UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

MEMORIA DE LA PRÁCTICA 3

SISTEMAS INFORMÁTICOS

Sergio Larriba Moreno
Javier Valero Velázquez

ÍNDICE

Descripción del material entregado

Hemos entregado 3 carpetas diferentes:

- "**app-mongodb-etl**" - En ella se encuentran todos los archivos solicitados en la práctica en relación a la base de datos documental mongodb:
  - ➢ "*create_mongodb_from_postgresqldb.py*": En este archivo se crea una base de datos documental que contiene una colección de documentos que representará las películas francesas (o de colaboración francesa) en base a la información almacenada en la base de datos relacional que se nos da en el código fuente de la práctica. Los documentos creados tienen la siguiente estructura:

```
moviesDocument.append({
    'title': movieTitle,
    'genres': movieGenres,
    'year': movieYear,
    'directors': movieDirectors,
    'actors': movieActors,
})
```

```
collection_france.update_one({'_id': movie['_id']}, {'$set': {
    'most_related_movies': most_related,
    'related_movies': related
}})
```

Todos estos documentos se encuentran en la colección "france" en la nueva base de datos en MongoDB llamada "si1". Así es como establecemos la conexion a postgres y configuramos el cliente de MongoDB para poder así crear dicha colección e insertar los documentos solicitados:

```
# Configurar el motor de sqlalchemy
"""
Url con:
    Usuario: alumnodb
    Contraseña: 1234
    Base de datos: si1
    host: localhost
"""
db_engine = create_engine("postgresql://alumnodb:1234@localhost/si1", echo=False)

# Configurar mongodb -> mongodb_client es el cliente que usaremos para interactuar con mongodb
mongodb_client = pymongo.MongoClient("mongodb://localhost:27017/")

# Creamos una bas de datos en mongodb llamada si1 y una coleccion llamada france
mongodb = mongodb_client["si1"] # Base de datos que estoy utilizando
collection_france = mongodb["france"]

try:
    # Conexion a la base de datos de postgress
    db_connection = None
    db_connection = db_engine.connect()
```

  - ➢ "*mongodb_queries.py*": Este archivo contiene 3 consultas que se aplican sobre la nueva base de datos de Mongodb:

1. Una tabla con toda la información de aquellas películas (documentos) de ciencia ficción comprendidas entre 1994 y 1998:

```
query = {"year":1998, "genres":"Drama", "title":{'$regex':'The'}}
films_drama_1998 = list(collection_france.find(query))
```

2. Una tabla con toda la información de aquellas películas (documentos) que sean dramas del año 1998, y que empiecen por la palabra “The”:

3. Una tabla con toda la información de aquellas películas (documentos) en las que Faye Dunaway y Viggo Mortensen hayan compartido reparto:

```
query = {"$and": [{"actors": "Dunaway, Faye"}, {"actors": "Mortensen, Viggo"}]}
films_faye_viggo = list(collection_france.find(query))
```

- **“app-neo4j-etl”** - En esta carpeta se encuentran los archivos relacionados con la creación y conexión de la nueva base de datos en neo4j. Además, se entregan 3 ficheros .cypher que sirven para ejecutar las consultas solicitadas en la base de datos en neo4j.
  - ➢ “*create_neo4jdb_from_postgresqldb.py*” - En este archivo se crea una base de datos basada en grados que contiene las 20 películas estadounidenses más vendidas almacenadas en la base de datos relacional. Se crea una nueva base de datos en neo4j llamada “si1” con contraseña para el usuario “neo4j”: “si1-password”. Más adelante explicaremos con más detalle dicho archivo.
  - ➢ “*winston-hattie-co-co-actors.cypher*” - Consulta que devuelve una tabla con 10 actores ordenados alfabéticamente que no han trabajado con “Winston, Hattie”, pero que en diferentes momentos han trabajado con un tercero en común:

```
MATCH (a1:Actor)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(a2:Actor)
WHERE a1.name <> "Winston, Hattie" AND a2.name <> "Winston, Hattie"
RETURN DISTINCT a1.name AS Actor
ORDER BY a1.name ASC
LIMIT 10
```

  - ➢ “*pair-persons-mostoccurrences.cypher*” - Consulta que devuelve una tabla que muestre en cada fila pares de personas que han trabajado juntas en más de una película, sin distinguir categoría de actores o directores:

```
MATCH (p1:Person)-[:ACTED_IN|DIRECTED]->(:Movie)<-[:ACTED_IN|DIRECTED]-(p2:Person)
WHERE p1 <> p2
WITH p1, p2, count(*) AS occurrences
RETURN p1.name AS Person1, p2.name AS Person2, occurrences
ORDER BY occurrences DESC
```

  - ➢ “degrees-reiner-to-smyth.cypher” - Consulta que halla el camino mínimo por el cual el director “Reine, Carl” podría conocer a la actriz “Smyth, Lisa(I)”:

```
MATCH (d:Director {name: "Reiner, Carl"}), (a:Actor {name: "Smyth, Lisa (I)"})
MATCH path = shortestPath((d)-[*]-(a))
RETURN path
```

- **“app-redis-etl”** - Esta carpeta contiene todos los documentos solicitados en el enunciado de la práctica relacionado con la base de datos en memoria en Redis. Solamente contiene 1 archivo:
  - ➢ “*create_redis_from_postgresqldb.py*” - Este archivo crea una nueva base de datos en redis llamada “si1” que guarda la cantidad de visitas de los clientes a la página web de dicha base de datos. Se lleva a cabo usando la estructura de datos *hashes* proporcionada por Redis. Para cada cliente que sea de España, almacena toda su información en la base de datos Redis. La información almacenada incluye el nombre del cliente, el teléfono y un número aleatorio de visitas entre 1 y 99. La clave utilizada para almacenar información de cada cliente es “customers:email”.

```python
def create_redis_db():
    print("Creando base de datos Redis...")
    # Obtener clientes de la base de datos PostgreSQL que son de España
    query = customers_table.select().where(customers_table.c.country == 'Spain')
    result = session.execute(query)
    customers = result.fetchall()

    # Almacenar clientes en la base de datos Redis
    for customer in customers:
        email = customer.email
        name = customer.firstname + " " + customer.lastname
        phone = customer.phone
        visits = random.randint(1, 99)

        hash_key = f"customers:{email}"
        redis_db.hset(hash_key, "name", name)
        redis_db.hset(hash_key, "phone", phone)
        redis_db.hset(hash_key, "visits", visits)

    print("Base de datos Redis creada!.")
```

Además, este archivo contiene 3 funciones adicionales:

1. Una función que incremente una visita dado el correo electrónico:

```python
def increment_by_email(email):
    hash_key = f"customers:{email}"
    redis_db.hincrby(hash_key, "visits", 1)
```

2. Una función que devuelve el email del usuario con mayor cantidad de visitas:

```python
def customer_most_visits():
    customers = redis_db.keys("customers:*")
    max_visits = 0
    max_email = ""

    for customer in customers:
        visits = int(redis_db.hget(customer, "visits"))
        if visits > max_visits:
            max_visits = visits
            max_email = customer.decode("utf-8").split(":")[1]

    return max_email
```

3. Una función que muestre el nombre, el teléfono y el número de visitas dado el email:

```python
def get_field_by_email(email):
    hash_key = f"customers:{email}"
    name = redis_db.hget(hash_key, "name").decode("utf-8")
    phone = redis_db.hget(hash_key, "phone").decode("utf-8")
    visits = int(redis_db.hget(hash_key, "visits"))

    return name, phone, visits
```

Resultados de cada ejercicio solicitado y su discusión

---

1 - NoSQL

1.1 - Base de datos documental

Apartados a) y b)

En este apartado se nos pide la creación de una base de datos documental que contenga una colección de documentos llamada “france” que contenga películas francesas o de colaboración francesa. Los documentos deben de tener la siguiente estructura:

- Título sin el año - cadena
- Géneros - listado de géneros
- Año - número
- Directores - listado de directores
- Actores participantes - listado de actores
- Hasta 10 películas más actuales y relacionadas - listado de títulos y años

Para ello hemos desarrollado el script “create_mongodb_from_postgresqldb.py”.

```python
# Configurar el motor de sqlalchemy
"""
Url con:
    Usuario: alumnodb
    Contraseña: 1234
    Base de datos: si1
    host: localhost
"""
db_engine = create_engine("postgresql://alumnodb:1234@localhost/si1", echo=False)

# Configurar mongodb -> mongodb_client es el cliente que usaremos para interactuar con mongodb
mongodb_client = pymongo.MongoClient("mongodb://localhost:27017/")

# Creamos una bas de datos en mongodb llamada si1 y una coleccion llamada france
mongodb = mongodb_client["si1"] # Base de datos que estoy utilizando
collection_france = mongodb["france"]

try:
    # Conexion a la base de datos de postgress
    db_connection = None
    db_connection = db_engine.connect()
```

```
    # Creamos una tabla con las peliculas francesas de la base de datos
    print("Creando tabla de peliculas francesas...")
    france_table = text("CREATE TEMPORARY TABLE actualFRMovies AS (\
                        SELECT m.movieid, m.movietitle, m.year\
                        FROM imdb_moviecountries mc\
                        JOIN imdb_movies m ON mc.movieid = m.movieid\
                        WHERE mc.country = 'France'\
                        ORDER BY m.year DESC\
                        )"
                        )

    # Ejecuta la consulta CREATE TABLE
    db_connection.execute(france_table)
```

1 - Nos conectamos a la base de datos de PostgreSQL “si1” en el host “localhost” con el usuario “alumnodb” con contraseña “1234”.
2 - Nos conectamos a la base de datos de MongoDB en el host “MongoDB” en el host “localhost” en el puerto “27017”.
3 - Creamos una base de datos llamada “si1” y una colección llamada “france”.
4 - Obtenemos mediante la query “france_table” una tabla que contiene todas las películas francesas de la base de datos, para poder así posteriormente, obtener los datos necesarios de dichas películas.

```
# Lista con las peliculas francesas de la base de datos
movieList = list(db_connection.execute(text("SELECT * FROM actualFRMovies")))
recentFRIds = list(db_connection.execute(text("SELECT movieid FROM actualFRMovies")))

moviesDocument = []
print("Obteniendo los datos de las peliculas en Postgres y pasandolos a mongoDB...")
# Conseguimos todos los datos de las peliculas francesas para pasarlos a mongodb
for movie in movieList:
    # Obtenemos el id, titulo, genero, año, directores y actores de las peliculas
    movieId = movie[0]
    movieTitle = movie[1]
    movieYear = int(movie[2])

    # Obtenemos los generos de la pelicula
    db_movieGenres = list(db_connection.execute(text("select genre from imdb_moviegenres\
                                                     where movieid = '" + str(movieId) + "'")))
    movieGenres = []
    for genre in db_movieGenres:
        movieGenres.append(genre[0])

    # Obtenemos los directores de la pelicula
    db_movieDirectors = list(db_connection.execute(text("select directorname from imdb_directormovies, imdb_directors\
                                                        where movieid = '" + str(movieId) + "' and\
                                                        imdb_directormovies.directorid = imdb_directors.directorid")))
    movieDirectors = []
    for directorname in db_movieDirectors:
        movieDirectors.append(directorname[0])

    # Obtenemos los actores de la pelicula
    db_movieActors = list(db_connection.execute(text("select actorname from imdb_actormovies, imdb_actors\
                                                     where movieid = '" + str(movieId) + "' and\
                                                     imdb_actormovies.actorid = imdb_actors.actorid")))
    movieActors = []
    for actorname in db_movieActors:
        movieActors.append(actorname[0])

    # Creamos y añadimos a la coleccion el documento para la pelicula en mongoDB
    moviesDocument.append({
        'title': movieTitle,
        'genres': movieGenres,
        'year': movieYear,
        'directors': movieDirectors,
        'actors': movieActors,
    })

# Insertamos las peliculas a mongodb
collection_france.insert_many(moviesDocument)
```

