Memoria de la Práctica 3 por Bernardo Andrés Zambrano Ferreira

1) NoSQL

1.1) Base de datos documental

Para esta parte, se utilizó MongoDB para crear una base de datos documental a partir de una base de datos relacional de películas en PostgreSQL. Se realizó un script en Python, que extrae la información de la BD PostgreSQL y crea otra BD en MongoDB llamada `si1`, que contiene la colección llamada `france` donde se almacenan los documentos.

A. Estructura del Documento

Cada documento en la colección `france` tiene 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

Las películas más relacionadas son aquellas que tienen una coincidencia en el 100% de los géneros, mientras que las relacionadas son aquellas que tienen una coincidencia en el 50%. Ambos conjuntos de películas relacionadas deben ser disjuntos y, obviamente, la película principal no puede aparecer como película relacionada.

B. Script `create_mongodb_from_postgresqldb.py`

Este script realiza las siguientes tareas:

1. **Configuración de las conexiones a PostgreSQL y MongoDB.**
2. **Extracción de los datos de la base de datos PostgreSQL.**
3. **Transformación de los datos en documentos de MongoDB.**
4. **Cálculo de las películas más relacionadas y relacionadas.**
5. **Inserción de los documentos en MongoDB.**

Archivo: `create_mongodb_from_postgresqldb.py`

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from pymongo import MongoClient
import itertools

# Configuración de la conexión a PostgreSQL
engine = create_engine('postgresql://alumnodb:alumnodb@localhost:5432/si1')
```

```
Session = sessionmaker(bind=engine)
session = Session()

# Prueba de conexión a la base de datos
try:
    session.execute('SELECT 1')
    print("Conexión a la base de datos establecida")
except Exception as e:
    print("Error en la conexión a la base de datos", e)

# Configuración de la conexión a MongoDB
client = MongoClient('localhost', 27017)
db = client.si1

# Prueba de conexión a la base de datos
try:
    db.command('ping')
    print("Conexión a MongoDB establecida")
except Exception as e:
    print("Error en la conexión a MongoDB", e)

# Definir la consulta para extraer películas francesas
query = """
SELECT
    m.movietitle AS title,
    STRING_AGG(DISTINCT g.genre, ', ') AS genres,
    m.year::int,
    array_agg(DISTINCT d.directorname) AS directors,
    array_agg(DISTINCT a.actorname) AS actors
FROM
    imdb_movies m
JOIN
    imdb_moviegenres g ON m.movieid = g.movieid
JOIN
    imdb_directormovies md ON m.movieid = md.movieid
JOIN
    imdb_directors d ON md.directorid = d.directorid
JOIN
    imdb_actormovies ma ON m.movieid = ma.movieid
JOIN
    imdb_actors a ON ma.actorid = a.actorid
JOIN
    imdb_moviecountries mc ON m.movieid = mc.movieid
WHERE
    mc.country = 'France' OR mc.country LIKE '%France%'
GROUP BY
    m.movietitle, m.year;
"""

# Ejecutar la consulta
movies = session.execute(query).fetchall()

# Transformar los resultados en una lista de diccionarios para facilitar el
procesamiento
```

```
movie_list = []
for movie in movies:
    movie_dict = {
        "title": movie.title,
        "genres": movie.genres.split(', '),
        "year": movie.year,
        "directors": movie.directors,
        "actors": movie.actors,
        "most_related_movies": [],
        "related_movies": []
    }
    movie_list.append(movie_dict)

# Función para encontrar películas relacionadas
def find_related_movies(movie, movie_list):
    most_related = []
    related = []

    for other_movie in movie_list:
        if other_movie["title"] == movie["title"]:
            continue

        common_genres =
set(movie["genres"]).intersection(set(other_movie["genres"]))

        if len(common_genres) == len(movie["genres"]):
            most_related.append({"title": other_movie["title"], "year":
other_movie["year"]})
        elif len(common_genres) >= len(movie["genres"]) / 2:
            related.append({"title": other_movie["title"], "year":
other_movie["year"]})

