MEMORIA P3
Autores: Daniel Aquino y Enrique Gómez

1.1) Base de datos documental

a. & b.

```python
from pymongo import MongoClient
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey, text

db_engine = create_engine("postgresql://alumnodb:1234@localhost/si1", echo=False)
db_connect = db_engine.connect()

def clean_year_title(title):
    ntitle = title[:-7]
    return ntitle

def main():
    db_connect = db_engine.connect()

    query = text("Select M.movieid from imdb_movies AS M JOIN imdb_moviecountries AS MC ON M.movieid = MC.movieid WHERE MC.country = 'France'")
    db_result = db_connect.execute(query).fetchall()

    mc = MongoClient("localhost", 27017)
    db = mc.si11

    for ids in db_result:
        query = text(f"SELECT movietitle FROM imdb_movies WHERE movieid = {ids[0]}")
        title = db_connect.execute(query).fetchall()
        ntitle = clean_year_title(title[0][0])

        query = text(f"SELECT DISTINCT MG.genre FROM imdb_moviegenres AS MG WHERE MG.movieid = {ids[0]}")
        genres = db_connect.execute(query).fetchall()
        query = text(f"SELECT imdb_movies.year FROM imdb_movies WHERE movieid = {ids[0]}")
        year = db_connect.execute(query).fetchall()
        query = text(f"SELECT directorname FROM imdb_directors D JOIN imdb_directormovies DM ON D.directorid = DM.directorid WHERE DM.movieid = {ids[0]}")
        directors = db_connect.execute(query).fetchall()
        query = text(f"SELECT actorname FROM imdb_actors A JOIN imdb_actormovies AM ON A.actorid = AM.actorid WHERE AM.movieid = {ids[0]}")
        actors = db_connect.execute(query).fetchall()
        query = text(f"SELECT M2.movieid, M2.movietitle, M2.year FROM imdb_moviegenres MG1 JOIN imdb_movies M1 ON MG1.movieid = M1.movieid JOIN imdb_moviegenres
        most_related = db_connect.execute(query).fetchall()
        query = text(f"SELECT M2.movieid, M2.movietitle, M2.year FROM imdb_moviegenres MG1 JOIN imdb_movies M1 ON MG1.movieid = M1.movieid JOIN imdb_moviegenres
        related = db_connect.execute(query).fetchall()

        pelicula = {
            "title": str(ntitle),
            "genres": [str(genre[0]) for genre in genres],
            "year": int(year[0][0]),
            "directors": [str(director[0]) for director in directors],
            "actors": [str(actor[0]) for actor in actors],
            "most_related_movies": [{"title": clean_year_title(movie[1]), "year": int(movie[2])} for movie in most_related],
            "related_movies": [{"title": clean_year_title(movie[1]), "year": int(movie[2])} for movie in related]
        }

        result=db.france.insert_one(pelicula)

    count = db.france.count_documents({})
    print(f"Número de documentos en 'france': {count}")

    mc.close()

if __name__ == '__main__':
    main()
```

Para generar la nueva base de datos documental de películas francesas, primero nos conectamos a la base de datos de postgres mediante “sqlalchemy” y hacemos la consulta para obtener el id de todas estas películas, para las que crearemos un documento para cada una. A su vez, creamos con “MongoClient” la base de datos documental que llamamos si1. Después, hacemos un bucle para ir sacando la información necesaria de cada película (borrando las fechas de los títulos llamando a la función “clean_llear_title”) y la insertamos de la forma en la que nos piden en la práctica mediante la llamada a “insert_one” de “MongoClient” en la cual especificamos la colección(france).

Comprobaciones:

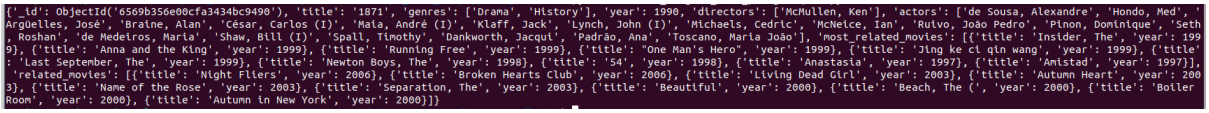
Primero, comprobamos el número de documentos que se habían creado en la colección y vimos que coincidía con el número de películas francesas en la BBDD inicial.

```
count = db.france.count_documents({})
print(f"Número de documentos en 'france': {count}")
```

```
enrique@enrique-VirtualBox:~/ING_INFO/3-TERCERO/sistemas1/si1p3_1321_P10/app-mongodb-et1$ python3 create_mongodb_from_postgresqldb.py
Número de documentos en 'france': 313
```

También comprobamos que como se habían insertado los datos en el primer documento y vimos que se habían insertado correctamente:

```
first_document = db.france.find_one()
print(first_document)
```



c. Las consultas las realizamos de la siguiente forma:

```
from pymongo import MongoClient

def query_sci_fi_movies_between_years(db):
        genre="Sci-Fi"
        result = db.france.find({
            "genres": {"$in": [genre]},
            "year": {"$gte": 1994, "$lte": 1998}
        })
        return result

def query_dramas_starting_with_the_in_1998(db):
    genre="Drama"
    result = db.france.find({
        "genres": {"$in": [genre]},
        "year": 1998,
        "title": {"$regex": "The$"}
    })
    return result

def query_movies_with_faye_dunaway_and_viggo_mortensen(db):
    result = db.france.find({
        "actors": "Dunaway, Faye"
    })
    movies_with_dunaway = [movie["title"] for movie in result]
    result = db.france.find({
        "title": {"$in": movies_with_dunaway},
        "actors": "Mortensen, Viggo"
    })
    return result
```

