SISTEMAS INFORMÁTICOS

~PRÁCTICA 3~

Memoria

Elena Balseiro García y Paula Beltrán Marianini

Grupo 1392 Pareja 08

NoSQL

A. Creación de la Base de Datos de Películas.
B. Creación de script *createMongoDBFromPostgreSQLDB.py*

Para crear la base de datos documental a partir de la Base de Datos de Películas para PostgreSQL, hemos definido la siguiente estructura para almacenar los datos de cada película en MongoDB:

```
{"title": title,
  "genres": genres,
  "year": year,
  "number_of_votes": number_of_votes,
  "average_rating": average_rating,
  "directors": directors,
  "actors": actors,
  "related_movies": relMovies})
```

Para automatizar el proceso de filtrado y creación, hemos creado el script *createMongoDBFromPostgreSQLDB.py* el cual se ejecutará manualmente antes de arrancar la aplicación web para poder cargar con los datos que se trabaja.

Desarrollo del script:

1. Conexión con la Base de Datos:

```
# configurar el motor de sqlalchemy
db_engine = sqlalchemy.create_engine("postgresql://alumnodb:alumnodb@localhost/si1", echo=False)

#conexión con la base de datos
try:
    db_conn = db_engine.connect()
    print("conectado")
except Exception:
    print("Error connecting database")
```

Extraemos los datos con *sqlalchemy* igual que en la práctica anterior. De esta forma, podremos obtener los datos de interés de la Base de Datos. Por ello, hay que realizar la conexión al principio del script y la desconexión al final de este, con la función *close()*.

2. Filtros:

Para obtener las 400 películas de UK (para ello hemos definido unas variables globales *país=UK* y *nPelis=400*), géneros, actores y directores, hemos realizado unas sentencias preparadas para cada una de ellas. De esta forma, están predefinidas y más adelante aceptan parámetros.

Filtro Top Películas:

```
#filtro top películas
q_topMovies = "SELECT MV.movieid, MV.movietitle, MV.year, MC.country, MV.ratingcount, MV.ratingmean " \
                "FROM imdb_movies AS MV, imdb_moviecountries AS MC " \
                "WHERE MC.country= '%%PAIS%%' AND MV.movieid=MC.movieid "  \
                "ORDER BY MV.year DESC " \
                "LIMIT '%%nPelis%%';"
```

Filtro géneros:

```
#filtro géneros películas
q_movieGenres = "SELECT * FROM imdb_moviegenres WHERE movieid = %%ID%%;"
```

Filtro actores:

```
#filtro actores películas
q_movieActors = "SELECT AM.movieid, AR.actorname as actorname " \
                    "FROM imdb_actormovies as AM NATURAL JOIN imdb_actors as AR " \
                    "WHERE AM.movieid = %%ID%%;"
```

Filtro directores:

```
#filtro directores películas
q_movieDirectors = "SELECT DM.movieid, DR.directorname as directorname " \
                       "FROM imdb_directormovies as DM NATURAL JOIN imdb_directors as DR " \
                       "WHERE DM.movieid = %%ID%%;"
```

Después, realizamos un bucle que recorra cada película de la lista de películas que devuelve la consulta ejecutada de Top Películas.
Luego recurrimos a las anteriores sentencias sustituyendo el parámetro ID por el de la película dependiendo de la iteración del bucle en la que estemos.
Cuando obtenemos todos los datos que necesitamos, los introducimos en la estructura que mencionamos anteriormente.
Para terminar, realizamos la conexión con MongoDB con los datos del cliente que nos dan e insertamos nuestras películas en la colección *topUk*, como se pide.

C. Aplicación Python que integre MongoDB
Hemos editado el archivo *topUk.html* de forma que se muestren los resultados de las consultas que se realicen con nuestra base de datos, como será el caso del siguiente apartado.