    # Limitar a 10 resultados
    most_related = sorted(most_related, key=lambda x: x["year"],
reverse=True)[:10]
    related = sorted(related, key=lambda x: x["year"], reverse=True)[:10]

    return most_related, related

# Procesar cada película para encontrar las relacionadas y cargar en
MongoDB
for movie in movie_list:
    most_related, related = find_related_movies(movie, movie_list)
    movie["most_related_movies"] = most_related
    movie["related_movies"] = related

    # Insertar el documento en la colección 'france'
    db.france.insert_one(movie)

print("Datos transferidos exitosamente a MongoDB")

# Cerrar las conexiones
session.close()
client.close()
```

C. Script `mongodb_queries.py`

Este script ejecuta y muestra los resultados de las siguientes consultas en MongoDB:

1. **Películas de ciencia ficción entre 1994 y 1998.**
2. **Películas de drama de 1998 que empiecen por "The".**
3. **Películas en las que Faye Dunaway y Viggo Mortensen hayan compartido reparto.**

Archivo: `mongodb_queries.py`

```
from pymongo import MongoClient

# Conexión a la base de datos
client = MongoClient('localhost', 27017)
db = client.si1

# Consulta 1: Ciencia ficción entre 1994 y 1998
query1 = {
    "genres": "Sci-Fi",
    "year": {"$gte": 1994, "$lte": 1998}
}

sci_fi_movies = db.france.find(query1)

# Mostrar los resultados
print("Ciencia ficción entre 1994 y 1998")
for movie in sci_fi_movies:
    print(movie)

# Consulta 2: Películas de drama del año 1998 que empiezan por "The"
query2 = {
    "genres": { '$in': ["Drama"]},
    "year": 1998,
    "title": {"$regex": "^The", "$options": "i"}
}

drama_movies = db.france.find(query2)

# Mostrar los resultados
print("\nPelículas de drama del año 1998 que empiezan por 'The'")
for movie in drama_movies:
    print(movie)

# Consulta 3: Películas en las que Faye Dunaway y Viggo Mortensen hayan
trabajado juntos
query3 = {
    "actors": {"$all": ["Faye Dunaway", "Viggo Mortensen"]}
}

shared_movies = db.france.find(query3)
```

```
# Mostrar los resultados
print("\nPelículas en las que Faye Dunaway y Viggo Mortensen han trabajado juntos")
for movie in shared_movies:
    print(movie)

# Cerrar la conexión
client.close()
```

Descripción del material entregado

Se han entregado los siguientes archivos:

- **`create_mongodb_from_postgresqldb.py`**: Script para extraer datos de PostgreSQL y cargar una base de datos documental en MongoDB.
- **`mongodb_queries.py`**: Script para ejecutar y mostrar los resultados de consultas en MongoDB.

1.2) BBDD basadas en Grafos

A. Creación de la Base de Datos basada en grafos en Neo4j

Archivo: `create_neo4jdb_from_postgresqldb.py`

Este script se encarga de extraer datos de la base de datos PostgreSQL y crear una base de datos en Neo4j. La base de datos contiene las 20 películas estadounidenses más vendidas junto con sus respectivos actores y directores. A continuación, se muestra el código del script:

```
from neo4j import GraphDatabase
from sqlalchemy import create_engine, text
from sqlalchemy.orm import sessionmaker

# Configuración de la conexión a PostgreSQL
engine = create_engine('postgresql://alumnodb:alumnodb@localhost:5432/si1')
Session = sessionmaker(bind=engine)
session = Session()

# Configuración de la conexión a Neo4j
uri = "bolt://localhost:7687"
driver = GraphDatabase.driver(uri, auth=("neo4j", "si1-password"))

# Prueba de conexión a PostgreSQL
try:
    session.execute('SELECT 1')
    print("Conexión a la base de datos PostgreSQL establecida")
except Exception as e:
    print("Error en la conexión a la base de datos PostgreSQL:", e)

# Prueba de conexión a Neo4j
```

```
try:
    with driver.session() as session:
        session.run("RETURN 1")
    print("Conexión a Neo4j establecida")
except Exception as e:
    print("Error en la conexión a

 Neo4j:", e)

