

Hoja de referencia de MongoDB

Terminología

Base de datos

Un contenedor para colecciones. Esto es lo mismo que una base de datos en SQL y, por lo general, cada proyecto tendrá su propia base de datos llena de diferentes colecciones.

Colección

Un grupo de documentos dentro de una base de datos. Esto es lo mismo que una tabla en SQL y, por lo general, cada tipo de datos (usuarios, publicaciones, productos) tendrá su propia colección.

Documento

Un registro dentro de una colección. Esto es lo mismo que una fila en SQL y, por lo general, habrá un documento por objeto en la colección. Un documento es también esencialmente un objeto JSON.

Campo

Un par clave-valor dentro de un documento. Esto es lo mismo que una columna en SQL. Cada documento tendrá un número determinado de campos que contienen información como nombre, dirección, pasatiempos, etc. Una diferencia importante entre SQL y MongoDB es que un campo puede contener valores como objetos JSON y arreglos en lugar de solo cadenas de texto, números, booleanos, etc.

Comandos básicos

`mongosh`

Abre una conexión a tu instancia local de MongoDB. Todos los demás comandos se ejecutarán dentro de esta conexión `mongosh`.

`show dbs`

Muestra todas las bases de datos en la instancia actual de MongoDB.

`use <nombre_base_de_datos>`
`use miBaseDeDatos`

Cambia a la base de datos proporcionada por `nombre_base_de_datos`.

`db`

Muestra el nombre de la base de datos actual.

`cls`

Limpia la pantalla del terminal.

`show collections`

Muestra todas las colecciones en la base de datos actual.

`db.dropDatabase()`

Elimina la base de datos actual.

exit

Sale de la sesión de mongosh.

Crear

Cada uno de estos comandos se ejecuta en una colección específica

db.<nombreColeccion>.<comando>

insertOne

```
db.usuarios.insertOne({ nombre: "Kyle" })
```

Crea un nuevo documento dentro de la colección especificada.

Añade un nuevo documento con el nombre de Kyle en la colección usuarios.

insertMany

```
db.usuarios.insertMany([{ edad: 26 }, { edad: 20 }])
```

Crea múltiples documentos dentro de una colección específica.

Añade dos nuevos documentos con las edades de 26 y 20 en la colección usuarios.

Leer

Cada uno de estos comandos se ejecuta en una colección específica

db.<nombreColeccion>.<comando>

find

```
db.usuarios.find()
```

Obtiene todos los documentos.

Obtiene todos los usuarios.

find(<objeto_filtro>)

```
db.usuarios.find({ nombre: "Kyle" })
db.usuarios.find({ "direccion.calle": "123 Main St" })
```

Encuentra todos los documentos que coinciden con el objeto de filtro.

Obtiene todos los usuarios con el nombre Kyle. Obtiene todos los usuarios cuyo campo direccion tiene un campo calle con el valor "123 Main St".

find(<objeto_filtro>, <objeto_seleccion>)

```
db.usuarios.find({ nombre: "Kyle" }, { nombre: 1, edad: 1 })
```

```
db.usuarios.find({}, { edad: 0 })
```

Encuentra todos los documentos que coinciden con el objeto de filtro pero solo devuelve los campos especificados en el objeto de selección.

Obtiene todos los usuarios con el nombre Kyle pero solo devuelve su nombre, edad y `_id`. Obtiene todos los usuarios y devuelve todas las columnas excepto la edad.

`findOne`

```
db.usuarios.findOne({ nombre: "Kyle" })
```

Es igual a `find`, pero solo devuelve el primer documento que coincida con el objeto de filtro.

Obtiene el primer usuario con el nombre Kyle.

`countDocuments`

```
db.usuarios.countDocuments({ nombre: "Kyle" })
```

Devuelve la cantidad de documentos que coinciden con el objeto de filtro proporcionado.

Obtiene el número de usuarios con el nombre Kyle.

Actualizar

Cada uno de estos comandos se ejecuta en una colección específica

```
db.<nombreColeccion>.<comando>
```

`updateOne`

```
db.usuarios.updateOne({ edad: 20 }, { $set: { edad: 21 } })
```

Actualiza el primer documento que coincida con el filtro especificado.

`updateMany`

```
db.usuarios.updateMany({ edad: 12 }, { $inc: { edad: 3 } })
```

Actualiza todos los documentos que coincidan con el filtro sumando 3 a la edad.

`replaceOne`

```
db.usuarios.replaceOne({ edad: 12 }, { edad: 13 })
```

Reemplaza el primer documento que coincida con el filtro por un nuevo documento.

Eliminar

Cada uno de estos comandos se ejecuta en una colección específica

```
db.<nombreColeccion>.<comando>
```

deleteOne

```
db.usuarios.deleteOne({ edad: 20 })
```

Elimina el primer documento que coincida con el filtro.

deleteMany

```
db.usuarios.deleteMany({ edad: 12 })
```

Elimina todos los documentos que coincidan con el filtro.