5 - Para cada película, obtenemos información específica en la que se incluyen: los géneros, directores y actores de cada película. Con ello, creamos un documento para cada película y lo añadimos a una lista de documentos llamada “moviesDocument” para su posterior inserción en la base de datos MongoDB.

```
# Obtenemos las peliculas mas relacionadas y relacionadas
max_related_movies = 10
movies = list(collection_france.find())
for movie in movies:
    # Coincidencia 100% -> Busco todas las peliculas que tengan los mismos generos que la pelicula actual
    most_related = collection_france.find(
        {
            'genres': {'$all': movie['genres']},
            '_id': {'$ne': movie['_id']}
        },
        {
            'title': 1,
            'year': 1,
            '_id': 0
        }
    ).sort('year', pymongo.DESCENDING).limit(max_related_movies)
    most_related = list(most_related)

    # Coincidencia 50%
    # Contamos numeros de coincidencia con un 'group_by'
    aggregate = collection_france.aggregate([
        {'$unwind': '$genres'}, # Descompongo los documentos por generos
        {'$match': {'genres': {'$in': movie['genres']}}}, # Filtro los documentos que tengan al menos un genero en comun con la pelicula actual
        {'$group': { # Agrupo los documentos por id, titulo y año y cuento el numero de coincidencias
            '_id': {'_id': '$_id', 'title': '$title', 'year': '$year'},
            'number': {'$sum': 1}
        }},
        {'$match': { # Filtro los documentos que tengan un numero de coincidencias entre 50% y 100%
            'number': {
                '$gte': len(movie['genres'])*0.5,
                '$lt': len(movie['genres'])
            }
        }},
        {'$sort': { # Ordeno los documentos por numero de coincidencias
            'number': pymongo.DESCENDING
        }},
        {'$limit': max_related_movies},
        {'$sort': {
            '_id.year': pymongo.DESCENDING
        }}
    ])
    related = [{'title': rel['_id']['title'], 'year': rel['_id']['year']}
               for rel in aggregate]

    # Insertamos aquellos con 100% > number > 50%
    collection_france.update_one({'_id': movie['_id']}, {'$set': {
        'most_related_movies': most_related,
        'related_movies': related
    }})

print("Películas insertadas a mongodb correctamente")

# Desconectamos la base de datos de postgres
if db_connection is not None:
    db_connection.close()
```

6 - Para cada película de la colección insertada anteriormente, buscamos las películas más relacionadas y las películas relacionadas. Las más relacionadas son aquellas que tienen todos los mismos géneros que la película actual, y las relacionadas son aquellas que tienen al menos el 50% de los mismos géneros que la película actual. Toda esta información la añadimos a cada documento de la colección “france”.
7 - Finalmente, cerramos la conexión a la base de datos de PostgreSQL.

A continuación, mostramos los resultados de la ejecución del archivo “create_mongodb_from_postgresqldb.py”:

```
larry@DESKTOP-IMR556L:/mnt/c/users/Sergio/Documents/3_curso/Sistemas_Informaticos/Practicas/P3/app/app-mongodb-etl$ mongosh
Current Mongosh Log ID: 6575a9deb41bee744c0ca7df
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.1.1
Using MongoDB:          7.0.4
Using Mongosh:          2.1.1

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

------
   The server generated these startup warnings when booting
   2023-12-10T11:29:52.888+01:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. See h
   2023-12-10T11:29:53.156+01:00: Access control is not enabled for the database. Read and write access to data and configur
   2023-12-10T11:29:53.156+01:00: /sys/kernel/mm/transparent_hugepage/enabled is 'always'. We suggest setting it to 'never'
   2023-12-10T11:29:53.156+01:00: vm.max_map_count is too low
------

test> show dbs
admin           40.00 KiB
config         108.00 KiB
database_name   24.00 KiB
local           80.00 KiB
si1              3.78 MiB
test> use si1
switched to db si1
si1> show collections
france
si1>
```

Desde la terminal, si ejecutamos el comando “mongosh” podemos acceder a las bases de datos de MongoDB. Como podemos ver, al mostrar las bases que tenemos “show dbs” está la base “si1”. Si accedemos a ella “use si1” y mostramos las colecciones que tiene “show collections” podemos observar la existencia de la colección “france”, lo que nos indica que se ha creado con éxito.

```
larry@DESKTOP-IMR556L:/mnt/c/users/Sergio/Documents/3_curso/Sistemas_Informaticos/Practicas/P3/app/app-mongodb-etl$ python3 create_mongodb_from_postgresqldb.py
Creando tabla de peliculas francesas...
Obteniendo los datos de las películas en Postgres y pasandolos a mongoDB...
Películas insertadas a mongodb correctamente
larry@DESKTOP-IMR556L:/mnt/c/users/Sergio/Documents/3_curso/Sistemas_Informaticos/Practicas/P3/app/app-mongodb-etl$
```

Apartado c)

Ahora, procederemos a mostrar el resultado de las consultas solicitadas (todas ellas están en el archivo “mongodb_queries.py”):

1ª Consulta (Apartado a) - Películas de ciencia ficción entre 1994 y 1998 :

```
si1> db.france.find({"genres": {"$in": ["Sci-Fi"]}, "year": {"$gte": 1994, "$lte": 1998}})
[
  {
    _id: ObjectId('656f61fcec6f742bfd5f77dd'),
    title: 'Fifth Element, The (1997)',
    genres: [ 'Action', 'Adventure', 'Sci-Fi', 'Thriller' ],
    year: 1997,
    directors: [ 'Besson, Luc' ],
    actors: [
      'Davis, Shaun (I)',
      'Dawodu, J.D.',
      'Dekker, Leon',
      'Douglas, Sam (I)',
      'Dunwell, Peter',
      'Ellwood, Eddie',
      'Evans, Lee (I)',
      'Ezekiel, Jerry',
      'Ezenagu, Derek',
      'Fairbank, Christopher',
      'Fishley, David',
      'Garcia, Roy (I)',
      'Garvey, David',
      'Georgijev, Alex',
      'Hamlett, Nathan',
      'Harvey-Wilson, Stewart',
      'Heng, Ivan',
      'Holm, Ian',
      'Hughes, John (VI)',
      'Adamson, Christopher (I)',
      'Alexander, Robert (I)',
      'Ashton, Richard (I)',
      'Barrass, David',
      'Beckett, Ian',
      'Bennett, John (I)',
      'Blake, Jerome (I)',
      'Bluthal, John',
```

(Al ser tantos datos, no hemos incluido una captura completa del resultado, solo un fragmento para observar su ejecución).

2ª Consulta (apartado b) - Dramas del año 1998 que empiecen por la palabra “The”:

```
si1> db.france.find({"year":1998, "genres":"Drama", "title":{'$regex':'The'}})
[
  {
    _id: ObjectId('656f61fcec6f742bfd5f77c2'),
    title: 'Quarry, The (1998)',
    genres: [ 'Drama' ],
    year: 1998,
    directors: [ 'Hänsel, Marion' ],
    actors: [
      'Abrahams, Jody',
      'Lynch, John (I)',
      'Petersen, Oscar (I)',
      'Valcke, Serge-Henri',
      'Phillips, Jonathan (I)',
      'Esau, Sylvia'
    ],
    most_related_movies: [
      { title: 'Under Suspicion (2000)', year: 2000 },
      { title: 'Dancer in the Dark (2000)', year: 2000 },
      { title: 'Under Suspicion (2000)', year: 2000 },
      { title: 'U-571 (2000)', year: 2000 },
      { title: 'Dancer in the Dark (2000)', year: 2000 },
      { title: 'Golden Bowl, The (2000)', year: 2000 },
      { title: 'U-571 (2000)', year: 2000 },
      { title: 'Dancer in the Dark (2000)', year: 2000 },
      { title: 'U-571 (2000)', year: 2000 },
      { title: 'Golden Bowl, The (2000)', year: 2000 }
    ],
    related_movies: []
  },
```

Como son muchos resultados, adjuntamos una parte de ellos.

3ª Consulta (apartado c): Películas en las que “Faye Dunaway” y “Viggo Mortensen” hayan compartido reparto

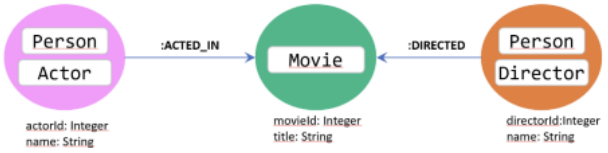
```
si1> db.france.find({"$and": [{"actors": "Dunaway, Faye"}, {"actors": "Mortensen, Viggo"}]})
[
  {
    _id: ObjectId('656f61fcec6f742bfd5f77f3'),
    title: 'Albino Alligator (1996)',
    genres: [ 'Crime', 'Drama', 'Thriller' ],
    year: 1996,
    directors: [ 'Spacey, Kevin (I)' ],
    actors: [
      'Dillon, Matt (I)',      'Faison, Frankie',
      'Fichtner, William',     'Garrett, Spencer (I)',
      'Hoffman, Jeffrey M.',   'Appel, Travis',
      'Ball, Tulsy',           'Carpenter, Willie C.',
      'Colantoni, Enrico',     'Koepenick, Brad',
      'Mantegna, Joe',         'Moore, Anthony (II)',
      'Sinise, Gary',          'Mortensen, Viggo',
      'Smith, Alexander (I)',  'Spinuzza, Doug',
      'Worthen, Jock',         'Spencer, John (I)',
      'Ulrich, Skeet',         'Unger, Michael (I)',
      'Walsh, M. Emmet',       'Figueiredo, Aura',
      'Dunaway, Faye',         'Montgomery, Toni',
      'McGraw, Melinda'
    ],
```

Ocurre lo mismo que en la 1ª query. Son demasiados datos, pero ponemos un fragmento de ellos para poder observar que existen películas en las que “Faye Dunaway” y “Viggo Mortensen” han compartido reparto, como por ejemplo la película “Albino Alligator”. Se pueden apreciar ambos nombres en el array de “actors”.

1.2 - BBDD basadas en Grafos

Apartado a)

Se nos pide, a partir de la base de datos de películas para PostgreSQL, crear una base de datos basada en grafos que contenga las 20 películas estadounidenses más vendidas almacenadas en la base de datos relacional, junto con sus respectivos actores y directores. Además, el grado que se cree deberá tener la siguiente estructura:



En primer lugar, hemos diseñado una query para obtener las 20 películas más vendidas de USA. Aquí se muestra el resultado de la ejecución del archivo “getTopFilmsUsa.sql”:

```sql
DROP FUNCTION IF EXISTS getTopFilmsUsa();

CREATE OR REPLACE FUNCTION getTopFilmsUsa(OUT movietitle VARCHAR, OUT ventas INT)
RETURNS SETOF RECORD
AS $$
BEGIN
    RETURN QUERY
    SELECT
        CAST(m.movietitle AS VARCHAR(100)),
        SUM(quantity)::INT
    FROM orders o
    JOIN orderdetail od ON o.orderid = od.orderid
    JOIN products p ON p.prod_id = od.prod_id
    JOIN imdb_movies m ON m.movieid = p.movieid
    JOIN imdb_moviecountries ic ON ic.movieid = m.movieid
    WHERE ic.country = 'USA'
    GROUP BY m.movietitle
    ORDER BY SUM(quantity) DESC
    LIMIT 20;
END;
$$ LANGUAGE plpgsql;
```

```
larry@DESKTOP-IMR556L:/mnt/c/users/Sergio/Documents/3_curso/Sistemas_Informaticos/Practicas/P3/app/app-neo4j-etl$ psql -U alumnodb -d si1
psql (14.9 (Ubuntu 14.9-0ubuntu0.22.04.1))
Type "help" for help.

si1=# \i getTopFilmsUsa.sql
DROP FUNCTION
CREATE FUNCTION
si1=# SELECT * FROM getTopFilmsUsa();
           movietitle             | ventas
----------------------------------+--------
 Life Less Ordinary, A (1997)     |    600
 Only You (1994)                  |    577
 Glimmer Man, The (1996)          |    568
 Illtown (1996)                   |    564
 Very Thought of You, The (1945)  |    556
 Blob, The (1958)                 |    554
 Lethal Weapon 4 (1998)           |    553
 Ilsa, She Wolf of the SS (1975)  |    548
 Handle with Care (1977)          |    546
 Doctor Zhivago (1965)            |    544
 Jerk, The (1979)                 |    544
 Angel Heart (1987)               |    544
 Heavy (1995)                     |    543
 Friends & Lovers (1999)          |    543
 Apostle, The (1997)              |    542
 Living Out Loud (1998)           |    542
 Bound for Glory (1976)           |    541
 Pyromaniac's Love Story, A (1995) |    540
 Ordinary People (1980)           |    540
 Stranger, The (1994)             |    540
(20 rows)

si1=#
```

Antes de integrarlo a la base de datos de neo4j, decidimos comprobar el correcto funcionamiento de la query. Podemos ver que obtiene 20 películas (20 rows), todas ellas estadounidenses ordenadas en orden decreciente en función del número de ventas totales.

