



Universidad Autónoma de Madrid

Escuela Politécnica Superior

Sistemas Informáticos I

Práctica 3

Iker González Sánchez

Gonzalo Segovia Oblanca

Grupo 1332 - Pareja 8

Noviembre de 2023



Las funciones para las consultas han sido realizadas con el lenguaje de consultas de mongo y quedan de la siguiente forma:

```
def scifi_from_1994_to_1998(mongodb_collection):
    query = {'$and': [{'year': {'$gte': '1994', '$lte': '1998'}}, {'genres': {'$all': ['Sci-Fi']}}]}
    movies = mongodb_collection.find(query)
    for movie in movies:
        print(movie)

def dramas_from_1998_the(mongodb_collection):
    query = {'$and': [{'year': '1998'}, {'genres': 'Drama'}, {'title': {'$regex': 'The$'}}]}
    movies = mongodb_collection.find(query)
    for movie in movies:
        print(movie)

def faye_dunaway_and_viggo_mortensen(mongodb_collection):
    query = {'$and': [{'actors': {'$regex': 'Dunaway, Faye'}}, {'actors': {'$regex': 'Mortensen, Viggo'}}]}
    movies = mongodb_collection.find(query)
    for movie in movies:
        print(movie)
```

Al ejecutar el programa de Python, obtenemos los siguientes resultados.