Operadores de filtros complejos

Cualquier combinación de los siguientes operadores puede ser utilizada dentro de un objeto de filtro para realizar consultas complejas.

\$eq

```
db.usuarios.find({ nombre: { $eq: "Kyle" } })
```

Comprueba la igualdad.

Obtiene todos los usuarios con el nombre Kyle.

\$ne

```
db.usuarios.find({ nombre: { $ne: "Kyle" } })
```

Comprueba que un valor sea diferente.

Obtiene todos los usuarios con un nombre distinto de Kyle.

\$gt / \$gte

```
db.usuarios.find({ edad: { $gt: 12 } })
db.usuarios.find({ edad: { $gte: 15 } })
```

Comprueba si un valor es mayor o mayor o igual a otro.

Obtiene todos los usuarios con una edad mayor a 12. Obtiene todos los usuarios con una edad mayor o igual a 15.

\$lt / \$lte

```
db.usuarios.find({ edad: { $lt: 12 } })
db.usuarios.find({ edad: { $lte: 15 } })
```

Comprueba si un valor es menor o menor o igual a otro.

Obtiene todos los usuarios con una edad menor a 12. Obtiene todos los usuarios con una edad menor o igual a 15.

\$in

```
db.usuarios.find({ nombre: { $in: ["Kyle", "Mike"] } })
```

Verifica si un valor pertenece a un conjunto de valores.

Obtiene todos los usuarios con el nombre Kyle o Mike.

\$nin

```
db.usuarios.find({ nombre: { $nin: ["Kyle", "Mike"] } })
```

Verifica si un valor no pertenece a un conjunto de valores.

Obtiene todos los usuarios cuyo nombre no es Kyle ni Mike.

\$and

```
db.usuarios.find({ $and: [{ edad: 12 }, { nombre: "Kyle" }] })
```

Comprueba que todas las condiciones sean verdaderas.

Obtiene todos los usuarios que tienen 12 años y se llaman Kyle.

\$or

```
db.usuarios.find({ $or: [{ edad: 12 }, { nombre: "Kyle" }] })
```

Comprueba que al menos una condición sea verdadera.

Obtiene todos los usuarios con el nombre Kyle o con una edad de 12 años.

\$not

```
db.usuarios.find({ nombre: { $not: { $eq: "Kyle" } } })
```

Niega el filtro dentro de \$not.

Obtiene todos los usuarios cuyo nombre no es Kyle.

\$exists

```
db.usuarios.find({ nombre: { $exists: true } })
```

Verifica si un campo existe.

Obtiene todos los usuarios que tienen un campo nombre.

`$expr`

```
db.usuarios.find({ $expr: { $gt: ["$saldo", "$deuda"] } })
```

Realiza comparaciones entre diferentes campos.

Obtiene todos los usuarios cuyo saldo es mayor que su deuda.

Modificadores de lectura

Cualquier combinación de los siguientes modificadores puede ser agregada al final de cualquier operación de lectura.

`sort`

```
db.usuarios.find().sort({ nombre: 1, edad: -1 })
```

Ordena los resultados de una consulta según los campos dados.

Obtiene todos los usuarios ordenados por nombre en orden alfabético y luego, si los nombres son iguales, por edad en orden descendente.

`limit`

```
db.usuarios.find().limit(2)
```

Limita la cantidad de documentos devueltos.

Devuelve solo los dos primeros usuarios.

`skip`

```
db.usuarios.find().skip(4)
```

Omite una cantidad de documentos desde el inicio de los resultados.

Omite los primeros 4 usuarios al devolver los resultados. Esto es útil para la paginación cuando se combina con `limit`.

1. Índices en MongoDB

MongoDB permite la creación de índices para optimizar las consultas. Sin índices, las búsquedas pueden ser ineficientes en bases de datos con grandes volúmenes de datos.

- **Crear un índice simple:**

```
db.usuarios.createIndex({ nombre: 1 })
```

Este índice mejora la velocidad de búsqueda en el campo `nombre`, ya que los documentos se almacenarán en orden ascendente basado en ese campo. Esto es especialmente útil cuando realizamos muchas búsquedas por este campo.

- **Crear un índice compuesto:**

```
db.usuarios.createIndex({ nombre: 1, edad: -1 })
```

Este índice mejora el rendimiento en consultas que filtran por `nombre` y ordenan por `edad` en orden descendente. Es útil cuando queremos recuperar registros ordenados por múltiples criterios de búsqueda.

- **Eliminar un índice:**

```
db.usuarios.dropIndex("nombre_1")
```

Borra el índice previamente creado en el campo `nombre`, lo que puede ralentizar futuras búsquedas si ese campo era clave en las consultas. Se recomienda eliminar índices solo cuando ya no sean necesarios.