# Extraer datos de PostgreSQL
query = """
SELECT
    m.movieid,
    m.movietitle AS title,
    CASE
        WHEN m.year ~ '^[0-9]{4}$' THEN m.year::int
        ELSE NULL
    END AS year,
    array_agg(DISTINCT d.directorid || ':' || d.directorname) AS directors,
    array_agg(DISTINCT a.actorid || ':' || a.actorname) AS actors
FROM
    imdb_movies m
JOIN
    imdb_moviegenres g ON m.movieid = g.movieid
JOIN
    imdb_directormovies md ON m.movieid = md.movieid
JOIN
    imdb_directors d ON md.directorid = d.directorid
JOIN
    imdb_actormovies ma ON m.movieid = ma.movieid
JOIN
    imdb_actors a ON ma.actorid = a.actorid
JOIN
    imdb_moviecountries mc ON m.movieid = mc.movieid
WHERE
    mc.country = 'USA'
    AND m.year ~ '^[0-9]{4}$'
GROUP BY
    m.movieid, m.movietitle, m.year
ORDER BY
    m.year DESC
LIMIT 20;
"""

connection = engine.connect()
movies = connection.execute(text(query)).fetchall()

# Funciones para crear nodos y relaciones en Neo4j
def create_movie_node(tx, movieid, title, year):
    tx.run("MERGE (m:Movie {movieid: $movieid, title: $title, year:
$year})", movieid=movieid, title=title, year=year)

def create_person_node(tx, personid, name, role):
    if role == 'Director':
```

```
        tx.run("MERGE (p:Person {personid: $personid, name: $name}) SET
p:Director", personid=personid, name=name)
    elif role == 'Actor':
        tx.run("MERGE (p:Person {personid: $personid, name: $name}) SET
p:Actor", personid=personid, name=name)

def create_relationship(tx, movieid, personid, role):
    if role == 'Director':
        tx.run("""
            MATCH (m:Movie {movieid: $movieid}),
                  (p:Person {personid: $personid})
            MERGE (p)-[:DIRECTED]->(m)
            """, movieid=movieid, personid=personid)
    elif role == 'Actor':
        tx.run("""
            MATCH (m:Movie {movieid: $movieid}),
                  (p:Person {personid: $personid})
            MERGE (p)-[:ACTED_IN]->(m)
            """, movieid=movieid, personid=personid)

# Crear nodos y relaciones en Neo4j
with driver.session() as session:
    for movie in movies:
        session.execute_write(create_movie_node, movie.movieid,
movie.title, movie.year)
        for director in movie.directors:
            director_id, director_name = director.split(':')
            session.execute_write(create_person_node, director_id,
director_name, 'Director')
            session.execute_write(create_relationship, movie.movieid,
director_id, 'Director')
        for actor in movie.actors:
            actor_id, actor_name = actor.split(':')
            session.execute_write(create_person_node, actor_id, actor_name,
'Actor')
            session.execute_write(create_relationship, movie.movieid,
actor_id, 'Actor')

print("Datos transferidos exitosamente a Neo4j")
connection.close()
driver.close()
```

Este script realiza las siguientes acciones:

1. Conecta a la base de datos PostgreSQL y a Neo4j.
2. Extrae las 20 películas estadounidenses más vendidas junto con sus directores y actores de la base de datos PostgreSQL.
3. Crea nodos de película y persona en Neo4j.
4. Establece relaciones entre películas y personas en Neo4j.

B. Consultas en Neo4j

Archivo: `execute_neo4j_queries.py`

Este script se utiliza para ejecutar las consultas especificadas en el punto 1.2 C. A continuación, se muestra el código del script:

```
from neo4j import GraphDatabase

# Conexión a la base de datos Neo4j
uri = "bolt://localhost:7687"
driver = GraphDatabase.driver(uri, auth=("neo4j", "si1-password"))

def execute_query(query_file):
    with driver.session() as session:
        with open(query_file, 'r') as file:
            query = file.read()
        result = session.run(query)
        return list(result)  # Recolectar todos los registros en una lista