Apartado b)

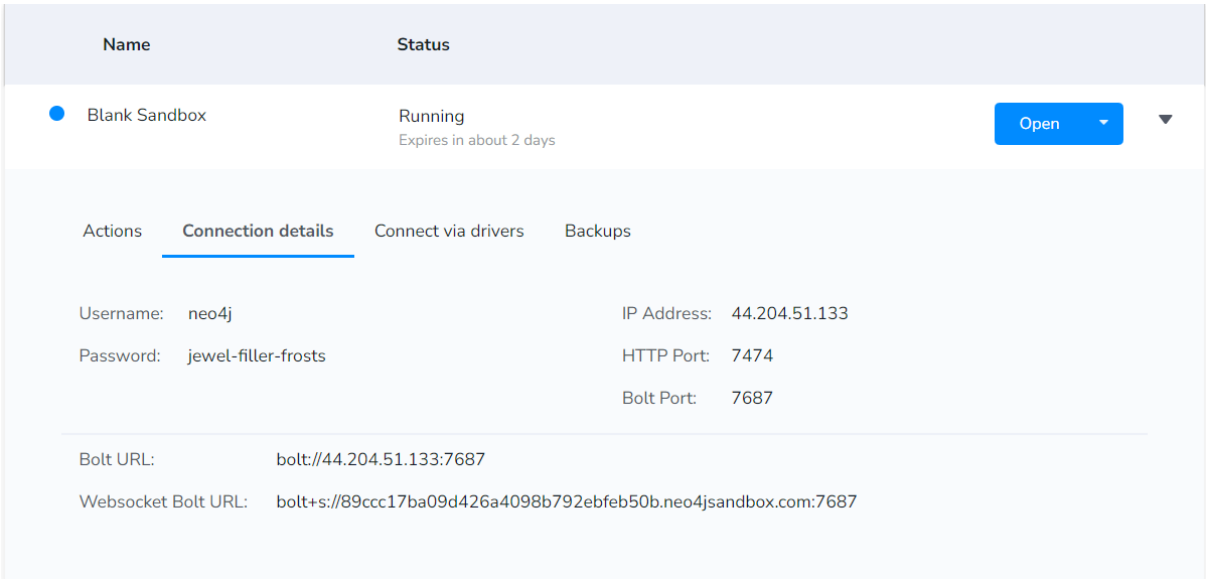
Este apartado consiste en la creación de una base de datos en neo4j con las 20 películas obtenidas gracias a la query anterior. Para ello, hemos implementado el archivo "create_neo4jdb_from_postgresqldb.py".

```
# Conexion a la base de datos si1
print("Conectando a la base de datos si1...")
postgres_conn = psycopg2.connect(
    host="localhost",
    database="si1",
    user="alumnodb",
    password="1234"
)

# Conexion a la base de datos neo4j
print("Conectando a la base de datos neo4j...")
neo4j_driver = GraphDatabase.driver("bolt://44.204.51.133:7687", auth=("neo4j", "jewel-filler-frosts"))
neo4j_session = neo4j_driver.session()
```

En primer lugar, nos conectamos a la base de datos "si1" para poder extraer los datos necesarios para ejecutar la query de las 20 películas estadounidenses más vendidas. Después establecemos la conexión con la base de datos en neo4j con:

Usuario: "neo4j"
Contraseña: "jewel-filler-frosts"
URL del servidor de Neo4j: "bolt://44.204.51.133:7687"
Protocolo de comunicación usado por neo4j - "bolt"
Dirección IP del servidor: "44.204.51.133"
Puerto en el que se ejecuta el servidor: "7687"

Hemos utilizado "neo4 sandbox" para poder observar en forma de grafos el resultado de las querys solicitadas. Los datos mencionados para establecer la conexión con la base de datos los hemos obtenido de dicho sandbox:

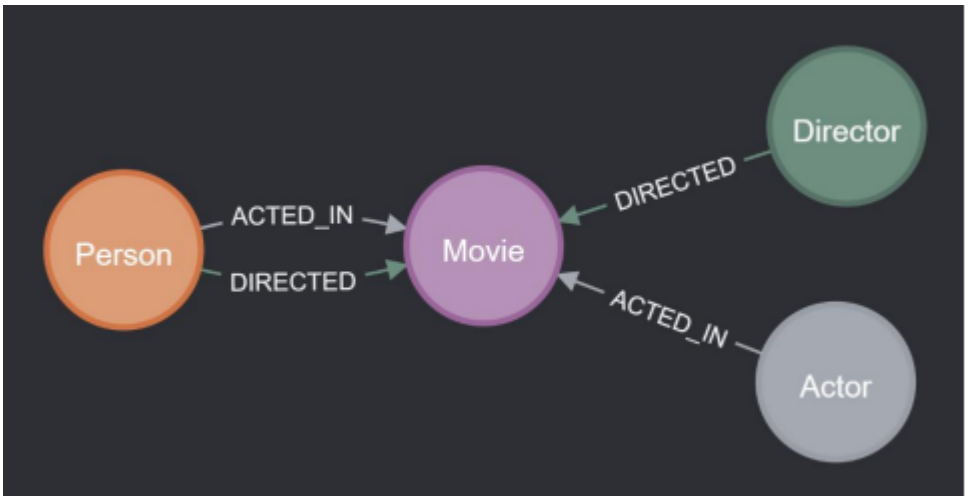


Una vez establecida la conexión con Neo4j, procedemos a la ejecución de la consulta para obtener las 20 películas estadounidenses más vendidas y a la creación de nodos y relaciones en Neo4j con el formato indicado en el enunciado:

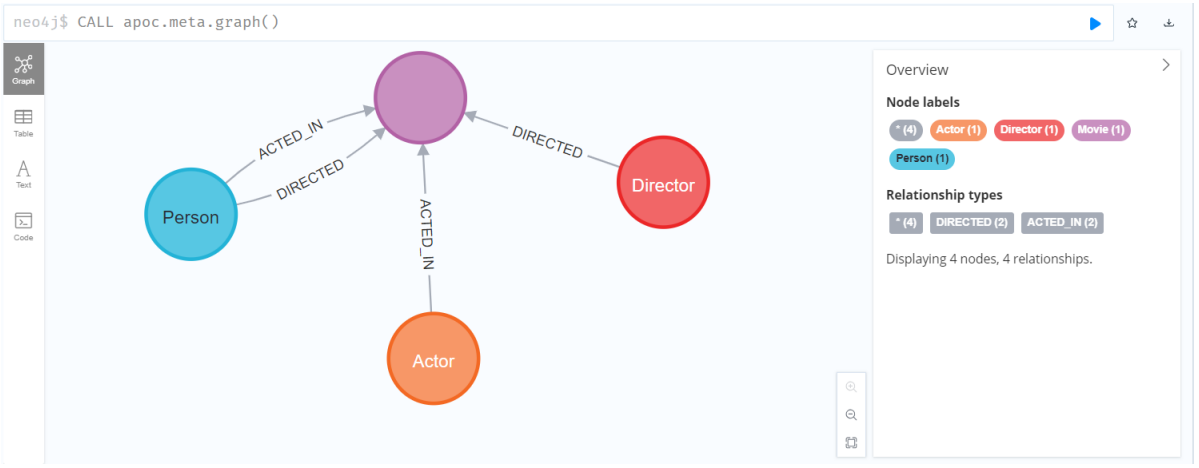
```
# Obtener las 20 peliculas mas vendidas de USA
print("Obteniendo las 20 peliculas mas vendidas de USA...")
postgres_cursor = postgres_conn.cursor()
postgres_cursor.execute("""
    WITH top_movies AS (
        SELECT m.movieid, m.movietitle
        FROM orders o
        JOIN orderdetail od ON o.orderid = od.orderid
        JOIN products p ON p.prod_id = od.prod_id
        JOIN imdb_movies m ON m.movieid = p.movieid
        JOIN imdb_moviecountries ic ON ic.movieid = m.movieid
        WHERE ic.country = 'USA'
        GROUP BY m.movieid, m.movietitle
        ORDER BY SUM(od.quantity) DESC
        LIMIT 20
    )
    SELECT tm.movieid, tm.movietitle, a.actorid, a.actorname, d.directorid, d.directorname
    FROM top_movies tm
    LEFT JOIN imdb_actormovies am ON am.movieid = tm.movieid
    LEFT JOIN imdb_actors a ON a.actorid = am.actorid
    LEFT JOIN imdb_directormovies dm ON dm.movieid = tm.movieid
    LEFT JOIN imdb_directors d ON d.directorid = dm.directorid;
""")
movies = postgres_cursor.fetchall()

# Crear nodos y relaciones en neo4j
print("Creando nodos y relaciones en neo4j...")
with neo4j_driver.session(database="neo4j") as session:
    for movie in movies:
        if movie[0] is not None and movie[1] is not None:
            session.execute_write(
                lambda tx: tx.run("""
                    MERGE (m:Movie {id: $movieid, title: $movietitle})
                    """, movieid=movie[0], movietitle=movie[1])
            )
        if movie[2] is not None and movie[3] is not None:
            session.execute_write(
                lambda tx: tx.run("""
                    MERGE (a:Actor:Person {id: $actorid, name: $actorname})
                    MERGE (a)-[:ACTED_IN]->(m)
                    """, actorid=movie[2], actorname=movie[3], movieid=movie[0])
            )
        if movie[4] is not None and movie[5] is not None:
            session.execute_write(
                lambda tx: tx.run("""
                    MERGE (d:Director:Person {id: $directorid, name: $directorname})
                    MERGE (d)-[:DIRECTED]->(m)
                    """, directorid=movie[4], directorname=movie[5], movieid=movie[0])
            )

print("Terminado!")
neo4j_driver.close()
```

Podemos ver la correcta creación de la base de datos con el procedimiento: “CALL apoc.meta.graph()”

Asi debe de quedar la estructura de la base de la nueva base de datos según las indicaciones de la práctica:

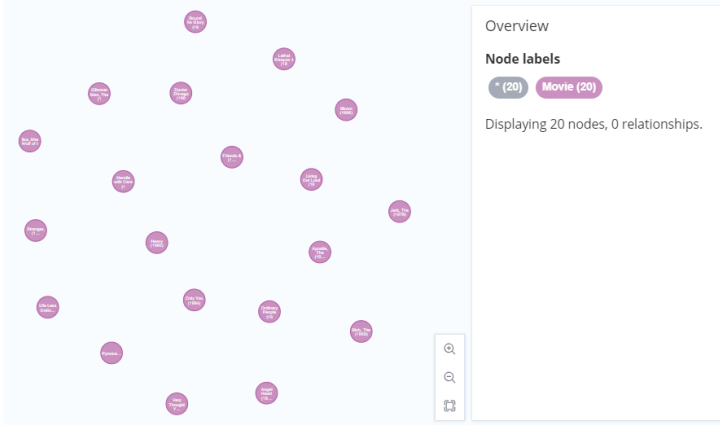


Así es como nos queda a nosotros:



Podemos apreciar que se han creado con éxito las etiquetas de los nodos ACTOR, DIRECTOR, MOVIE y PERSON y las relaciones entre ellos ACTED_IN, DIRECTED.