Realizamos 3 funciones para las consultas:

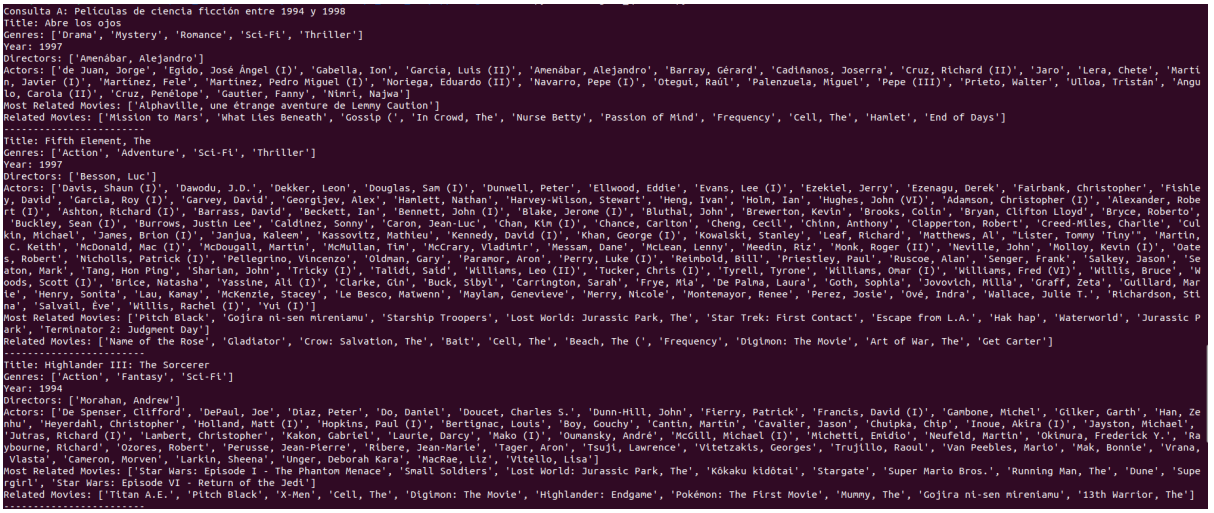
1. En la primera utilizamos un find en la cual buscamos que el “Sci-Fi” se encuentre en el apartado de genres de las películas mediante “$in”, y la otra condición, la obtenemos gracias a el año que sea mayor o igual que 1994 (“$gte”) y menor o igual que 1998 (“$lte”)
2. En la segunda, utilizamos un find, cuyas condiciones son:
   - Drama debe ser un Género de la película(“$in”)

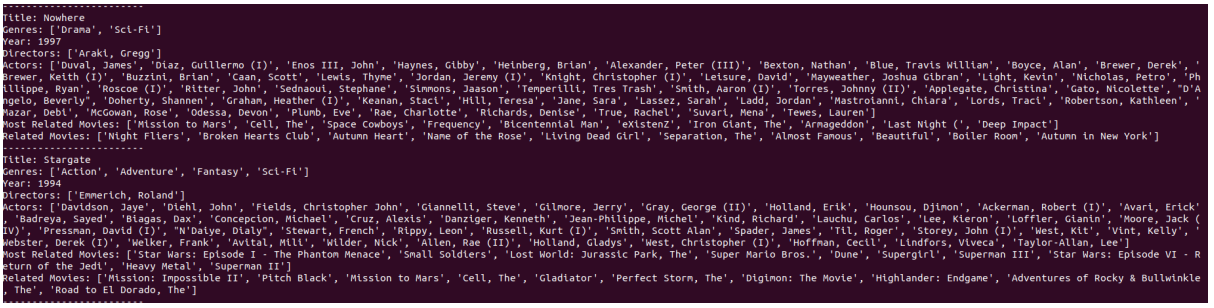
-El año debe ser 1998
-Y que el título, debe acabar por “The”, ya que las películas que empiezan por “The”, lo contienen al final del título. (“$regex”) para buscar títulos que coinciden con la cadena “The” añadiendo el $ al final para que coja los que están al final del título.

3. En la última, deben coincidir los 2 actores en una misma película, por tanto, primero buscamos las películas en las cuales aparezca el primer actor, y después buscamos entre las películas en las que sale el primero, el nombre del segundo actor.

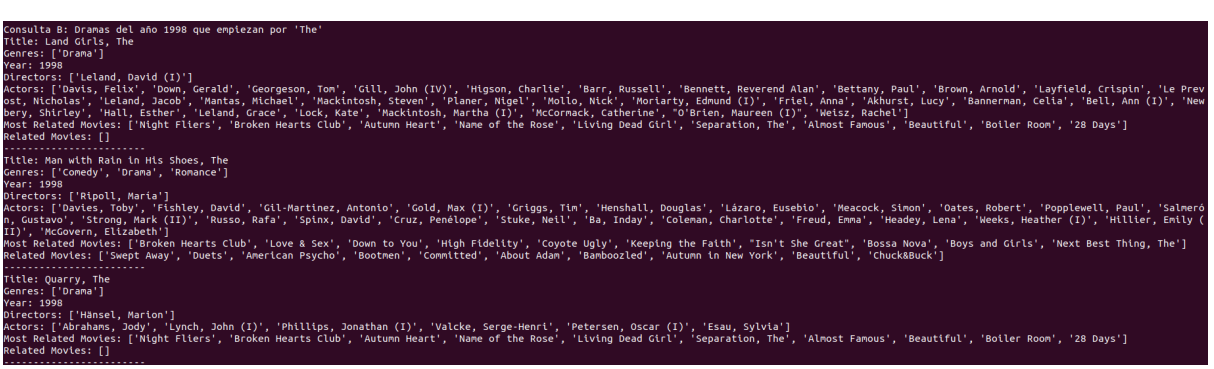
Resultados de las consultas:

1.