# Ejecutar y mostrar los resultados de las consultas
queries = [
    ("winston-hattie-co-co-actors.cypher", "Actores que no han trabajado
con 'Winston, Hattie' pero han trabajado con un tercero en común:"),
    ("pair-persons-most-occurrences.cypher", "Pares de personas que han
trabajado juntas en más de una película:"),
    ("degrees-reiner-to-smyth.cypher", "Camino mínimo entre 'Reiner, Carl'
y 'Smyth, Lisa (I)':")
]

for query_file, description in queries:
    print(description)
    result = execute_query(f"app-neo4j-etl/{query_file}")
    for record in result:
        print(record)
    print("\n")

driver.close()
```

Este script realiza las siguientes consultas en Neo4j:

1. `winston-hattie-co-co-actors.cypher`: Devuelve 10 actores ordenados alfabéticamente que no han trabajado con "Winston, Hattie", pero que en diferentes momentos han trabajado con un tercero en común.
2. `pair-persons-most-occurrences.cypher`: Muestra en cada fila pares de personas que han trabajado juntas en más de una película, sin distinguir categoría de actores o directores.
3. `degrees-reiner-to-smyth.cypher`: Halla el camino mínimo por el cual el director "Reiner, Carl" podría conocer a la actriz "Smyth, Lisa (I)".

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'
WITH a1, a2, COUNT(DISTINCT m) AS movies_together
WHERE movies_together > 1
RETURN DISTINCT a1.name AS Actor1, a2.name AS Actor2
ORDER BY a1.name, a2.name
LIMIT 10;
```

pair-persons-most-occurrences.cypher

```
MATCH (p1:Person)-[:ACTED_IN|DIRECTED]->(m:Movie)<-[:ACTED_IN|DIRECTED]-
(p2:Person)
WHERE p1 <> p2
WITH p1, p2, COUNT(m) AS movies_together
WHERE movies_together > 1
RETURN p1.name AS Person1, p2.name AS Person2, movies_together
ORDER BY movies_together DESC, Person1, Person2;
```

degrees-reiner-to-smyth.cypher

```
MATCH p=shortestPath((d:Director {name: 'Reiner, Carl'})-
[:ACTED_IN|DIRECTED*]-(a:Actress {name: 'Smyth, Lisa (I)'}))
RETURN p;
```

---

2) Uso de tecnología caché de acceso rápido con Redis

A. Creación de la base de datos en memoria con Redis

Archivo: `create_redis_from_postgresqldb.py`

Este script se encarga de extraer datos de la base de datos PostgreSQL y crear una base de datos en memoria en Redis para guardar la cantidad de visitas de los clientes a la página web. Se utiliza la estructura de datos hashes proporcionada por Redis. A continuación, se muestra el código del script:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
import redis
import random

# Configuración de la conexión a PostgreSQL
engine = create_engine('postgresql://alumnodb:alumnodb@localhost:5432/si1')
Session = sessionmaker(bind=engine)
session = Session()

# Prueba de conexión a PostgreSQL
```

```
try:
    session.execute('SELECT 1')
    print("Conexión a la base de datos PostgreSQL establecida")
except Exception as e:
    print("Error en la conexión a la base de datos PostgreSQL:", e)

# Configuración de la conexión a Redis
r = redis.Redis(host='localhost', port=6379, db=0)

# Prueba de conexión a Redis
try:
    r.ping()
    print("Conexión a Redis establecida")
except Exception as e:
    print("Error en la conexión a Redis:", e)

# Definir la consulta para extraer usuarios de España
query = """
SELECT
    c.email,
    c.firstname || ' ' || c.lastname AS name,
    c.phone
FROM
    customers c
WHERE
    c.country = 'Spain'
"""

# Ejecutar la consulta
customers = session.execute(query).fetchall()

# Guardar los datos en Redis
for customer in customers:
    email = customer.email
    name = customer.name
    phone = customer.phone
    visits = random.randint(1, 99)

    r.hset(f"customers:{email}", mapping={
        "name

": name,
        "phone": phone,
        "visits": visits
    })

print("Datos transferidos exitosamente a Redis")
```

Este script realiza las siguientes acciones:

1. Conecta a la base de datos PostgreSQL y a Redis.
2. Extrae los usuarios de España de la base de datos PostgreSQL.

3. Guarda los datos de los usuarios en Redis usando la estructura de datos hashes.

B. Funciones para interactuar con Redis

En el mismo script, se definen las funciones requeridas para interactuar con la base de datos en memoria en Redis:

