Práctica avanzada con MongoDB

Taller de NoSQL

Pablo Hidalgo

En la **primera sesión** vimos un repaso de las **operaciones CRUD básicas para trabajar con MongoDB**. En esta práctica vamos ir un poco más allá y hacer operaciones más complejas con MongoDB.

1 Los datos

Hasta ahora hemos creado una colección de forma *manual* a través del comando `insert()`. En esta práctica vamos a trabajar de una forma algo más *real*, importando un conjunto de documentos (ya sabes, las colecciones están compuestas de documentos) con *más gracia*.

Vamos a utilizar datos que el ayuntamiento de Madrid tiene en su portal de datos abiertos. En particular, vamos a trabajar con los datos en este enlace en el que se puede encontrar datos del uso del servicio de bicicletas eléctricas de ayuntamiento de Madrid **BiciMAD**.

En la web aparecen dos conjuntos de datos, los que se llaman `Datos de uso de ...` contienen la traza de todos los movimientos que se han producido. Es decir, información de cada vez que un usuario ha cogido una bicicleta en una estación y la ha dejado en otra.

Los que se denominan `Situación estaciones bicimad por día y hora de ...` recogen información de en qué estado se encuentra cada estación en diversos instantes temporales.

> **Nota:** en esta práctica se han usado los conjuntos de datos `201812_Usage_Bicimad.json` y `Bicimad_Stations_201812.json`. Si utilizas otros, los resultados que obtengas serán ligeramente distintos.


```
{
    "_id" : "2018-11-01T00:05:32.277622",
    "stations" : [
        {
            "activate" : 1,
            "name" : "Segovia 26",
            "reservations_count" : 0,
            "light" : 0,
            "total_bases" : 24,
            "free_bases" : 18,
            "number" : "166",
            "longitude" : "-3.7135833",
            "no_available" : 0,
            "address" : "Calle Segovia, 26",
            "latitude" : "40.4138333",
            "dock_bikes" : 6,
            "id" : 174
        }
    ]
}
```

2 Dónde guardar los datos *(opcional)*

MongoDB es una base de datos y, como tal, tiene que guardar los datos físicamente en el disco duro. Podemos *levantar* el servicio de MongoDB eligiendo **dónde se van a almacenar los datos**.

Para ello, primero creamos una nueva carpeta donde queremos que se guarden los datos. En una distribución Linux o Mac, para crear una nueva carpeta, lo podemos hacer con el comando `mkdir` (la opción `-p` significa que creará las carpetas intermedias necesarias)

```
mkdir -p mongodb/data/db
```

En Windows

```
md "\data\db"
```

Ahora podemos *lanzar* el servicio de MongoDB especificando la ruta donde tiene que almacenar los datos. En Mac/Linux:

```
mongod --dbpath ruta
```

En Windows:

```
"C:\Program Files\MongoDB\Server\4.0\bin\mongod.exe" --dbpath="ruta"
```

Esto habrá lanzado el *motor* de MongoDB. Si queremos entrar en la *shell* de MongoDB, abrimos una nueva ventana de la terminal y escribimos `mongo` o ejecutamos `C:\Program Files\MongoDB\Server\4.0\bin\mongod.exe`.

3 Importar los datos

Para importar un conjunto de datos podemos utilizar `mongoimport`.

> El comando `mongoimport` se utiliza directamente desde la línea de comandos del sistema, no desde la shell de mongo.

```
mongoimport --db bicimad --collection usage --file 201812_Usage_Bicimad.json
mongoimport --db bicimad --collection stations --file Bicimad_Stations_201812.json
```

> Si estás en Windows: abre **símbolo de sistema** y sustituye `mongoimport` por `C:\Program Files\MongoDB\Server\4.0\bin\mongo.exe` en las sentencias anteriores.

Si ahora entramos en la shell de mongo mediante el comando `mongo` y escribimos `show dbs`, podemos ver la base de datos `bicimad` que hemos creado al importar el conjunto de datos.

