MongoDB

Pablo Hidalgo

MongoDB es una de las bases de datos NoSQL más utilizada. Ya hemos visto en clase que se trata de una base de datos orientada a documentos.

Primeros pasos

Una vez que se ha completado la instalación, ya podemos empezar a divertirnos con MongoDB.

> Asegúrate de que MongoDB está funcionando. Para ello, debes haber ejecutado en la línea de comandos `mongod` y tienes que haber obtenido algo como `initandlisten] waiting for connections on port 27017`. Ten cuidado de no cerrar esta ventana ya que matarías la sesión de MongoDB.

Abre una nueva ventana de la línea de comandos y escribe

```
mongo mbads
```

Esta sentencia genera una conexión al *motor* de MongoDB y crea una base de datos `mbads` en el caso de que no exista previamente.

Escribe `help` para ver qué sentencias puedes utilizar.

Como nos hemos conectado a MongoDB con la sentencia `mongo mbads` estamos en la base de datos `mbads`. Podemos ver qué otras bases de datos tenemos disponibles escribiendo `show dbs` y cambiar a otra base de datos escribiendo `use nombre_bbdd`.

Ahora mismo nuestra base de datos está vacía; tenemos que empezar a rellenarla con datos. Para crear una colección (recuerda la nomenclatura en Mongo y piensa en que las colecciones son análogas a las tablas en un base de datos relacional) es tan fácil como empezar a añadir registros, ya que no es necesario crear una estructura (ni un esquema) simplemente hay que empezar a usarla. Además, si te diste cuenta antes, cuando ejecutamos `show dbs` no aparecía por ningún lado nuestra base de datos `mbads` ya que ésta no existe propiamente hasta que no comienza a tener valores. Parar crear una colección `ciudades` tenemos que **insertar** un valor con ayuda del comando `db.nombre_coleccion.insert()`:

```
db.ciudades.insert({
  nombre: "Madrid",
  poblacion: 3182175,
  ultimo_censo: ISODate("2017-01-01"),
  monumentos: ["Cibeles", "Puerta de Alcalá"],
  alcalde: {
    nombre: "Manuela Carmena",
    partido: "Ahora Madrid"
  }
})
```

Como hemos visto en clase, los documentos (recuerda, una colección está compuesta de documentos, algo similar a una fila o un registro en una base de datos relacional) siguen un formato al estilo JSON (estrictamente, se tratan de BSON) por lo que cualquier nuevo documento que añadamos debe seguir ese formato donde las llaves `{...}` denotan un objeto compuesto por parejas de atributo-valor y donde `[...]` significa un array.

Si te fijas en el documento que acabamos de insertar, uno de los valores es `monumentos` al que lo hemos declarado como un array cosa que, por construcción, no era posible hacer en una base de datos relacional ya que una de los principios es que una celda (la intersección entre una fila y una columna) solo podía

contener un único valor. Eso nos podía llevar en algún caso a tener alguna información duplicada. Además, un propiedad relevante de los documentos es que se introduce de forma natural el concepto de jerarquía como puedes ver en el atributo `alcalde`.

Ahora podemos verificar que existe nuestra base de datos con `show dbs` y ver la colección con `show collections`. Podemos listar el contenido de una colección con `find()`:

```
db.ciudades.find()
```

Como puedes ver, la sintaxis es muy diferente a la misma operación escrita en lenguaje SQL:

```
SELECT *
  FROM ciudades;
```

Como habrás observado, el documento contiene un campo `_id` que Mongo genera automáticamente y que se trata de un identificador único, algo así como el datatype `SERIAL` en PostgreSQL. Este `_id` siempre tiene 12 bytes y está compuesto por un *timestamp*, un identificador de la máquina cliente, un identificador del proceso del cliente y un contador de 3-bytes incremental.