D. Consultas en MongoDB
Para facilitar estas consultas hemos creado una función llamada *searchBy*() en *database.py* cuya finalidad es devolver una lista de películas que cumpla con los parámetros que recibe como argumentos. Estos parámetros pueden ser título, año, género(s) y/o actor(es).

```python
def searchBy(title=None, year=None, genre=None, actor=None):
    #creamos la estructura donde se guardarán las expresiones de búsqueda
    search ={"$and":[]} #se usa $and para que se cumplan todas las condiciones

    #si se ha introducido un título, se añade a la búsqueda
    if title:
        search["$and"].append({"title": {"$regex": str(title), "$options": "i"}})

    #si se ha introducido un año, se añade a la búsqueda
    if year:
        if isinstance(year, list):
            search["$and"].append({"year": {"$in": year}})
        else:
            search["$and"].append({"year": {"$eq": year}})

    #si se ha introducido un género, se añade a la búsqueda
    if genre:
        if isinstance(genre, list):
            for g in genre:
                search["$and"].append({"genres": {"$regex": str(g), "$options": "i"}})
        else:
            search["$and"].append({"genres": {"$regex": str(genre), "$options": "i"}})

    #si se ha introducido un actor, se añade a la búsqueda
    if actor:
        if isinstance(actor, list):
            for a in actor:
                search["$and"].append({"actors": {"$regex": str(a), "$options": "i"}})
        else:
            search["$and"].append({"actors": {"$regex": str(actor), "$options": "i"}})

    movies = db.get_collection('topUk')
    return list(movies.find(search))
```

a. Películas Comedia entre 1990 y 1992

```python
m1 = database.searchBy(genre="Comedy", year=list(range(1990, 1992+1)))
```

b. Películas Acción 1995, 1997, 1998 que empiecen por 'The'

```python
m2 = database.searchBy(title='', The", genre="Action", year=[1995,1997,1998])
```

c. Películas donde están tanto Darren McAree como Katie Lockett en el reparto

```python
m3 = database.searchBy(actor=["McAree, Darren", "Lockett, Katie"])
```

Las llamadas a searchBy se hacen en la ruta */topUK* en el archivo *routes.py* que devuelve *movies*:

```python
movies=[[m1],[m2],[m3]]
return render_template('topUK.html', movies=movies)
```

Para poder ser mostradas en el html.

Optimización

E. Estudio del impacto de un índice

Creamos la consulta *estadosDistintos.sql* que muestra el número de estados distintos con clientes que tienen pedidos en un año dado, por ejemplo 2017, y que además pertenecen a un país, por ejemplo, Perú.

```
explain SELECT count(distinct(c.state)) FROM customers c NATURAL JOIN orders ord
    WHERE '2017' = date_part('year', ord.orderdate)
    AND 'Peru' = c.country;
```

Resultado de la sentencia EXPLAIN:

QUERY PLAN
Aggregate (cost=5089.32..5089.33 rows=1 width=8)
-> Gather (cost=1529.04..5089.31 rows=5 width=118)
Workers Planned: 1
-> Hash Join (cost=529.04..4088.81 rows=3 width=118)
Hash Cond: (o.customerid = c.customerid)
-> Parallel Seq Scan on orders o (cost=0.00..3558.37 rows=535 width=4)
Filter: ('2017'::double precision = date_part('year'::text, (orderdate)::timestamp without time zone))
-> Hash (cost=528.16..528.16 rows=70 width=122)
-> Seq Scan on customers c (cost=0.00..528.16 rows=70 width=122)
Filter: ('Peru'::text = (country)::text)

Analizando el output al ejecutar la sentencia *explain* podremos apreciar de arriba abajo:

1. La función *aggregate*, que nos muestra los cálculos realizados sobre el conjunto de datos dándonos información acerca del coste y las filas y columnas obtenidas. También observamos el elevado coste de 5089.
2. A continuación, aparecen los pasos del plan de ejecución de la consulta:
    a. *Gather*: representa la recolección de los datos necesarios.
    b. *Hash Join*: representa la operación NATURAL JOIN de nuestra query.

c. *Sequential Scan*: es un método que escanea los datos de la tabla. El sistema comprueba cada fila y descarta aquellas que no coinciden con la condición de nuestra query.
d. *Hash*: en este caso no es necesario hacer JOIN de varias tablas porque la última condición de la query solo requiere datos de una tabla.
e. *Sequential Scan*: de nuevo escanea los datos de la tabla descartando las filas que no satisfagan la condición.

