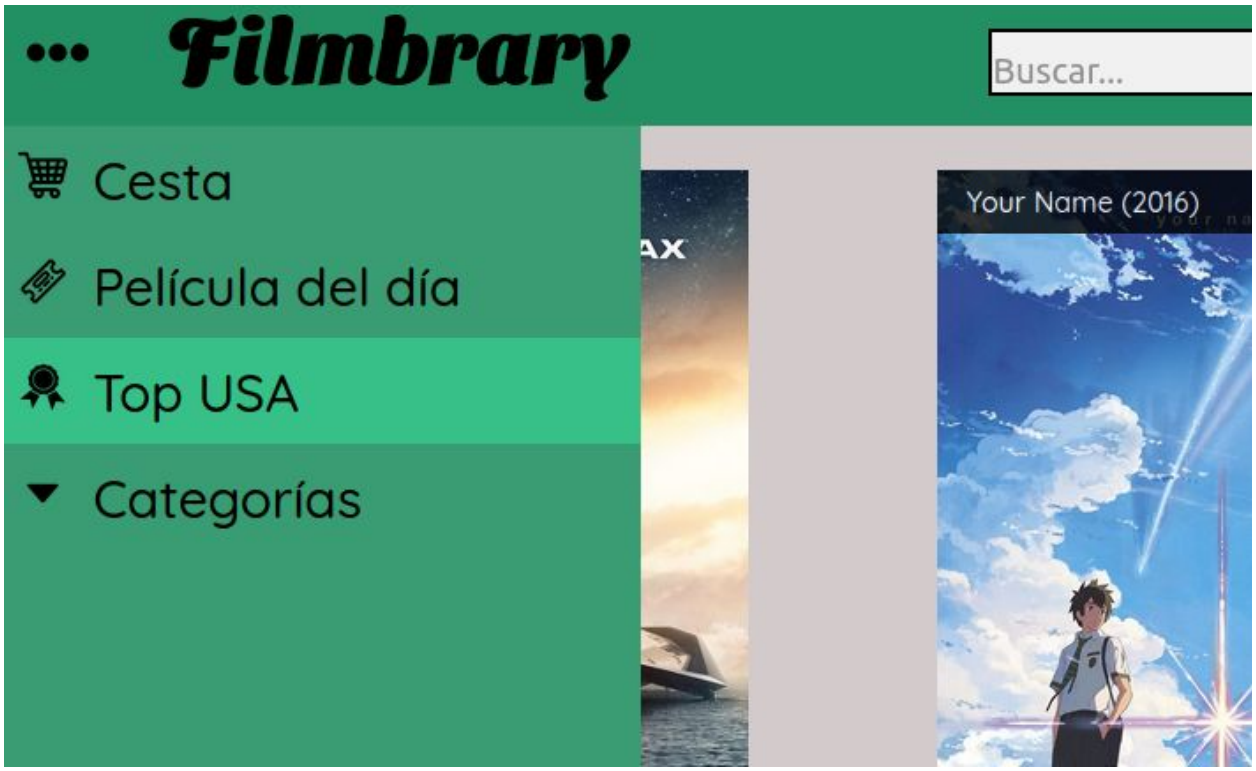
Miguel Herrera y José Manuel Freire

SI - Práctica 3

Group 1391

PARTE 1

Para esta primera parte se nos pedía implementar la consulta top USA, que es el top de las 800 películas más recientes de Estados Unidos. En la página web, hemos añadido en el menú de navegación una nueva pestaña llamada Top USA, que lleva a la página que muestra las películas que se nos pedían. Para poder ejecutar la web y acceder a la página, es necesario ejecutar el script crear.sh dentro de la carpeta funciones, el cual creará las bases de datos de Postgresql y MongoDB (con el createMongoDBFromPostgreSQLDB.py) y actualizará la de Postgresql para que la página funcione.