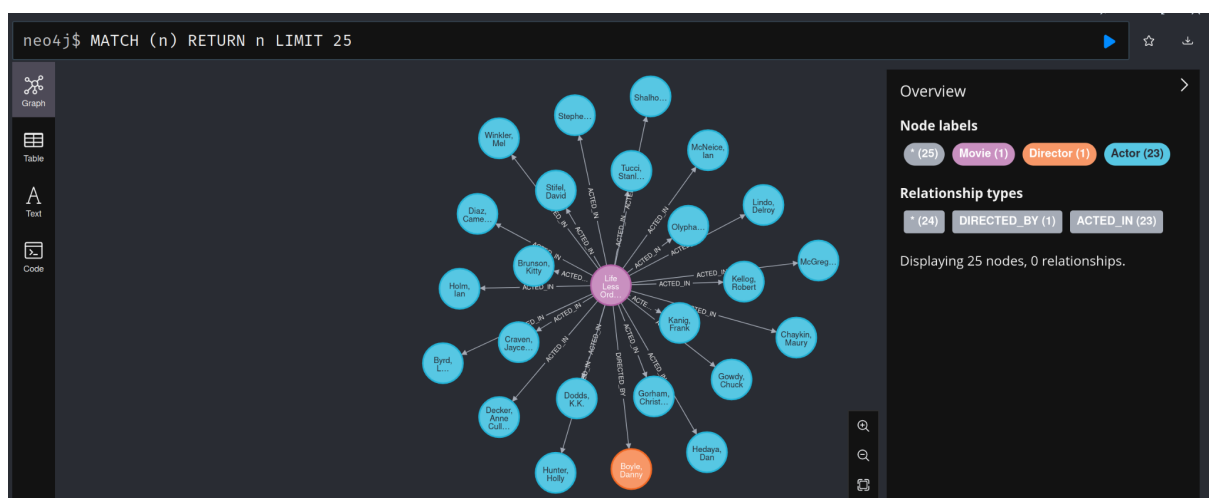
```
Sci-Fi movies from 1994 to 1998:
{'_id': ObjectId('657dfeald842f87897d92c03'), 'title': 'Abre los ojos', 'genres': ['Drama', 'Mystery', 'Romance', 'Sci-Fi', 'Thriller'], 'year': '1994', 'directors': ['Luis Buñuel', 'Luis Buñuel'], 'actors': ['Faye Dunaway', 'Viggo Mortensen'], 'relationships': ['ACTED_IN', 'DIRECTED_BY']}
{'_id': ObjectId('657dfeald842f87897d92c63'), 'title': 'Fifth Element, The', 'genres': ['Action', 'Adventure', 'Sci-Fi', 'Thriller'], 'year': '1998', 'directors': ['Luc Besson'], 'actors': ['Milla Jovovich', 'Bruce Willis'], 'relationships': ['ACTED_IN', 'DIRECTED_BY']}
{'_id': ObjectId('657dfeald842f87897d92c80'), 'title': 'Highlander III: The Sorcerer', 'genres': ['Action', 'Fantasy', 'Sci-Fi'], 'year': '1994', 'directors': ['Russel Mulcahy'], 'actors': ['Michael York', 'Michael York'], 'relationships': ['ACTED_IN', 'DIRECTED_BY']}
{'_id': ObjectId('657dfeald842f87897d92ccd'), 'title': 'Nowhere', 'genres': ['Drama', 'Sci-Fi'], 'year': '1997', 'directors': ['Araki, Gregg'], 'actors': ['Viggo Mortensen', 'Faye Dunaway'], 'relationships': ['ACTED_IN', 'DIRECTED_BY']}
{'_id': ObjectId('657dfeald842f87897d92d09'), 'title': 'Stargate', 'genres': ['Action', 'Adventure', 'Fantasy', 'Sci-Fi'], 'year': '1994', 'directors': ['Roland Emmerich'], 'actors': ['Kurt Russell', 'Val Kilmer'], 'relationships': ['ACTED_IN', 'DIRECTED_BY']}

Dramas from 1998 starting with "The":
{'_id': ObjectId('657dfeald842f87897d92c9d'), 'title': 'Land Girls, The', 'genres': ['Drama'], 'year': '1998', 'directors': ['Leland, David'], 'actors': ['Leland, David'], 'relationships': ['ACTED_IN', 'DIRECTED_BY']}
{'_id': ObjectId('657dfeald842f87897d92cb0'), 'title': 'Man in the Iron Mask, The', 'genres': ['Action', 'Adventure', 'Drama'], 'year': '1998', 'directors': ['Randall Kosslyn'], 'actors': ['Jean-Claude Van Damme', 'Jean-Claude Van Damme'], 'relationships': ['ACTED_IN', 'DIRECTED_BY']}
{'_id': ObjectId('657dfeald842f87897d92cbl'), 'title': 'Man with Rain in His Shoes, The', 'genres': ['Comedy', 'Drama', 'Romance'], 'year': '1998', 'directors': ['John Dahl'], 'actors': ['John Dahl'], 'relationships': ['ACTED_IN', 'DIRECTED_BY']}
{'_id': ObjectId('657dfeald842f87897d92ce6'), 'title': 'Quarry, The', 'genres': ['Drama'], 'year': '1998', 'directors': ['Hänsel, Marion'], 'actors': ['Hänsel, Marion'], 'relationships': ['ACTED_IN', 'DIRECTED_BY']}

Movies with Faye Dunaway and Viggo Mortensen:
{'_id': ObjectId('657dfeald842f87897d92c06'), 'title': 'Albino Alligator', 'genres': ['Crime', 'Drama', 'Thriller'], 'year': '1996', 'directors': ['John Dahl'], 'actors': ['Faye Dunaway', 'Viggo Mortensen'], 'relationships': ['ACTED_IN', 'DIRECTED_BY']}
```

Neo4j

Para la creación de la base de datos para Neo4j, filtramos las 20 películas estadounidenses con más ventas, y después creamos las 3 clases: Actor, Movie y Director; cada uno con los atributos vistos en el dibujo, es decir: para Actor utilizamos actorid y name; para Movie el movieid y name; y para Director el directorid y name. Además creamos las relaciones entre las películas y los actores y directores, y por último, creamos los nodos correspondientes. La base de datos queda creada:



Para la realización de las consultas pedidas se ha utilizado cypher:

Para la primera consulta:


```
// Encontrar actores que no han trabajado con "Winston, Hattie"
MATCH (actor:Actor)
WHERE actor.name <> "Winston, Hattie"

// Encontrar actores que han trabajado con un tercer actor en común
MATCH (actor:Actor)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(coActor:Actor),
      (coActor:Actor)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(thirdActor:Actor)


// Filtrar actores que no han trabajado con "Winston, Hattie" y no son "Winston, Hattie" ni el tercer actor
WHERE NOT (actor:Actor)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(Actor {name: "Winston, Hattie"})
      AND actor.name <> "Winston, Hattie"
      AND actor.name <> thirdActor.name

// Devolver los resultados ordenados alfabéticamente y limitar a 10 resultados
RETURN actor.name
ORDER BY actor.name
LIMIT 10;
```

No obtenemos ningún registro:


Table

(no changes, no records)


Code

Completed after 28 ms.