Aunque pueda parecer que para nosotros, como usuarios, cómo se genere `_id` no es relevante, lo cierto es que esa forma de generarlo permite que cada máquina no tenga que conocer los identificadores que hayan generado otras siendo fundamental para poder lanzar procesos distribuidos (el *dataype* SERIAL de PostgreSQL necesita conocer el último generado para incremetnarlo en una unidad).

> Cuando ejecutamos la función `find()`, la consola muestra cada documento en una sola línea que hace difícil la visualización. Puedes añadir al final `.pretty` para obtener una vista más intuitiva. Pruébalo con `db.ciudades.find().pretty()`

Por completar nuestra colección, vamos a añadir algún documento más. Para añadir varios documentos a la vez en una colección, utilizamos `[...]`:

```
db.ciudades.insert(
  [
    {
      nombre: "Barcelona",
      poblacion: 1620809,
      ultimo_censo: ISODate("2017-01-01"),
      monumentos: ["Sagrada Familia", "Pedrera"],
      alcalde: {
        nombre: "Ada Colau",
        partido: "Barcelona en Comú"
      }
    },
    {
      nombre: "Sevilla",
      poblacion: 689434,
      ultimo_censo: ISODate("2017-01-01"),
      monumentos: ["Giralda", "Torre del Oro"],
      alcalde: {
        nombre: "Juan Espadas",
        partido: "PSOE"
      }
    },
    {
      nombre: "Oviedo",
      poblacion: 220301,
      ultimo_censo: ISODate("2017-01-01"),
      monumentos: ["Giralda", "Torre del Oro"],
```

```
      alcalde: {
        nombre: "Wenceslao López",
        partido: "PSOE"
      }
    }
  ]
)
```

Una función que puede resultar útil es saber cuándos documentos existen dentro de una colección. Esto lo podemos hacer con la función `count()`:

```
db.ciudades.count()
```

Leyendo datos

Ya hemos visto que la función `find()` sin parámetros nos da todos los documentos. Para especificar un documento en concreto, tenemos que hacerlo de la siguiente forma

```
db.ciudades.find({nombre: "Madrid"})
```

La función `find()` acepta un segundo parámetro opcional con el que especificar qué campos queremos obtener. Por ejemplo, si solo queremos la población de una ciudad (además del `_id`), podemos escribir:

```
db.ciudades.find({nombre: "Madrid"}, {poblacion: true})
```

Si queremos obtener todos los campos excepto la población, podemos hacerlo con:

```
db.ciudades.find({nombre: "Madrid"}, {poblacion: false})
```

> ¿Cómo harías para devolver los campos `poblacion`, `alcalde` de la ciudad de Madrid?
>
> ¿Qué ocurre si escribimos `db.ciudades.find({nombre: "Madrid"}, {poblacion: false})`?

Igual que en PostgreSQL, en Mongo podemos construir queries específicas que nos permitan encontrar aquellos documentos que cumplan algún requisito.

Por ejemplo, para encontrar aquellas ciudades cuya población es mayor que 1.000.000 debemos escribir:

```
db.ciudades.find({poblacion: {$gt: 1000000}})
```

En Mongo, los operaradores condicionales se escribe como `{$operador: valor}`.

Aquí se pueden consultar los operadores que existen en Mongo. Algunos habituales son:

- `$eq`: Valores **idénticos** a un valor especificado.
- `$gt`: Valores **mayores** que un valor especificado.
- `$lt`: Valores **menores** que un valor especificado.
- `$gte`: Valores **mayores o iguales** que un valor especificado.
- `$lte`: Valores **menores o iguales** que un valor especificado.
- `$in`: valores en un **array** de valores.
- `$ne`: Valores **distintos** a un valor especificado.