```
# Definir las funciones requeridas

def increment_by_email(email):
    if r.hexists(f"customers:{email}", "visits"):
        r.hincrby(f"customers:{email}", "visits", 1)
        print(f"Visitas incrementadas para {email}")
    else:
        print(f"No se encontró el cliente con el email {email}")

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().split(":", 1)[1]

    return max_email

def get_field_by_email(email):
    if r.hexists(f"customers:{email}", "name"):
        name = r.hget(f"customers:{email}", "name").decode()
        phone = r.hget(f"customers:{email}", "phone").decode()
        visits = int(r.hget(f"customers:{email}", "visits"))
        return {"name": name, "phone": phone, "visits": visits}
    else:
        print(f"No se encontró el cliente con el email {email}")
        return None

# Ejecución de las funciones requeridas para los puntos del ejercicio

# Incrementar visitas para un email específico
test_email = "ballsy.cobra@jmail.com"
increment_by_email(test_email)

# Obtener el cliente con más visitas
most_visits_email = customer_most_visits()
print(f"Cliente con más visitas: {most_visits_email}")

# Obtener campos para un email específico
fields = get_field_by_email(test_email)
```

```
if fields:
    print(f"Datos del cliente {test_email}: {fields}")
```

Las funciones definidas son:

1. **increment_by_email(email)**: Incrementa en 1 el número de visitas del cliente con el email especificado.
2. **customer_most_visits()**: Devuelve el email del cliente con mayor cantidad de visitas.
3. **get_field_by_email(email)**: Muestra el nombre, el teléfono y número de visitas del cliente con el email especificado.

Estas funciones permiten interactuar con la base de datos en memoria en Redis para realizar las operaciones requeridas en el ejercicio.

---

3) Transacciones

A. Estudio de transacciones

a. La página recibirá la city de los clientes a borrar, que será solicitada mediante un formulario por la propia página.

En este subapartado, se configura una página web para recibir la ciudad (`city`) de los clientes que se desean borrar. Se utiliza un formulario HTML para este propósito.

b. La transacción deberá tener mecanismos de rollback.

Implementamos mecanismos de rollback para asegurar que cualquier error durante la transacción revierte todos los cambios realizados hasta ese momento. Esto se maneja en el archivo `database.py`.

c. Se usarán sentencias SQL (vía execute()) para gestionar la transacción.

Utilizamos sentencias SQL para manejar las transacciones. Esto se implementa en las funciones `delState` y `delCity`.

d. Para poder realizar este ejercicio, se deberán desactivar (eliminar) en la base de datos, si las hubiera, todas las restricciones ON DELETE CASCADE referentes al cliente y sus pedidos, pero manteniendo las foreign keys que aseguran la integridad.

Este paso se asegura manualmente en la base de datos PostgreSQL.

e. Elaborar una versión de la página, controlada por un argumento, que implemente este borrado de registros en un orden incorrecto, de forma que se provoque un fallo de restricción de foreign key.

Se implementa la lógica para provocar un fallo de restricción de clave foránea mediante un argumento que controla el orden de las operaciones de borrado.

f. Se deberá entonces realizar un rollback de la transacción para volver a la situación original, deshaciendo los cambios realizados hasta ese momento.

El rollback se maneja en caso de error dentro de las funciones de transacción.

g. Elaborar otra versión con el orden de borrado correcto, y que funcione de la forma esperada y borre todos los registros asociados a un cliente.

Se implementa la lógica de borrado en el orden correcto para asegurar que todos los registros asociados se borren sin errores.

h. Se deberá realizar un control de errores adecuado, y no considerar el NOT FOUND como un error.

El control de errores se maneja adecuadamente en las funciones de transacción.

i. Se mostrarán en la página resultante trazas de los cambios parciales que se van realizando, y cómo estos cambios se deshacen al realizar un rollback.

Las trazas de los cambios parciales y los rollbacks se registran y muestran en la página resultante.

j. Alterar la versión incorrecta de la página para que se realice algún COMMIT intermedio (seguido de BEGIN, ¿por qué?), antes de producirse el error, y comprobar que los cambios realizados antes del COMMIT persisten tras el ROLLBACK.

Se añade lógica para realizar commits intermedios y verificar que los cambios realizados antes del commit persisten después de un rollback.

Archivo: `database.py`