Nos piden tres tablas que hacer con los datos de las películas en MongoDB. La primera es de las películas que son comedias del año 1997 y contienen la palabra “Life” en el título. Para buscarlas hemos usado el siguiente filtro en mongo:

```
{ "title": { "$regex": "Life" }, "year": "1997", "genres": { "$in": ["Comedy"] } }
```

Películas que son comedias del año 1997 y que contienen la palabra "Life" en el título

Película	Géneros	Año	Directores	Actores	Películas más relacionadas	Películas relacionadas
Life During Wartime (1997)	Comedy	1997	Dunsky, Evan	Dell Jr., Gabriel Gorman, Brad (I) Henderson, Gerald (III) Howell, Hoke Arquette, David Arquette, Lewis Arquette, Richmond Bilancio, Vinnie Brisbin, David (I) Campbell, Colin (II) Y más...	2006, Broken Hearts Club (2006) 2006, Big Bang Theory (2006) 2005, After Life (2005) 2000, Next Best Thing, The (2000) 2000, Flintstones in Viva Rock Vegas, The (2000) 2000, Down to You (2000) 2000, Opportunists, The (2000) 2000, Small Time Crooks (2000) 2000, Kid, The (2000) 2000, Beautiful (2000)	
Life Less Ordinary, A (1997)	Comedy Crime Drama Fantasy Romance	1997	Boyle, Danny	Gorham, Christopher Gowdy, Chuck Hedaya, Dan Holm, Ian Kanig, Frank Chaykin, Maury Stephens, Duane Kellog, Robert Lindo, Delroy McGregor, Ewan Y más...		2000, Here on Earth (2000) 2000, Crow: Salvation, The (2000) 2000, Passion of Mind (2000) 2000, Autumn in New York (2000) 2000, Small Time Crooks (2000) 2000, Beautiful (2000) 2000, Wonder Boys (2000) 2000, Crew, The (2000/I) 2000, Bamboozled (2000) 2000, Nutty Professor II: The Klumps (2000)

La segunda tabla es de las películas que han sido dirigidas por Woody Allen en los años 90:

```
{ "year": { "$gt": "1989", "$lt": "2000" }, "directors": { "$in": ["Allen, Woody"] } }
```

Películas que han sido dirigidas por Woody Allen en los años 90

Película	Géneros	Año	Directores	Actores	Películas más relacionadas	Películas relacionadas
Sweet and Lowdown (1999)	Comedy Drama Music	1999	Allen, Woody	Darrow, Tony Davis, Eddy Duncan, Ben (I) Edelson, Kenneth Erskine, Drummond Francis, Alan (III) Garrett, Brad (I) Garvey, Ray Gayton, Clark Giles III, Brooks Y más...	2000, Coyote Ugly (2000) 2000, High Fidelity (2000) 2000, Duets (2000) 1997, Slaves to the Underground (1997)	2006, Broken Hearts Club (2006) 2000, Next Best Thing, The (2000) 2000, Down to You (2000) 2000, Opportunists, The (2000) 2000, Kid, The (2000) 2000, Beautiful (2000) 2000, Whatever It Takes (2000) 2000, Nurse Betty (2000) 2000, Where the Money Is (2000) 2000, Wonder Boys (2000)
Celebrity (1998)	Comedy	1998	Allen, Woody	Dark, Peter Darrow, Tony DiCaprio, Leonardo Doumanian, John Edelson, Kenneth Eisenberg, Ned (I) Erskine, Howard Faragallah, Ramsey Fitzgerald, Donegal Ford, Clebert Y más...	2006, Broken Hearts Club (2006) 2006, Big Bang Theory (2006) 2005, After Life (2005) 2000, Next Best Thing, The (2000) 2000, Flintstones in Viva Rock Vegas, The (2000) 2000, Down to You (2000) 2000, Opportunists, The (2000) 2000, Small Time Crooks (2000) 2000, Kid, The (2000) 2000, Beautiful (2000)	
Deconstructing Harry (1997)	Comedy	1997	Allen, Woody	Darrow, Tony Doumanian, John Edelson, Kenneth Frazer, Dan Garvey, Ray Giamatti, Paul Glazer, Brett Harper, Robert (I) Howard, David S. Allen, Woody Y más...	2006, Broken Hearts Club (2006) 2006, Big Bang Theory (2006) 2005, After Life (2005) 2000, Next Best Thing, The (2000) 2000, Flintstones in Viva Rock Vegas, The (2000) 2000, Down to You (2000) 2000, Opportunists, The (2000) 2000, Small Time Crooks (2000) 2000, Kid, The (2000) 2000, Beautiful (2000)	

La tercera tabla es de las películas en las que Jim Parsons y Johnny Galecki han compartido reparto. En este caso, hemos buscado al actor Jim Parsons (II) ya que es el que existe en mongo:

```
{ "actors": { "$all": ["Parsons, Jim (II)", "Galecki, Johnny"] } }
}
```