Además, hemos realizado una comprobación extra para ver si se han obtenido las 20 películas estadounidenses más vendidas correctamente. Para ello vemos todos los nodos con la etiqueta “MOVIE” y comprobamos que sus nombres coincidan con los nombres de las películas obtenidas anteriormente en la terminal:



pelicula
(:Movie {id: 291835,title: "Only You (1994)"})
(:Movie {id: 20335,title: "Angel Heart (1987)"})
(:Movie {id: 293247,title: "Ordinary People (1980)"})
(:Movie {id: 49493,title: "Blob, The (1958)"})
(:Movie {id: 23084,title: "Apostle, The (1997)"})
(:Movie {id: 201944,title: "Jerk, The (1979)"})
(:Movie {id: 233000,title: "Living Out Loud (1998)"})
(:Movie {id: 189256,title: "Illtown (1996)"})
(:Movie {id: 227752,title: "Lethal Weapon 4 (1998)"})
(:Movie {id: 145256,title: "Friends & Lovers (1999)"})

(:Movie {id: 54396,title: "Bound for Glory (1976)"})
(:Movie {id: 107209,title: "Doctor Zhivago (1965)"})
(:Movie {id: 156403,title: "Glimmer Man, The (1996)"})
(:Movie {id: 189363,title: "Ilsa, She Wolf of the SS (1975)"})
(:Movie {id: 167065,title: "Handle with Care (1977)"})
(:Movie {id: 381390,title: "Stranger, The (1994)"})
(:Movie {id: 229764,title: "Life Less Ordinary, A (1997)"})
(:Movie {id: 171259,title: "Heavy (1995)"})
(:Movie {id: 323235,title: "Pyromaniac's Love Story, A (1995)"})
(:Movie {id: 425473,title: "Very Thought of You, The (1945)"})

Apartado c)

En este apartado se pide hacer 3 consultas diferentes sobre la base de datos creada en el apartado b).

1º Consulta (apartado a) - Una tabla que devuelva 10 actores ordenados alfabéticamente que no han trabajado con “Winston, Hattie”, pero que en diferentes momentos han trabajado con un tercero en común.

La consulta se encuentra en el archivo “winston-hattie-co-co-actors.cypher”:

```
MATCH (a1:Actor)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(a2:Actor)
WHERE a1.name <> "Winston, Hattie" AND a2.name <> "Winston, Hattie"
RETURN DISTINCT a1.name AS Actor
ORDER BY a1.name ASC
LIMIT 10
```

Resultado de su ejecución tanto desde terminal con la interfaz “cypher-shell” como desde neo4j sandbox/browser:

```
neo4j@neo4j> MATCH (a1:Actor)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(a2:Actor)
             WHERE a1.name <> "Winston, Hattie" AND a2.name <> "Winston, Hattie"
             RETURN DISTINCT a1.name AS Actor
             ORDER BY a1.name ASC
             LIMIT 10;
+------------------------+
| Actor                  |
+------------------------+
| "Adam, Joel"           |
| "Adams, Catlin"        |
| "Adams, Lillian"       |
| "Adams, Melanie (II)"  |
| "Addington, Constance" |
| "Addota, Kip"          |
| "Ahern, Alston"        |
| "Albright, Gerald"     |
| "Alcañiz, Luana"       |
| "Alderman, Jane"       |
+------------------------+

10 rows available after 43 ms, consumed after another 82 ms
neo4j@neo4j>
```

Actor
"Adam, Joel"
"Adams, Catlin"
"Adams, Lillian"
"Adams, Melanie (II)"
"Addington, Constance"
"Addota, Kip"
"Ahern, Alston"
"Albright, Gerald"
"Alcañiz, Luana"
"Alderman, Jane"

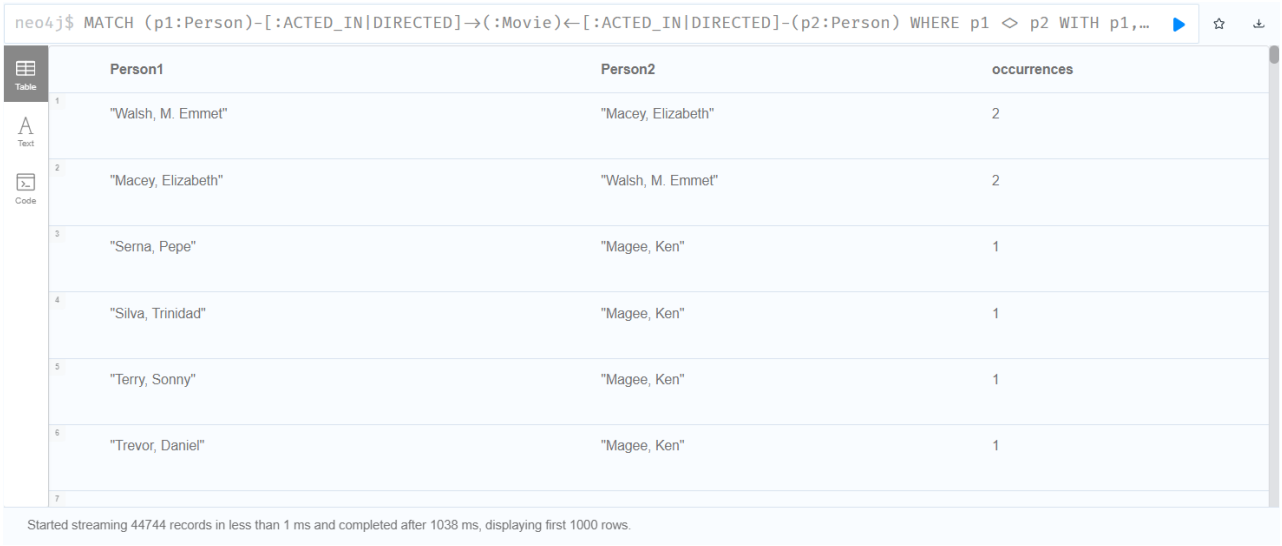
Como era de esperar, el resultado es el mismo tanto desde neo4j browser como desde la propia terminal de nuestra base de datos.

2ª Consulta (apartado b) - Una tabla que muestre en cada fila pares de personas que han trabajado juntas en más de una película, sin distinguir categoría de actores o directores.

La consulta se encuentra en el archivo “pair-persons-mostoccurrences.cypher”.

```
MATCH (p1:Person)-[:ACTED_IN|DIRECTED]->(:Movie)<-[:ACTED_IN|DIRECTED]-(p2:Person)
WHERE p1 <> p2
WITH p1, p2, count(*) AS occurrences
RETURN p1.name AS Person1, p2.name AS Person2, occurrences
ORDER BY occurrences DESC
```

Resultado de su ejecución desde neo4j sandbox/browser:

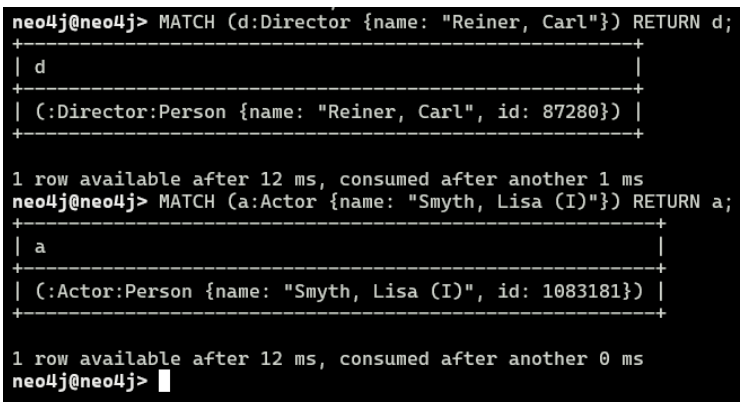


3ª Consulta (apartado C) - Hallar el camino mínimo por el cual el director “Reiner, Carl” podría conocer a la actriz “Smyth, Lisa (I)”.

La consulta se encuentra en el archivo “degrees-reiner-to-smyth.cypher”.

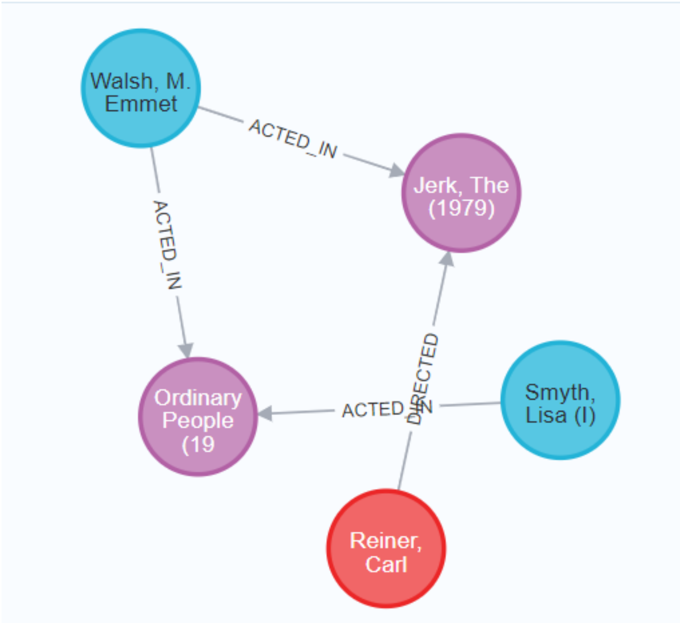
```
MATCH (d:Director {name: "Reiner, Carl"}), (a:Actor {name: "Smyth, Lisa (I)"})
MATCH path = shortestPath((d)-[*]-(a))
RETURN path
```

Antes de ejecutar esta consulta, nos aseguramos que existiesen el director “Reiner, Carl” y la actriz “Smyth, Lisa (I)”.



Resultado de la ejecución de la consulta tanto desde la “cypher-shell” como desde el neo4j browser:

```
neo4j@neo4j> MATCH (d:Director {name: "Reiner, Carl"}), (a:Actor {name: "Smyth, Lisa (I)"})
             MATCH path = shortestPath((d)-[*]-(a))
             RETURN path;
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------+
| path
                                                                                                                                          |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------+
| (:Director:Person {name: "Reiner, Carl", id: 87280})-[:DIRECTED]->(:Movie {title: "Jerk, The (1979)", id: 201944})<-[:ACTED_IN]-(:Actor:Person {name: "Walsh, M. Emmet", id: 682727})-[:AC
TED_IN]->(:Movie {title: "Ordinary People (1980)", id: 293247})<-[:ACTED_IN]-(:Actor:Person {name: "Smyth, Lisa (I)", id: 1083181}) |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------+

1 row available after 41 ms, consumed after another 11 ms
neo4j@neo4j>
```



Ambos obtienen el mismo camino. El director Reiner Carl dirigió la película Jerk, The (1979), la actriz participó en la película Ordinary People (1980) y en ambas películas actuó el actor Walsh, M.Emmet.

2 - Uso de tecnología caché de acceso rápido

Apartados a) y b)

En esta parte de la práctica nos piden crear una base de datos en memoria para guardar la cantidad de visitas de los clientes a la página web que utiliza esta base de datos. Esto se debe llevar a cabo usando la estructura de los datos hashes proporcionado por Redis. El hash debe tener como llave customers:email. Además del email se deben guardar los siguientes campos:

- name - firstname + “” + lastname
- phone - El teléfono del cliente en la base de datos
- visits - El número total de visitas. Debido a que este campo no se encuentra en la base de datos, se debe rellenar con un número aleatorio del 1..99.

Todo ello se tiene que hacer en el script “create_redis_from_postgresqldb.py”.

Nuestro script consiste en lo siguiente:

```
# Conexión a la base de datos PostgreSQL
"""
Url con:
    Usuario: alumnodb
    Contraseña: 1234
    Base de datos: si1
    host: localhost
"""
print("Conectando a la base de datos PostgreSQL...")
engine = create_engine('postgresql://alumnodb:1234@localhost:5432/si1')
metadata = MetaData()
metadata.bind = engine
customers_table = Table('customers', metadata, autoload_with=engine)

# Conexión a la base de datos Redis
print("Conectando a la base de datos Redis...")
redis_db = redis.Redis(host='localhost', port=6379, db=0)

# Crear una sesión
session = Session(engine)
```

1 - Nos conectamos a la base de datos de PostgreSQL llamada “si1” en el host “localhost” con el usuario “alumnodb” y la contraseña “1234”. La conexión la realizamos a través de SQLAlchemy, una biblioteca de Python para interactuar con bases de datos SQL.
2 - Cargamos la tabla “customers” en un objeto Table de SQLAlchemy. Esto nos permite interactuar con dicha tabla a través de este objeto.
3 - Nos conectamos a una base de datos Redis en el host “localhost” en el puerto “6379”.
4 - Creamos una sesión con la base de datos de PostgresSQL a través de SQLAlchemy, Dicha sesión maneja todas las interacciones con la base de datos y proporciona una interfaz para realizar diferentes operaciones como por ejemplo consultas, inserciones y actualizaciones.

```
def create_redis_db():
    print("Creando base de datos Redis...")
    # Obtener clientes de la base de datos PostgreSQL que son de España
    query = customers_table.select().where(customers_table.c.country == 'Spain')
    result = session.execute(query)
    customers = result.fetchall()

    # Almacenar clientes en la base de datos Redis
    for customer in customers:
        email = customer.email
        name = customer.firstname + " " + customer.lastname
        phone = customer.phone
        visits = random.randint(1, 99)

        hash_key = f"customers:{email}"
        redis_db.hset(hash_key, "name", name)
        redis_db.hset(hash_key, "phone", phone)
        redis_db.hset(hash_key, "visits", visits)

    print("Base de datos Redis creada!.")
```

5 - Obtenemos los clientes que sean de España para posteriormente almacenarlos en la base de datos de redis.
6 - Para cada cliente creamos una clave única en el formato solicitado customers:email y almacenamos información del cliente en la base de datos de Redis usando dicha clave. La información almacenada incluye el nombre, el teléfono y el número de visitas.