Creamos un índice que disminuya el coste anterior. Hemos elegido crear un índice para obtener el año del pedido. Así será mucho más rápido comprobar esta condición.

```
create index idx_orders on orders(date_part('year', orders.orderdate));

explain SELECT count(distinct(c.state)) FROM customers c NATURAL JOIN orders ord
    WHERE '2017' = date_part('year', ord.orderdate)
    AND 'Peru' = c.country;

drop index idx_orders;
```

Y ahora al ejecutar:

QUERY PLAN
Aggregate (cost=2032.50..2032.51 rows=1 width=8)
-> Hash Join (cost=548.50..2032.49 rows=5 width=118)
Hash Cond: (ord.customerid = c.customerid)
-> Bitmap Heap Scan on orders ord (cost=19.46..1501.07 rows=909 width=4)
Recheck Cond: ('2017'::double precision = date_part('year'::text, (orderdate)::timestamp without time zone))
-> Bitmap Index Scan on idx_orders (cost=0.00..19.24 rows=909 width=0)
Index Cond: (date_part('year'::text, (orderdate)::timestamp without time zone) = '2017'::double precision)
-> Hash (cost=528.16..528.16 rows=70 width=122)
-> Seq Scan on customers c (cost=0.00..528.16 rows=70 width=122)
Filter: ('Peru'::text = (country)::text)

En este nuevo output observamos:
1. El coste ha disminuido sustancialmente de 5089 a 2032.
2. En vez de realizar un *Sequential Scan* para obtener los datos del año, realiza un *Bitmap Heap Scan* lo que reduce bastante el coste.

3. Otro factor que destacar en por qué se ha reducido tanto el coste es que en este nuevo plan de ejecución no se realiza un *Gather*.

Ahora decidimos crear un índice para obtener el país del cliente en lugar del año del pedido.

```
create index idx_country on customers(country);

explain SELECT count(distinct(c.state)) FROM customers c NATURAL JOIN orders ord
    WHERE '2017' = date_part('year', ord.orderdate)
    AND 'Peru' = c.country;

drop index idx_country;
```

Y ahora al ejecutar:

QUERY PLAN
Aggregate (cost=4740.99..4741.00 rows=1 width=8)
-> Gather (cost=1180.71..4740.98 rows=5 width=118)
Workers Planned: 1
-> Hash Join (cost=180.71..3740.48 rows=3 width=118)
Hash Cond: (ord.customerid = c.customerid)
-> Parallel Seq Scan on orders ord (cost=0.00..3558.37 rows=535 width=4)
Filter: ('2017'::double precision = date_part('year'::text, (orderdate)::timestamp without time zone))
-> Hash (cost=179.83..179.83 rows=70 width=122)
-> Bitmap Heap Scan on customers c (cost=4.83..179.83 rows=70 width=122)
Recheck Cond: ('Peru'::text = (country)::text)
-> Bitmap Index Scan on idx_country (cost=0.00..4.81 rows=70 width=0)
Index Cond: ((country)::text = 'Peru'::text)

Observamos que el coste es menor que sin índice pero que no es una diferencia notable.

Si decidiéramos realizar los dos índices, primero el del año y luego el del país:

```
create index idx_orders on orders(date_part('year', orders.orderdate));
create index idx_country on customers(country);

explain SELECT count(distinct(c.state)) FROM customers c NATURAL JOIN orders ord
    WHERE '2017' = date_part('year', ord.orderdate)
    AND 'Peru' = c.country;

drop index idx_orders;
drop index idx_country;
```

Obtendríamos:

QUERY PLAN
Aggregate (cost=1684.17..1684.18 rows=1 width=8)
-> Hash Join (cost=200.17..1684.16 rows=5 width=118)
Hash Cond: (ord.customerid = c.customerid)
-> Bitmap Heap Scan on orders ord (cost=19.46..1501.07 rows=909 width=4)
Recheck Cond: ('2017'::double precision = date_part('year'::text, (orderdate)::timestamp without time zone))
-> Bitmap Index Scan on idx_orders (cost=0.00..19.24 rows=909 width=0)
Index Cond: (date_part('year'::text, (orderdate)::timestamp without time zone) = '2017'::double precision)
-> Hash (cost=179.83..179.83 rows=70 width=122)
-> Bitmap Heap Scan on customers c (cost=4.83..179.83 rows=70 width=122)
Recheck Cond: ('Peru'::text = (country)::text)
-> Bitmap Index Scan on idx_country (cost=0.00..4.81 rows=70 width=0)
Index Cond: ((country)::text = 'Peru'::text)

Un coste mucho más reducido. Este coste es el menor de todos puesto que reduce el tiempo de obtención del año de la función de datepart y también del país. Daría lo mismo realizarlo en el orden inverso.