```
# -*- coding: utf-8 -*-

import os
import sys, traceback, time
import sqlalchemy as sa
from sqlalchemy import create_engine, text
from pymongo import MongoClient

# Configurar el motor de sqlalchemy
db_engine = create_engine("postgresql://alumnodb:alumnodb@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 dbConnect():
    return db_engine.connect()
```

```
def dbCloseConnect(db_conn):
    db_conn.close()

def delState(state, bFallo, bSQL, duerme, bCommit):
    # Array de trazas a mostrar en la página
    dbr = []

    # Consultas de borrado
    query_orderdetail = """
    WITH city_orders AS (
        SELECT o.orderid
        FROM orders o JOIN customers c ON c.customerid = o.customerid
        WHERE c.city = :city
    )
    DELETE FROM orderdetail
    WHERE orderid IN (SELECT orderid FROM city_orders);
    """

    query_orders = """
    WITH city_customers AS (
        SELECT customerid
        FROM customers
        WHERE city = :city
    )
    DELETE FROM orders
    WHERE customerid IN (SELECT customerid FROM city_customers);
    """

    query_customers = """
    DELETE FROM customers
    WHERE city = :city;
    """

    try:
        # Establecer conexión y comenzar transacción
        connection = db_engine.connect()
        transaction = connection.begin()
        if bSQL:
            connection.execute(text("BEGIN"))

        if bFallo:
            # Orden de borrado incorrecto que provocará un fallo
            dbr.append("Intentando borrar registros en un orden que
provocará un error")
            connection.execute(text(query_customers), {"city": state})
        else:
            # Orden de borrado correcto
            dbr.append("Borrando registros de forma segura")
            connection.execute(text(query_orderdetail), {"city": state})
            connection.execute(text(query_orders), {"city": state})
            connection.execute(text(query_customers), {"city": state})

        # Suspender ejecución si se especifica
        if duerme > 0:
```

```
            dbr.append(f"Suspendiendo ejecución por {duerme} segundos")
            time.sleep(duerme)

        # Commit intermedio
        if bCommit:
            dbr.append("Realizando commit intermedio")
            if bSQL:
                connection.execute(text("COMMIT"))
                connection.execute(text("BEGIN"))
            else:
                transaction.commit()
                transaction = connection.begin()

        # Confirmar cambios si todo va bien
        if bSQL:
            connection.execute(text("COMMIT"))
        else:
            transaction.commit()
        dbr.append("Borrado exitoso")

    except Exception as e:
        # Manejar errores y realizar rollback
        if bSQL:
            connection.execute(text("ROLLBACK"))
        else:
            transaction.rollback()
        dbr.append(f"Error: {str(e)}")
        return dbr

    finally:
        connection.close()

    return dbr

def delCity(city, bFallo, bSQL, seg, bCommit):
    dbr = []  # Array para guardar trazas

    dbr.append(f"Running function delCity with arguments: city = {city},
bFallo = {bFallo}, bSQL = {bSQL}, seg = {seg}, bCommit = {bCommit}")
    query_orderdetail = """
    WITH city_orders AS (
        SELECT o.orderid
        FROM orders o JOIN customers c ON c.customerid = o.customerid
        WHERE c.city = :city
    )
    DELETE


    FROM orderdetail
    WHERE orderid IN (SELECT orderid FROM city_orders);
    """

    query_orders = """
    WITH city_customers AS (
```

```
        SELECT customerid
        FROM customers
        WHERE city = :city
    )
    DELETE
    FROM orders
    WHERE customerid IN (SELECT customerid FROM city_customers);
    """

    query_customers = """
    DELETE
    FROM customers
    WHERE city = :city;
    """