Películas en las que Jim Parsons y Johnny Galecki han compartido reparto

Película	Géneros	Año	Directores	Actores	Películas más relacionadas	Películas relacionadas
Big Bang Theory (2006)	Comedy	2006		Galecki, Johnny Parsons, Jim (II) Walsh, Amanda (II)	2006, Broken Hearts Club (2006) 2005, After Life (2005) 2000, Next Best Thing, The (2000) 2000, Flintstones in Viva Rock Vegas, The (2000) 2000, Down to You (2000) 2000, Opportunists, The (2000) 2000, Small Time Crooks (2000) 2000, Kid, The (2000) 2000, Beautiful (2000) 2000, Coyote Ugly (2000)	

PARTE 2.1

En este ejercicio hemos creado la consulta clientesDistintos que devuelve el número de clientes distintos que hayan comprado películas por un importe mayor al dado en el mes dado. El índice que encontramos para optimizar la consulta fue status (para confirmar que el pedido fue pagado); esto se debe a que status es una string y requiere más procesamiento que comparar enteros. Con EXPLAIN obtuvimos el plan de ejecución de la consulta.

Con índice:

QUERY PLAN text
Aggregate (cost=2078.90..2078.91 rows=1 width=8)
-> GroupAggregate (cost=2078.86..2078.89 rows=1 width=36)
Group Key: orders.customerid
Filter: (sum(orders.totalamount) >= '200'::numeric)
-> Sort (cost=2078.86..2078.87 rows=1 width=36)
Sort Key: orders.customerid
-> Bitmap Heap Scan on orders (cost=19.24..2078.85 rows=1 width=36)
Filter: (((status)::text ~~ 'Paid'::text) AND (date_part('year'::text, (orderdate)::
-> Bitmap Index Scan on myindex (cost=0.00..19.24 rows=909 width=0)
Index Cond: ((status)::text = 'Paid'::text)

Sin índice:

QUERY PLAN text
Aggregate (cost=6455.16..6455.17 rows=1 width=8)
-> GroupAggregate (cost=6455.12..6455.14 rows=1 width=36)
Group Key: orders.customerid
Filter: (sum(orders.totalamount) >= '200'::numeric)
-> Sort (cost=6455.12..6455.13 rows=1 width=36)
Sort Key: orders.customerid
-> Gather (cost=1000.00..6455.11 rows=1 width=36)
Workers Planned: 2
-> Parallel Seq Scan on orders (cost=0.00..5455.01 rows=1 width=36)
Filter: (((status)::text ~~ 'Paid'::text) AND (date_part('year'::text,

Comparación de ejecuciones: En el primer caso (con índice), podemos ver que utiliza Index Cond con un coste mucho menor que el que tiene el filter en el caso sin índice. Este coste mayor lo arrastra en cada operación, haciendo que el coste total de la consulta sea, en el caso sin índice, tres veces mayor que con él.

Otros índices que hemos probado fueron orderdate, totalamount, customerid, orderdate + totalamount, totalamount + customerid y customerid + orderdate. Ninguno de ellos modificaron el plan de consulta, por lo que su tiempo de ejecución tampoco cambió respecto a la consulta sin índice.

PARTE 2.2

Impacto al cambiar la forma de las consultas:

QUERY PLAN text
Seq Scan on customers (cost=5646.65..6175.81 rows=7046 width=4)
Filter: (NOT (hashed SubPlan 1))
SubPlan 1
-> Seq Scan on orders (cost=0.00..5644.38 rows=909 width=4)
Filter: ((status)::text = 'Paid'::text)

Con `where ~ not in` podemos ver que el plan es mucho más sencillo que en el resto de consultas. Esta consulta solo hace un Seq Scan on orders y luego en customers.