Jerarquías de datos

Mongo está muy bien preparado para trabajar con jerarquías de datos. Para filtrar *subdocumento*, hay que separar los campos anidados por puntos. Por ejemplo, para encontrar aquellas ciudades en las que el partido que gobierna es el PSOE, lo haríamos

```
db.ciudades.find(
  {"alcalde.partido" : "PSOE"}
)
```

Operaciones `OR`

Cuando utilizamos la función `find()` estamos utilizando implícitamente operaciones `and`, es decir, solo devuelve aquellos documentos que cumplan **todos** los criterios que hayamos especificado. Por ejemplo, la siguiente sentencia no devuelve ningún resultado ya que ningún documento satisface las dos condiciones **a la vez**:

```
db.ciudades.find(
  {
    "nombre": "Sevilla",
    "alcalde.nombre": "Manuela Carmena"
  }
)
```

Sin embargo, si queremos obtener aquellos registros que satisfacen *alguna* de las condiciones, necesitamos utilizar el operador `or`:

```
db.ciudades.find(
  {
    $or: [
      {"nombre": "Sevilla"},
      {"alcalde.nombre": "Manuela Carmena"}
    ]
  }
)
```

> **Ejercicio:** encuentra aquellos documentos que hagan referencia a las ciudades de Madrid y Sevilla. ¿Sabrías hacerlo de dos formas distintas?

Actualizando datos

Hasta ahora hemos añadido nuevos documentos a una colección. Otra tarea que querremos saber hacer es **cómo actualizar documentos ya existentes**. Para ello utilizamos la función `update(criterio, operacion)`. El primero de los parámetros, `criterio`, es el mismo tipo de objeto que se le pasa a la función `find()`; el segundo parámetro, `operacion`, es un objeto cuyos campos reemplazarán a aquellos documentos que concuerden con lo escrito en `criterio`.

Por ejemplo, si quisiésemos actualizar la población de Madrid, tendríamos que

```
db.ciudades.update(
  {nombre: "Madrid"},
  {$set : {poblacion: 4000000, ultimo_censo: ISODate("2018-01-01")}}
)
```

Quizá te preguntes si es necesaria escribir `$set`. Si en vez de la sentencia anterior escribiésemos

```
db.ciudades.update(
  {nombre: "Madrid"},
  {poblacion: 4000000, ultimo_censo: ISODate("2018-01-01")}
)
```

Mongo entendería que lo que quieres es **reeemplazar por completo** aquellos el documento de la ciudad de Madrid por `{poblacion: 4000000, ultimo_censo: ISODate("2018-01-01")}`. Así que, presta atención cuando quieras actualizar datos.

Podemos hacer más cosas que modificar un determinado valor. Por ejemplo, `$inc` nos permite incrementar un número por una canitdad determinada:

```
db.ciudades.update(
{nombre: "Madrid"},
{$inc: {poblacion: 1000}}
)
```

Se van añadiendo constantemente nuevos operadores, por lo que es aconsejable consultar la documentación para estar al día de los cambios. Algunos relevantes son:

- `$set`: cambia un campo con un valor dado.
- `$unset`: elimina un campo.
- `$inc`: incrementa un campo por un número dado.
- `$pop`: eliman el último (o el primer) elemento de un array.
- `$push`: añade un elemento a un array.

Eliminar datos

Otra operación habitual es la de **eliminar documentos** de una colección. En Mongo esto se puede hacer simplemente reemplazando `find()` por `remove()` y todos los documentos que satisfagan la condición se eliminarán.

Antes de eliminar cualquier documento, es recomendable ejecutar `find()` para comprobar que el criterio está bien construido y hace lo que queremos que haga.

> **Ejercicio:** Elimina todos las ciudades en las que gobierne el PSOE.

Eliminar una colección y una base de datos

Por último, dos tareas que pueden ser útiles son las de eliminar una colección entera (junto con todos su documentos) y eliminar completamente una base de datos.

Para eliminar una colección utilizamos `drop()`. Para comprobar su funcionamiento, primero vamos a asegurarnos de las colecciones existentes en nuestra base de datos:

```
show collections
```

Para eliminar la colección `ciudades` podemos ejecutar

```
db.ciudades.drop()
```

Para eliminar una base de datos completa, necesitamos utilizar `dropDatabase()`. Igual que en el caso anterior, primero mostramos las bases de datos disponibles:

```
show dbs
```