Apartado c)

En el mismo fichero que los apartados anteriores, se nos pide realizar 3 funciones:

Una función que incremente una visita dado el correo electrónico:

```python
def increment_by_email(email):
    hash_key = f"customers:{email}"
    redis_db.hincrby(hash_key, "visits", 1)
```

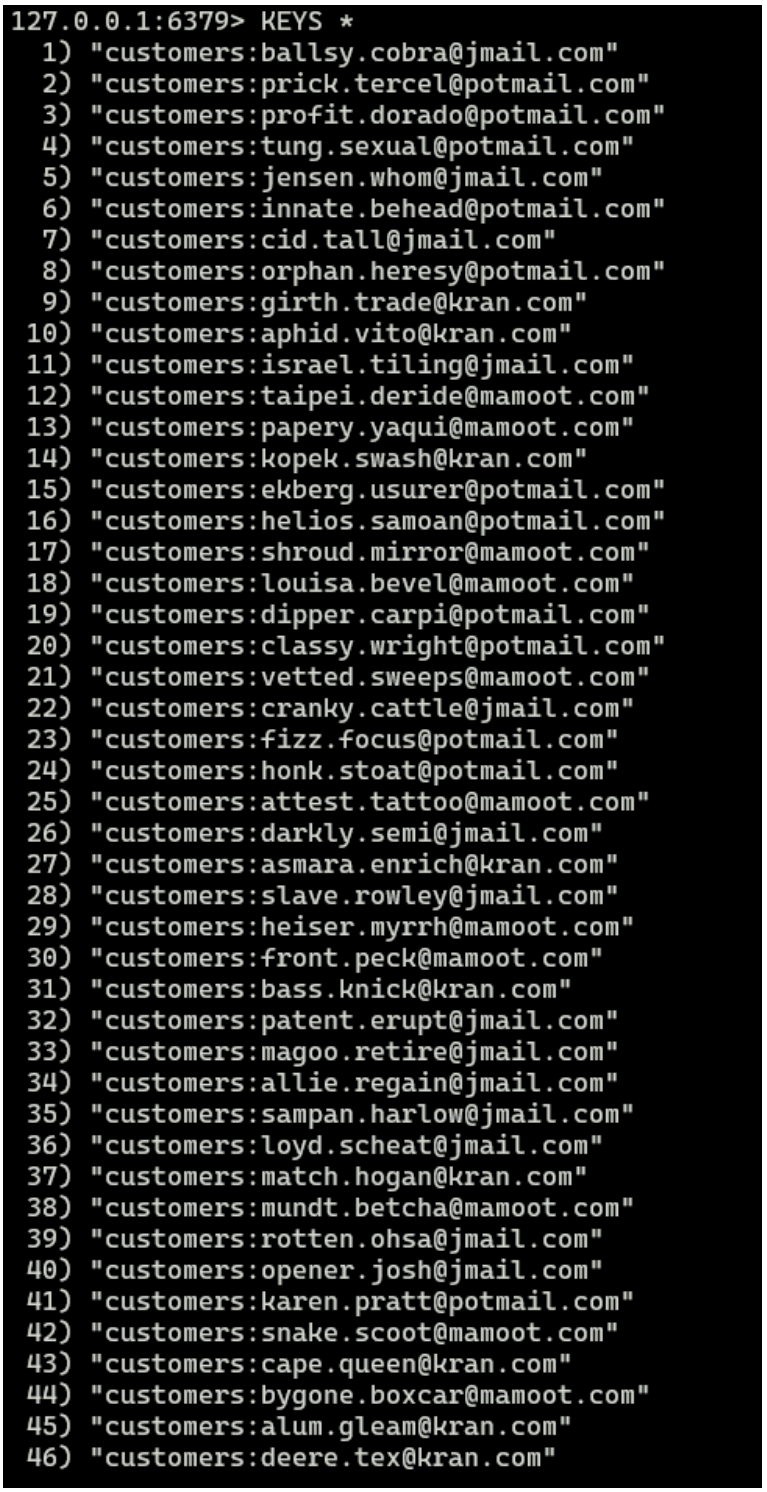
Una función que devuelva el email del usuario con mayor cantidad de visitas:

```python
def customer_most_visits():
    customers = redis_db.keys("customers:*")
    max_visits = 0
    max_email = ""

    for customer in customers:
        visits = int(redis_db.hget(customer, "visits"))
        if visits > max_visits:
            max_visits = visits
            max_email = customer.decode("utf-8").split(":")[1]

    return max_email
```

Una función que muestre el nombre, teléfono y número de visitas dado el email:

```python
def get_field_by_email(email):
    hash_key = f"customers:{email}"
    name = redis_db.hget(hash_key, "name").decode("utf-8")
    phone = redis_db.hget(hash_key, "phone").decode("utf-8")
    visits = int(redis_db.hget(hash_key, "visits"))

    return name, phone, visits
```

Para comprobar que la base de datos ha sido creada correctamente nos hemos metido desde la terminal en la interfaz “redis-cli” y hemos puesto “KEYS *” para ver si se han creado las claves de la base de datos:



Como podemos observar, las claves se han creado correctamente. Cabe destacar que hay todavía muchas más claves, sólo hemos capturado una parte de ellas.

Una vez que sabemos que se han creado bien las claves, procederemos a ejecutar el script:

```
larry@DESKTOP-IMR556L:/mnt/c/users/Sergio/Documents/3_curso/Sistemas_Informaticos/Practicas/P3/app/app-redis-etl$ python3 create_redis_from_postgresqldb.py
Conectando a la base de datos PostgreSQL...
Conectando a la base de datos Redis...
Creando base de datos Redis...
Base de datos Redis creada!.

Query 1: Incrementar en 1 las visitas de un cliente - Cliente: ballsy.cobra@jmail.com
Visitas antes: 40
Visitas despues: 41

Query 2: obtener el email con mas visitas: runty.patty@kran.com

Query 3: Obtener el nombre, telefono y visitas de un cliente - Cliente: ballsy.cobra@jmail.com
Nombre: ballsy cobra - Telefono: +45 446945576 - Visitas: 41
larry@DESKTOP-IMR556L:/mnt/c/users/Sergio/Documents/3_curso/Sistemas_Informaticos/Practicas/P3/app/app-redis-etl$
```

Hemos ejecutado las funciones realizadas con un email en concreto, simplemente para ver cómo se comportan:

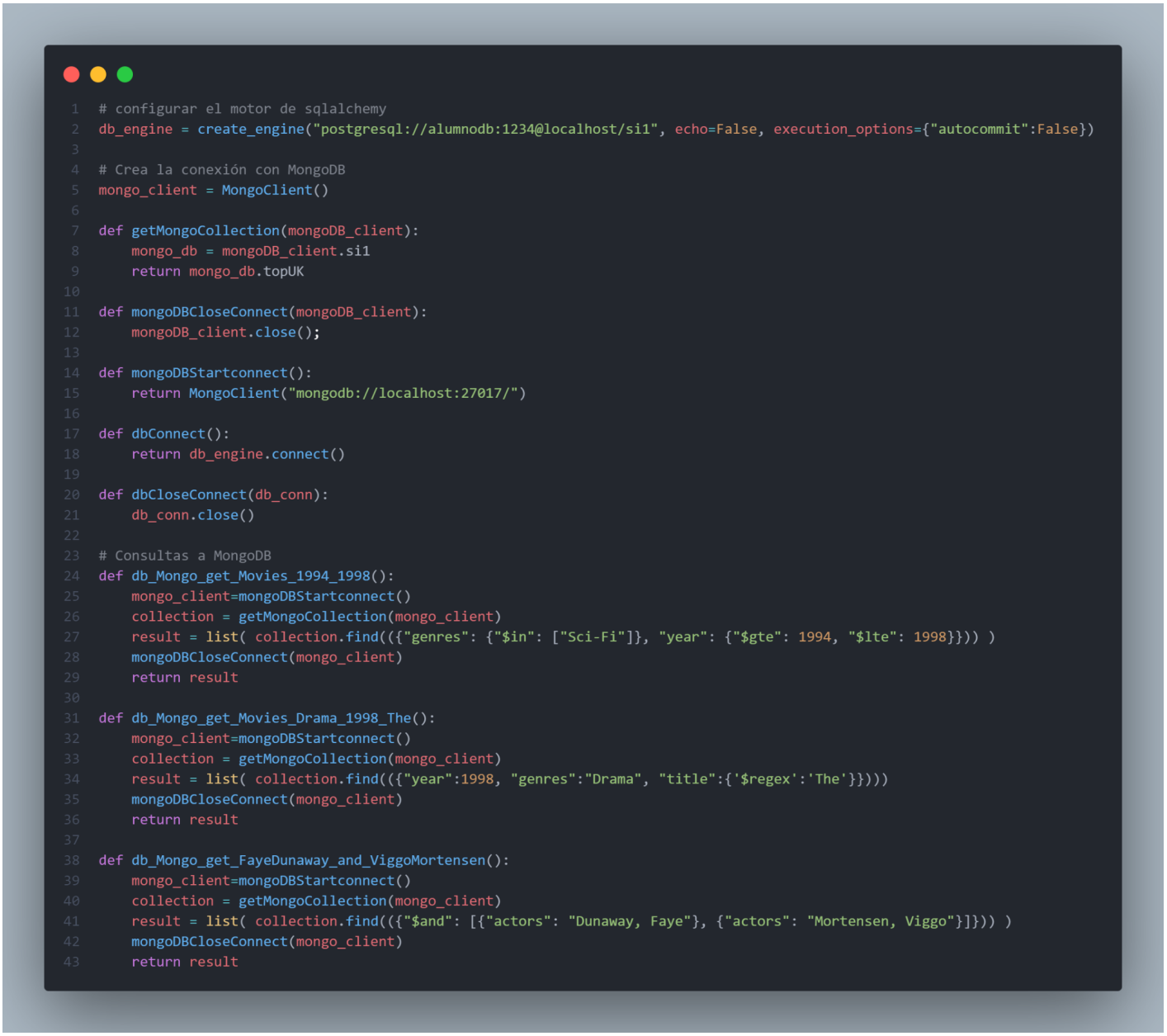
```python
# Consulta 1: Incrementar en 1 las visitas de un cliente
email = "ballsy.cobra@jmail.com"
print(f"\nQuery 1: Incrementar en 1 las visitas de un cliente - Cliente: {email}")
print(f"Visitas antes: {get_visits_by_email(email)}")
increment_by_email(email)
print(f"Visitas despues: {get_visits_by_email(email)}")

# Consulta 2: Obtener el email del usuario con mas visitas
print(f"\nQuery 2: obtener el email con mas visitas: {customer_most_visits()}")

# Consulta 3: Obtener el nombre, telefono y visitas de un cliente
print(f"\nQuery 3: Obtener el nombre, telefono y visitas de un cliente - Cliente: {email}")
name, phone, visits = get_field_by_email(email)
print(f"Nombre: {name} - Telefono: {phone} - Visitas: {visits}")
```

3 - Transacciones

Apartado A - Estudio de Transacciones

En este apartado se nos pide realizar una página “borrarCiudad” que ejecute una transacción que borre todos los clientes de una ciudad, y toda su información asociada (carrito e historial y pedidos con su detalle), de la base de datos.