En el archivo , hemos dejado solo la sentencia explain y las líneas de código que corresponden a los índices creados, las hemos dejado comentadas para poder observarlas y descomentarlas en función de qué índice queramos usar.

```
--create index idx_country on customers(country); --descomentar para usar idx_country
--create index idx_orders on orders(date_part('year', orders.orderdate)); --descomentar para usar idx_orders


explain SELECT count(distinct(c.state)) FROM customers c NATURAL JOIN orders ord
    WHERE '2017' = date_part('year', ord.orderdate)
    AND 'Peru' = c.country;

--drop index idx_orders; --descomentar si se usa idx_orders
--drop index idx_country; --descomentar si se usa idx_country
```

F. Estudio del impacto de cambiar la forma de realizar una consulta

Consulta 1	`select customerid`
	`from customers`
	`where customerid not in (`
	`  select customerid`
	`  from orders`
	`  where status='Paid'`
	`);`
		QUERY PLAN
		Index Only Scan using customers_pkey on customers (cost=3961.93..4372.56 rows=7046 width=4)
		Filter: (NOT (hashed SubPlan 1))
		SubPlan 1
		-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)
		Filter: ((status)::text = 'Paid'::text)

Consulta 2	`select customerid`
	`from (`
	`  select customerid`
	`  from customers`
	`  union all`
	`  select customerid`
	`  from orders`
	`  where status='Paid'`
	`) as A`
	`group by customerid`
	`having count(*) =1;`
		QUERY PLAN
		Finalize GroupAggregate (cost=4425.26..4476.43 rows=1 width=4)
		Group Key: customers.customerid
		Filter: (count(*) = 1)
		-> Gather Merge (cost=4425.26..4471.93 rows=400 width=12)
		Workers Planned: 2
		-> Sort (cost=3425.24..3425.74 rows=200 width=12)
		Sort Key: customers.customerid
		-> Partial HashAggregate (cost=3415.59..3417.59 rows=200 width=12)
		Group Key: customers.customerid
		-> Parallel Append (cost=0.00..3383.01 rows=6516 width=4)
		-> Parallel Index Only Scan using customers_pkey on customers (cost=0.29..317.65 rows=8290 width=4)
		-> Parallel Seq Scan on orders (cost=0.00..3023.69 rows=535 width=4)
		Filter: ((status)::text = 'Paid'::text)

Consulta 3	`select customerid`
	`from customers`
	`except`
	`  select customerid`
	`  from orders`
	`  where status='Paid';`
		QUERY PLAN
		HashSetOp Except (cost=0.29..4597.59 rows=14093 width=8)
		-> Append (cost=0.29..4560.09 rows=15002 width=8)
		-> Subquery Scan on "*SELECT* 1" (cost=0.29..516.61 rows=14093 width=8)
		-> Index Only Scan using customers_pkey on customers (cost=0.29..375.68 rows=14093 width=4)
		-> Subquery Scan on "*SELECT* 2" (cost=0.00..3968.47 rows=909 width=8)
		-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)
		Filter: ((status)::text = 'Paid'::text)

Consulta 1:

Se puede observar que en esta consulta hay una subconsulta. En esta última se seleccionan todos los *Ids* de los *Customers* que han pagado (columna *status* de la tabla *orders*) mientras que la de fuera lo que hace es seleccionar todos los Ids

de la tabla *Customers* que no aparecen en la anterior. Por tanto, obtendríamos los Ids de los Customers que nunca han pagado un pedido.

Consulta 2:
En esta consulta, cabe destacar el alto coste debido a la unión de ambas tablas donde los elementos duplicados no se eliminan. Después de agrupar las tablas se agrupan los resultados comprobando aquellos que solo aparezcan una vez.

Consulta 3:
Esta consulta es casi idéntica a la Consulta 1. La única diferencia, aunque es una diferencia importante, es que sustituye *not in* por *except*. Este cambio hace que esta consulta al usar except elimine las filas duplicadas del resultado.