Cambiamos la base de datos con `use bicimad` y escribiendo `show collections` veremos la colección `usage`.

4 Datos de uso

4.1 Manipulando los datos

Para ver la magnitud de la colección, es conveniente hacer `db.usage.count()` de donde obtendremos 273.272 colecciones.

Vamos a intentar entender la estructura de los documentos. Si ejecutamos `db.usage.findOne()` obtendremos el primer documento para una inspección rápida.


```
{
    "_id" : ObjectId("5c085aae2f384324b8a10925"),
    "user_day_code" : "39d847f3af0e9d8ab5f29fa7b5af3612c87e84f4d35a1cb479edf8d38f1ce8ac",
    "idplug_base" : 7,
    "user_type" : 3,
    "idunplug_base" : 7,
    "travel_time" : 16,
    "idunplug_station" : 75,
    "ageRange" : 5,
    "idplug_station" : 75,
    "unplug_hourTime" : ISODate("2018-11-30T23:00:00Z"),
    "zip_code" : ""
}
```


Si ejecutamos `db.usage.find().limit(3).pretty()` para acceder a los tres primeros documentos, tenemos que el tercero de ellos tiene la siguiente estructura


```
{
    "_id" : ObjectId("5c085aae2f384324b8a10928"),
    "user_day_code" : "697479f320b109464122ad3ce33eb368a6bc6755682f95545586c240b433f24a",
    "idplug_base" : 19,
    "track" : {
        "type" : "FeatureCollection",
        "features" : [
            {
                "geometry" : {
                    "type" : "Point",
                    "coordinates" : [
                        -3.6949683,
                        40.4031057997222
                    ]
                },
                "type" : "Feature",
                "properties" : {
                    "var" : "28045,ES,Madrid,Madrid,CALLE PALOS DE LA FRONTERA 25,Madrid",
                    "speed" : 1.94,
                    "secondsfromstart" : 246
                }
            },
            {
                "geometry" : {
                    "type" : "Point",
                    "coordinates" : [
```

```
                        -3.6972233,
                        40.4053758
                    ]
                },
                "type" : "Feature",
                "properties" : {
                    "var" : "28012,ES,Madrid,Madrid,CALLE JOSE ANTONIO DE ARMONA 12,Madrid",
                    "speed" : 8,
                    "secondsfromstart" : 186
                }
            },
            {
                "geometry" : {
                    "type" : "Point",
                    "coordinates" : [
                        -3.69792949972222,
                        40.406854
                    ]
                },
                "type" : "Feature",
                "properties" : {
                    "var" : "28012,ES,Madrid,Madrid,CALLE MALLORCA 7,Madrid",
                    "speed" : 0.66,
                    "secondsfromstart" : 126
                }
            },
            {
                "geometry" : {
                    "type" : "Point",
                    "coordinates" : [
                        -3.69815579972222,
                        40.4078321
                    ]
                },
                "type" : "Feature",
                "properties" : {
                    "var" : "28012,ES,Madrid,Madrid,CALLE SOMBRERERIA 24,Madrid",
                    "speed" : 6.36,
                    "secondsfromstart" : 66
                }
            }
        ]
    },
    "user_type" : 1,
    "idunplug_base" : 14,
    "travel_time" : 262,
    "idunplug_station" : 57,
    "ageRange" : 4,
    "idplug_station" : 128,
    "unplug_hourTime" : ISODate("2018-11-30T23:00:00Z"),
    "zip_code" : "28045"
}
```


Como ya sabemos, MongoDB es una base de datos *schemaless* y que ++permite que cada documento de una

colección pueda tener una estructura ligeramente distinta**. En este último documento tenemos el campo `track` que no aparecía en los dos documentos anteriores.