- **Listar los índices de una colección:**

```
db.usuarios.getIndexes()
```

Devuelve un listado de todos los índices existentes en la colección `usuarios`, mostrando su estructura y optimización. Es útil para evaluar el rendimiento de la base de datos.

- **Selección de índices en consultas:**

MongoDB decide automáticamente qué índice utilizar basándose en las estadísticas de uso y la estructura de la consulta. Sin embargo, se puede forzar el uso de un índice específico con `hint`:

- ```
db.usuarios.find({ nombre: "Carlos" }).hint({ nombre: 1 })
```

*Esto obliga a MongoDB a usar el índice sobre `nombre`, lo cual puede ser útil para pruebas de rendimiento o depuración de consultas lentas.*

## 2. Agregaciones

La función `aggregate()` en MongoDB permite realizar operaciones avanzadas en los documentos de una colección. Se compone de una serie de etapas (`stages`), donde cada etapa transforma los datos en un proceso secuencial.

### Funcionamiento de `aggregate()`

`aggregate()` recibe un array de etapas. Cada etapa aplica una transformación específica a los documentos y los pasa a la siguiente etapa en la secuencia.

Ejemplo de estructura básica:

```
db.usuarios.aggregate([
 { $match: { edad: { $gte: 18 } } }, // Filtra documentos
 { $group: { _id: "$ciudad", totalUsuarios: { $sum: 1 } } }, //
Agrupar datos
 { $sort: { totalUsuarios: -1 } } // Ordena resultados
])
```

Este ejemplo primero filtra usuarios mayores de 18 años (`$match`), luego agrupa los usuarios por ciudad (`$group`) y finalmente ordena los resultados en orden descendente (`$sort`).

### Principales operadores de agregación

- **Filtrar datos con `$match`**

```
db.usuarios.aggregate([
 { $match: { edad: { $gte: 18 } } }
])
```

*Filtra los documentos para incluir solo aquellos usuarios cuya edad es 18 o más. Es similar a la cláusula `WHERE` en SQL y mejora el rendimiento de la agregación reduciendo el número de documentos procesados.*

- **Agrupar datos con `$group`**

```
db.usuarios.aggregate([
 { $group: { _id: "$ciudad", totalUsuarios: { $sum: 1 } } }
])
```

*Agrupar los documentos por ciudad y cuenta cuántos usuarios hay en cada una, permitiendo generar estadísticas de uso o reportes.*

- **Ordenar datos con `$sort`**

```
db.usuarios.aggregate([
 { $sort: { totalUsuarios: -1 } }
])
```



*Ordena los resultados en orden descendente basado en totalUsuarios, útil para mostrar rankings o tendencias.*

- **Proyectar y renombrar campos con \$project**

```
db.usuarios.aggregate([
 { $project: { nombreCompleto: { $concat: ["$nombre", " ",
"$apellido"] }, _id: 0 } }
])
```

*Permite modificar la estructura de salida combinando nombre y apellido en un nuevo campo nombreCompleto, ocultando \_id. Esto es útil para mejorar la presentación de los datos en reportes.*

### 3. Uniones entre colecciones con \$lookup

El operador \$lookup permite realizar una unión entre documentos de diferentes colecciones, equivalente a JOIN en SQL.

- **Unir datos entre pedidos y usuarios:**

```
db.pedidos.aggregate([
 {
 $lookup: {
 from: "usuarios",
 localField: "usuario_id",
 foreignField: "_id",
 as: "informacion_usuario"
 }
 }
])
```

*Asocia cada pedido con la información del usuario correspondiente, almacenando los datos en un array informacion\_usuario. Esto permite recuperar información detallada sin necesidad de hacer múltiples consultas.*

- **Expandir los resultados con \$unwind**

```
db.pedidos.aggregate([
 { $lookup: { from: "usuarios", localField: "usuario_id",
foreignField: "_id", as: "usuario" } },
 { $unwind: "$usuario" }
])
```

*Convierte el array de usuarios en documentos individuales, facilitando la manipulación y análisis de datos.*

## 4. Almacenamiento de resultados con `$merge`

El operador `$merge` permite almacenar los resultados de una agregación en una colección existente o crear una nueva.

- **Guardar resultados de agregación en una nueva colección:**

```
db.usuarios.aggregate([
 { $group: { _id: "$ciudad", totalUsuarios: { $sum: 1 } } },
 { $merge: { into: "resumen_ciudades", whenMatched: "merge",
whenNotMatched: "insert" } }
])
```

*Agrupar los usuarios por ciudad, cuenta cuántos hay en cada una y almacena los resultados en `resumen_ciudades`. Si la colección ya existe, combina los datos.*