Hemos implementado los requisitos solicitados en los archivos “*database.py*” y “*routes.py*”.



```python
# configurar el motor de sqlalchemy
db_engine = create_engine("postgresql://alumnodb:1234@localhost/si1", echo=False, execution_options={"autocommit":False})

# Crea la conexión con MongoDB
mongo_client = MongoClient()

def getMongoCollection(mongoDB_client):
    mongo_db = mongoDB_client.si1
    return mongo_db.topUK

def mongoDBCloseConnect(mongoDB_client):
    mongoDB_client.close();

def mongoDBStartconnect():
    return MongoClient("mongodb://localhost:27017/")

def dbConnect():
    return db_engine.connect()

def dbCloseConnect(db_conn):
    db_conn.close()

# Consultas a MongoDB
def db_Mongo_get_Movies_1994_1998():
    mongo_client=mongoDBStartconnect()
    collection = getMongoCollection(mongo_client)
    result = list( collection.find(({"genres": {"$in": ["Sci-Fi"]}, "year": {"$gte": 1994, "$lte": 1998}})) )
    mongoDBCloseConnect(mongo_client)
    return result

def db_Mongo_get_Movies_Drama_1998_The():
    mongo_client=mongoDBStartconnect()
    collection = getMongoCollection(mongo_client)
    result = list( collection.find(({"year":1998, "genres":"Drama", "title":{'$regex':'The'}})))
    mongoDBCloseConnect(mongo_client)
    return result

def db_Mongo_get_FayeDunaway_and_ViggoMortensen():
    mongo_client=mongoDBStartconnect()
    collection = getMongoCollection(mongo_client)
    result = list( collection.find(({"$and": [{"actors": "Dunaway, Faye"}, {"actors": "Mortensen, Viggo"}]})) )
    mongoDBCloseConnect(mongo_client)
    return result
```

En esta parte del fichero “database.py” realizamos lo siguiente:

1. Creamos el motor de SQLALchemy para el usuario: alumnodb, contraseña: 1234, host:localhost, y base de datos: si1. Cabe destacar que autocommit = False por que no queremos que las transacciones se confirman automáticamente después de cada operación de base de datos, ya que estamos implementándolo con “rollback”.
2. Definimos funciones para garantizar la conexión y cierre con la base de datos de mongodb como explicamos en el archivo ”create_mongodb_from_postgresqldb.py” .
3. Creamos 3 funciones adicionales con las 3 consultas solicitadas para posteriormente poder ejecutarlas mediante las transacciones.

```python
def delState(state, bFallo, bSQL, duerme, bCommit):

    # Array de trazas a mostrar en la página
    dbr=[]

    # TODO: Ejecutar consultas de borrado
    # - ordenar consultas según se desee provocar un error (bFallo True) o no
    # - ejecutar commit intermedio si bCommit es True
    # - usar sentencias SQL ('BEGIN', 'COMMIT', ...) si bSQL es True
    # - suspender la ejecución 'duerme' segundos en el punto adecuado para forzar deadlock
    # - ir guardando trazas mediante dbr.append()

    try:
        # TODO: ejecutar consultas
        db_conn = dbConnect()
        db_session = Session(db_conn)  # Crear una nueva sesión

        # Si bSQL es True, ejecutamos las consultas sql
        if bSQL:
            # Primero empezamos con begin
            db_session.begin()  # Iniciar una nueva transacción
            dbr.append("BEGIN")

        # Si no hay fallo, ejecutamos las consultas en orden
        if (bFallo != True):
            query_in_order(db_session, state, duerme, bCommit, dbr)
        else:
            query_not_in_order(db_session, state, duerme, bCommit, dbr)
    except Exception as e:
        if 'db_session' in locals():  # Comprueba si db_session está definida
            db_session.rollback()  # Hacer rollback de la transacción
        dbr.append("ROLLBACK" + str(e))
        if db_session is not None:
            db_session.close()  # Cerrar la sesión
    else:
        # TODO: confirmar cambios si todo va bien
        db_session.commit()  # Hacer commit de la transacción
        dbr.append("COMMIT")
        if db_session is not None:
            db_session.close()  # Cerrar la sesión

    return dbr
```

La función “delState” es la función principal que realiza una serie de consultas y operaciones sobre la base de datos.

- Si no se produce ningún fallo, las consultas se ejecutarán en orden mediante la llamada a la función “query_in_order”. Por otro lado, si hay fallo, las consultas no se ejecutarán en orden mediante la llamada al procedimiento “query_not_in_order”.
- Además, si se produce alguna excepción, implica que algo no ha sucedido como se esperaba, por lo que se ejecuta el “rollback”. De esta manera, se revierte la transacción de base de datos.
- A lo largo del código, vamos poniendo trazas para poder apreciar de una manera ordenada las operaciones realizadas.
- Si todo funciona correctamente, se ejecuta la transacción y devolvemos las trazas capturadas.

```python
def query_in_order(db_conn, state, duerme, bCommit, dbr):

    # Empiezo obteniendo el id de todos los customer que estén en el estado especificado
    query = text("select customerID from customers where state='" + str(state) + "';")
    idsC = list(db_conn.execute(query))
    idCustomers = [a[0] for a in idsC]
    dbr.append("LEER CUSTOMERS")

    # Obtengo las orderID de las orders de los customerid de la ciudad especificados
    if idCustomers:
        query = text("select orderID from orders where customerid IN " + str(idCustomers).replace("[", "(").replace("]", ")") + ";")
        idsO = list(db_conn.execute(query))
        idOrders = [a[0] for a in idsO]
    else:
        idOrders = []
    dbr.append("LEER PEDIDOS")

    if idOrders:
        # Borro pedidos de orderdetail
        query = text("delete from orderdetail where orderid in " + str(idOrders).replace("[", "(").replace("]", ")") + ";")
        db_conn.execute(query)
        dbr.append("BORRAR PRODUCTOS DE PEDIDOS")

    if bCommit:
        db_conn.execute(text("COMMIT;"))
        dbr.append("COMMIT")
        db_conn.execute(text("BEGIN;"))
        dbr.append("BEGIN")

    if duerme != 0:
        db_conn.execute(text("SELECT pg_sleep(" + str(duerme) + ");"))
        dbr.append("SLEEP")

    if idOrders:
        # Borro pedidos de orders
        query = text("delete from orders where orderid in " + str(idOrders).replace("[", "(").replace("]", ")") + ";")
        db_conn.execute(query)
        dbr.append("BORRAR PEDIDOS")

    if bCommit:
        db_conn.execute(text("COMMIT;"))
        dbr.append("COMMIT")
        db_conn.execute(text("BEGIN;"))
        dbr.append("BEGIN")

    if idCustomers:
        # Borramos customers
        query = text("delete from customers where customerid in " + str(idCustomers).replace("[", "(").replace("]", ")") + ";")
        db_conn.execute(query)
        dbr.append("BORRAR USUARIOS")
```

Este método sirve para ejecutar las consultas en orden. La función realiza lo siguiente:

1. Obtiene los ID de todos los clientes (customerID) que están en el estado especificado (state). Los ID obtenidos se almacenan en la lista idCustomers.
2. Si idCustomers no está vacío, obtiene los ID de los pedidos (orderID) que pertenecen a los clientes con los ID obtenidos en el paso anterior. Los ID de los pedidos se almacenan en la lista idOrders.
3. Si idOrders no está vacío, borra los detalles de los pedidos (orderdetail) que pertenecen a los pedidos con los ID obtenidos en el paso anterior.
4. Si bCommit es True, realiza un COMMIT para guardar los cambios en la base de datos y luego inicia una nueva transacción con BEGIN.
5. Si duerme no es 0, hace una pausa en la ejecución durante el número de segundos especificado por duerme.

6. Si idOrders no está vacío, borra los pedidos (orders) que tienen los ID obtenidos en el paso 2.
7. Si bCommit es True, realiza un COMMIT para guardar los cambios en la base de datos y luego inicia una nueva transacción con BEGIN.
8. Si idCustomers no está vacío, borra los clientes (customers) que tienen los ID obtenidos en el paso 1.

Además, en cada paso, se añade una traza de la operación realizada a la lista dbr. Al final de la función, dbr contendrá una lista de todas las operaciones realizadas.

```python
def query_not_in_order(db_conn, state, duerme, bCommit, dbr):
    # Empiezo obteniendo el id de todos los customer que estén en el state especificado
    query = text("select customerID from customers where state='" + str(state) + "';")
    idsC = list(db_conn.execute(query))
    idCustomers = [a[0] for a in idsC]
    dbr.append("LEER CUSTOMERS")

    # Obtengo las orderID de las orders de los customerid de la ciudad especificados
    if idCustomers:
        query = text("select orderID from orders where customerid IN " + str(idCustomers).replace("[", "(").replace("]", ")") + ";")
        idsO = list(db_conn.execute(query))
        idOrders = [a[0] for a in idsO]
    else:
        idOrders = []
    dbr.append("LEER PEDIDOS")

    if idOrders:
        # Borro pedidos de orderdetail
        query = text("delete from orderdetail where orderid in " + str(idOrders).replace("[", "(").replace("]", ")") + ";")
        db_conn.execute(query)
        dbr.append("BORRAR PRODUCTOS DE PEDIDOS")

    if bCommit:
        db_conn.execute(text("COMMIT;"))
        dbr.append("COMMIT")
        db_conn.execute(text("BEGIN;"))
        dbr.append("BEGIN")

    if duerme != 0:
        db_conn.execute(text("SELECT pg_sleep(" + str(duerme) + ");"))
        dbr.append("SLEEP")

    if idCustomers:
        # Borro customers
        query = text("delete from customers where customerid in " + str(idCustomers).replace("[", "(").replace("]", ")") + ";")
        db_conn.execute(query)
        dbr.append("BORRAR USUARIOS")

    if bCommit:
        db_conn.execute(text("COMMIT;"))
        dbr.append("COMMIT")
        db_conn.execute(text("BEGIN;"))
        dbr.append("BEGIN")

    if idOrders:
        # Borro pedidos de orders
        query = text("delete from orders where orderid in " + str(idOrders).replace("[", "(").replace("]", ")") + ";")
        db_conn.execute(query)
        dbr.append("BORRAR PEDIDOS")
```

Esta función realiza una serie de consultas y operaciones en la base de datos, similar a la función anterior “query_in_order”, pero en un orden diferente.

La principal diferencia con query_in_order es el orden en que se borran los clientes y los pedidos: en query_not_in_order, los clientes se borran antes que los pedidos.

Además del archivo “database.py”, hemos completado el archivo suministrado “routes.py”.

```python
@app.route('/', methods=['POST','GET'])
@app.route('/index', methods=['POST','GET'])
def index():
    return render_template('index.html')


@app.route('/borraEstado', methods=['POST','GET'])
def borraEstado():
    if 'state' in request.form: # Si contiene el campo state, se ha enviado el formulario -> POST
        state    = request.form["state"]
        bSQL     = request.form["txnSQL"]
        bCommit = "bCommit" in request.form
        bFallo  = "bFallo"  in request.form
        duerme  = request.form["duerme"]
        dbr = database.delState(state, bFallo, bSQL=='1', int(duerme), bCommit)
        return render_template('borraEstado.html', dbr=dbr)
    else: # Si no contiene el campo state, se ha accedido por primera vez -> GET
        return render_template('borraEstado.html')


@app.route('/topUK', methods=['POST','GET'])
def topUK():
    # TODO: consultas a MongoDB ...
    movies=[[],[],[]]

    movies[0] = database.db_Mongo_get_Movies_1994_1998()
    movies[1] = database.db_Mongo_get_Movies_Drama_1998_The()
    movies[2] = database.db_Mongo_get_FayeDunaway_and_ViggoMortensen()

    return render_template('topUK.html', movies=movies)
```

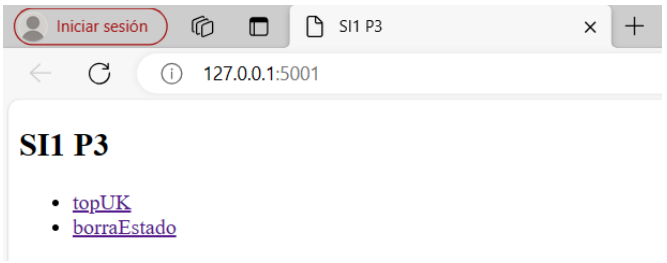
Hemos añadido las 3 querys implementadas anteriormente en este archivo para poder actualizar de manera correcta los datos de nuestra base de datos.