Una cosa que sí comparten los tres primeros documentos es el **código postal**. Para extraer solamente el código postal de las colecciones, podemos hacer con `db.usage.find({}, {zip_code: 1})` de donde obtendremos:

```
{ "_id" : ObjectId("5c085aae2f384324b8a10925"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aae2f384324b8a1092c"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aae2f384324b8a10928"), "zip_code" : "28045" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10948"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10958"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aae2f384324b8a10931"), "zip_code" : "28020" }
{ "_id" : ObjectId("5c085aae2f384324b8a10940"), "zip_code" : "28010" }
{ "_id" : ObjectId("5c085aaf2f384324b8a1095c"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aae2f384324b8a10938"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aaf2f384324b8a1095b"), "zip_code" : "28045" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10941"), "zip_code" : "28014" }
{ "_id" : ObjectId("5c085aae2f384324b8a1093a"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10962"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10966"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aae2f384324b8a10937"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aaf2f384324b8a1095f"), "zip_code" : "28004" }
{ "_id" : ObjectId("5c085aaf2f384324b8a1095e"), "zip_code" : "28012" }
{ "_id" : ObjectId("5c085aaf2f384324b8a1096d"), "zip_code" : "11207" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10964"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10971"), "zip_code" : "" }
```

En muchos documentos, el campo `zip_code` está sin informar. Como cada documento puede tener una estructura distinta, vamos a hacer que desaparezca este campo en aquellos documentos en los que esté sin informar y así optimizar el espacio en disco.

El operador `$unset` elimina un campo en particular. Así, podemos hacer `db.usage.update({zip_code: ""}, {$unset: {zip_code: ""}})`. Si volvemos a ejecutar `db.usage.find({}, {zip_code: 1})`, obtenemos

```
{ "_id" : ObjectId("5c085aae2f384324b8a10925") }
{ "_id" : ObjectId("5c085aae2f384324b8a1092c"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aae2f384324b8a10928"), "zip_code" : "28045" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10948"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10958"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aae2f384324b8a10931"), "zip_code" : "28020" }
{ "_id" : ObjectId("5c085aae2f384324b8a10940"), "zip_code" : "28010" }
{ "_id" : ObjectId("5c085aaf2f384324b8a1095c"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aae2f384324b8a10938"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aaf2f384324b8a1095b"), "zip_code" : "28045" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10941"), "zip_code" : "28014" }
{ "_id" : ObjectId("5c085aae2f384324b8a1093a"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10962"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10966"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aae2f384324b8a10937"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aaf2f384324b8a1095f"), "zip_code" : "28004" }
{ "_id" : ObjectId("5c085aaf2f384324b8a1095e"), "zip_code" : "28012" }
{ "_id" : ObjectId("5c085aaf2f384324b8a1096d"), "zip_code" : "11207" }
```

```
{ "_id" : ObjectId("5c085aaf2f384324b8a10964"), "zip_code" : "" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10971"), "zip_code" : "" }
```