2.

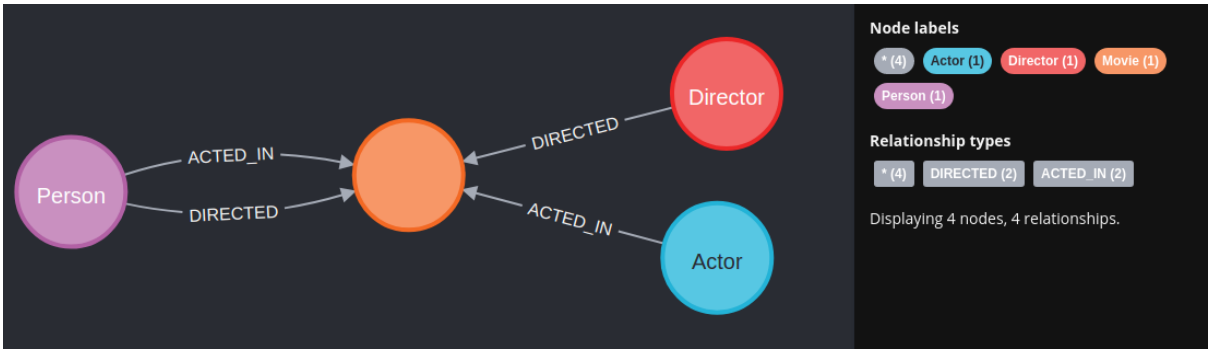


3.

```
Consulta C: Peliculas con Faye Dunaway y Viggo Mortensen en el reparto
Title: Albino Alligator
Genres: ['Crime', 'Drama', 'Thriller']
Year: 1996
Directors: ['Spacey, Kevin (I)']
Actors: ['Dillon, Matt (I)', 'Falson, 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)', 'Mortensen, Viggo', 'Sinise, Gary', 'Smith, Alexander (I)', 'Spencer, John (I)', 'Ulrich, Skeet', 'Spinuzza, Doug', 'Unger, Michael (I)', 'Worthen, Jock',
 'Walsh, M. Emmet', 'Dunaway, Faye', 'McGraw, Melinda', 'Figueiredo, Aura', 'Montgomery, Toni']
Most Related Movies: ['Name of the Rose', 'Reindeer Games', 'Cell, The', 'Hamlet', 'Nurse Betty', 'Boiler Room', 'Get Carter', 'Crow: Salvation, The', 'Circus', 'Romeo Must Die']
Related Movies: ['In Crowd, The', 'Mission to Mars', 'Crime and Punishment in Suburbia', 'Gossip (', 'Bait', 'Beach, The (', 'Contender, The', 'American Psycho', 'Frequency', 'Opportunists, The']
-------------------------
```

1.2) BBDD basadas en Grafos

```
CALL apoc.meta.graph()
```



A la base de datos si1 creada en Neo4j se deben realizar las siguientes consultas.

a.

```
MATCH (a:Actor)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(commonActor)
WHERE a.name <> 'Winston, Hattie'
WITH a, commonActor
MATCH (commonActor)-[:ACTED_IN]->(movieWithThirdActor)<-[:ACTED_IN]-(thirdActor)
RETURN DISTINCT a.actorid, a.name
ORDER BY a.name
LIMIT 10;
```

Esta query, primero busca las personas que han trabajado en alguna película con Winston, para luego descartar dichos actores, por último selecciona una tercera persona que si haya trabajado con las personas con las que ha colaborado Winston. Finalmente retorna dichas personas, ordenando sus nombre alfabéticamente y se aplica un límite de 10 personas como máximo en la salida

a.actorid	a.name
7133	"Adam, Joel"
11553	"Adams, Catlin"
13873	"Adams, Lillian"
17919	"Adams, Melanie (II)"
8187	"Addington, Constance"
26690	"Addota, Kip"
46924	"Ahern, Alston"
16193	"Albright, Gerald"
36411	"Alcañiz, Luana"
20621	"Alderman, Jane"

b.

```
MATCH (p1:Person)-[:ACTED_IN|DIRECTED]-(m:Movie)-[:ACTED_IN|DIRECTED]-(p2:Person)
WHERE p1 < p2
WITH p1, p2, COLLECT(DISTINCT m) AS movies
WHERE SIZE(movies) > 1
RETURN p1.name AS person1, p2.name AS person2, movies;
```

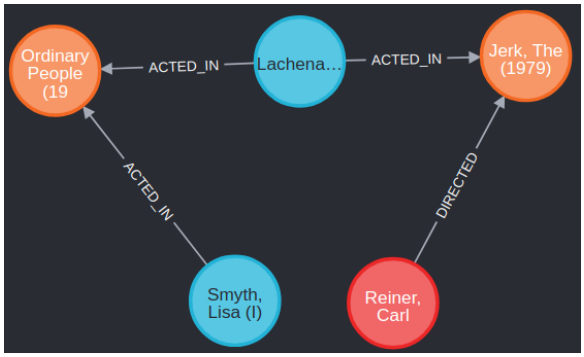
Primero se buscan las personas que hayan trabajado en una misma película, da igual si fue como director o como actor, luego se asegura que no se repitan las parejas de personas, y se establece que han debido de trabajar en más de una película juntos. Finalmente se retorna el nombre de la primera persona, el de la segunda y por último la película

person1	person2	movies
"Lachenauer, Gustave"	"Velasco, Jerry G."	[(:Movie {movieid: 54396,title: "Bound for Glory (1976)"}), (:Movie {movieid: 201944,title: "Jerk, The (1979)"})]

c.

```
MATCH p = shortestPath(
  (director:Director {name: 'Reiner, Carl'})-[*]-(actress:Actor {name: 'Smyth, Lisa (I)'})
)
RETURN p;
```

Se utiliza la función shortest path para encontrar el recorrido más corto entre ambos, y se especifica el principio del camino, que es un director llamado “Reiner, Carl”, y el final del camino, la actriz “Smyth, Lisa (I)”.