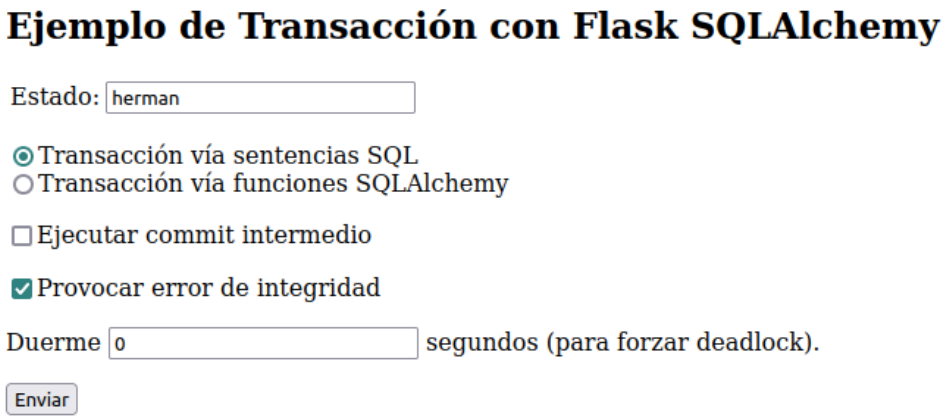
Cuestiones:

- **¿Qué consulta devuelve algún resultado nada más comenzar su ejecución?**
  La tercera consulta es la que empieza a devolver resultados nada más ejecutarse. Esto se debe al except, que comprueba si el resultado está duplicado o no determinando si lo tiene que añadir.
- **¿Qué consulta se puede beneficiar de la ejecución en paralelo?**
  Tanto la primera como la tercera (ya comentamos antes que eran muy parecidas) se podrían beneficiar de ejecutar paralelamente sus subqueries.

G. Estudio del impacto de la generación de estadísticas
:) porfa no nos bajes 0.5

Transacciones

H. Estudio de transacciones:
   a. Formulario de la página que recibe el state:



Además de solicitarnos el nombre del estado, podemos elegir si queremos que la transacción se ejecute con sentencias SQL o funciones SQLAlquemy.
También podremos marcar si queremos que haya un commit intermedio o que se provoque un error de integridad.

   b. Hemos tenido en cuenta al añadir los mecanismos de rollback si se está ejecutando mediante sentencias SQL o funciones SQLAlchemy.
   Begin para realizar el rollback:
      - SQL

```
if bSQL is True:
    db_conn.execute('BEGIN; ')
```

      - SQLAlchemy

```
else:
    transaccion = db_conn.begin()
```

   Rollback:
      - SQL

```
if bSQL is True:
    dbr.append('Rollback SQL')
    db_conn.execute('ROLLBACK; ')
```

- SQLAlchemy

```
else:
    dbr.append('Rollback Transaccion')
    transaccion.rollback()
```

c. Se puede apreciar el uso de las distintas funciones SQL (execute()) y SQLAlchemy (begin()) en el apartado anterior.

d. Después de comprobar en *phppgadmin* que no existen dependencias de este tipo, no tendremos que preocuparnos por eliminarlas. Sin embargo, si las hubiera, habría que hacerlo puesto que en nuestra transacción borraríamos gran parte de ellas.

e. Según nuestra implementación de la transacción, después de la primera eliminación, en el momento en el que el valor *bFallo* (obtenido del formulario) sea **True**, cambiamos el orden de borrado: primero eliminamos lo relacionado a la tabla *customers* y luego lo relacionado a la tabla *orders*.
Si hacemos esto, no habrá columna *state* por lo que los elementos de la tabla *orders* no los podremos eliminar.

   De esta forma, nos damos cuenta de que el orden lógico para el borrado sería:
   1. *Orderdetail*
   2. *Orders*
   3. *Customers*

f. Como hemos explicado en apartados anteriores, hemos implementado los rollbacks para que cuando se produzca una excepción se realice el *rollback* para volver a la situación original, deshaciendo los cambios realizados hasta ese momento.