¿Qué ha pasado? El comando `db.usage.update()` solo actualiza el **primer** documento que satisface la condición. Para actualizar todos los documentos que cumplen la condición podemos hacerlo usando el parámetro `multi` `db.usage.update({zip_code: ""}, {$unset: {zip_code: ""}}, {multi: 1})` o con `db.usage.updateMany({zip_code: ""}, {$unset: {zip_code: ""}})`. Ahora sí que tenemos lo que queremos:

```
{ "_id" : ObjectId("5c085aae2f384324b8a10925") }
{ "_id" : ObjectId("5c085aae2f384324b8a1092c") }
{ "_id" : ObjectId("5c085aae2f384324b8a10928"), "zip_code" : "28045" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10948") }
{ "_id" : ObjectId("5c085aaf2f384324b8a10958") }
{ "_id" : ObjectId("5c085aae2f384324b8a10931"), "zip_code" : "28020" }
{ "_id" : ObjectId("5c085aae2f384324b8a10940"), "zip_code" : "28010" }
{ "_id" : ObjectId("5c085aaf2f384324b8a1095c") }
{ "_id" : ObjectId("5c085aae2f384324b8a10938") }
{ "_id" : ObjectId("5c085aaf2f384324b8a1095b"), "zip_code" : "28045" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10941"), "zip_code" : "28014" }
{ "_id" : ObjectId("5c085aae2f384324b8a1093a") }
{ "_id" : ObjectId("5c085aaf2f384324b8a10962") }
{ "_id" : ObjectId("5c085aaf2f384324b8a10966") }
{ "_id" : ObjectId("5c085aae2f384324b8a10937") }
{ "_id" : ObjectId("5c085aaf2f384324b8a1095f"), "zip_code" : "28004" }
{ "_id" : ObjectId("5c085aaf2f384324b8a1095e"), "zip_code" : "28012" }
{ "_id" : ObjectId("5c085aaf2f384324b8a1096d"), "zip_code" : "11207" }
{ "_id" : ObjectId("5c085aaf2f384324b8a10964") }
{ "_id" : ObjectId("5c085aaf2f384324b8a10971") }
```

Al haber eliminado el campo `zip_code` de algunos documentos, si ejecutamos `db.usage.find({zip_code: ""})` no obtendremos ningún resultado. Para extraer aquellos documentos que tengan o no tengan un determinado campo, podemos utilizar el operador '$exists

```
db.usage.find({zip_code: {$exists: 0}}).pretty()
```

4.2 Agregación

Una operación habitual en cualquier base de datos es la de **agregar** datos. Los operadores de agregación agrupan valores de múltiples documentos juntos y pueden realizar operaciones para devolver un único resultado. Tenemos el *aggregation pipeline* para hacer agregaciones. Aquí hay un pequeño vídeo explicativo.

El *aggregation pipeline* de MongoDB consiste en *stages*. Cada *stage* transforma los documentos como si pasasen por el *pipeline*. El *stage pipeline* no es necesario que produzca un *output* para cada documento de *input*.

Vamos a empezar calculando **cuántos viajes de producen para cada estación de origen (campo `idunplug_station`)**. En código SQL, haríamos

```
SELECT COUNT(*) AS count
FROM usage
GROUP BY idunplug_base;
```

En MongoDB necesitamos el *aggregation pipeline stage* `$group`. La estructura de `$group` es

```
{ $group: { _id: <expression>, <field1>: { <accumulator1> : <expression1> }, ... } }
```

El campo `_id` es **obligatorio**. Para referirnos a un campo del documento, haremos `$idunplug_station`. Para hacer el conteo, deberemos recurrir al operador suma `$sum`. La sentencia sería

```
db.usage.aggregate([
  {
    $group: {
      _id: "$idunplug_station",
      count: { $sum: 1 }
      }
  }
])
```

> Hemos escrito `$sum: 1` para hacer el conteo (asignar 1 a cada documento). Podemos utilizar este operador para sumar valores numéricos. Como MongoDB es flexible, puede que en un mismo campo de documentos distintos se almacenen valores numéricos y no numéricos. El operador `$sum` solo tendrá en cuenta aquellos que sean numéricos.

De forma parecida, podemos calcular el tiempo medio de uso (campo `travel_time`) de una bici en función de la estación de partida. con el operardor `$avg`.

```
db.usage.aggregate([
  {
    $group: {
      _id: "$idunplug_station",
      avg_travel_time:{ $avg: "$travel_time" }
      }
  }
])
```

Podemos usar `$group` también para devolver los valores únicos. Por ejemplo, para conocer qué estaciones de origen hay

```
db.usage.aggregate([
  {
    $group: {
      _id: "$idunplug_station"
      }
  }
])
```

