Pablo Soëtard García y Rodrigo Juez

SISTEMAS INFORMÁTICOS - P3

18 de Diciembre 2020

Material Entregado:

- RPPelis/*: Contiene la página web de las prácticas anteriores, incluida la página TopUSA.
  ¡IMPORTANTE! Para ejecutar RPPelis debes hacer primero make en la carpeta de database, esto generará las bases de datos necesarias, (la antigua, la nueva y la de mongoDB junto al script de python).
- database/*: Contiene las consultas y actualizaciones de la base de datos de la práctica anterior así como (contenido en la carpeta p3) el script createMongoDBFromPostgreSQLDB.py, y las consultas de sql anexo1.sql, anexo2.sql, clientesDistintos.sql, countStatus.sql, updPromo.sql solicitadas, además de algunos tests para ellas.
- transacciones/* incluye la app auxiliar de la parte 3 de la práctica.
  ¡IMPORTANTE! El updPromo y testupdPromo (para probar el deadlock se encuentran en /database).

1.- noSQL

A) B)
Para realizar estos apartados hemos creado, como se nos pedía, un script en python createMongoDBFromPostgreSQLDB.py el cual, haciendo uso de consultas ejecutadas mediante SQLAlchemy, crea en python el diccionario “movies” que contiene todos los datos que se deben insertar en la base de datos de MongoDB. A continuación, en el mismo script, creamos una nueva DB de Mongo llamada si1 en la cual se inserta el diccionario “movies” creado anteriormente.

C)
Como se indica en el enunciado hemos realizado los cambios necesarios en los templates y en los archivos routes.py y database.py para que en el menú de navegación aparezca un nuevo botón “Top USA”.

D)
Para realizar este apartado hemos creado tres funciones mongo_a(), mongo_b() y mongo_c() en database.py las cuales devuelven la información requerida por cada subapartado realizando consultas a la base de datos de MongoDB mediante pymongo.
La primera query es muy directa, ponemos los requisitos y usamos regex para buscar películas por el título (‘Life+’ significa que el título debe contener mínimo un ‘Life’).
Para la segunda query, en vez de complicarnos con rangos de años, como está guardado en una string simplemente usamos regex de nuevo para que busque años de 1990-1999.
En la tercera query buscamos en el array de actores que estén incluidos tanto Parsons como Galecki, tuvimos problemas porque no sabíamos que Jim Parsons tenía un (II) en el nombre.

A la hora de integrarlo en la web seguimos la línea del resto de pestañas y usamos tablas que se expanden con información adicional, puesto que la lista de películas relacionadas era muy larga (como se puede ver en The Big Bang Theory) y de esta manera se veían los resultados sin hacer scrolling.

```python
def mongodb_a():
    myclient = pymongo.MongoClient('mongodb://localhost:27017/')
    col = myclient['si1']['topUSA']
    a = col.find({'year': '1997',
                  'title': {"$regex": "Life+"},
                  'genres': 'Comedy'
                  })
    myclient.close()
    return a


def mongodb_b():
    myclient = pymongo.MongoClient('mongodb://localhost:27017/')
    col = myclient['si1']['topUSA']
    b = col.find({'directors': "Allen, Woody",
                  'year': {"$regex": "199+"}
                  })
    myclient.close()
    return b


def mongodb_c():
    myclient = pymongo.MongoClient('mongodb://localhost:27017/')
    col = myclient['si1']['topUSA']
    c = col.find({'actors': {"$all": ["Parsons, Jim (II)",
                                      "Galecki, Johnny"]}})
    myclient.close()
    return c
```

Inicio

Top USA

Registrar

Iniciar Sesión

Películas del 1997 con "Life"

Título	Genero	Año	Directores	Actores	Películas más relacionadas	Películas relacionadas
Life During Wartime (1997)	Comedy	1997				
Life Less Ordinary, A (1997)	Comedy Crime Drama Fantasy Romance	1997				

Películas de los 90 por Woody Allen

Título	Genero	Año	Directores	Actores	Películas más relacionadas	Películas relacionadas
Sweet and Lowdown (1999)	Comedy Drama Music	1999				
Celebrity (1998)	Comedy	1998				
Deconstructing Harry (1997)	Comedy	1997				