2)Uso de tecnología caché de acceso rápido

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey, text
import random
import redis

db_engine = create_engine("postgresql://alumnodb:1234@localhost/si1", echo=False)
db_connect = db_engine.connect()
redis_conn = redis.StrictRedis(host='localhost', port=6379, decode_responses=True)


def main():
    db_connect = db_engine.connect()

    query = text("select email, firstname, lastname, phone from customers where country = 'Spain'")
    db_result = db_connect.execute(query).fetchall()


    for customer in db_result:
        visits= random.randint(1, 99)
        email= customer[0]
        name = f"{customer[1]} {customer[2]}"
        phone = customer[3]

        redis_key = f"customers:{email}"

        redis_conn.hset(redis_key, 'name', name)
        redis_conn.hset(redis_key, 'phone', phone)
        redis_conn.hset(redis_key, 'visits', visits)
```

Para la realización de la nueva base de datos, primero nos conectamos a la original mediante “sqlalchemy” y a la nueva con “StrictRedis” de redis en el puerto 6379 por defecto e hicimos la consulta obteniendo el teléfono, email y el nombre completo de los clientes españoles. Tras esto, para cada cliente, asignamos un número aleatorio entre el 1 y el 99 llamando a la función randint de random. Tras esto asignamos el email como clave del cliente en la nueva BBDD y finalmente para esa clave añadimos con “hset” cada par de clave-valor del cliente.

Para comprobar que los datos se habían insertado correctamente, hicimos una función que imprimía los 5 primeros clientes:

```python
#SOLO PARA PROBAR QUE SE INSERTAN BIEN
def check_inserted_data(limit=5):
    customer_keys = redis_conn.keys("customers:*")

    if not customer_keys:
        print("No hay clientes en Redis.")
        return

    customer_keys = customer_keys[:limit]

    for customer_key in customer_keys:
        customer_info = redis_conn.hgetall(customer_key)

        print(f"{customer_key}")
        for field, value in customer_info.items():
            print(f"{field}: {value}")
        print("-" * 30)
```

```
enrique@enrique-VirtualBox:~/ING_INFO/3-TERCERO/sistemas1/si1p3_1321_P10/a
customers:dipper.carpi@potmail.com
name: dipper carpi
phone: +25 264766107
visits: 5
------------------------------
customers:ibero.mamma@potmail.com
name: ibero mamma
phone: +12 283698855
visits: 11
------------------------------
customers:shroud.mirror@mamoot.com
name: shroud mirror
phone: +39 815826855
visits: 37
------------------------------
customers:remind.nudism@potmail.com
name: remind nudism
phone: +35 106296474
visits: 66
------------------------------
customers:aphid.vito@kran.com
name: aphid vito
phone: +46 437981969
visits: 7
------------------------------
```

c)

a. increment_by_email(email).

```python
def increment_by_email(email):
    redis_key = f"customers:{email}"

    if redis_conn.exists(redis_key):

        redis_conn.hincrby(redis_key, 'visits', 1)
        print(f"Visita incrementada para {email}")
    else:
        print(f"No se encontró el correo electrónico {email} en Redis")
```

Primero se obtiene la clave redis y si existe, se incrementa para esa clave el campo visits a 1, finalmente se imprime a que email se ha incrementado su campo visits o si no se encontró el correo.

Ejemplo de ejecución.
Antes de la ejecución:

```
Información para el correo electrónico binder.yarrow@mamoot.com:
Nombre: binder yarrow
Teléfono: +53 495846539
Número de visitas: 65
```

Después de la ejecución:

```
Visita incrementada para binder.yarrow@mamoot.com
Información para el correo electrónico binder.yarrow@mamoot.com:
Nombre: binder yarrow
Teléfono: +53 495846539
Número de visitas: 66
```

b. customer_most_visits().

```python
def customer_most_visits():
    customer_keys = redis_conn.keys("customers:*")

    if not customer_keys or len(customer_keys) == 0:
        print("No hay clientes en Redis.")
        return None

    max_visits = -1
    max_visits_email = None

    for customer_key in customer_keys:
        customer_info = redis_conn.hgetall(customer_key)

        if 'visits' in customer_info:
            visits = int(customer_info['visits'])

            if visits > max_visits:
                max_visits = visits
                max_visits_email = customer_key.split(":")[1]

    return max_visits_email
```

Primero se obtiene la key y se verifica que existe en nuestro cliente redis, después se recorren todos los costumers verificando su campo visits y uno por uno verificamos si es o no mayor que nuestro max_visits_email, que será el correo con el valor visits más alto en cada iteración.

El resultado de aplicar dicha función es:

```
most_visits_email = customer_most_visits()

if most_visits_email:
    print(f"El cliente con más visitas tiene el correo electrónico: {most_visits_email}")
else:
    print("No se encontraron clientes en Redis.")
```

```
El cliente con más visitas tiene el correo electrónico: andrew.kook@jmail.com
Información para el correo electrónico andrew.kook@jmail.com:
Nombre: andrew kook
Teléfono: +28 808818624
Número de visitas: 99
```

c. get_field_by_email(email).