    # Establece la conexión y comienza la transacción
    with db_engine.connect() as connection:
        if bSQL:
            connection.execute(text("BEGIN TRANSACTION;"))
            dbr.append("La transaccion ha comenzado")

            try:
                if bFallo:
                    # Orden de borrado incorrecto que provocará un fallo
                    dbr.append("Intentando borrar registros en un orden que
provocará un error...")
                    connection.execute(text(query_customers), {"city":
city})

                    if bCommit:
                        connection.execute(text("COMMIT;"))
                        dbr.append("Eliminacion de customers confirmada")
                    else:
                        dbr.append("Customers eliminado")

                    connection.execute(text(query_orders), {"city": city})

                    if bCommit:
                        connection.execute(text("COMMIT;"))
                        dbr.append("Eliminacion de orders confirmada")
                    else:
                        dbr.append("Orders eliminado")

                    connection.execute(text(query_orderdetail), {"city":
city})

                    if bCommit:
                        dbr.append("Eliminacion de orderdetail confirmada")
                    else:
                        dbr.append("Orderdetail eliminado")
                else:
                    connection.execute(text(query_orderdetail), {"city":
city})
```

```
                    if bCommit:
                        connection.execute(text("COMMIT;"))
                        dbr.append("Orderdetail deletion committed")
                    else:
                        dbr.append("Orderdetail eliminado")

                    connection.execute(text(query_orders), {"city": city})

                    if bCommit:
                        connection.execute(text("COMMIT;"))
                        dbr.append("Eliminacion de orders confirmada")
                    else:
                        dbr.append("Orders eliminado")

                    connection.execute(text(query_customers), {"city":
city})

                    if bCommit:
                        dbr.append("Eliminacion de customers confirmada")
                    else:
                        dbr.append("Customers eliminado")

            except Exception as e:
                dbr.append(f"Error ejecutando consultas: {e}")
                connection.execute(text("ROLLBACK;"))
                dbr.append("SQL rollback completado")
            else:
                time.sleep(seg)
                connection.execute(text("COMMIT;"))
                dbr.append("SQL commit completado")

        else:
            transaction = connection.begin()
            dbr.append("Transaccion con funciones SQLAlchemy")
            try:
                if bFallo:
                    dbr.append("Intentando borrar registros en un orden que
provocará un error...")
                    connection.execute(text(query_customers), {"city":
city})

                    if bCommit:
                        connection.execute(text("COMMIT;"))
                        dbr.append("Eliminacion de customers confirmada")
                    else:
                        dbr.append("Customers eliminado")

                    connection.execute(text(query_orders), {"city": city})

                    if bCommit:
                        connection.execute(text("COMMIT;"))
                        dbr.append("Eliminacion de orders confirmada")
                    else:
                        dbr.append("Orders eliminado")
```

```
                    connection.execute(text(query_orderdetail), {"city":
city})

                    if bCommit:
                        dbr.append("Eliminacion de orderdetail confirmada")
                    else:
                        dbr.append("Orderdetail eliminado")
                else:
                    # Successful path
                    connection.execute(text(query_orderdetail), {"city":
city})

                    if bCommit:
                        connection.execute(text("COMMIT;"))
                        dbr.append("Orderdetail deletion committed")
                    else:
                        dbr.append("Orderdetail eliminado")

                    connection.execute(text(query_orders), {"city": city})

                    if bCommit:
                        connection.execute(text("COMMIT;"))
                        dbr.append("Eliminacion de orders confirmada")
                    else:
                        dbr.append("Orders eliminado")

                    connection.execute(text(query_customers), {"city":
city})

                    if bCommit:
                        dbr.append("Eliminacion de customers confirmada")
                    else:
                        dbr.append("Customers eliminado")

            except Exception as e:
                dbr.append(f"Error ejecutando consultas: {e}")
                transaction.rollback()
                dbr.append("SQLAlchemy rollback completado")
            else:
                time.sleep(seg)
                transaction.commit()
                dbr.append("SQLAlchemy commit completado")

    dbr.append("Proceso completado")

    return dbr
```

Archivo: routes.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```
from app import app
from app import database
from flask import render_template, request, url_for
import os
import sys
import time

@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:
        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:
        return render_template('borraEstado.html')

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

@app.route('/borraCiudad', methods=['GET', 'POST'])
def borraCiudad():
    if request.method == 'POST':
        city = request.form['city']
        bFallo = request.form.get('bFallo') == 'on'
        bSQL = request.form.get('bSQL') == 'on'
        seg = int(request.form.get('seg', 0))
        bCommit = request.form.get('bCommit') == 'on'

        result = database.delCity(city, bFallo, bSQL, seg, bCommit)

        return render_template('resultado.html', result=result)
    return render_template('borraCiudad.html')
```

En estos archivos, `database.py` maneja las operaciones de base de datos y las transacciones, mientras que `routes.py` define las rutas y la lógica para recibir y procesar los formularios web. Las funciones de borrado implementan los requisitos de transacciones, incluyendo rollback y commit, y proporcionan trazas detalladas para el seguimiento de la ejecución y los errores.