QUERY PLAN text
HashAggregate (cost=5987.75..5989.75 rows=200 width=4)
Group Key: customers.customerid
Filter: (count(*) = 1)
-> Append (cost=0.00..5912.74 rows=15002 width=4)
-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4)
-> Gather (cost=1000.00..5409.72 rows=909 width=4)
Workers Planned: 2
-> Parallel Seq Scan on orders (cost=0.00..4318.82 rows=379 width=4)
Filter: ((status)::text = 'Paid'::text)

QUERY PLAN text
HashSetOp Except (cost=0.00..6091.18 rows=14093 width=8)
-> Append (cost=0.00..6053.67 rows=15002 width=8)
-> Subquery Scan on "*SELECT* 1" (cost=0.00..634.86 rows=14093 width=8)
-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4)
-> Subquery Scan on "*SELECT* 2" (cost=1000.00..5418.81 rows=909 width=8)
-> Gather (cost=1000.00..5409.72 rows=909 width=4)
Workers Planned: 2
-> Parallel Seq Scan on orders (cost=0.00..4318.82 rows=379 width=4)
Filter: ((status)::text = 'Paid'::text)

Estas dos últimas consultas son las que se benefician de la ejecución en paralelo, ya que ambas crean dos workers y realizan el scan on orders de forma paralela. Mientras que la primera consulta utiliza un escáner secuencial, estas dos utilizan hashes, lo cual reduce el coste de la operación mucho. Entre las dos últimas consultas, la primera de ellas utiliza HashAggregate, para operar la unión entre las queries, mientras que la segunda utiliza HashSetOp, para operar con el resultado de las dos consultas (la de dentro y la de fuera del except).

PARTE 2.3

Estudio del impacto de la generación de estadísticas. Resultados del explain con cada una de las consultas del anexo 2 (consulta countStatus.sql):

QUERY PLAN text
Aggregate (cost=3507.17..3507.18 rows=1 width=8)
-> Seq Scan on orders (cost=0.00..3504.90 rows=909 width=0)
Filter: (status IS NULL)

QUERY PLAN text
Aggregate (cost=3961.65..3961.66 rows=1 width=8)
-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=0)
Filter: ((status)::text = 'Shipped'::text)

Ahora probamos creando un index sobre orders.status:

QUERY PLAN text
Aggregate (cost=1496.52..1496.53 rows=1 width=8)
-> Bitmap Heap Scan on orders (cost=19.46..1494.25 rows=909 width=0)
Recheck Cond: (status IS NULL)
-> Bitmap Index Scan on myindex (cost=0.00..19.24 rows=909 width=0)
Index Cond: (status IS NULL)

QUERY PLAN text
Aggregate (cost=1498.79..1498.80 rows=1 width=8)
-> Bitmap Heap Scan on orders (cost=19.46..1496.52 rows=909 width=0)
Recheck Cond: ((status)::text = 'Shipped'::text)
-> Bitmap Index Scan on myindex (cost=0.00..19.24 rows=909 width=0)
Index Cond: ((status)::text = 'Shipped'::text)

En ambos casos podemos ver como el plan de ejecución se hizo más complicado, pero el coste de ejecución se ha reducido a menos de la mitad.

QUERY PLAN text
Aggregate (cost=1496.52..1496.53 rows=1 width=8) (actual time=0.268..0.269 rows=1 loops=1)
-> Bitmap Heap Scan on orders (cost=19.46..1494.25 rows=909 width=0) (actual time=0.265..0.265 rows=0 loops=1)
Recheck Cond: (status IS NULL)
-> Bitmap Index Scan on myindex (cost=0.00..19.24 rows=909 width=0) (actual time=0.230..0.230 rows=0 loops=1)
Index Cond: (status IS NULL)
Planning time: 0.134 ms
Execution time: 0.309 ms

QUERY PLAN text
Aggregate (cost=1498.79..1498.80 rows=1 width=8) (actual time=70.172..70.172 rows=1 loops=1)
-> Bitmap Heap Scan on orders (cost=19.46..1496.52 rows=909 width=0) (actual time=41.711..62.903 rows=127323 loops=1)
Recheck Cond: ((status)::text = 'Shipped'::text)
Heap Blocks: exact=1686
-> Bitmap Index Scan on myindex (cost=0.00..19.24 rows=909 width=0) (actual time=41.485..41.485 rows=127323 loops=1)
Index Cond: ((status)::text = 'Shipped'::text)
Planning time: 0.317 ms
Execution time: 70.312 ms

En este caso podemos ver que la segunda consulta requiere una instrucción más, ya que necesita encontrar la palabra “shipped”, mientras que la primera no compara palabras sino null.