Para la segunda:

```

// Encontrar pares de personas que han trabajado juntas en más de una película
MATCH (person1)-[:ACTED_IN|DIRECTED_BY]->(movie)<-[:ACTED_IN|DIRECTED_BY]-(person2)

// Filtrar pares de personas distintas
WHERE id(person1) < id(person2)

// Contar el número de películas en las que han trabajado juntas
WITH person1, person2, count(movie) AS collaborations

// Filtrar pares con más de una colaboración
WHERE collaborations > 1

// Devolver los resultados
RETURN id(person1), id(person2), collaborations
ORDER BY collaborations DESC;

```

	id(person1)	id(person2)	collaborations
1	502	1562	477
2	201	1261	164
3	983	2043	152
4	373	1433	146
5	281	1341	117
6	144	1204	114

Para la tercera:

```

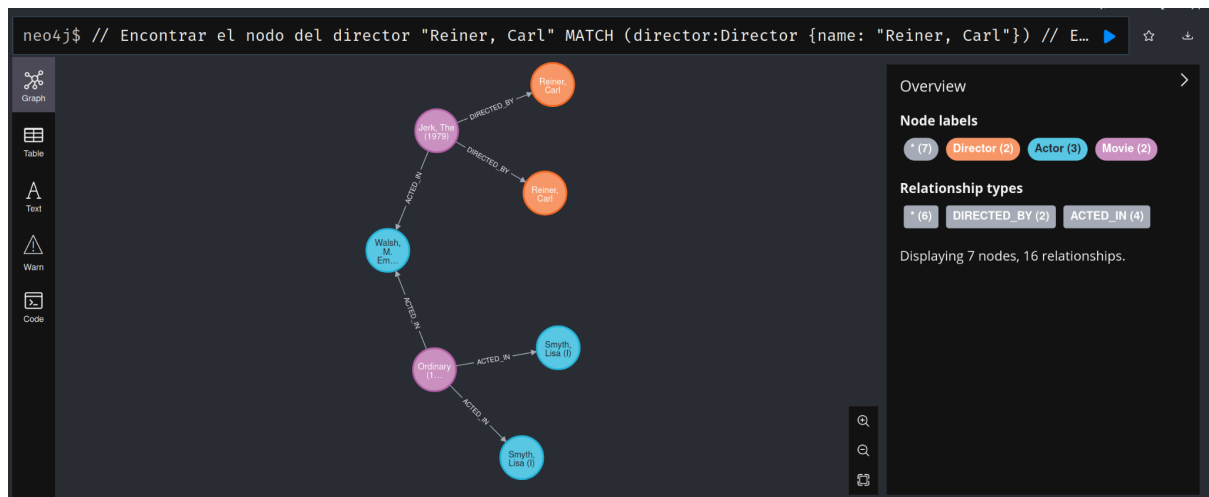
// Encontrar el nodo del director "Reiner, Carl"
MATCH (director:Director {name: "Reiner, Carl"})

// Encontrar el nodo de la actriz "Smyth, Lisa (I)"
MATCH (actress:Actor {name: "Smyth, Lisa (I)"})

// Encontrar el camino mínimo entre el director y la actriz
MATCH path = shortestPath((director)-[*]-(actress))

// Devolver el camino mínimo y la longitud del camino
RETURN path, length(path) AS degrees;

```



Parte 2

Redis

Creación de la base de datos en memoria

Para crear la base de datos de este apartado tenemos que seleccionar los clientes que cumplen que `customers.country == 'Spain'` mediante una query en una sesión de nuestra base de datos. Después, con los resultados de la query creamos la clave y extraemos los valores que queremos reflejar en la base de datos, para después asociar los valores a la clave creada con la función `hset`. Como se puede observar, la base de datos se crea correctamente:

Total: 199			Last refresh: < 1 min	
HASH	customers.rotten.ohsa@gmail.com	No limit	144 B	
HASH	customers.gorgon.strike@potmail.com	No limit	144 B	
HASH	customers.remind.nudism@potmail.com	No limit	144 B	
HASH	customers.taine.daren@mamoot.com	No limit	144 B	
HASH	customers.fist.zero@kran.com	No limit	128 B	
HASH	customers.attest.tattoo@mamoot.com	No limit	144 B	
HASH	customers.oboe.hooey@kran.com	No limit	136 B	
HASH	customers.opener.josh@gmail.com	No limit	144 B	
HASH	customers.puke.groom@mamoot.com	No limit	144 B	
HASH	customers.tung.sexual@potmail.com	No limit	144 B	
HASH	customers.phekda.downy@mamoot.com	No limit	144 B	
HASH	customers.signed.mood@kran.com	No limit	136 B	
HASH	customers.tenon.fry@mamoot.com	No limit	128 B	
HASH	customers.gregg.dicky@kran.com	No limit	136 B	
HASH	customers.queasy.smoggy@mamoot.com	No limit	144 B	

HASH customers.rotten.ohsa@gmail.com		Key Size: 144 B	Length: 3	TTL: No limit	< 1 min	Unicode	Add Fields	
Field	Value							
name	rotten ohsa							
phone	+41 794258283							
visits	87							

Funciones

A continuación, se muestran las distintas funciones y sus pruebas de funcionalidad:

Función que devuelve el email del usuario con mayor cantidad de visitas:

```
def customer_most_visits():
    keys = r.keys('customers:*')
    max_visits = -1
    max_email = None
    for key in keys:
        visits = int(r.hget(key, 'visits'))
        if visits > max_visits:
            max_visits = visits
            max_email = key.decode('utf-8').split(':')[1]
    return max_email

print(customer_most_visits())
```

```
liz.taoist@jmail.com
```

Función que muestra el nombre, el teléfono y número de visitas dado el email:

```
def get_field_by_email(email):
    key = f"customers:{email}"
    name = r.hget(key, 'name').decode('utf-8')
    phone = r.hget(key, 'phone').decode('utf-8')
    visits = int(r.hget(key, 'visits').decode('utf-8'))
    return name, phone, visits

print(get_field_by_email('liz.taoist@jmail.com'))
```

```
('liz taoist', '+60 645897838', 99)
```

Función en python que incrementa una visita dado el correo electrónico:

```
def increment_by_email(email):
    key = f"customers:{email}"
    r.hincrby(key, 'visits', 1)

increment_by_email('liz.taoist@jmail.com')

print(get_field_by_email('liz.taoist@jmail.com'))
```

```
('liz taoist', '+60 645897838', 100)
```

Parte 3

Estudio de transacciones

Para crear la función *borraCiudad*, nos hemos apoyado en la estructura que venía provista en el ejemplo *borraEstado* para la página y la función *delCity*. A continuación, explicaremos el funcionamiento de la función con ayuda de unos ejemplos prácticos:

Transacción sin errores vía sentencias SQL

Ejemplo de Transacción con Flask SQLAlchemy

Ciudad:

☒ Transacción vía sentencias SQL
☐ Transacción vía funciones SQLAlchemy

☐ Ejecutar commit intermedio

☒ Provocar error de integridad

Duerme segundos (para forzar deadlock).

Trazas

1. Ejecución de consultas en el orden correcto
2. Ejecutando: DELETE FROM orderdetail od WHERE od.orderid IN (SELECT o.orderid FROM orders o JOIN customers c ON c.customerid = o.customerid WHERE c.city = 'herman');
3. Ejecutando: DELETE FROM orders WHERE customerid IN (SELECT customerid FROM customers WHERE city = 'herman');
4. Ejecutando: DELETE FROM customers WHERE city = 'herman';
5. Duerme 10 segundos
6. Ejecutando commit

Esta transacción no tiene errores ya que pasamos un parámetro a la función (*bFallo*) para escoger si queremos que se provoquen errores de integridad o no, y, en este caso, dicho parámetro se encuentra a cero. Para que no se produzcan errores, debemos ejecutar las sentencias en orden (trazas 2, 3 y 4). Después, para forzar el deadlock, ejecutamos la función de python *sleep()* con el tiempo especificado. Finalmente, al no producirse errores, se ejecuta el *commit*.