g. Implementación del orden correcto de borrado:

```
if bSQL is True:
    db_conn.execute('BEGIN; ')

    # Comprobamos el numero de coincidencias a borrar antes y despues de ejecutar la consulta
    # para comprobar que ha funcionado
    query_nCustomers = "SELECT count(*) " \
                       "FROM customers NATURAL JOIN orders NATURAL JOIN orderdetail " \
                       "WHERE state='"+str(state)+"';"

    nCustomers = list(db_conn.execute(query_nCustomers))

    dbr.append("Entradas coincidentes de orderdetails antes de borrar:" + str(nCustomers[0][0]))

    del_orderdetail = "DELETE " \
                      "FROM orderdetail using customers NATURAL JOIN orders " \
                      "WHERE state='"+str(state)+"' AND orders.orderid=orderdetail.orderid;"

    db_conn.execute(del_orderdetail)
    nCustomers = list(db_conn.execute(query_nCustomers))

    dbr.append("Entradas coincidentes de orderdetails despues de borrar:" + str(nCustomers[0][0]))
```

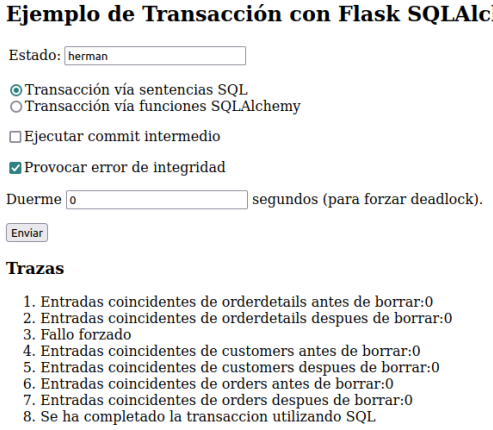
```
    # Si forzamos el fallo, alteramos el orden de ejecucion para producir un error
    if bFallo is True:
```

```
# Si no hay fallo a producir, la ejecucion sigue el orden correcto
else:
    dbr.append('Durmiendo ' + str(duerme) + ' segundo(s)')
    time.sleep(duerme)


    query_totalC2 = "SELECT count(*) " \
                    "FROM customers NATURAL JOIN orders " \
                    "WHERE state='"+str(state)+"';"

    totalC2 = list(db_conn.execute(query_totalC2))

    dbr.append("Entradas coincidentes de orders antes de borrar:" + str(totalC2[0][0]))

    db_conn.execute(del_orders)
    totalC2 = list(db_conn.execute(query_totalC2))

    dbr.append("Entradas coincidentes de orders despues de borrar:" + str(totalC2[0][0]))


    query_totalC = "SELECT count(*) " \
                    "FROM customers " \
                    "WHERE state='"+str(state)+"';"

    totalC = list(db_conn.execute(query_totalC))

    dbr.append("Entradas coincidentes de customers antes de borrar:" + str(totalC[0][0]))

    db_conn.execute(del_customers)
    totalC = list(db_conn.execute(query_totalC))

    dbr.append("Entradas coincidentes de customers despues de borrar:" + str(totalC[0][0]))
```

Estas capturas de nuestro código representan el orden correcto de ejecución en SQL. Para ver nuestra implementación en SQLAlchemy revisar archivo completo.

h. Si ejecutamos la transacción cuando se ha eliminado todo lo correspondiente a una ciudad, obtenemos que no hay entradas a eliminar.

i. Trazas de los cambios parciales que se van realizando:



j. Tras realizar la primera eliminación *orderdetail*, se comprueba la variable *bCommit* y si su valor es **True**, se realizará un *commit* intermedio. Este guardará los cambios realizados tras la eliminación, y si tras el ejecutamos *begin*, el *rollback* ya no volverá al inicio, si no a ese punto.

- Implementación en SQL:

```
if bCommit is True:
    dbr.append('Primer Commit')
    db_conn.execute('COMMIT; ')
    db_conn.execute('BEGIN; ')
```

- Implementación en SQLAlchemy:

```
if bCommit is True:
    dbr.append('Primer Commit')
    transaccion.commit()
    transaccion = db_conn.begin()
```