Películas por Jim Parsons y Johnny Galecki

Título	Genero	Año	Directores	Actores	Películas más relacionadas	Películas relacionadas
Big Bang Theory (2006)	Comedy	2006		Galecki, Johnny Parsons, Jim (II) Walsh, Amanda (II)	1. Broken Hearts Club (2006): 2006 2. After Life (2005): 2005 3. 3 Strikes (2000): 2000 4. About Adam (2000): 2000 5. Adventures of Rocky & Bullwinkle, The (2000): 2000 6. American Psycho (2000): 2000 7. American Virgin (2000): 2000 8. Bamboozled (2000): 2000 9. Beautiful (2000): 2000 10. Best in Show (2000): 2000	1. Big Momma's House (2000): 2000 2. Bootmen (2000): 2000 3. Bossa Nova (2000): 2000 4. Boys and Girls (2000): 2000 5. Bring It On (2000): 2000 6. Cecil B. DeMented (2000): 2000 7. Chain of Fools (2000): 2000 8. Chuck&Buck (2000): 2000 9. Clerks (2000): 2000 10. Committed (2000):

2.- Optimización

E)

sin índices:

```
Aggregate  (cost=5627.93..5627.94 rows=1 width=8)
  ->  Gather  (cost=1000.00..5627.92 rows=2 width=4)
        Workers Planned: 1
        ->  Parallel Seq Scan on orders  (cost=0.00..4627.72 rows=1 width=4)
                    Filter: ((totalamount > '100'::numeric) AND (date_part('year'::text, (orderdate)::timestamp without time zone) =
'2015'::double precision) AND (date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision))
(5 filas)
```

índice de totalamount:

```
Aggregate  (cost=4480.32..4480.33 rows=1 width=8)
  ->  Bitmap Heap Scan on orders  (cost=1126.90..4480.32 rows=2 width=4)
        Recheck Cond: (totalamount > '100'::numeric)
                    Filter:  ((date_part('year'::text, (orderdate)::timestamp without time zone) = '2015'::double precision) AND
(date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision))
        ->  Bitmap Index Scan on ordersindex  (cost=0.00..1126.90 rows=60597 width=0)
              Index Cond: (totalamount > '100'::numeric)
(6 filas)
```

índice de (totalamount, (EXTRACT(YEAR FROM orderdate)), EXTRACT(MONTH FROM orderdate)):

```
Aggregate  (cost=1973.91..1973.92 rows=1 width=8)
  ->  Index Scan using ordersindex on orders  (cost=0.42..1973.90 rows=2 width=4)
            Index Cond: ((totalamount > '100'::numeric) AND (date_part('year'::text, (orderdate)::timestamp without time zone) =
'2015'::double precision) AND (date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision))
(3 filas)
```

El código entregado está dentro de una función con argumento de formato YYYYMM pero para testear el rendimiento extrajimos la query y harcodeamos los valores (totalamount>100 y 201504). Probamos índices con totalamount, combinandolo con el orderdate o con el customerid.

El primero que nos dió buen rendimiento fue el totalamount, este rendimiento es gracias a un bitmap que comprueba la condición de totalamount.

Adicionalmente probamos a crear el índice sobre la función de la orderdate en el formato que la usamos (extrayendo el year y el month) esto nos dió una ganancia de rendimiento muy alta. Como se puede ver el planificador sólo necesitó buscar la condición directamente en el índice y devolverla, a diferencia con el anterior que solo podía buscar el totalamount.

El customerid no se usa para realizar el filtrado y entorpece el índice, pero es que además empeora el tiempo de ejecución, el número de filas visitadas es el mismo, así que creemos que el aumento en coste se debe a que tiene que comparar más elementos..

F)

```
Query1:
                          QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on customers  (cost=3961.65..4490.81 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)
(5 filas)


Query2:
                          QUERY PLAN
-------------------------------------------------------------------------------
 HashAggregate  (cost=4537.41..4539.41 rows=200 width=4)
   Group Key: customers.customerid
   Filter: (count(*) = 1)
   ->  Append  (cost=0.00..4462.40 rows=15002 width=4)
         ->  Seq Scan on customers  (cost=0.00..493.93 rows=14093 width=4)
         ->  Seq Scan on orders  (cost=0.00..3959.38 rows=909 width=4)
               Filter: ((status)::text = 'Paid'::text)
(7 filas)
```

Query3:

```
                                QUERY PLAN
-------------------------------------------------------------------------------------
 HashSetOp Except  (cost=0.00..4640.83 rows=14093 width=8)
   ->  Append  (cost=0.00..4603.32 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=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)
(7 filas)
```

i. La query3 es la consulta que antes empieza a devolver resultados, lo hace nada más empezar su ejecución (como se ve en el cost=0.00..) Esto se debe a que los resultados que no cumplen el except se van añadiendo al resultado a medida que se van obteniendo en vez de ir obteniendo resultados parciales y después juntarlos como hacen las otras 2 consultas, cuyos costes de la planificación de inicio más externos son mayores que 0.