B. Estudio de bloqueos y deadlocks

En este apartado se busca entender cómo funcionan los bloqueos y deadlocks en una base de datos PostgreSQL. Los bloqueos y deadlocks ocurren cuando dos o más transacciones están esperando que el otro libere un recurso, lo que resulta en una situación en la que ninguna transacción puede proceder.

El archivo `updPromo.sql` contiene un script para crear una nueva columna `promo` en la tabla `customers` y un trigger que aplica un descuento a los artículos en el carrito del cliente cuando se actualiza la columna `promo`. El trigger incluye un `pg_sleep` para simular un retraso en la ejecución, lo cual puede ayudar a ilustrar cómo se producen los bloqueos y deadlocks.

a. Partir de una base de datos limpia (recién creada y cargada de datos).

Esto implica que la base de datos se debe reiniciar y recargar con los datos iniciales para asegurar un estado conocido y predecible.

b. Crear un script `updPromo.sql`, que creará una nueva columna, `promo`, en la tabla `customers`. Esta columna contendrá un descuento (en porcentaje) promocional.

El script `updPromo.sql` agrega una columna `promo` a la tabla `customers`:

```
ALTER TABLE customers ADD COLUMN promo NUMERIC;
```

c. Añadir al script la creación de un trigger sobre la tabla `customers` de forma que al alterar la columna `promo` de un cliente, se le haga un descuento en los artículos de su cesta o carrito del porcentaje indicado en la columna `promo` sobre el precio de la tabla `products`.

Se crea una función `apply_promo` que se ejecutará como un trigger cuando la columna `promo` de la tabla `customers` sea actualizada. Esta función aplica el descuento a los artículos en el carrito del cliente.

```
CREATE OR REPLACE FUNCTION apply_promo() RETURNS TRIGGER AS $$
BEGIN
    -- Introduce una pausa de 10 segundos
    PERFORM pg_sleep(10);

    UPDATE orderdetail
    SET price = price * (1 - NEW.promo / 100)
    WHERE orderid IN (
        SELECT orderid
        FROM orders
        WHERE customerid = NEW.customerid
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_apply_promo
AFTER UPDATE OF promo ON customers
```

```
FOR EACH ROW
EXECUTE FUNCTION apply_promo();
```

d. Modificar el trigger para que haga un `sleep` durante su ejecución (`sentencia PERFORM pg_sleep(nn)`) en el momento adecuado.

El `pg_sleep(10)` introduce un retraso de 10 segundos en la ejecución del trigger. Este retraso puede causar bloqueos si otra transacción intenta acceder a los mismos recursos mientras el trigger está en ejecución.

j. Ajustar el punto en que se hacen los `sleep` para conseguir un deadlock y explicar por qué se produce.

Introducir `sleep` en puntos específicos para forzar un deadlock y explicar las razones por las que ocurre, destacando cómo las transacciones compiten por los mismos recursos en un orden que conduce a un impasse.

k. Discutir cómo afrontar o evitar este tipo de problema.

Para evitar deadlocks, se pueden seguir varias estrategias, como asegurar un orden consistente de acceso a los recursos, utilizar tiempos de espera (`timeouts`) para detectar y resolver deadlocks, y revisar y optimizar las transacciones para minimizar el tiempo durante el cual los bloqueos se mantienen.

En resumen, el archivo `updPromo.sql` y las instrucciones relacionadas ayudan a comprender y demostrar cómo se producen los bloqueos y deadlocks en una base de datos, y cómo se pueden manejar mediante el uso adecuado de transacciones, retrasos simulados (`pg_sleep`), y la observación de los efectos en tiempo real utilizando herramientas de administración de bases de datos.