```
def get_field_by_email(email):
    redis_key = f"customers:{email}"

    if not redis_conn.exists(redis_key):
        print(f"No se encontró el correo electrónico {email} en Redis.")
        return None

    customer_info = redis_conn.hgetall(redis_key)

    if not customer_info:
        print(f"No se encontró información para el correo electrónico {email}.")
        return None

    name = customer_info.get('name', 'Nombre no disponible')
    phone = customer_info.get('phone', 'Teléfono no disponible')
    visits = customer_info.get('visits', 'Número de visitas no disponible')

    return {
        'name': name,
        'phone': phone,
        'visits': visits,
    }
```

Esta función, como las anteriores, primero verifica que exista la clave asociada al email correspondiente, se obtiene la información de dicha clave y por último se imprime por pantalla.

Como habrás podido observar, ya se ha comprobado la correcta ejecución de esta función en las anteriores ejecuciones.

```
email_to_query = "binder.yarrow@mamoot.com"

customer_data = get_field_by_email(email_to_query)

if customer_data:
    print(f"Información para el correo electrónico {email_to_query}:")
    print(f"Nombre: {customer_data['name']}")
    print(f"Teléfono: {customer_data['phone']}")
    print(f"Número de visitas: {customer_data['visits']}")
else:
    print("No se pudo obtener información.")
```

```
Información para el correo electrónico binder.yarrow@mamoot.com:
Nombre: binder yarrow
Teléfono: +53 495846539
Número de visitas: 66
```

3) Transacciones

a)

```
select distinct c.customerid
FROM orderdetail od JOIN orders o ON od.orderid = o.orderid JOIN customers c ON o.customerid = c.customerid
WHERE c.state = 'minuit'
```

	customerid integer
1	3865

Ejemplo de Transacción con Flask SQLAlchemy

Estado: minuit

- ○ Transacción vía sentencias SQL
- ◉ Transacción vía funciones SQLAlchemy

☐ Ejecutar commit intermedio

☑ Provocar error de integridad

Duerme 0 segundos (para forzar deadlock).

Enviar

Trazas

1. No se utilizan sentencias SQL
2. Se provoca un fallo de restricción de foreign key.
3. Customer borrado correctamente de la tabla orderdetail
4. Error en la transacción, ejecutamos rollback.


```
select distinct c.customerid
FROM orderdetail od JOIN orders o ON od.orderid = o.orderid JOIN customers c ON o.customerid = c.customerid
WHERE c.state = 'minuit'
```

	customerid integer
1	3865

En este caso, al no hacer commit intermedio, el estado final es el mismo que el estado inicial en la base de datos, es decir, no se ha realizado ningún cambio en ninguna tabla al finalizar la transacción, ya que hemos provocado un fallo, el cual se detecta y se vuelve al estado inicial. Como podemos comprobar, la tabla “orderdetail” de la cual hemos borrado el un momento ese customer, al final la transacción lo sigue teniendo.

PRUEBA 2:

Ejemplo de Transacción con Flask SQLAlchemy

Estado: herman

- ○ Transacción vía sentencias SQL
- ◉ Transacción vía funciones SQLAlchemy

☑ Ejecutar commit intermedio

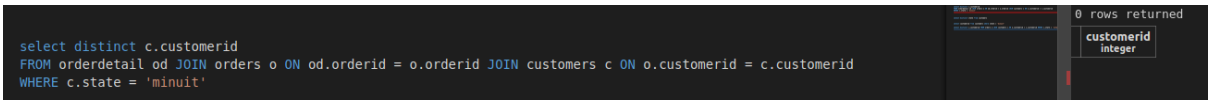
☑ Provocar error de integridad

Duerme 0 segundos (para forzar deadlock).

Enviar

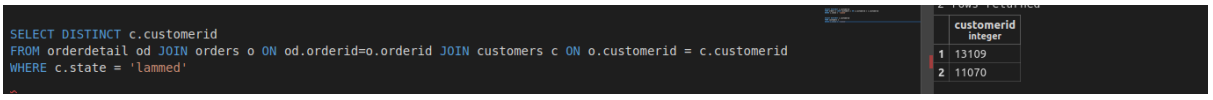
Trazas

1. No se utilizan sentencias SQL
2. Se provoca un fallo de restricción de foreign key.
3. Customer borrado correctamente de la tabla orderdetail
4. Commit intermedio antes de la siguiente operación.
5. Error en la transacción, ejecutamos rollback.



Como podemos observar, se ha ejecutado un rollback, ya que hemos provocado un error de integridad. Sin embargo, al haber hecho el commit después de borrar todas las compras de ese “customer” en la tabla “orderdetail”, esa sentencia ha sido ya ejecutada y el rollback deshace los cambios hasta el rollback. Como podemos observar, el customer en la tabla customers no ha sido eliminado gracias al rollback.

PRUEBA 3:
En este caso la prueba la haremos sin ningún fallo y utilizando las sentencias sql (execute()).
BBDD antes de realizar la transacción:



Ejemplo de Transacción con Flask SQLAlchemy

Estado: lammed

- (•) Transacción vía sentencias SQL
- ( ) Transacción vía funciones SQLAlchemy

- [ ] Ejecutar commit intermedio

- [ ] Provocar error de integridad

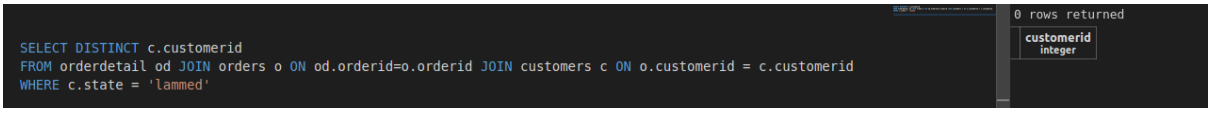
Duerme 0 segundos (para forzar deadlock).

Enviar

Trazas

1. Se utilizan sentencias SQL
2. Customer borrado correctamente de la tabla orderdetail
3. Customer borrado correctamente de la tabla orders
4. Customer borrado correctamente de la tabla customers
5. La transacción es correcta.

BBDD después de realizar la transacción:



Explicación de la transacción:

```python
def delState(state, bFallo, bSQL, duerme, bCommit):
    dbr = []

    # TODO: Ejecutar consultas de borrado
    # - ordenar consultas según se desee provocar un error (bFallo True) o no
    # - 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()
    db_conn = dbConnect()
    try:
        # TODO: ejecutar consultas
        if bSQL == True:
            dbr.append("Se utilizan sentencias SQL")
            query = text(f"BEGIN;")
            db_conn.execute(query)

            if bFallo == False:
                query = text(f"DELETE FROM orderdetail od USING orders o, customers c WHERE od.orderid = o.orderid AND o.customerid = c.customerid AND c.state = '{state}'")
                db_conn.execute(query)
                dbr.append("Customer borrado correctamente de la tabla orderdetail")

                query = text(f"DELETE FROM orders USING customers c WHERE orders.customerid = c.customerid AND c.state = '{state}'")
                db_conn.execute(query)
                dbr.append("Customer borrado correctamente de la tabla orders")

                if bCommit:
                    dbr.append("Commit final antes de finalizar la transacción.")
                    query = text(f"COMMIT;")
                    db_conn.execute(query)
                    # Insert sleep before final commit
                    time.sleep(duerme)
                    query = text(f"BEGIN;")
                    db_conn.execute(query)

                query = text(f"DELETE FROM customers WHERE state = '{state}'")
                db_conn.execute(query)
                dbr.append("Customer borrado correctamente de la tabla customers")

            if bFallo == True:
                dbr.append("Se provoca un fallo de restricción de foreign key.")

                query = text(f"DELETE FROM orderdetail od USING orders o, customers c WHERE od.orderid = o.orderid AND o.customerid = c.customerid AND c.state = '{state}'")
                db_conn.execute(query)
                dbr.append("Customer borrado correctamente de la tabla orderdetail")

                if bCommit:
                    dbr.append("Commit final antes de finalizar la transacción.")
                    query = text(f"COMMIT;")
                    db_conn.execute(query)
                    # Insert sleep before final commit
                    time.sleep(duerme)
                    query = text(f"BEGIN;")
                    db_conn.execute(query)

                query = text(f"DELETE FROM customers WHERE state = '{state}'")
                db_conn.execute(query)
                dbr.append("Customer borrado correctamente de la tabla customers")

                query = text(f"DELETE FROM orders USING customers c WHERE orders.customerid = c.customerid AND c.state = '{state}'")
                db_conn.execute(query)
                dbr.append("Customer borrado correctamente de la tabla orders")
```

```
    else:
        dbr.append("No se utilizan sentencias SQL")
        transaction = db_conn.begin()

        if bFallo == False:
            query = text(f"DELETE FROM orderdetail od USING orders o, customers c WHERE od.orderid = o.orderid AND o.customerid = c.customerid AND c.state = '{state}'")
            db_conn.execute(query)
            dbr.append("Customer borrado correctamente de la tabla orderdetail")

            query = text(f"DELETE FROM orders USING customers c WHERE orders.customerid = c.customerid AND c.state = '{state}'")
            db_conn.execute(query)
            dbr.append("Customer borrado correctamente de la tabla orders")

            if bCommit == True:
                dbr.append("Commit final antes de finalizar la transacción.")
                transaction.commit()
                # Insert sleep before final commit
                time.sleep(duerme)
                transaction = db_conn.begin()

            query = text(f"DELETE FROM customers c WHERE c.state = '{state}'")
            db_conn.execute(query)
            dbr.append("Customer borrado correctamente de la tabla customers")

        if bFallo == True:
            dbr.append("Se provoca un fallo de restricción de foreign key.")

            query = text(f"DELETE FROM orderdetail od USING orders o, customers c WHERE od.orderid = o.orderid AND o.customerid = c.customerid AND c.state = '{state}'")
            db_conn.execute(query)
            dbr.append("Customer borrado correctamente de la tabla orderdetail")

            if bCommit == True:
                dbr.append("Commit final antes de finalizar la transacción.")
                transaction.commit()
                # Insert sleep before final commit
                time.sleep(duerme)
                transaction = db_conn.begin()

            query = text(f"DELETE FROM customers c WHERE c.state = '{state}'")
            db_conn.execute(query)
            dbr.append("Customer borrado correctamente de la tabla customers")

            query = text(f"DELETE FROM orders USING customers c WHERE orders.customerid = c.customerid AND c.state = '{state}'")
            db_conn.execute(query)
            dbr.append("Customer borrado correctamente de la tabla orders")

except Exception as e:
    # TODO: deshacer en caso de error
    dbr.append("Error en la transacción, ejecutamos rollback.")
    if bSQL == True:
        query = text(f"ROLLBACK;")
        db_conn.execute(query)
    else:
        transaction.rollback()
    dbCloseConnect(db_conn)