ii. Solo se benefician de la paralelización la query2 y la query3. La query3 es debido a que se pueden ejecutar ambas subqueries al mismo tiempo y luego solo se tiene que hacer en serie el EXCEPT, como se ve en el explain tiene dos flechas al mismo nivel de subqueries, por lo tanto no depende una de la otra.
La query2 es debido a lo mismo, se hacen dos escaneos paralelamente y se une secuencialmente con el UNION, como se puede observar los escaneos están al mismo nivel.

G)

SIN INDEX:

query1:

```
                        QUERY PLAN
-----------------------------------------------------------------
 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)
(3 filas)
```

query2:

```
                        QUERY PLAN
-----------------------------------------------------------------
 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)
(3 filas)
```

B. Ambas hacen un escaneo secuencial tanto para mirar si es NULL o si es ‘Shipped’ lo cual es muy costoso, la diferencia de rendimiento se debe a que la **query2** debe comparar una string, pero accede al mismo número de filas.

CON INDEX:

query1:

```
                                QUERY PLAN
-------------------------------------------------------------------------------------
 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 ordersindex  (cost=0.00..19.24 rows=909 width=0)
               Index Cond: (status IS NULL)
(5 filas)
```

query2:

```
                                QUERY PLAN
-------------------------------------------------------------------------------------
 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 ordersindex  (cost=0.00..19.24 rows=909 width=0)
               Index Cond: ((status)::text = 'Shipped'::text)
(5 filas)
```

D. De nuevo el plan de ejecución de ambas es el mismo, esto es debido a que la condición es de un valor fijo, no es un rango de strings (como lo sería: ‘Shipped%’). Eso sí, es mucho más rápido debido a que usa el índice, además con un Bitmap por

lo que accederá a cada página del disco una única vez. El que busca 'Shipped' tarda ligeramente más pero no mucho puesto que el índice ordena los strings por cada carácter, por lo tanto es muy rápido al buscar.

ANALYZE:

query1:

```
                        QUERY PLAN
-----------------------------------------------------------------------------------
 Aggregate  (cost=7.28..7.29 rows=1 width=8)
   ->  Index Only Scan using ordersindex on orders  (cost=0.42..7.28 rows=1 width=0)
         Index Cond: (status IS NULL)
(3 filas)
```

query2:

```
                          QUERY PLAN
-------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=4210.49..4210.50 rows=1 width=8)
   ->  Gather  (cost=4210.37..4210.48 rows=1 width=8)
         Workers Planned: 1
         ->  Partial Aggregate  (cost=3210.37..3210.38 rows=1 width=8)
               ->  Parallel Seq Scan on orders  (cost=0.00..3023.69 rows=74673 width=0)
                     Filter: ((status)::text = 'Shipped'::text)
(6 filas)
```

F. A diferencia de los casos anteriores se puede observar una increíble mejora en la **query1** pero no en la **query2** que incluso empeora.
Está bastante claro porque mejora la primera query, el analyze ha visto que solo hay una row con NULL y accede directamente (*rows=1* nos dice que solo accede a una row).
Creemos que la segunda tarda más debido a que, tras haber ejecutado el analyze, el planificador de consultas considera más eficiente ejecutar la consulta en paralelo en vez que hacerlo usando los índices ya creados (como hace en la primera), y esto hace que el coste de crear los hilos paralelos y después juntarlos no compense debido a que el volumen de datos no es lo suficientemente grande.

query3:

```
                          QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate  (cost=2312.31..2312.32 rows=1 width=8)
   ->  Bitmap Heap Scan on orders  (cost=355.71..2267.37 rows=17973 width=0)
         Recheck Cond: ((status)::text = 'Paid'::text)
         ->  Bitmap Index Scan on ordersindex  (cost=0.00..351.22 rows=17973 width=0)
               Index Cond: ((status)::text = 'Paid'::text)
(5 filas)
```

query4:

```
                          QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate  (cost=2936.65..2936.66 rows=1 width=8)
   ->  Bitmap Heap Scan on orders  (cost=710.82..2846.84 rows=35922 width=0)
         Recheck Cond: ((status)::text = 'Processed'::text)
         ->  Bitmap Index Scan on ordersindex  (cost=0.00..701.83 rows=35922 width=0)
               Index Cond: ((status)::text = 'Processed'::text)
(5 filas)
```

G. Estas dos consultas vuelven a utilizar un bitmap, es decir se aprovechan de los índices ya creados para buscar los elementos requeridos. Creemos que son más lentas que la **query2** cuando usaba el índice (apartado D) porque seguramente allá muchos más casos en la base de datos y tenga que hacer más comparaciones.

