 documentos, cada uno con un array que contenga los identificadores del otro documento. Hay que tener cuidado porque podemos tener problemas de inconsistencia de datos si no actualizamos correctamente.

db.libro\_con\_autores.insertMany([

{

\_id: 1,

titulo: "La historia interminable",

anyo: 1979,

autores: [1]

},

{

\_id: 2,

titulo: "Momo",

anyo: 1973,

autores: [1]

}

])

db.autor\_con\_libros.insertOne({

\_id: 1,

nombre: "Michael Ende",

pais: "Alemania",

libros: [1, 2]

})

**OPCIÓN 3**

Embeber un documento dentro de otro:

db.libros\_con\_autores\_embebidos.insertMany([

{

\_id: 1,

titulo: "La historia interminable",

anyo: 1979,

autores: [{ nombre:"Michael Ende", pais:"Alemania" }]

},

{

\_id: 2,

titulo: "Momo",

anyo: 1973,

autores: [{ nombre:"Michael Ende", pais:"Alemania" }]

}

])

En principio este enfoque puede dar problemas porque el documento puede crecer mucho y provocar anomalías de modificaciones donde la información no es consistente.

### RENDIMIENTO E INTEGRIDAD

A modo de resumen, embeber documentos ofrece un mejor rendimiento que referenciar, ya que con una única operación (ya sea una lectura o una escritura) podemos acceder a varios documentos.

Pero, en general, arrays con una alta cardinalidad son una clara pista para no embeber.

Más información en <https://www.mongodb.com/docs/manual/applications/data-models-tree-structures/>

## INDEXACIÓN

El mecanismo de indexación que utiliza MongoDB hace que los datos se almacenen en forma de Árbol-B, manteniendo cada uno de los nodos de este balanceado. Esto incrementa la velocidad a la hora de buscar y también a la hora de devolver resultados ya ordenados.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

Los índices en MongoDB son una parte esencial de la optimización de consultas y el rendimiento de tus bases de datos.

Un índice en MongoDB es una estructura de datos que mejora la velocidad de las consultas al permitir un acceso más rápido a los datos en una colección.

Algunos de los diferentes tipos de índices son:

* **Índice por defecto**: todas las colecciones dispondrán de un índice individual por el campo \_id (identificador único del documento dentro de la colección).
* **Índices individuales**: índices que se configuran sobre un único atributo, pudiendo tener orden ascendente o descendente.
* **Índices compuestos**: índices que permiten acceder a los datos a través de dos atributos simultáneamente.
* **Índices geoespaciales**: permiten la consulta eficiente a bases de datos espaciales, pudiendo existir únicamente un índice de este tipo por colección.

### CREACIÓN DE ÍNDICES

Puedes crear índices en MongoDB usando el método createIndex() en una colección.

* Para crear un índice ascendente en el campo especificado (-1 para que sea descendente):

db.miColeccion.createIndex({ campo: 1 })

* Para crear un índice compuesto en los campos "campo1" y "campo2":

db.miColeccion.createIndex({ campo1: 1, campo2: -1 })

* Para crear un índice único (para garantizar que no haya duplicados en un campo específico):

db.miColeccion.createIndex({ campoUnico: 1 }, { unique: true })

Podemos ver los índices existentes en una colección utilizando getIndexes().

db.miColeleccion.getIndexes()

## USO DESDE PYTHON

Lo primero que tenemos que hacer es instalar las librerías de Python para MongoDB:

pip install pymongo

En este ejemplo vamos a guardar en una colección de la base de datos que llamaremos "Books", información relacionada sobre libros.

Primeramente vamos a crear la clase Book en Python:

class Book:

def \_\_init\_\_(self, titulo, autor, anyo, descripcion):

self.titulo = titulo

self.autor = autor

self.anyo = anyo

self.descripcion = descripcion

En nuestro script principal vamos a declarar una lista de objetos de la clase Book:

# Creamos una lista de objetos a insertar en la BD

books = [

Book('MongoDB','author',2019,'MongoDB'),

Book('Cassandra','author',2020,'Cassandra'),

Book('Neo4j','author',2021,'Neo4j'),

Book('ElasticSearch','author',2022,'Elastic')

]

Una vez definido lo que queremos guardar en nuestra base de datos, vamos a ver cómo nos conectamos a la misma:

# PASO 1: Conexión al servidor de MongoDB pasándole por parámetro el host y el puerto

mongoClient = MongoClient('localhost',27017)

# PASO 2: Conexión a la base de datos

db = mongoClient.Book

# PASO 3: Obtenemos una colección para trabajar con ella

collection = db.Books

El tipo de dato diccionario de Python se adapta muy bien a MongoDB ya que ambos utilizan la estructura "<clave, valor>".

