Práctica - NoSql

Práctica para utilizar alguna de la herramientas NoSql vistas en clase.

Redis

Realizar el tutorial que provee la página

https://try.redis.io/

Este tutorial permite utilizar un redis online no persistente para poder probar el uso de Redis.
Además tiene una serie de ejercicios para poder conocer el uso de los diferentes tipos de valores que se pueden usar. Para ir pasando entre ejercicio y ejercicio se utiliza el comando Next.
Una vez terminado el tutorial se puede seguir probando pero una vez cerrado el browser la información guardada se pierde.

Mongo

Al igual que con Redis utilizaremos una página que nos provee un entorno tipo sandbox (no persistente y de prueba) para probar el uso de Mongo.
Para utilizarlo acceder a:

https://www.tutorialspoint.com/mongodb_terminal_online.php

La idea es seguir los ejercicios enumerados a continuación, los mismos intenta explicar los usos de los comandos proporcionado ejemplos a ejecutar. Cada comando está en un cuadro para facilitar su copia para llevarlo a la consola de mongo

1. **Ingreso a la consola de mongo:**

   ```
   mongo
   ```

2. **Uso de bases y colecciones**:

   a. Primero veamos las bases de datos existen:

      ```
      show databases
      ```

   b. Ninguna es la que queremos usar así que, indiquemos cual queremos usar (por más que no exista se puede decir quiero usar la base hdbd):

```
use hdbd
```

```
show databases
```

A pesar de que estamos en la base, la misma “no existe" porque hasta que no se le cree algún contenido la misma no será considerada “creada".

c. Creemos una colección, y veamos si se creó la base de datos y la colección :

```
db.createCollection("series")
```

```
show databases
```

```
show collections
```

d. Veamos el contenido de la misma

```
db.series.find()
```

3. **Inserción**:
   a. Insertamos un documento y chequeamos que el mismo se encuentra utilizando el find sin parámetros (o sea diciendo “quiero ver todos los documentos”)

```
db.series.insert({"name":"Games Of Thrones",  "abbr": "GOT", "channel":"HBO"})
```

```
db.series.find()
```

b. Agregar .pretty() al final hace que el json sea más legible

```
db.series.find().pretty()
```

c. Probemos insertar otra vez el mismo documento

```
db.series.insert({"name":"Games Of Thrones",  "abbr": "GOT", "channel":"HBO"})
```

```
db.series.find()
```

Se puede realizar porque nada me impide insertar dos documentos “iguales”. Lo que varía en cada uno es el _id autogenerado.

d. **Algo importante** la inserción de un documento en una colección que no existe provoca que se cree la colección. Ejemplo creamos la colección movies insertando una película y chequeamos que existe y su contenido:

```
db.movies.insert({ title: 'Matrix', description: 'Pelicula muy volada',  director: 'Los Wachowski', tags: ['sci fy', 'neo', 'red pill'],likes: 10000})
```

```
show collections
```

```
db.movies.find({})
```

4. **Búsqueda simple**: El comando find recibe un documento como parámetro. El mismo describe las condiciones que quiero que cumplan los documentos que busco. En un caso de igualdad simple simplemente se pone el campo y el valor. Ejemplo:

```
db.movies.find({ title: 'No existe'})
```

```
db.movies.find({ title: 'Matrix'})
```

5. **Borrado**:
   a. Para borrar un documento de usa el comando remove pasando como parámetro la condición (al igual que en el find).

```
db.movies.remove({ title: 'Matrix'})
```

   b. En caso de querer borrar una colección se utiliza el comando drop:

```
db.movies.drop()
```

6. **Update**:
   a. Probemos modificar uno de esos documentos para ello necesitamos una búsqueda que lo identifique (hay que reemplazar el valor del objectId por el que se les haya generado):
   Para ello primero tenemos que saber que una búsqueda se realiza pasando un documento con el campo

```
db.series.find({_id: ObjectId("5d17f8af3f75e18581809708")})
```

   b. Ahora si modificamos, recordando que el update indica dos documentos el primero es el que me dice cual documento se modifica y el segundo indica que se modifica:

```
db.series.update({_id: ObjectId("5d17f8af3f75e18581809708")},
{"name":"The Walking Dead",  "abbr": "TWD",
"channel":"AMC"}})
```

c. Detalles del update:
   i. Por defecto solo modifica un solo documento, en caso de que la query de múltiples documentos y se quieran modificarlos todos hay que agregar un tercer parámetro (con el valor **multi: true**).
   ii. Por otro lado si quiero modificar una parte del documento se debe utilizar el operador **$set** en el segundo parámetro (si no el documento se pisa con lo que puse de segundo parámetro).