Cuando agregamos datos estamos tratando de resumir la información. Tal y como hemos usado hasta ahora `$group` nos devuelve el resultado en un orden indeterminado. Si queremos ordenar los documentos obtenidos, tenemos que utilizar el *stage* `$sort`. La estructura es

```
{ $sort: { <field1>: <sort order>, <field2>: <sort order> ... } }
```

donde `<sort order>` toma valor 1 para especificar un orden ascendente y -1 descendente.

```
db.usage.aggregate([
  {
    $group: {
      _id: "$idunplug_station",
      count:{ $sum: 1 }
      }
  },
  {$sort: {_id: 1}}
])
```

Por último, `$match` puede ser útil para filtrar primero los documentos de los que nos interese agregar.

```
db.usage.aggregate([
  { $match: {idunplug_station: {$in: [1, 3, 5] }}},
  {
    $group: {
      _id: "$idunplug_station",
      count:{ $sum: 1 }
      }
  },
  {$sort: {_id: 1}}
])
```

> El *aggregate pipeline* lleva un orden secuencial. Por tanto, utiliza `$match` tan pronto como sea posible para limitar los documentos que se propagan en el *pipeline*.

Como en cualquier base de datos, el modelo, es decir, la estructura que tenga los datos, será relevante para que unas operaciones sean más fáciles de realizar que otras.

Por ejemplo, si nuestro interés principal es el de analizar el tiempo de viaje en función de las estaciones de partida, sería interesante tener una estructura como

```
{ "_id" : 24, "travel_time" : [ 1010, 852, 1027 ] }
{ "_id" : 6, "travel_time" : [ 383, 412, 901, 263, 287, 1395 ] }
{ "_id" : 16, "travel_time" : [ 477, 570, 794, 512, 626 ] }
{ "_id" : 19, "travel_time" : [ 206, 404, 334, 416, 356, 983, 493 ] }
{ "_id" : 7, "travel_time" : [ 16, 801, 623, 688, 298, 210 ] }
{ "_id" : 26, "travel_time" : [ 410 ] }
{ "_id" : 1, "travel_time" : [ 210, 394, 601, 542, 428, 426, 961 ] }
{ "_id" : 17, "travel_time" : [ 325, 799, 904, 427 ] }
{ "_id" : 2, "travel_time" : [ 386, 883, 538, 989, 379, 490, 642, 525, 777, 412 ] }
{ "_id" : 15, "travel_time" : [ 564 ] }
{ "_id" : 14, "travel_time" : [ 262, 589, 343, 760 ] }
{ "_id" : 8, "travel_time" : [ 546, 537, 432 ] }
{ "_id" : 23, "travel_time" : [ 250, 714, 1455 ] }
{ "_id" : 3, "travel_time" : [ 280, 385, 320, 207, 266, 894 ] }
{ "_id" : 22, "travel_time" : [ 456, 493, 1191, 1266, 388 ] }
{ "_id" : 11, "travel_time" : [ 259, 234, 465, 733, 11 ] }
{ "_id" : 20, "travel_time" : [ 405, 546, 382, 594 ] }
{ "_id" : 4, "travel_time" : [ 517, 988, 209 ] }
{ "_id" : 10, "travel_time" : [ 316, 766, 355, 381, 519 ] }
{ "_id" : 13, "travel_time" : [ 466, 1049, 970 ] }
```

Esto lo podemos hacer con el operador `$push`. Este operador devuelve un array con todos los valores que resultan de aplicar la operación de agrupación.

```
db.usage.aggregate([
  {$limit: 100},
  {
  $group:{
    _id: "$idunplug_station",
    travel_time: { $push: "$travel_time"}
  }
  }
])
```

> Para hacer legible la salida, hemos utilizado `$limit` para hacer la operación solamente sobre los 100 primeros documentos.