else:
    # TODO: confirmar cambios si todo va bien
    if bSQL == True:
        query = text(f"COMMIT;")
        db_conn.execute(query)
    else:
        transaction.commit()
    dbr.append("La transacción es correcta.")
    dbCloseConnect(db_conn)

return dbr
```

En primer lugar nos conectamos a la base de datos. La función se basa en un try, except, else:

-Try: Probamos a hacer la transacción con todo lo que nos piden. Primero comprobamos si todas las sentencias de la transacción se deben hacer vía sql o utilizando funciones de SQLAlchemy. Esto hace que hagamos un if, else que hará lo mismo solo que cambiando esto. Tras esto, comenzamos con la transacción ejecutando el begin, y después comprobaremos si se pide realizar un fallo de restricción de foreign key. La diferencia de esto será que si queremos realizar un fallo, debemos eliminar primero ese customer de una tabla que luego necesitemos para eliminar en otra tabla. Por ejemplo, nosotros hemos optado por eliminar primero todos los customers de ese estado de la tabla orderdetail, después, hemos eliminado el customer de la tabla customer, sin embargo, cuando intentamos acceder a la tabla orders para eliminarlo, no podemos relacionarlo con la tabla customers ya que dicho customer ya no existe por lo que habrá un fallo y se ejecutará el rollback. Sin embargo, si no queremos provocar el fallo, el orden correcto que se seguirá al eliminar será: eliminar de la tabla orderdetail, eliminar de la tabla orders,

eliminar de customer. El commit intermedio se dará la opción de hacerlo en cualquier caso, falle o no, y este lo realizamos después de eliminar de la primera tabla. Este commit lo que hará será guardar los cambios realizados hasta el momento por lo que si en algún momento falla la transacción, esto que borramos antes del commit intermedio seguirá borrado, como podemos ver en la prueba 2.
-Except: Aquí entrará el programa si alguna parte de la transacción da error por lo que en ese caso ejecutaremos el rollback o bien por sentencias SQL (execute("ROLLBACK")) o bien por funciones de la biblioteca SQLAlchemy (transaction.rollback()). Finalmente cerramos la conexión con la base de datos.
-Else: Este se ejecutará si todo en la transacción ha salido bien por lo que podremos hacer el commit final para guardar los cambios realizados. Esto también se puede hacer vía sentencias SQL o funciones de SQLAlchemy. Finalmente cerramos la conexión con la base de datos.

B)

Creamos la columna Promo para la tabla customers

```
ALTER TABLE customers
ADD COLUMN promo INTEGER DEFAULT 0;
```

Primero creamos un trigger 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.

```
CREATE OR REPLACE FUNCTION update_orderdetail_price() RETURNS TRIGGER AS $$
BEGIN
    UPDATE orderdetail
    SET
        price = (products.price - (products.price * (NEW.promo/100))) * quantity
    FROM
        orders,
        products
    WHERE
        NEW.customerid = orders.customerid and
        orders.status = 'null' and
        orders.orderid = orderdetail.orderid and
        orderdetail.prod_id = products.prod_id;

    UPDATE orders
    SET
        netamount = totalSum.sum
    FROM
        (
        SELECT orders.orderid, SUM(orderdetail.price) AS sum
        FROM orderdetail, orders
        WHERE
        orderdetail.orderid = orders.orderid and
            NEW.customerid = orders.customerid
        GROUP BY orders.orderid
        ) AS totalSum
     WHERE totalSum.orderid = orders.orderid;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_orderdetail_price
AFTER UPDATE OF promo ON customers
FOR EACH ROW
    EXECUTE PROCEDURE update_orderdetail_price();
```

Después, hemos modificado el trigger, como se ha indicado, realizando un sleep antes de actualizar la columna orders

```
CREATE OR REPLACE FUNCTION update_orderdetail_price() RETURNS TRIGGER AS $$
BEGIN
    UPDATE orderdetail
    SET
        price = (products.price - (products.price * (NEW.promo/100))) * quantity
    FROM
        orders,
        products
    WHERE
        NEW.customerid = orders.customerid and
        orders.status = 'null' and
        orders.orderid = orderdetail.orderid and
        orderdetail.prod_id = products.prod_id;

    PERFORM pg_sleep(5);

    UPDATE orders
    SET
        netamount = totalSum.sum
    FROM
        (
        SELECT orders.orderid, SUM(orderdetail.price) AS sum
        FROM orderdetail, orders
        WHERE
        orderdetail.orderid = orders.orderid and
            NEW.customerid = orders.customerid
        GROUP BY orders.orderid
        ) AS totalSum
     WHERE totalSum.orderid = orders.orderid;

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

Se ha insertado un sleep en el final de cada commit antes de finalizar la transacción, en la función delState

```
if bCommit:
    dbr.append("Commit final antes de finalizar la transacción.")
    time.sleep(duerme)  # Insert sleep before final commit
    query = text(f"COMMIT;")
    db_conn.execute(query)
    query = text(f"BEGIN;")
    db_conn.execute(query)
```

Ahora procederemos con la comprobación
Primero obtenemos los tres primeros pedidos por ejemplos, y observamos el valor de sus columnas “status”

```
SELECT
    od.orderid, od.status
FROM
    orders od
ORDER BY
    od.orderid
LIMIT 3;
```

	orderid integer	status character varying
1	1	Shipped
2	2	Shipped
3	3	Paid

Ahora procedemos a poner a null los valores de las columnas status de los anteriores carritos mediante la query, volviendo buscar esos carritos comprobamos que el valor para su columna status es null

```
UPDATE orders
SET status = NULL
WHERE orderid IN (
    SELECT od.orderid
    FROM orders od
    ORDER BY od.orderid
    LIMIT 3
);
```

