Instituto Tecnológico de Buenos Aires

Informe TPO - Grupo 5

(20241Q) 72.41 - Base de Datos II - Comisión: S

Integrantes:

Carrica, Florencia [1] - 62040

Ferrutti, Francisco Marcos [2] - 62780

Perri, Lucas David [3] - 62746

Docentes:

Guillermo Rodríguez

Cecilia Rodríguez Babino

Fecha de entrega: 11 de Junio de 2024

[1] fcarrica@itba.edu.ar
[2] fferrutti@itba.edu.ar
[3] lperri@itba.edu.ar

Índice

Introducción

El presente informe detalla el desarrollo de los ejercicios del trabajo práctico obligatorio de la materia Base de Datos II. Se verá en detalle los comandos ejecutados para cada base de datos (MongoDB, Neo4j y Redis), junto con su explicación. Para preparar el ambiente de trabajo, por favor leer el archivo README.md adjuntado en el repositorio de la entrega.

Ejercicio 1

Primero, para realizar el item A, importamos los datos a la colección tpDb utilizando el comando mongoimport:

```
mongoimport --headerline --db tpDb --collection albumlist --type csv <
    ../../data/albumlist.csv
```

Luego debemos indicar qué base de datos vamos a utilizar. Llamaremos tpDb a la base de datos que utilizaremos. Para crearla (y seleccionarla) corremos el siguiente comando:

```
use('tpDb');
```

Para resolver el item B (cantidad de álbumes por año, ordenados de manera descendente) utilizamos los operadores de agregación $group y $sort en la operación aggregate de la siguiente manera:

```
db.getCollection('albumlist').aggregate([
    {
        $group: {
            _id: "$Year",
            cantidad: {$sum: 1}
        }
    },
    {
        $sort: {
            cantidad: -1
        }
    }
]);
```

Para resolver el item C (agregarle a cada documento un nuevo atributo llamado "score"que sea 501-Number) lo que hacemos es utilizar la operación updateMany con los operadores de agregación

$set y $substract:

```
db.getCollection('albumlist').updateMany(
    {},
    [{
        $set: {
            score: {
                $subtract: [501, "$Number"]
            }
        }
    }]
);
```

Para resolver el item D (realizar una consulta que muestre el "score"de cada artista) utilizamos la operación find sin parámetros en la query.

Mientras que en el parámetro de proyección de la operación find, indicamos lo siguiente. Con '_id: 0' nos aseguamos de que no se mueste el ID que asigna mongo a sus documentos y con 'Artista: 1' y 'score: 1' indicamos que se proyecten estos campos:

```
db.getCollection('albumlist').find(
    {},
    {
        _id: 0,
        Artist: 1,
        score: 1
    }
);
```

Ejercicio 2

Decidimos realizar el ejercicio utilizando la herramienta de línea de comandos de Neo4j: Cypher. En primer lugar, cargamos los datos utilizando el script indicado en la página (ver archivo /src/ejercicio2/.load.cypher).

Luego, para responder al item a (cantidad de productos en la base de datos), ejecutamos la funcion count() sobre el match de todos los productos:

```
MATCH (p:Product)
RETURN count(p) AS total
```

Para responder al item B (Precio de "Queso Cabrales") debemos buscar el nodo cuyo nombre de producto sea "Queso Cabrales", y retornar su precio:

```
MATCH (p:Product {productName: "Queso Cabrales"})
RETURN p.unitPrice
```

Para el item C (cantidad de productos que pertenecen a la categoría Çondiments") lo que hacemos es buscar aquellos nodos "Product"que tienen una realción del tipo "PART_OFçon un nodo vecino de tipo categoría con nombre Çondiments". Y luego retornamos la cantidad con la función count():

```
MATCH (p:Product)-[:PART_OF]->(c:Category {categoryName: "Condiments"})
RETURN count(p) AS numberOfProducts
```

Cabe destacar que el código anterior sólo matchea aquellos productos que tienen una arista dirigida hacia un nodo tipo category. Es decir que no considera aquellas relaciones con una arista dirigida de un nodo Category a un nodo Product.

Para responder al item D (nombre y precio de los productos más caros de los proveedores de UK) debemos buscar los nodos tipo Supplier que tienen el atributo country con valor ÜKž que tienen un vecino

```
MATCH (s:Supplier {country: "UK"})-[:SUPPLIES]->(p:Product)
RETURN p.productName, p.unitPrice
ORDER BY p.unitPrice DESC
LIMIT 3
```

Al igual que ocurre con el item C, sólo se se produce un match para aquellas aristas que salen de los nodos tipos Supplier hacia un nodo tipo Product, y no viceversa.

Ejercicio 3

Para realizar el ejercicio 3, primero debemos cargar los datos del archivo bataxi.csv. Para ello, recorremos las filas del archivo bataxi.csv, ejecutando por cada una de ellas el comando GEOADD, donde los parámetros $6, $7 y $1 son la longitud, latitud y el id del viaje, respectivamente.

```
awk -F ',' 'FNR > 1 {print "GEOADD bataxi "$6" "$7" "$1}' ../../data/bataxi.csv |
    redli
```

Una vez cargados los datos al sorted set bataxi, realizamos el item B (cantidad de viajes generados a 1km de distancia de los lugares especificados) de la siguiente manera.

```
for row in (jq -r '.[] | [.place, .lon, .lat] | @csv' .places.json)
    set arr (string split , $row)

    set place (string unescape $arr[1])
    set longitude $arr[2]
    set latitude $arr[3]

    set count (echo "GEOSEARCH bataxi FROMLONLAT $longitude $latitude BYRADIUS
        1 KM" | redli | count)
    echo "1km from $place: $count"
end
```

Para obtener el número de claves en el conjunto de datos, usamos el comando DBSIZE:

```
echo "Number of keys"
echo "DBSIZE" | redli
```

Para obtener el número de miembros en el conjunto de datos bataxi, usamos el comando ZCOUNT con los límites -inf y +inf para contar todos los miembros:

```
echo "Number of members"
echo "ZCOUNT bataxi -inf +inf" | redli
```

Finalmente, para responder a la pregunta del item E sobre sobre qué estructura de Redis trabaja el GeoADD, es importante destacar que GeoADD utiliza la estructura de conjuntos ordenados de Redis. Esta implementación utiliza una técnica llamada Geohash para poblar el conjunto ordenado. Los bits de latitud y longitud se .entrelazan"para formar un entero único de 52 bits. GeoADD aprovecha esta estructura para almacenar datos geoespaciales, donde cada miembro representa un punto en el espacio geográfico.

Conclusión

A lo largo de este informe, abordamos diferentes aspectos relacionados con el manejo de bases de datos, utilizando tanto MongoDB como Neo4J y Redis. Durante el desarrollo de los ejercicios propuestos, pudimos explorar diversas operaciones de importación, consulta y manipulación de datos.

En el primer ejercicio, nos centramos en la importación y manipulación de datos de álbumes musicales, utilizando operaciones de agregación y actualización en MongoDB para resolver consultas específicas. Luego, en el segundo ejercicio, exploramos el modelado y la consulta de datos de productos y proveedores utilizando Cypher, el lenguaje de consulta de Neo4j. Finalmente, en el tercer ejercicio, profundizamos en el manejo de datos geoespaciales utilizando Redis, específicamente la estructura de conjuntos ordenados.

En resumen, este trabajo nos permitió adquirir experiencia práctica en el manejo de distintos tipos de bases de datos y en la implementación de consultas complejas para resolver problemas específicos. Además, nos proporcionó una visión más amplia sobre las diversas herramientas y tecnologías disponibles para el almacenamiento y gestión de datos.