Un *stage* importante del *aggregation pipeline* es `$project`. Este permite hacer transformaciones sobre los datos. Es útil, por ejemplo, para trabajar con fechas que tienen sus propios operadores (bastante intuitivos).

```
db.usage.aggregate([
  {$limit: 5},
  {$project: {
    unplug_month: { $month: "$unplug_hourTime"},
    unplug_day: { $dayOfMonth: "$unplug_hourTime"},
    unplug_year: { $year: "$unplug_hourTime"}
  }
  }
])
```

Podemos utilizar `$project` como fase previa a utilizar `$group`

```
db.usage.aggregate([

  {$project: {
    unplug_month: { $month: "$unplug_hourTime"},
    unplug_day: { $dayOfMonth: "$unplug_hourTime"},
    unplug_year: { $year: "$unplug_hourTime"}
  }
  },
  {$group: {_id: {month: "$unplug_month", day: "$unplug_day"}, cuenta: {$sum: 1}}}
])
```

Aunque también podemos utilizar los operadores dentro del propio `$group`

```
db.usage.aggregate([
  {
    $group:
    {
      _id: {month: { $month: "$unplug_hourTime"}, day: { $dayOfMonth: "$unplug_hourTime"}},
      cuenta: {$sum: 1}
    }
  }
])
```

Ejercicio: utilizando la última sentencia, haz que el resultado esté ordenado primero por mes y luego por día del mes.

5 Datos de situación de las estaciones

Ya sabemos que en MongoDB no existe el concepto de esquema como en las bases de datos relacionales y que cada documento puede tener una estructura distinta. Esto es un problema cuando no hemos formado parte de la arquitectura de los datos o no están lo suficientemente bien documentados.

Si el tamaño de los documentos es relativamente pequeño, podemos utilizar `db.coleccion.find()` para hacernos una idea de la estructura (siendo conscientes de que cada documento puede variar su estructura).

Sin embargo, esto no siempre es posible. La colección `stations` que hemos creado es uno de estos casos. Si ejecutamos `db.stations.findOne()` obtenemos un documento bastante largo que nos impide conocer la estructura.

Podemos conocer los campos de este primer documento así

```
Object.keys(db.stations.findOne())
```

Como vemos, tiene dos campos: `_id` y `stations`.

Ejercicio: ¿De cuántos documentos está compuesto el campo `stations` del primer documento? Pista: utiliza `$size`.

5.1 Trabajando con arrays

Para seguir haciéndonos la idea de la estructura de la colección, vamos a quedarnos con solamente los tres primeros elementos del campo `stations`. Para ello, tenemos el operador `$slice` que devuelve parte de un array. La siguiente sintaxis devuelve los `<n>` primer elementos del `<array>` (si `<n>` es negativo, devuelve los `<n>` últimos elementos).

```
{ $slice: [ <array>, <n> ] }
```

La operación que queremos sería

```
db.stations.aggregate(
  {$limit: 1},
  {$project:{stations: {$slice: ["$stations", 3]}}}
  )
```

Ejercicio: revisando la documentación de `$slice`. ¿Cómo devolverías los documentos del 15 al 18?

Imaginemos que queremos extraer la información de solamente la estación de la dirección `Serrano 210`. Podríamos intentar hacerlo así

```
db.stations.find({"stations.name": "Serrano 210"})
```

Esta sentencia nos devuelve todos los documentos donde aparezca `Serrano 210` que, en este caso, es la colección entera.

El operador `$elemMatch` limita el contenido de un campo *array* para devolver solo aquel **primer** elemento que contenga la condición expresada. Es decir, la siguiente sentencia nos devolverá solo los documentos de `stations` con la dirección `Serrano 210`:

```
db.stations.find({}, {stations: {$elemMatch: {name: "Serrano 210"}}}).limit(3).pretty()
```

Ejercicio: Escribe una sentencia que devuelva lo siguiente

```
{
    "_id" : "2018-12-01T00:30:12.524146",
    "stations" : [
        {
            "name" : "Paseo de las Delicias",
            "free_bases" : 19
        }
    ]
}
{
    "_id" : "2018-12-01T01:30:12.684386",
    "stations" : [
        {
            "name" : "Paseo de las Delicias",
            "free_bases" : 9
        }
    ]
}
{
    "_id" : "2018-12-01T03:30:15.860607",
    "stations" : [
        {
            "name" : "Paseo de las Delicias",
            "free_bases" : 0
        }
    ]
}
```

Una pregunta que nos gustaría responder sobre la base de datos es **cuántas bases libres hay de media en cada estación**. La estructura de los documentos complica un poco la tarea, ya que tenemos un documento por cada instante temporal.