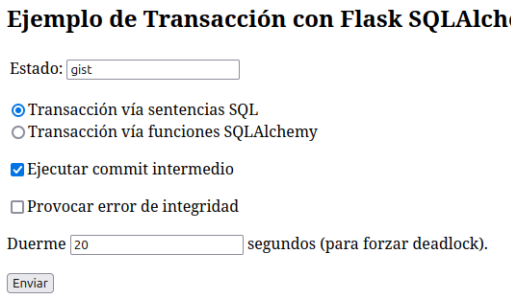
	orderid integer	status character varying
1	1	*null*
2	2	*null*
3	3	*null*

Tomaremos para este apartado el siguiente cliente

	customerid integer	promo integer	state character varying
1	1121	30	gist

Actualizamos su valor de promo para este customer para hacer saltar el trigger

```
UPDATE customers
SET promo = promo + 10
WHERE customerid = '1121';
```



Al ejecutar el siguiente comando sobre la base de datos se genera la siguiente salida:

```
si1_p3=# SELECT * FROM pg_locks;
```

pg_locks muestra información sobre los bloqueos actuales en la base de datos y como podemos comprobar se están generando desde la misma base de datos.

locktype	database	relation	page	tuple	virtualxid	transactionid	classid	objid	objsubid	virtualtransaction	pid	mode	granted	fastpath	waitstart
relation	41434	41542								10/1510	14127	AccessShareLock	t	t	
relation	41434	41950								10/1510	14127	AccessShareLock	t	t	
relation	41434	41950								10/1510	14127	RowExclusiveLock	t	t	
relation	41434	41949								10/1510	14127	AccessShareLock	t	t	
relation	41434	41949								10/1510	14127	RowExclusiveLock	t	t	
relation	41434	41540								10/1510	14127	AccessShareLock	t	t	
relation	41434	41540								10/1510	14127	RowExclusiveLock	t	t	
relation	41434	41952								10/1510	14127	AccessShareLock	t	t	
relation	41434	41952								10/1510	14127	RowExclusiveLock	t	t	
relation	41434	41951								10/1510	14127	AccessShareLock	t	t	
relation	41434	41951								10/1510	14127	RowExclusiveLock	t	t	
relation	41434	41503								10/1510	14127	AccessShareLock	t	t	
relation	41434	41497								10/1510	14127	AccessShareLock	t	t	
relation	41434	41497								10/1510	14127	RowExclusiveLock	t	t	
relation	41434	41492								10/1510	14127	AccessShareLock	t	t	
relation	41434	41492								10/1510	14127	RowExclusiveLock	t	t	
relation	41434	41948								10/1510	14127	RowExclusiveLock	t	t	
relation	41434	41947								10/1510	14127	RowExclusiveLock	t	t	
relation	41434	41520								10/1510	14127	RowExclusiveLock	t	t	
relation	41434	41435								10/1510	14127	RowExclusiveLock	t	t	
virtualxid					10/1510					10/1510	14127	ExclusiveLock	t	t	
relation	41434	41948								8/1591	14002	AccessShareLock	t	t	
relation	41434	41947								8/1591	14002	AccessShareLock	t	t	
relation	41434	41520								8/1591	14002	AccessShareLock	t	t	
relation	41434	41950								8/1591	14002	AccessShareLock	t	t	
relation	41434	41950								8/1591	14002	RowShareLock	t	t	
relation	41434	41950								8/1591	14002	RowExclusiveLock	t	t	
relation	41434	41949								8/1591	14002	AccessShareLock	t	t	
relation	41434	41949								8/1591	14002	RowExclusiveLock	t	t	
relation	41434	41540								8/1591	14002	AccessShareLock	t	t	
relation	41434	41540								8/1591	14002	RowExclusiveLock	t	t	
relation	41434	41952								8/1591	14002	RowExclusiveLock	t	t	
relation	41434	41951								8/1591	14002	RowShareLock	t	t	
relation	41434	41951								8/1591	14002	RowExclusiveLock	t	t	
relation	41434	41435								8/1591	14002	AccessShareLock	t	t	
relation	41434	41497								8/1591	14002	AccessShareLock	t	t	
relation	41434	41497								8/1591	14002	RowShareLock	t	t	
relation	41434	41497								8/1591	14002	RowExclusiveLock	t	t	
relation	41434	41492								8/1591	14002	RowShareLock	t	t	
relation	41434	41492								8/1591	14002	RowExclusiveLock	t	t	

Para comprobar que esta salida es generada por nuestra base de datos, ejecutamos la query cuando haya terminado la transacción.

locktype	database	relation	page	tuple	virtualxid	transactionid	classid	objid	objsubid	virtualtransaction	pid	mode	granted	fastpath	waitstart
relation	41434	12073								9/358	11112	AccessShareLock	t	t	
virtualxid					9/358					9/358	11112	ExclusiveLock	t	t	

(2 rows)

Como se puede observar ya no hay locks generados, y el único que aparece hace referencia al lock que se genera al ejecutar esta misma query.

Al ejecutar durante la transacción la query que muestra el valor

```
si1_p3=# SELECT
        c.customerid, c.promo, c.state
FROM
        customers c
WHERE
        c.customerid = '1121';
 customerid | promo | state
------------+-------+-------
       1121 |    30 | gist
(1 row)
```

Por lo que se comprueba que no se generó la transacción.

Un deadlock se genera si dos o más transacciones están a la espera de que otras transacciones liberen recursos, por lo que aplicando un sleep antes del último commit se generaría dicho deadlock