I. Estudio de bloqueos y deadlocks:
   a. Creamos y cargamos la base de datos.
   b. Creamos script updPromo.sql y creamos una columna de tipo decimal con valor por defecto 0:

```
--creacion de la columna promo con valor default 0
ALTER TABLE customers ADD COLUMN promo DECIMAL(6,2) DEFAULT 0;
```

   c. Trigger:

```
DROP TRIGGER IF EXISTS tr_Promo ON customers;
CREATE OR REPLACE FUNCTION updPromo() RETURNS TRIGGER AS $$
BEGIN
    UPDATE orderdetail
    SET price = pr.price - (pr.price*(NEW.promo/100)) FROM products AS pr, orders AS od WHERE
    od.customerid = NEW.customerid AND od.status IS NULL AND od.orderid = orderdetail.orderid AND pr.prod_id = orderdetail.prod_id;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER tr_Promo AFTER UPDATE ON customers FOR EACH ROW EXECUTE PROCEDURE updPromo();
```

   d. Modificamos trigger para que haga un sleep durante su ejecución:

```
DROP TRIGGER IF EXISTS tr_Promo ON customers;
CREATE OR REPLACE FUNCTION updPromo() RETURNS TRIGGER AS $$
BEGIN
    UPDATE orderdetail
    SET price = pr.price - (pr.price*(NEW.promo/100)) FROM products AS pr, orders AS od WHERE
    od.customerid = NEW.customerid AND od.status IS NULL AND od.orderid = orderdetail.orderid AND pr.prod_id = orderdetail.prod_id;
    PERFORM pg_sleep(10);
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER tr_Promo AFTER UPDATE ON customers FOR EACH ROW EXECUTE PROCEDURE updPromo();
```

   e. El momento ideal de ejecución del sleep en la función *delState*, es después de haber realizado la primera eliminación si no estamos forzando el fallo, ya que, si lo ejecutamos durante un borrado, este podría no aplicarse, o si lo hacemos durante un fallo en la consulta, la integridad de la base de datos podría verse afectada, ya que otros procedimientos o transacciones podrían modificar la base de datos.
   Implementación:

```
dbr.append('Durmiendo ' + str(duerme) + ' segundo(s)')
time.sleep(duerme)
```

   f. Creación de carrito (status a NULL) con UPDATE:

```
UPDATE orders
SET status = null
WHERE orderid = 42135
```

   g. Sin realizar la consulta:

Acciones	orderid	orderdate	customerid	netamount	tax	totalamount	status
Editar Eliminar	42135	2021-04-08	3199	139.8	18	164.96	Paid

   Con la consulta realizada:

Acciones	orderid	orderdate	customerid	netamount	tax	totalamount	status
Editar Eliminar	42135	2021-04-08	3199	139.8	18	164.96	*NULL*

h. Tenemos que activar el trigger y para ello vamos a darle un valor a la columna *promo* que hemos creado antes:

```
UPDATE customers
SET promo = 50
WHERE cutomerid = 3199
```

customerid	firstname	lastname	creditcardtype	creditcard	creditcardexpiration	username	password	age	income	gender	promo
3199	sabras	sitter	American	4890795373643202	201309	verona	luge	33	29592	F	0.00

...

customerid	firstname	lastname	creditcardtype	creditcard	creditcardexpiration	username	password	age	income	gender	promo
3199	sabras	sitter	American	4890795373643202	201309	verona	luge	33	29592	F	50.00

...

i. Mientras dure la transacción, el acceso a la base de datos no está permitido. Esto nos lo indica un pequeño candado que aparece en las tablas al visualizarlas en *phppgadmin.* Esto sucede porque la base de datos es atómica, es decir, no acepta operaciones simultáneas para no dañar la integridad de los datos.

j. Si intentamos acceder a uno de los recursos que se están actualizando en el trigger durante el sleep, se produce un bloqueo ya que ambos procedimientos intentarán acceder a los mismos recursos una vez acabe el sleep, produciendo un bucle en la cola de ejecución (una consulta no puede terminar sin los recursos que acapara la otra).

k. Para evitar los deadlocks es recomendable la brevedad de las transacciones, es decir reducirlas a lo mínimo posible permitiendo una mejor atomicidad. Otro enfoque sería estudiar los posibles bloqueos y llegar a una forma de solventarlos todos mediante esperas o un orden diferente de las transacciones.
Por último, la forma más efectiva de afrontar los deadlocks sería hacer uso de *savepoints (*situación de la transacción donde el estado es consistente*)*. Estos no evitarán la aparición de deadlocks, sino que reaccionan a ellos haciendo que cuando se ejecute un *rollback* a causa de un *deadlock* no se pierda toda la transacción, sino que se retome desde el último savepoint.