Una forma de resolverlo sería llegar a tener **un documento por cada instante temporal y estación**, es decir, tenemos que dividir el array `stations`. Para esta tarea tenemos el stage `$unwind`.

```
db.stations.aggregate({$unwind: "$stations"}, {$limit: 5}).pretty()
```

Ejercicio: Calcula la media de las bases libres para cada estación usando el stage `$unwind`.

6 Índices

Cuando vimos las bases de datos relacionales hablamos de la importancia de los índices para hacer consultas de forma eficiente. En MongoDB también podemos utilizar índices.

MongoDB crea un índice único en el campo `_id` durante la creación de una colección. Para crear un índice utilizamos `createIndex()`.

```
db.stations.createIndex({"stations.name": 1})
```

Para eliminar un índice, podemos usar `dropIndex()`.

7 Vistas

En las bases de datos relacionales existía el concepto de vista. Las vistas son consultas almacenadas (solo la información de cómo hacer la consulta, no los datos en sí mismos) que, de cara al usuario, actuan como si fuese una tabla física. En MongoDB podemos crear vistas de colecciones con `createView()` cuya sintaxis es

```
db.createView(<view>, <source>, <pipeline>, <options>)
```

Donde `<view>` es el nombre que le queremos dar a la vista entre comillas, `<source>` el nombre de la colección de la que queremos crear una vista y `<pipeline>` un *array* tal y como lo usaríamos en un *aggregation pipeline*.

```
db.createView(
  "stations2",
  "stations",
  [{$unwind: "$stations"}]
)
```

Ya podríamos acceder a la nueva vista:

```
db.stations2.find().limit(3).pretty()
```

> Ten en cuenta que `createView()` es un método de la base de datos. Por eso escribimos `db.createView()` y no `db.coleccion.createView()`.

8 Joins

Aunque en MongoDB se promueve una estructura *desnormalizada* de los datos, es posible hacer cruces en el caso de que sea necesario.

Vamos a hacer un ejemplo entre todos. Primero, vamos a crear una vista con el maestro de las estaciones. **Escribe** una sentencia que genere una vista llamada `info_estaciones` que devuelva

```
{ "name" : "Puerta del Sol A", "_id" : 1 }
{ "name" : "Puerta del Sol B", "_id" : 2 }
{ "name" : "Miguel Moya", "_id" : 3 }
{ "name" : "Plaza Conde Suchil", "_id" : 4 }
{ "name" : "Malasaña", "_id" : 5 }
...
```

Ahora vamos a añadir la dirección de la estación de origen para cada documento de la colección `usage` a través de el stage `$lookup`. Su sintaxis es

```
{
   $lookup:
     {
       from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection>,
       as: <output array field>
     }
}
```

Y podemos hacerlo como

```
db.usage.aggregate(
  {$limit: 1},
  { $lookup: {
      from: "info_estaciones",
      localField: "idunplug_base",
      foreignField: "_id",
      as: "direccion"
      }
  }
).pretty()
```

9 Creación de nuevas colecciones

Podemos crear una nueva colección a partir del resultado de `aggregate()`. Para ello tenemos que utilizar el *stage* `$out`:

```
db.usage.aggregate([
  {
    $group:
    {
      _id: {month: { $month: "$unplug_hourTime"}, day: { $dayOfMonth: "$unplug_hourTime"}},
      cuenta: {$sum: 1}
    }
  },
  {$out: "usage_month_day"}
])
```