Con las otras dos queries:

QUERY PLAN text
Aggregate (cost=1498.79..1498.80 rows=1 width=8) (actual time=10.015..10.015 rows=1 loops=1)
-> Bitmap Heap Scan on orders (cost=19.46..1496.52 rows=909 width=0) (actual time=2.011..8.284 rows=18163 loops=1)
Recheck Cond: ((status)::text = 'Paid'::text)
Heap Blocks: exact=1686
-> Bitmap Index Scan on myindex (cost=0.00..19.24 rows=909 width=0) (actual time=1.677..1.677 rows=18163 loops=1)
Index Cond: ((status)::text = 'Paid'::text)
Planning time: 0.081 ms
Execution time: 10.058 ms

QUERY PLAN text
Aggregate (cost=1498.79..1498.80 rows=1 width=8) (actual time=18.734..18.734 rows=1 loops=1)
-> Bitmap Heap Scan on orders (cost=19.46..1496.52 rows=909 width=0) (actual time=3.640..14.714 rows=36304 loops=1)
Recheck Cond: ((status)::text = 'Processed'::text)
Heap Blocks: exact=1685
-> Bitmap Index Scan on myindex (cost=0.00..19.24 rows=909 width=0) (actual time=3.267..3.267 rows=36304 loops=1)
Index Cond: ((status)::text = 'Processed'::text)
Planning time: 0.090 ms
Execution time: 18.780 ms

En este caso las consultas son iguales (Incluso el mismo coste) esto se debe a que pese a que la string cambie, la operación es exactamente la misma: buscar esa string.

PARTE 3.1

En el archivo database.py se nos pedía completar la función delCustomer que crea una transacción para borrar todos los datos asociados a un cliente. Aunque hemos utilizado el esqueleto suministrado, tuvimos que cambiar el routes.py para que pudiésemos coger bien los datos del formulario html. Como la página recibe los argumentos vía GET hemos sustituido ‘request.form’ por ‘request.values’, aparte de incluir la siguiente línea de código al principio de la función borraCliente:

```
@app.route('/', methods=['POST','GET'])
```

De tal forma podemos acceder a la página web directamente con localhost:5001.

Lo primero que debemos hacer para borrar el cliente es conectarnos a la base de datos en Postgres con el autocommit en False. Esto es muy importante ya que vamos a ser nosotros quienes manejemos cuándo hacer los commit. Dependiendo de los argumentos introducidos por el usuario en el formulario web, haremos fallar la transacción o no, meteremos un commit intermedio cuando falla, o utilizaremos SQL o SQLAlchemy.

Para hacer la transacción en SQL es tan fácil como ejecutar los comandos con db_conn.execute(), ya sea “BEGIN”, “COMMIT” o “ROLLBACK”. Mientras que en SQLAlchemy hay funciones específicas para cada acción:

```
t = db_conn.begin() # Siendo db_conn la conexión a la BD

t.commit() / t.rollback()
```

Al borrar los datos asociados al cliente, primero debemos borrar todas las orderid en la tabla orderdetail que sean del customerid que vamos a eliminar. Para ello hemos hecho lo siguiente:

```
db_orders = list(db_conn.execute("Select orderid from orders where customerid = '" + str(customerid) + "'"))

# Ahora iteramos a través de todas las orders del usuario y las borramos una a una en orderdetail

for orderid in db_orders:

    db_conn.execute("Delete from orderdetail where orderid = '" + str(orderid[0]) + "'")
```

Una vez borrada esa información, comprobamos si la transacción debe ser fallida o no. Si tiene que serla, solo nos basta con borrar de la tabla customers el customerid. De tal forma, cuando después queramos borrar las orders asociadas a ese customerid, nos dará error ya que no existe. Así forzamos el fallo de la transacción:

```
db_conn.execute("Delete from customers where customerid = '" +
str(customerid) + "'")

db_conn.execute("Delete from orders where customerid = '" +
str(customerid) + "'")
```

En el caso de que queramos hacer un commit intermedio, lo hacemos antes de estos dos delete. Ahora, **¿por qué hacemos commit seguido de begin?** Porque así al hacer rollback volvemos al estado de antes de hacer los últimos dos delete en orders y customers, en vez de volver al inicio antes del delete en orderdetail. Aunque en nuestro caso no tiene mucho sentido, ya que si hay algún fallo en la transacción queremos que tampoco se borre nada en orderdetail para mantener la integridad de la base de datos.

Para terminar, solo falta hacer rollback si la transacción es fallida, o commit si es correcta, ya sea con SQL o SQLAlchemy como hemos descrito antes.