Transacción sin errores vía SQLAlchemy

Ejemplo de Transacción con Flask SQLAlchemy

Ciudad:

☒ Transacción vía sentencias SQL
☐ Transacción vía funciones SQLAlchemy

☐ Ejecutar commit intermedio

☒ Provocar error de integridad

Duerme segundos (para forzar deadlock).

Trazas

1. Ejecución de consultas en el orden correcto
2. Ejecutando: DELETE FROM orderdetail od WHERE od.orderid IN (SELECT o.orderid FROM orders o JOIN customers c ON c.customerid = o.customerid WHERE c.city = 'punjab');
3. Ejecutando: DELETE FROM orders WHERE customerid IN (SELECT customerid FROM customers WHERE city = 'punjab');
4. Ejecutando: DELETE FROM customers WHERE city = 'punjab';
5. Duerme 10 segundos
6. Ejecutando commit

Uno de los parámetros de la función (*bSQL*) sirve para elegir si se quiere ejecutar sentencias SQL o funciones de SQLAlchemy. En este caso, la funcionalidad es idéntica a la anterior pero usando funciones de SQLAlchemy en lugar de sentencias SQL.

Transacción con errores sin commit intermedio

Ejemplo de Transacción con Flask SQLAlchemy

Ciudad:

☒ Transacción vía sentencias SQL
☐ Transacción vía funciones SQLAlchemy

☐ Ejecutar commit intermedio

☒ Provocar error de integridad

Duerme segundos (para forzar deadlock).

Trazas

1. Error provocado en la ejecución de las consultas
2. Ejecutando: DELETE FROM orderdetail od WHERE od.orderid IN (SELECT o.orderid FROM orders o JOIN customers c ON c.customerid = o.customerid WHERE c.city = 'ballys');
3. Ejecutando: DELETE FROM customers WHERE city = 'ballys';
4. Error: (psycopg2.errors.ForeignKeyViolation) update or delete on table "customers" violates foreign key constraint "orders_customerid_fkey" on table "orders" DETAIL: Key (customerid)=(52) is still referenced from table "orders". [SQL: DELETE FROM customers WHERE city = 'ballys;'] (Background on this error at: <https://sqlalche.me/e/20/gkpp>)
5. Ejecutando rollback

En esta transacción, hemos forzado un error de integridad alterando el orden natural de las sentencias SQL. Como se puede observar (traza 4), se produce la excepción que mostramos por pantalla y, posteriormente, se ejecuta el rollback.

Transacción con errores con commit intermedio

Ejemplo de Transacción con Flask SQLAlchemy

Ciudad:

☒ Transacción vía sentencias SQL
☐ Transacción vía funciones SQLAlchemy

☐ Ejecutar commit intermedio

☒ Provocar error de integridad

Duerme segundos (para forzar deadlock).

Trazas

1. Error provocado en la ejecución de las consultas
2. Ejecutando: DELETE FROM orderdetail od WHERE od.orderid IN (SELECT o.orderid FROM orders o JOIN customers c ON c.customerid = o.customerid WHERE c.city = 'chang');
3. Ejecutando commit intermedio
4. Ejecutando commit
5. Comenzando nueva transacción
6. Ejecutando: DELETE FROM customers WHERE city = 'chang';
7. Error: (psycopg2.errors.ForeignKeyViolation) update or delete on table "customers" violates foreign key constraint "orders_customerid_fkey" on table "orders" DETAIL: Key (customerid)=(103) is still referenced from table "orders". [SQL: DELETE FROM customers WHERE city = 'chang;'] (Background on this error at: <https://sqlalche.me/e/20/gkpp>)
8. Ejecutando rollback

En este caso, tenemos el parámetro *bCommit* a uno, por lo que realizamos un commit intermedio, que garantiza que los cambios que se realizan antes del error se hagan efectivos, sin que le afecte el rollback posterior. Después de este commit intermedio, se produce el error y el rollback posterior no afectará a los cambios en la columna *orderdetail*.