Ayuda:
El generador de estadísticas crea un fichero con los datos obtenidos de realizar las estadísticas que sirve al planificador de consultas para generar el plan de consulta más eficiente.

Como hemos mencionado anteriormente, tras obtener las estadísticas, el planificador de consultas tiene una idea más concisa de la organización de la BD, por ello adecua las consultas para que sean más eficientes (o lo intenta).

3.- Transacciones

H) Para la realización de este ejercicio hemos tomado como punto de partida la página web proporcionada en los ficheros auxiliares “FicherosAuxiliaresP3.zip”. En el archivo database.py hemos completado la función delCustomer(). Primero hemos declarado las consultas necesarias para eliminar los customers, orders y orderdetails, además de aquellas consultas que nos permitieran ver el estado de los customers y orderdetails de la BD para mostrarlas en las Trazas. A continuación hemos hecho que se ejecuten de acuerdo con los parámetros recibidos por el get, ya sea con o sin commit intermedio, con o sin fallo, con sentencias SQL o funciones SQLAlchemy. Aunque en las capturas se vea que está marcado “Transacciones SQL”, lo hemos probado con funciones también y el resultado es el mismo.

j) El BEGIN es necesario después del COMMIT porque este último finaliza la transacción, y para realizar una nueva hay que usar BEGIN, como al principio de la función.

Ejemplo de Transacción con Flask SQLAlchemy

Customer ID: 2

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

[x] Ejecutar commit intermedio

[x] Provocar error de integridad

Duerme 0 segundos (para forzar deadlock).