A continuación se muestra una ejecución de este apartado:

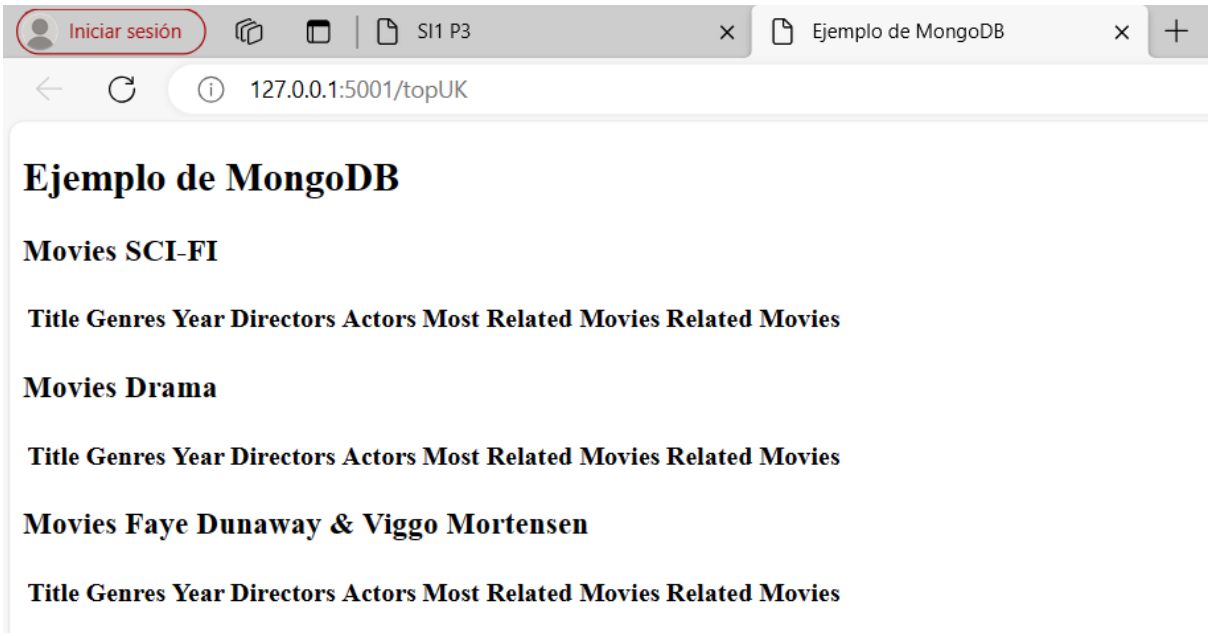
Primero, nos metemos en la terminal para activarlo y poder acceder desde nuestro navegador con los siguientes enlaces:

```
larry@DESKTOP-IMR556L:/mnt/c/users/Sergio/Documents/3_curso/Sistemas_Informaticos/Practicas/P3$ PYTHONPATH=. python3 -m app
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5001
 * Running on http://172.29.173.188:5001
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 106-237-374
```

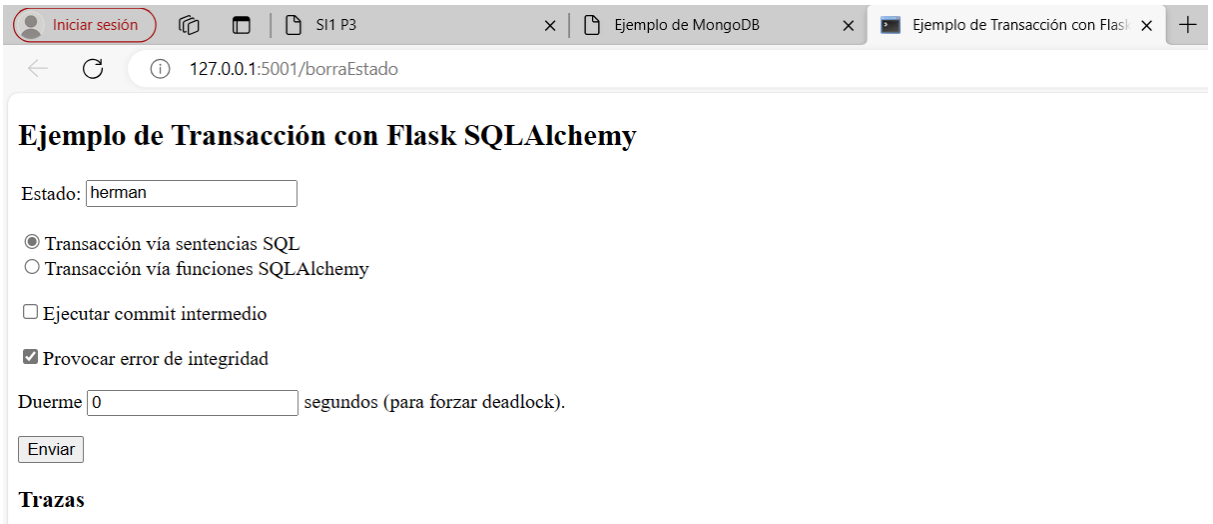
Accedemos a cualquiera de ellos y nos aparece este menú:



Si seleccionamos “topUK” nos encontramos la siguiente pantalla:

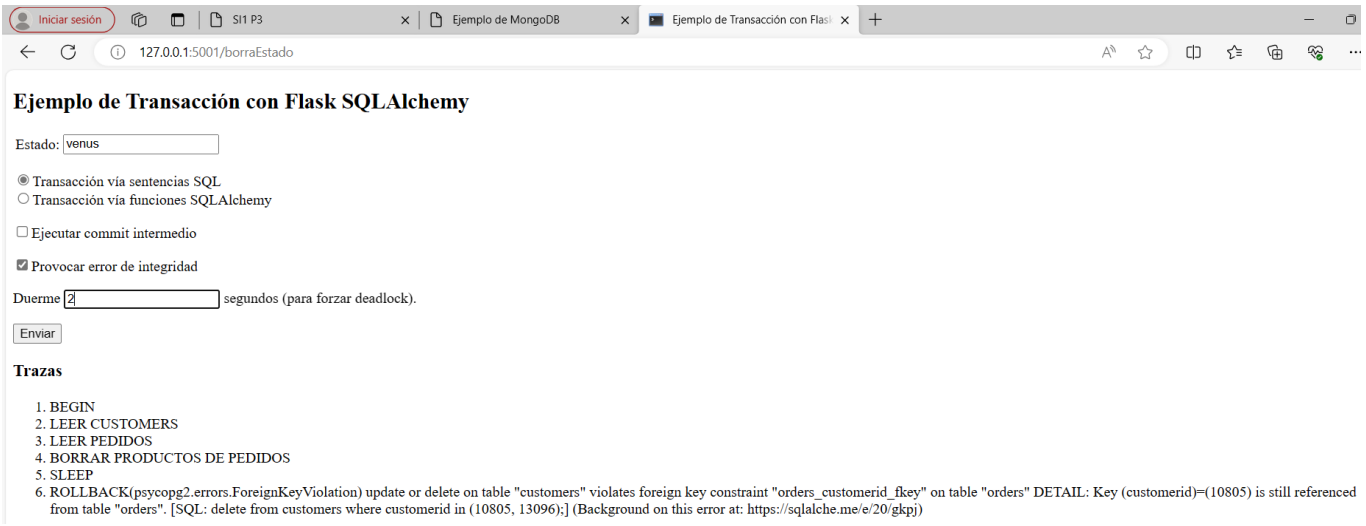


En ella podemos apreciar las 3 querys mencionadas anteriormente.
Por otro lado, si queremos realizar una transacción, seleccionaremos el botón “borraEstado” y nos llevará aqui:



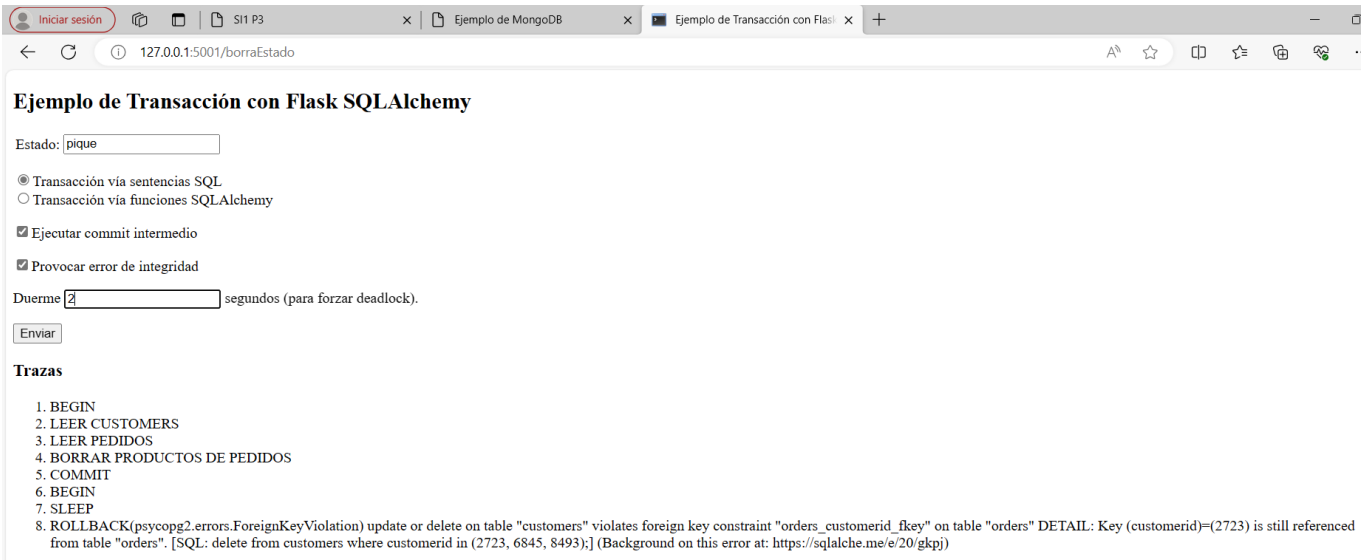
Una vez en este menú, ya podremos realizarlas correctamente.

Vamos a probar a borrar el estado “venus” mediante SQL, sin ejecutar commit intermedio y provocando un error de integridad con 2 segundos de deadlock:



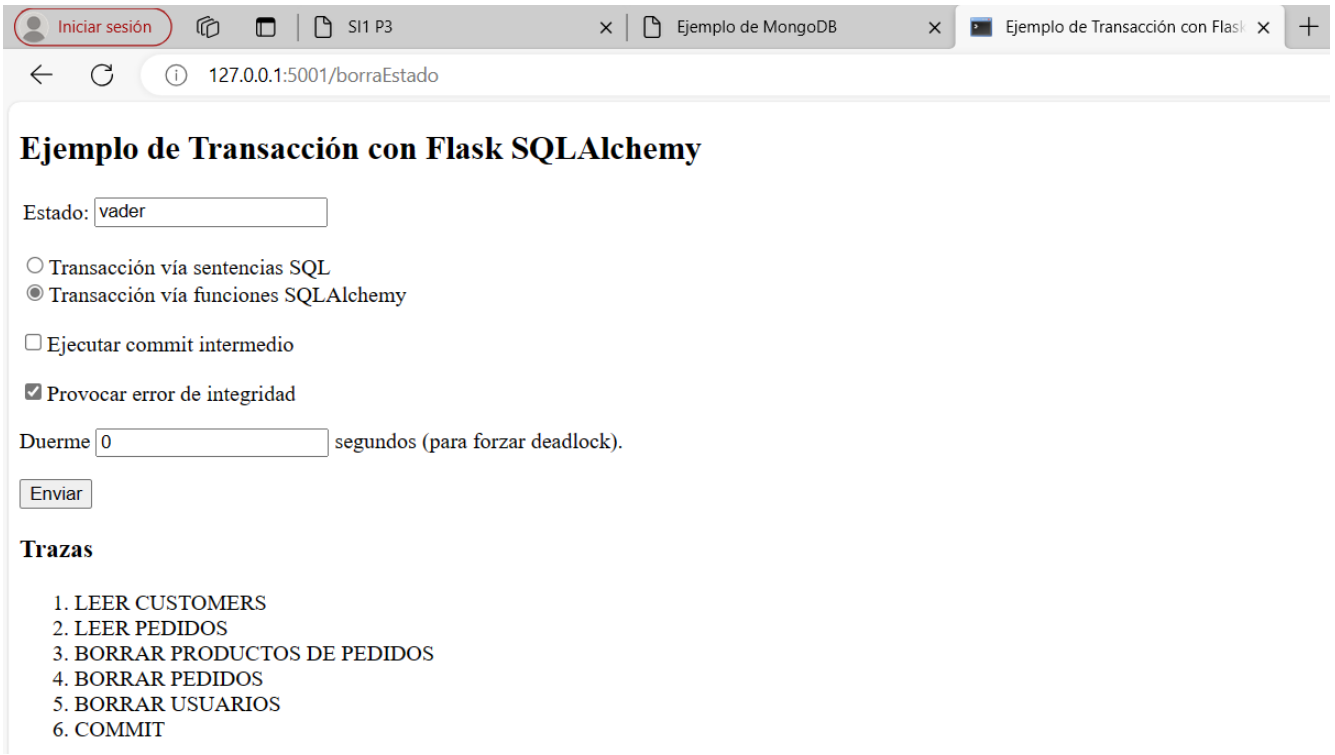
Como podemos observar, al provocar el error de integridad (situación en la que una operación viola las reglas de integridad de la base de datos) se produce un ROLLBACK, lo que nos indica que la transacción no se ha aplicado. Tras comprobarlo desde la base de datos, los datos de los clientes no se han borrado.