   Para ejemplificar esto vamos a agregarle un campo “popularity” con el valor 0 (con $set) a todos los documentos (por eso el documento de búsqueda es un documento vacío {}).

```
db.series.update({}, {$set : {"popularity": 0}} ,  { multi:
true })
```

d. Incrementemos la popularidad de GOT a 60 usando dos operaciones de incremento (50 y 10)y modifiquemos la de walking dead directamente a 100.

```
db.series.update({"abbr":"TWD"}, {$set:{"popularity": 100}})
db.series.find({})
db.series.update({"abbr":"GOT"}, {$inc:{"popularity": 50}})
db.series.find({})
db.series.update({"abbr":"GOT"}, {$inc:{"popularity": 10}})
db.series.find({})
```

e. Más operadores de actualización se pueden encontrar aquí.

7. Más sobre búsquedas:
   a. Primero queries por igualdad donde describo el campo y el valor que espero que tenga

```
db.series.find({"popularity": 10})
db.series.find({"popularity": 60})
db.series.find({"popularity": 100})
```

   b. Find puede recibir un segundo parámetro que es la proyección, o sea cuales campos quiero ver.

```
db.series.find({"popularity": 100}, {"abbr": 1})
```

   En este caso digo quiero ver el campo abbr (1 significa ver, 0 significa no ver).
   Cuando se agrega una proyección solo se muestran los campos específicamente indicados, excepto el _id, que hay que explicitamente decir "no quiero verlo".

```
db.series.find({"popularity": 100}, {"_id": 0, "abbr": 1})
```

   c. Se puede consultar con operadores relacionales, por ejemplo preguntar mayor, menor, etc (más operadores aquí).

```
db.series.find({"popularity": {$gt: 60} })
db.series.find({"popularity": {$gte: 60} })
db.series.find({"popularity": {$lte: 100} })
db.series.find({"popularity": {$lt: 100} })
```

   d. Para agregar más condiciones sobre otros campos simplemente agregarlas

```
db.series.find({"abbr" : "TWD", "popularity": {$lt: 100} })
db.series.find({"abbr" : "TWD", "popularity":  100 })
```

   e. Si se quiere que se cumpla al menos una de las dos condiciones se usa el operador **$or** y se arma una lista (usando [ ]), con un documento por cada condición.

```
db.series.find({ $or: [{"abbr" : "TWD"}, {"popularity":
{$lt: 100} }]})
```

8. **Importar:** vamos a importar documentos a una colección nueva para poder probar agregaciones.
    a. Bajar de campus los archivo **account.json y restaurant.json**.
    b. Para importar necesitamos salir de mongo y la consola que estamos usando no lo permite asi que lo mejor **es abrir otro tab en el browser y volver a entrar.**

https://www.tutorialspoint.com/mongodb_terminal_online.php

    c. Hacia la izquierda cliquear en el texto **New Project-20190628** (la fecha cambia).
    d. En la ventana que aparece hacer click en el **botón derecho sobre la carpeta root** y elegir la opción **upload file**.
    e. Subir el archivo **account.json**.
    f. importar el archivo en la colección account:

mongoimport --db hdbd --collection accounts --file accounts.json
mongoimport --db hdbd --collection restaurants --file restaurant.json

    g. volver a entrar en mongo.

mongo

9. Analizar la información
    a. Revisemos primero que se crearon las colecciones.

`show databases`
`use hdbd`
`show collections`

    b. ¿Cuántos documentos hay?

`db.restaurants.count()`
`db.accounts.count()`

    c. Vamos a usar accounts primero.Para ver la estructura de los documentos queremos verlos, pero si hacemos un find({}) voy a ver mil asi que mejor quedarnos con los primeros 5 (usando limit). En caso de no usar limit

muestra los primeros 20 y para los ver los siguientes hay que usar el comado **it** hasta terminar.

```
db.accounts.find({}).limit(5).pretty()
```

d. Como vemos hay dos tipos de documentos
   i. unos que tiene un solo campo “index”

```
{
      "_id": ObjectId("5d18cf8212f6ac4909b3a73d"),
      "index": {
            "_id": "6"
      }
}
```

   ii. otros que tienen datos de cuentas de clientes:

```
{
      "_id": ObjectId("5d18cf8212f6ac4909b3a73e"),
      "account_number": 6,
      "balance": 5686,
      "firstname": "Hattie",
      "lastname": "Bond",
      "age": 36,
      "gender": "M",
      "address": "671 Bristol Street",
      "employer": "Netagy",
      "email": "hattiebond@netagy.com",
      "city": "Dante",
      "state": "TN"
}
```

e. El operador **$exists** permite filtrar por los documentos que tengan (o no) algún campo. en este caso saquemos los documentos que tiene el campo index

```
db.accounts.find({"index": {$exists: false}}).limit(5)
```

f. Cuantos documentos tenemos sin “index”.

```
db.accounts.find({"index": {$exists: false}}).count()
```

Sería más simple borrar a los documentos con index, pero vamos a dejarlos para poder usar la condición en las siguientes consultas.