Enviar

Trazas

1. Ejecutado BEGIN
2. Contenido Customers: [(2, 'claim', 'portal', 'bethel scotty 16', 'ieyasu major', 'lanky', 'spryer', '40285', ' Vietnam', 'biddy ', 'claim.portal@jmail.com', '+56 222098611', 'VISA', '4517014291483479', '201203', 'laxity', 'cesar', 53, 35645, 'M')]
3. Contenido OrderDetail: [(111, 3260, Decimal('14.79426722145168 74'), 1, datetime.date(2016, 3, 1), 2, Decimal('55.4785020804438278'), Decimal('15'), Decimal('63.80'), 'Shipped'), (112, 5153, Decimal('12'), 1, datetime.date(2020, 2, 22), 2, Decimal('117.1'), Decimal('18'), Decimal('138.18'), 'Paid'), (112, 6161, Decimal('22.5'), 1, datetime.date(2020, 2, 22), 2, Decimal('117.1'), Decimal('18'), Decimal('138.18'), 'Paid'), (112, 2164, Decimal('14.4'), 1, datetime.date(2020, 2, 22), 2, Decimal('117.1'), Decimal('18'), Decimal('138.18'), 'Paid'), (112, 1019, Decimal('18'), 1, datetime.date(2020, 2, 22), 2, Decimal('117.1'), Decimal('18'), Decimal('138.18'), 'Paid'), (112, 2176, Decimal('19'), 1, datetime.date(2020, 2, 22), 2, Decimal('117.1'), Decimal('18'), Decimal('138.18'), 'Paid'), (112, 4150, Decimal('15.6'), 2, datetime.date(2020, 2, 22), 2, Decimal('117.1'), Decimal('18'), Decimal('138.18'), 'Paid'), (110, 3930, Decimal('13'), 1, datetime.date(2020, 10, 2), 2, Decimal('87.8'), Decimal('18'), Decimal('103.60'), 'Processed'), (110, 809, Decimal('19'), 1, datetime.date(2020, 10, 2), 2, Decimal('87.8'), Decimal('18'), Decimal('103.60'), 'Processed'), (110, 3307, Decimal('15'), 1, datetime.date(2020, 10, 2), 2, Decimal('87.8'), Decimal('18'), Decimal('103.60'), 'Processed'), (110, 4083, Decimal('18'), 1, datetime.date(2020, 10, 2), 2, Decimal('87.8'), Decimal('18'), Decimal('103.60'), 'Processed'), (110, 4626, Decimal('22.8'), 1, datetime.date(2020, 10, 2), 2, Decimal('87.8'), Decimal('18'), Decimal('103.60'), 'Processed'), (115, 5637, Decimal('12.9449838187702266'), 1, datetime.date(2018, 6, 21), 2, Decimal('122.4225612575127138'), Decimal('15'), Decimal('140.79'), 'Shipped'), (115, 331, Decimal('12.0203421174294961'), 1, datetime.date(2018, 6, 21), 2, Decimal('122.4225612575127138'), Decimal('15'), Decimal('140.79'), 'Shipped'), (115, 1039, Decimal('14.4244105409153952'), 1, datetime.date(2018, 6, 21), 2, Decimal('122.4225612575127138'), Decimal('15'), Decimal('140.79'), 'Shipped'), (115, 6432, Decimal('12.0203421174294961'), 1, datetime.date(2018, 6, 21), 2, Decimal('122.4225612575127138'), Decimal('15'), Decimal('140.79'), 'Shipped'), (115, 1816, Decimal('10.1710587147480351'), 1, datetime.date(2018, 6, 21), 2, Decimal('122.4225612575127138'), Decimal('15'), Decimal('140.79'), 'Shipped'), (115, 93, Decimal('17.5681923254738789'), 1, datetime.date(2018, 6, 21), 2, Decimal('122.4225612575127138'), Decimal('15'), Decimal('140.79'), 'Shipped'), (111, 125, Decimal('10.1710587147480351'), 1, datetime.date(2016, 3, 1), 2, Decimal('55.4785020804438278'), Decimal('15'), Decimal('63.80'), 'Shipped'), (111, 3331, Decimal('14.79426722145168 74'), 1, datetime.date(2016, 3, 1), 2, Decimal('55.4785020804438278'), Decimal('15'), Decimal('63.80'), 'Shipped'), (111, 2087, Decimal('15.7189089227924179'), 1, datetime.date(2016, 3, 1), 2, Decimal('55.4785020804438278'), Decimal('15'), Decimal('63.80'), 'Shipped'), (115, 3253, Decimal('15.5339805825242718'), 1, datetime.date(2018, 6, 21), 2, Decimal('122.4225612575127138'), Decimal('15'), Decimal('140.79'), 'Shipped'), (115, 3994, Decimal('17.56819232547387 89'), 1, datetime.date(2018, 6, 21), 2, Decimal('122.4225612575127138'), Decimal('15'), Decimal('140.79'), 'Shipped'), (115, 4864, Decimal('10.1710587147480351'), 1, datetime.date(2018, 6, 21), 2, Decimal('122.4225612575127138'), Decimal('15'), Decimal('140.79'), 'Shipped'), (113, 3582, Decimal('17.5681923254738789'), 1, datetime.date(2017, 2, 7), 2, Decimal('131.4840499306518725'), Decimal('15'), Decimal('151.21'), 'Shipped'), (113, 5670, Decimal('14.4244105409153952'), 1, datetime.date(2017, 2, 7), 2, Decimal('131.4840499306518725'), Decimal('15'), Decimal('151.21'), 'Shipped'), (113, 1369, Decimal('14.79426722145168 74'), 1, datetime.date(2017, 2, 7), 2, Decimal('131.4840499306518725'), Decimal('15'), Decimal('151.21'), 'Shipped'), (113, 3397, Decimal('12.9449838187702266'), 1, datetime.date(2017, 2, 7), 2, Decimal('131.4840499306518725'), Decimal('15'), Decimal('151.21'), 'Shipped'), (113, 6038, Decimal('9.2464170134073047'), 1, datetime.date(2017, 2, 7), 2, Decimal('131.4840499306518725'), Decimal('15'), Decimal('151.21'), 'Shipped'), (113, 2802, Decimal('19.9722607489597781'), 1, datetime.date(2017, 2, 7), 2, Decimal('131.4840499306518725'), Decimal('15'), Decimal('151.21'), 'Shipped'), (113, 4436, Decimal('12.0203421174294961'), 1, datetime.date(2017, 2, 7), 2, Decimal('131.4840499306518725'), Decimal('15'), Decimal('151.21'), 'Shipped'), (113, 4146, Decimal('17.5681923254738789'), 1, datetime.date(2017, 2, 7), 2, Decimal('131.4840499306518725'), Decimal('15'), Decimal('151.21'), 'Shipped'), (114, 1645, Decimal('12.2052704576976422'), 1, datetime.date(2018, 5, 30), 2, Decimal('127.6930189551548773'), Decimal('15'), Decimal('146.85'), 'Paid'), (114, 3449, Decimal('11.0957004160887656'), 1, datetime.date(2018, 5, 30), 2, Decimal('127.6930189551548773'), Decimal('15'), Decimal('146.85'), 'Paid'), (114, 6204, Decimal('14.4244105409153952'), 1, datetime.date(2018, 5, 30), 2, Decimal('127.6930189551548773'), Decimal('15'), Decimal('146.85'), 'Paid'), (114, 4581, Decimal('24.9653259361997226'), 1, datetime.date(2018, 5, 30), 2, Decimal('127.6930189551548773'), Decimal('15'), Decimal('146.85'), 'Paid'), (114, 922, Decimal('14.79426722145168 74'), 1, datetime.date(2018, 5, 30), 2, Decimal('127.6930189551548773'), Decimal('15'), Decimal('146.85'), 'Paid'), (114, 5590, Decimal('17.7531206657420250'), 1, datetime.date(2018, 5, 30), 2, Decimal('127.6930189551548773'), Decimal('15'), Decimal('146.85'), 'Paid'), (114, 2055, Decimal('18.0305131761442441'), 1, datetime.date(2018, 5, 30), 2, Decimal('127.6930189551548773'), Decimal('15'), Decimal('146.85'), 'Paid'), (109, 3102, Decimal('11.0957004160887656'), 1, datetime.date(2017, 7, 8), 2, Decimal('68.0536292186777623'), Decimal('15'), Decimal('78.26'), 'Shipped'), (109, 5129, Decimal('14.4244105409153952'), 1, datetime.date(2017, 7, 8), 2, Decimal('68.0536292186777623'), Decimal('15'), Decimal('78.26'), 'Shipped'), (109, 5139, Decimal('9.2464170134073047'), 1, datetime.date(2017, 7, 8), 2, Decimal('68.0536292186777623'), Decimal('15'), Decimal('78.26'), 'Shipped'), (109, 4219, Decimal('16.6435506241331484'), 1, datetime.date(2017, 7, 8), 2, Decimal('68.0536292186777623'), Decimal('15'), Decimal('78.26'), 'Shipped'), (109, 5025, Decimal('16.6435506241331484'), 1, datetime.date(2017, 7, 8), 2, Decimal('68.0536292186777623'), Decimal('15'), Decimal('78.26'), 'Shipped')]
4. Ejecutando COMMIT
5. Ejecución ROLLBACK por excepcion: (psycopg2.errors.ForeignKeyViolation) update o delete en «customers» viola la llave foránea «orders_customerid_fkey» en la tabla «orders» DETAIL: La llave (customerid)=(2) todavia es referida desde la tabla «orders». [SQL: DELETE FROM customers WHERE customerid = 2] (Background on this error at: http://sqlalche.me/e/13/gkpj)
6. Contenido Customers: [(2, 'claim', 'portal', 'bethel scotty 16', 'ieyasu major', 'lanky', 'spryer', '40285', ' Vietnam', 'biddy ', 'claim.portal@jmail.com', '+56 222098611', 'VISA', '4517014291483479', '201203', 'laxity', 'cesar', 53, 35645, 'M')]
7. Contenido OrderDetail: []