PARTE 3.2

Hemos creado el script updPromo.sql, que creará una nueva columna llamada promo en la tabla customers. Esta columna la usaremos para aplicar un descuento en el carrito del cliente al que se le ponga. También hemos creado un trigger, de forma que al cambiar la columna promo de un cliente, se actualice el precio de su carrito.

El trigger se compone de dos updates. El primero actualiza el precio en orderdetail para cada película y el segundo actualiza el netamount de orders (no hemos actualizado el precio total con impuestos). Además tuvimos que añadir un sleep al trigger entre los dos updates.

```
PERFORM pg_sleep(5);
```

Antes del descuento:

orderid integer	prod_id integer	price numeric	quantity integer	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying(10)
4187	6171	9.7087378640776699	1	4187	2019-02-04	305	112.4271844660194175	15	129.29	null
4190	2812	11.6504854368932039	1	4190	2019-06-22	305	74.9514563106796117	15	86.19	null
4190	3999	14.5631067961165049	1	4190	2019-06-22	305	74.9514563106796117	15	86.19	null
4190	4693	12.6213592233009709	1	4190	2019-06-22	305	74.9514563106796117	15	86.19	null
4190	553	18.6407766990291262	1	4190	2019-06-22	305	74.9514563106796117	15	86.19	null
4190	1876	17.4757281553398058	1	4190	2019-06-22	305	74.9514563106796117	15	86.19	null
4189	3772	19.8058252427184466	1	4189	2019-08-29	305	19.8058252427184466	15	22.78	null
4187	2999	13.5922330097087379	1	4187	2019-02-04	305	112.4271844660194175	15	129.29	null
4187	3214	18.4466019417475728	1	4187	2019-02-04	305	112.4271844660194175	15	129.29	null
4187	791	18.4466019417475728	1	4187	2019-02-04	305	112.4271844660194175	15	129.29	null
4187	2339	10.6796116504854369	1	4187	2019-02-04	305	112.4271844660194175	15	129.29	null
4187	3677	9.7087378640776699	2	4187	2019-02-04	305	112.4271844660194175	15	129.29	null

Después del descuento (50%):

orderid integer	prod_id integer	price numeric	quantity integer	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying(10)
4187	6171	5.0000000000000000000	1	4187	2019-02-04	305	57.900000000000000000000	15	129.29	null
4190	2812	6.0000000000000000000	1	4190	2019-06-22	305	38.600000000000000000000	15	86.19	null
4190	3999	7.5000000000000000000	1	4190	2019-06-22	305	38.600000000000000000000	15	86.19	null
4190	4693	6.5000000000000000000	1	4190	2019-06-22	305	38.600000000000000000000	15	86.19	null
4190	553	9.6000000000000000000	1	4190	2019-06-22	305	38.600000000000000000000	15	86.19	null
4190	1876	9.0000000000000000000	1	4190	2019-06-22	305	38.600000000000000000000	15	86.19	null
4189	3772	10.200000000000000000	1	4189	2019-08-29	305	10.200000000000000000	15	22.78	null
4187	2999	7.0000000000000000000	1	4187	2019-02-04	305	57.900000000000000000000	15	129.29	null
4187	3214	9.5000000000000000000	1	4187	2019-02-04	305	57.900000000000000000000	15	129.29	null
4187	791	9.5000000000000000000	1	4187	2019-02-04	305	57.900000000000000000000	15	129.29	null
4187	2339	5.5000000000000000000	1	4187	2019-02-04	305	57.900000000000000000000	15	129.29	null
4187	3677	10.000000000000000000	2	4187	2019-02-04	305	57.900000000000000000000	15	129.29	null

El sleep en la página que borra a un cliente ocurre cuando comprobamos que la transacción es exitosa. Por eso, si se provoca el error de integridad, el sleep no se realizará. Mientras la página está en el sleep, no se muestra nada y en caso de iniciar otra sesión, tiene que esperar a que el sleep de la anterior termine.

En la foto de abajo se puede ver como la segunda página tiene que esperar a que la primera termine el sleep (la que se ve tiene sleep de 2 segundos pero tuvo que esperar más, ya que la que está en segundo plano tenía un sleep de 120 segundos que se inició antes). Por esa razón en la sección de las trazas no se muestra ninguna.

Ejemplo de Transacción con Flask SQLAlchemy

Customer ID: 100

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

[ ] Ejecutar commit intermedio

[ ] Provocar error de integridad

Duerme 2 segundos (para forzar deadlock).

Enviar

Trazas