Funcionamiento de *commit*

“*Commit*” es una instrucción de transacción que sirve para aplicar los cambios realizados por una o más sentencias SQL en una base de datos. Cuando se ejecuta un commit, se finaliza la transacción actual y se hacen permanentes todos los cambios realizados en la base de datos. También se liberan los recursos bloqueados por la transacción y se permite que otras transacciones accedan a los datos modificados.

Estudio de bloqueos y deadlocks

Hemos creado el archivo updPromo.sql con todas las queries y requisitos:

Crear columna promo en tabla customers:

```
-- Crear la nueva columna "promo" en la tabla "customers"
ALTER TABLE customers
ADD COLUMN IF NOT EXISTS promo DECIMAL(5, 2);
```

Trigger sobre la tabla customers que descuenta los artículos de su cesta o carrito del porcentaje de la columna promo sobre el precio de la tabla products:

```
-- Crear el trigger para aplicar descuento en los artículos al alterar la columna "promo"
CREATE OR REPLACE FUNCTION trg_UpdatePromo_Discount()
RETURNS TRIGGER AS $$
BEGIN
    -- Verificar si la columna "promo" ha cambiado
    IF NEW.promo <> OLD.promo THEN
        -- Actualizar el descuento en los productos de las órdenes
        UPDATE products
        SET price = price * (1 - (NEW.promo / 100))
        FROM orderdetails od
        WHERE products.product_id = od.product_id
        AND od.order_id IN (SELECT order_id FROM orders WHERE customer_id = NEW.customer_id);

        RAISE NOTICE 'Descuento aplicado correctamente en los productos de las órdenes para el cliente %.', NEW.customer_id;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Vincular el trigger a la tabla "customers"
CREATE TRIGGER trg_UpdatePromo_Discount
AFTER UPDATE
ON customers
FOR EACH ROW
EXECUTE FUNCTION trg_UpdatePromo_Discount();
```

Además añadimos los sleeps requeridos justo después de comprobar si ha cambiado la columna promo y después de la actualización:

```
-- Modificar el trigger para agregar un sleep en el momento adecuado
CREATE OR REPLACE FUNCTION trg_UpdatePromo_Discount()
RETURNS TRIGGER AS $$
BEGIN
    -- Verificar si la columna "promo" ha cambiado
    IF NEW.promo <> OLD.promo THEN
        -- Realizar una pausa de 5 segundos antes de la actualización
        PERFORM pg_sleep(5);

        -- Actualizar el descuento en los productos de las órdenes
        UPDATE products
        SET price = price * (1 - (NEW.promo / 100))
        FROM orderdetails od
        WHERE products.product_id = od.product_id
        AND od.order_id IN (SELECT order_id FROM orders WHERE customer_id = NEW.customer_id);

        RAISE NOTICE 'Descuento aplicado correctamente en los productos de las órdenes para el cliente %.', NEW.customer_id;

        -- Realizar otra pausa de 5 segundos después de la actualización
        PERFORM pg_sleep(5);
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

En el proceso de prueba de bloqueos y deadlocks, se ha modificado la ubicación de los tiempos de espera, para lograr provocar un deadlock, donde dos transacciones concurrentes solicitaban recursos mutuos y ninguna podía completarse sin que la otra cediera el registro. Además respecto al apartado g, obtenemos un rollback. Respondiendo

al apartado h, concluimos que existe falta de visibilidad de los resultados debido a las propiedades ACID, especialmente la atomicidad.

Para afrontar estos problemas, sería conveniente hacer más breves las transacciones; también, asegurarnos de que se accede a los recursos en el mismo orden, mediante una secuencia consistente; también se podrían ordenar los recursos antes de adquirirlos o adquirirlos en una sola operación atómica para mayor brevedad. Por último, podría ser útil el uso de puntos de guardado ya que permite que una transacción se recupere desde el último punto guardado en lugar de realizar un rollback completo.