Ahora vamos a probar a ejecutarlo con fallo otra vez, pero esta vez haciendo un COMMIT intermedio:



Tras comprobarlo desde la base de datos, vemos que se han borrado los clientes de manera adecuada, incluso habiendo hecho ROLLBACK tal y como se puede apreciar en la imagen. Esto se debe a que al hacer el COMMIT intermedio, el ROLLBACK vuelve al momento después de borrar los detalles de los pedidos.

Vamos a probar a borrar el estado “vader”, sin ningún error, via funciones SQLAlchemy y con 0 segundos de deadlock:



Como podemos ver, la transacción se ha realizado correctamente.

Apartado B - Estudio de Bloqueos y Deadlocks

En este apartado se nos pide crear un script “updPromo.sql” con diferentes requisitos. El script que hemos diseñado tiene la siguiente funcionalidad:

- Creamos una columna “promo” en la tabla “customers” .
- Creamos un trigger y su función para aplicar el descuento de la columna anteriormente creada “promo” a los productos del carrito del cliente que actualice esta columna.
- Dicho trigger nos lleva a una modificación extra que es la adición de un SLEEP arbitrario entre las actualizaciones de las tablas “orderdetail” y “orders”. Este sleep se usa para generar el “deadlock” de la página del apartado anterior.
- Finalmente, como el enunciado nos pedía crear uno o varios carritos (status a NULL) mediante la sentencia UPDATE hemos creado 4 carritos en pedidos ya existentes.

A continuación se muestra nuestra implementación del script descrito:

```sql
ALTER TABLE customers ADD COLUMN IF NOT EXISTS promo REAL;

CREATE OR REPLACE FUNCTION updPromo_trigger()
RETURNS TRIGGER
AS $$
BEGIN
    UPDATE orderdetail as od
    SET price = round((price * (1 - NEW.promo/100))::NUMERIC, 2)
    FROM (SELECT od.orderid, od.prod_id
        FROM orders natural join orderdetail as od
        WHERE customerid = NEW.customerid and status IS NULL) as q
    WHERE q.prod_id = od.prod_id and q.orderid = od.orderid;
    PERFORM pg_sleep(10.5);
    UPDATE orders as o
    SET netamount = round(q.neto, 2),
        totalamount = round(q.neto * (1 + tax/100), 2)
    FROM (SELECT od.orderid, sum(price * quantity) as neto
            FROM orders natural join orderdetail as od
            WHERE NEW.customerid = customerid and status is NULL
            GROUP BY od.orderid) as q
    WHERE o.orderid = q.orderid;
    RETURN NEW;
END; $$
LANGUAGE plpgsql;

DROP TRIGGER updPromo ON customers;

CREATE TRIGGER updPromo
BEFORE UPDATE
OF promo ON customers
FOR EACH ROW
EXECUTE PROCEDURE updPromo_trigger();

-- Usuario 1 (U: shad, C: naples)
UPDATE orders SET status = NULL WHERE orderid = 103;

-- Usuario 2 (U: laxity, C: cesar)
UPDATE orders SET status = NULL WHERE orderid = 109;

-- Usuario 3 (U: flax, C: goat)
UPDATE orders SET status = NULL WHERE orderid = 116;

-- Usuario 4 (U: share, C: honor)
UPDATE orders SET status = NULL WHERE orderid = 124;
```

Una vez tenemos el script creado, procedemos a ejecutarlo:

```
si1=# SELECT * FROM orderdetail WHERE orderid IN (SELECT orderid FROM orders WHERE customerid = 1);
SELECT * FROM orders WHERE customerid = 1;
 orderid | prod_id |      price       | quantity
---------+---------+------------------+----------
     108 |    1256 | 13.3148404993065 |        1
     108 |    6125 | 9.15395284327323 |        1
     108 |    5873 | 14.9791955617198 |        1
     107 |    2648 | 14.9791955617198 |        1
     107 |    1204 | 15.8113730929265 |        1
     107 |    5825 | 14.1470180305132 |        1
     107 |    6422 | 23.7170596393897 |        1
     107 |    5066 | 10.8183079056865 |        1
     107 |    6088 | 16.2274618585298 |        1
     107 |    2597 | 11.9833564493759 |        1
     107 |     448 | 9.98613037447989 |        1
     105 |    1609 | 10.9847434119279 |        1
     106 |    4669 | 8.32177531206657 |        1
     106 |     724 | 15.8113730929265 |        1
     104 |    2105 | 13.9805825242718 |        1
     104 |    4709 | 9.98613037447989 |        1
     103 |    3500 |            9.882 |        1
     103 |     911 |           10.485 |        1
(18 rows)

 orderid | orderdate  | customerid |     netamount      | tax | totalamount |  status
---------+------------+------------+--------------------+-----+-------------+-----------
     108 | 2020-01-31 |          1 |  37.447988904299 53 |  15 |       43.07 | Shipped
     103 | 2016-12-28 |          1 |             20.367 |  15 |       23.42 |
     104 | 2019-03-07 |          1 |  23.96671289875169 |  15 |       27.56 | Shipped
     105 | 2016-11-29 |          1 |   10.98474341192 79 |  15 |       12.63 | Processed
     107 | 2016-10-28 |          1 | 117.66990291262129 |  15 |      135.32 | Shipped
     106 | 2019-12-31 |          1 |  24.133148404993 07 |  15 |       27.75 | Shipped
(6 rows)

si1=# \i updPromo.sql
psql:updPromo.sql:1: NOTICE:  column "promo" of relation "customers" already exists, skipping
ALTER TABLE
CREATE FUNCTION
DROP TRIGGER
CREATE TRIGGER
UPDATE 1
UPDATE 1
UPDATE 1
UPDATE 1
si1=# UPDATE customers SET promo = 10 WHERE customerid = 1;
UPDATE 1
si1=# SELECT * FROM orderdetail WHERE orderid IN (SELECT orderid FROM orders WHERE customerid = 1);
SELECT * FROM orders WHERE customerid = 1;
 orderid | prod_id |      price       | quantity
---------+---------+------------------+----------
     108 |    1256 | 11.9833564493759 |        1
     108 |    6125 | 8.23855755894591 |        1
     108 |    5873 | 13.4812760055478 |        1
     107 |    2648 | 13.4812760055478 |        1
     107 |    1204 | 14.2302357836338 |        1
     107 |    5825 | 12.7323162274619 |        1
     107 |    6422 | 21.3453536754507 |        1
     107 |    5066 | 9.73647711511785 |        1
     107 |    6088 | 14.6047156726768 |        1
     107 |    2597 | 10.7850208044383 |        1
     107 |     448 |  8.9875173370319 |        1
     105 |    1609 | 9.88626907073511 |        1
     106 |    4669 | 7.48959778085991 |        1
     106 |     724 | 14.2302357836338 |        1
     104 |    2105 | 12.5825242718446 |        1
     104 |    4709 |  8.9875173370319 |        1
     103 |    3500 |            8.001 |        1
     103 |     911 |            8.496 |        1
(18 rows)

 orderid | orderdate  | customerid |     netamount      | tax | totalamount |  status
---------+------------+------------+--------------------+-----+-------------+-----------
     108 | 2020-01-31 |          1 |  33.703190013869 61 |  15 |       38.76 | Shipped
     104 | 2019-03-07 |          1 |   21.5700416088765 |  15 |       24.81 | Shipped
     105 | 2016-11-29 |          1 |   9.88626907073511 |  15 |       11.37 | Processed
     107 | 2016-10-28 |          1 | 105.90291262135905 |  15 |      121.79 | Shipped
     106 | 2019-12-31 |          1 |  21.719833564493 71 |  15 |       24.98 | Shipped
     103 | 2016-12-28 |          1 |             16.497 |  15 |       18.97 |
(6 rows)

si1=#
```

Para ver su correcto funcionamiento ejecutamos la consulta:
“SELECT * FROM orderdetail WHERE orderid IN (SELECT orderid FROM orders WHERE customerid = 1); SELECT * FROM orders WHERE customerid = 1;”, la cuál selecciona todos los pedidos y los detalles para un cliente específico, en este caso, para el cliente con customerid = 1.

Después, ejecutamos el script creado “updPromo.sql” y podemos ver cómo se crea de forma correcta tanto la columna “promo” en la tabla customers como el trigger. Además, se actualizan de manera adecuada los 4 usuarios incluidos en el script.

Más adelante, ejecutamos la consulta:
“UPDATE customers SET promo = 10 WHERE customerid = 1;”, la cuál actualiza promo = 10 para el cliente de customerid = 1. Podemos apreciar cómo se actualiza correctamente.

Finalmente, comprobamos si se ha actualizado de manera adecuada los precios para el cliente con customerid = 1, y podemos observar que si. Se puede ver como varían las columnas: price, netamount y totalamount una vez aplicado el descuento.

Si comprobamos los cambios esperados tras la actualización de la columna promo de algún cliente, durante el proceso del trigger, desde otra terminal no se ven los cambios que esté haciendo e incluso poniendo el sleep no podemos ver los cambios que hace en la tabla orderdetail, y por supuesto, tampoco los de la tabla orders.

Esto sucede porque el proceso que está actualizando los registros de las tablas, las bloquea y hasta que no llegue al END, no confirma los datos ni los hace visibles. Pasa lo mismo cuando intentamos ver los datos que están siendo eliminados al eliminar una ciudad de la base de datos. Hasta que no se haga el COMMIT, no se podrán ver los cambios.

Si ajustamos los tiempos de los SLEEP, podemos comprobar que se llega a un “deadlock” con los bloqueos de las tablas “orderdetail” y “orders”. Esto sucede porque la transacción de la página web bloquea una de las tablas y se duerme, el trigger mientras tanto bloquea la otra tabla y se duerme. Cuando ambos se despierten, continuarán con sus cometidos y lo siguiente que les toca hacer es bloquear la tabla que tiene bloqueado el otro proceso. De manera que se esperan uno a otro sin fin.

Hemos discutido sobre cómo resolver este problema y hemos llegado a las siguiente conclusiones:

Evidentemente, si eliminamos los SLEEP se hace más improbable que pase esto, pero no imposible, así que como primera opción sería intentar cambiar el orden de bloqueos en caso de que se pueda, haciendo que ambos procesos bloqueen las tablas en el mismo orden, de manera que, si uno ha bloqueado una tabla, el otro intente bloquear primero la misma que este y no la otra tabla.Otra idea que hemos tenido es que se puede ejecutar un contador de espera a la hora de bloquear una tabla, y en caso de que el contador llegue a 0, por ejemplo, si están interbloqueados, salgan del bloqueo y vuelvan a intentar bloquear, volviendo a activar el contador. Es muy difícil que se llegue a interbloqueo todo el tiempo una vez hayan llegado al mismo punto.