g. Para luego hacer agregaciones vamos a proyectar ciertos campos para tener una idea de los valores
    i. edades

```
db.accounts.find({"index": {$exists: false}}, {"age": 1, _id:0}).limit(50)
```

    ii. ciudades y estados

```
db.accounts.find({"index": {$exists: false}}, {"city": 1, "state": 1, _id:0}).limit(50)
```

10. **Agregaciones**: Las agregaciones se armar a partir de una lista de operaciones que se le mandan al comando aggregate algunas de las operaciones posibles son (documentación sobre agregaciones):
    a. **$project** : se utiliza para modificar el conjunto de datos de entrada, añadiendo, eliminando o recalculando campos para que la salida sea diferente.
    b. **$match**: filtra la entrada para reducir el número de documentos, dejando solo los que cumplan las condiciones establecidas.
    c. **$limit**: restringe el número de resultados al número indicado.
    d. **$skip:** ignora un número determinado de registros, devolviendo los siguientes.
    e. **$unwind**: convierte un array para devolverlo separado en documentos.
    f. **$group:** agrupa documentos según una determinada condición.
    g. **$sort**: ordena un conjunto de documentos según el campo especificado.
    h. **$geoNear:** utilizado con datos geoespaciales, devuelve los documentos ordenados por proximidad según un punto geoespacial.
11. Lo primero que vamos a hacer es lograr el mismo “filtrado” que hicimos en el find. O sea sacar los documentos que no tengan index (o sea que queden las accounts válidas), para lo cual hay que usar el operador **$match**. Además limitemos la salida para que no tengamos que recorrer los 1000 registros (usando **$limit**).

```
db.accounts.aggregate( [
    { $match : {"index": {$exists: false}}},
    { $limit: 30}
])
```

12. Confirmamos que el filtro anterior da 1000 como el find (usando **$count**):

```
db.accounts.aggregate( [
    { $match : {"index": {$exists: false}}},
```

```
    { $count: "cantidad de cuentas validas"}
])
```

13. ¿Cuántos usuarios hay por género? O sea agrupamos (con **$group**), por el campo gender (se agrupa lo que esté en el campo _id del grupo), y genero el campo **total** sumando 1 por cada valor.

```
db.accounts.aggregate( [
    { $match : {"index": {$exists: false}}},
    { $group: {
            _id: "$gender",
            total: {
                $sum: 1
                }
             }
       }
])
```

14. Con el mismo agrupamiento hagamos otra agregación. Calculemos el promedio de edad por género.

```
db.accounts.aggregate( [
    { $match : {"index": {$exists: false}}},
    {$group: {
            _id: "$gender",
            "edad promedio": {
                $avg: "$age"
                }
             }
       }
])
```

15. Contemos por ciudad y veamos si hay más de una persona por ciudad, para eso ordenamos descendentemente (-1) y limitamos a los primeros 10

```
db.accounts.aggregate( [
    {$match : {"index": {$exists: false}}},
    {$group:{
             _id: {"city": "$city"},
             total: { $sum: 1}
            }
      },
      { $sort : {total : -1} },
      { $limit: 10}
])
```

Solo hay una ciudad con 2 accounts así que mucho no sirve

16. Para poder hacer alguna cuenta más potente proyectemos las edades para poder obtener la década de cada uno

```
db.accounts.aggregate( [
   {$match : {"index": {$exists: false}}},
   {$project:{_id:0, gender:1,decade:{$trunc:{$divide:["$age", 10]
}}}},
   { $sort : {decade : -1} },
   { $limit: 10}
])
```

17. Agrupemos para contar las cuentas por genero y década.

```
db.accounts.aggregate([
 {$match : {"index": {$exists: false}}},
 {$project: {_id:0, gender: 1, decade:{ $trunc: {$divide: [
"$age", 10 ] }}}},
 {$group:{_id: { "gender": "$gender", "decade": "$decade"}, total:
{$sum: 1} } }
 ])
```

18. Sobre la query anterior vamos a agregar un nuevo match para filtrar las que total sea mayor a 100

```
db.accounts.aggregate([
 {$match : {"index": {$exists: false}}},
 {$project: {_id:0, gender: 1, decade:{ $trunc: {$divide: [
"$age", 10 ] }}}},
 {$group:{_id: { "gender": "$gender", "decade": "$decade"}, total:
{$sum: 1} } },
  {$match : {"total": {$gt: 100}}},
 ])
```

19. Para la colección restaurants realizar las siguientes queries:
    a. Listar los primeros 5 restaurants de londre ordenados por rating.
    b. Listar los restaurants que estén repetidos en el set de datos (con que el nombre se repita alcanza).
    c. Indicar el promedio de rating por tipo de comida.
    d. Realizar la misma consulta anterior pero solo para los restaurants de Manchester.
    e. Cual es el tipo de comida más común.