Para trabajar de una forma más limpia y organizada, creamos en la clase "Book" la función toDBCollection que transforma nuestro objeto en un diccionario:

def toDBCollection (self):

return {

"titulo":self.titulo,

"autor":self.autor,

"anyo": self.anyo,

"descripcion":self.descripcion

}

Ahora insertamos en la colección Books cada uno de los objetos book que hemos creado y que tenemos en la lista books.

# PASO 4: Insertamos los documentos

for book in books:

print(book.toDBCollection())

collection.insert\_one(book.toDBCollection())

Ahora vamos a hacer consultas, primeramente vamos a contar cuantos documentos tenemos en nuestra colección.

# PASO 5: Obtenemos el número documentos

print(collection.count\_documents({}))

Al hacer consultas (find), Mongo nos devuelve una lista (o un cursor) con objetos por tanto, hecha la consulta, debemos de recorrer cada uno de los objetos y acceder a su clave para poder imprimirlos por pantalla.

# PASO 6: Leemos todos los documentos de la base de datos

book\_list = collection.find()

for book in book\_llist:

print(book['titulo'], book['autor'], book['anyo'], book['descripcion'])

Pasamos ahora a hacer una actualización o modificación de los documentos. Vamos a actualizar la descripción de los libros a todos aquellos libros cuyo año sea mayor de 2020.

Las consultas de actualización requieren 4 parámetros:

* Filtro para buscar los documentos a acutalizar
* Campos a actualizar con los nuevos datos
* upsert: si es True si no se encuentra una coincidencia, creará un nuevo documento con los datos proporcionados en la operación de actualización, si encuentra coincidencia actualiza los datos. Si es False (o no se proporciona), solo actualizará documentos que coincidan con los criterios de búsqueda.
* multi: si es True, la función update() actualizará todos los documentos que coincidan con los criterios de búsqueda. Si es False (o no se proporciona), solo actualizará el primer documento que coincida con los criterios de búsqueda.

# PASO 7: Actualizamos la descripción de los libros

collection.update\_many(

{"anyo": {"$gt":2020}},

{"$set": {"descripcion":"nuevadescripcion"}},

upsert = False

)

Por último vamos a mostrar cómo borrar documentos. En el ejemplo, vamos a eliminar todos los documentos que sean del año 2019.

# PASO 8: Borramos los libros donde anyo=2019

collection.delete\_one({"anyo":2019})

Veamos el código completo:

***main\_mongo.py***

from Book import Book

from pymongo import MongoClient

# Creamos una lista de objetos a insertar en la BD

books = [

Book('MongoDB','author',2019,'MongoDB'),

Book('Cassandra','author',2020,'Cassandra'),

Book('Neo4j','author',2021,'Neo4j'),

Book('ElasticSearch','author',2022,'Elastic')

]

# PASO 1: Conexión al Server de MongoDB Pasandole el host y el puerto

mongoClient = MongoClient('localhost',27017)

# PASO 2: Conexión a la base de datos

db = mongoClient.Book

# PASO 3: Obtenemos una colección para trabajar con ella

collection = db.Books

print(collection)

# PASO 4: Creamos los documentos

for book in books:

print(book.toDBCollection())

collection.insert\_one(book.toDBCollection())

# PASO 5:Obtenemos el número documentos

print(collection.count\_documents({}))

# PASO 6: Leemos todos los documentos de la base de datos

book\_list = collection.find()

for book in book\_list:

print(book['titulo'], book['autor'], book['anyo'], book['descripcion'])

# PASO 7: Actualizamos la descripción de los libros.

collection.update\_many({"anyo":{"$gt":2020}},{"$set":{"descripcion":"nueva descripcion"}}, upsert = False)

# PASO 8: Borramos todos los libros donde anyo=2019

collection.delete\_one({"anyo":2019})

***book.py***

class Book:

def \_\_init\_\_(self, titulo, autor, anyo, descripcion):

self.titulo = titulo

self.autor = autor

self.anyo = anyo

self.descripcion = descripcion

def toDBCollection (self):

return {

"titulo":self.titulo,

"autor":self.autor,

"anyo": self.anyo,

"descripcion":self.descripcion

}

def \_\_str\_\_(self):

return "Titulo: {} - Autor: {} - Anyo: {} - Descripcion: {}".format(self.titulo, self.autor, self.anyo, self.descripcion)