l)

bcd) Creamos el siguiente Trigger y comprobamos su correcto funcionamiento con testudpPromo.sql:

```
You, 20 minutes ago | 1 author (You)
ALTER TABLE customers
ADD promo double precision NOT NULL
CONSTRAINT promo_default DEFAULT 0;

CREATE OR REPLACE FUNCTION func_updPromo() RETURNS TRIGGER AS $$
DECLARE
BEGIN
    /*sabemos que el TG_OP no es necesario pero es por si acaso
    cambiamos el trigger a que salte con insert y delete*/
    IF TG_OP = 'UPDATE' THEN
        PERFORM pg_sleep(10);

        UPDATE orderdetail      You, 20 minutes ago • Uncommitted
        SET price = (products.price * (1-NEW.promo/100))
        FROM products
        WHERE
            products.prod_id = orderdetail.prod_id AND
            orderdetail.orderid IN (
                SELECT orderid
                FROM orders
                WHERE
                    orders.customerid = NEW.customerid
                    AND orders.status is NULL);

    END IF;
    RETURN NEW;
END;
$$ LANGUAGE "plpgsql";


DROP TRIGGER IF EXISTS tr_updPromo ON customers;

CREATE TRIGGER tr_updPromo
BEFORE UPDATE ON customers
FOR EACH ROW EXECUTE PROCEDURE func_updPromo();
```

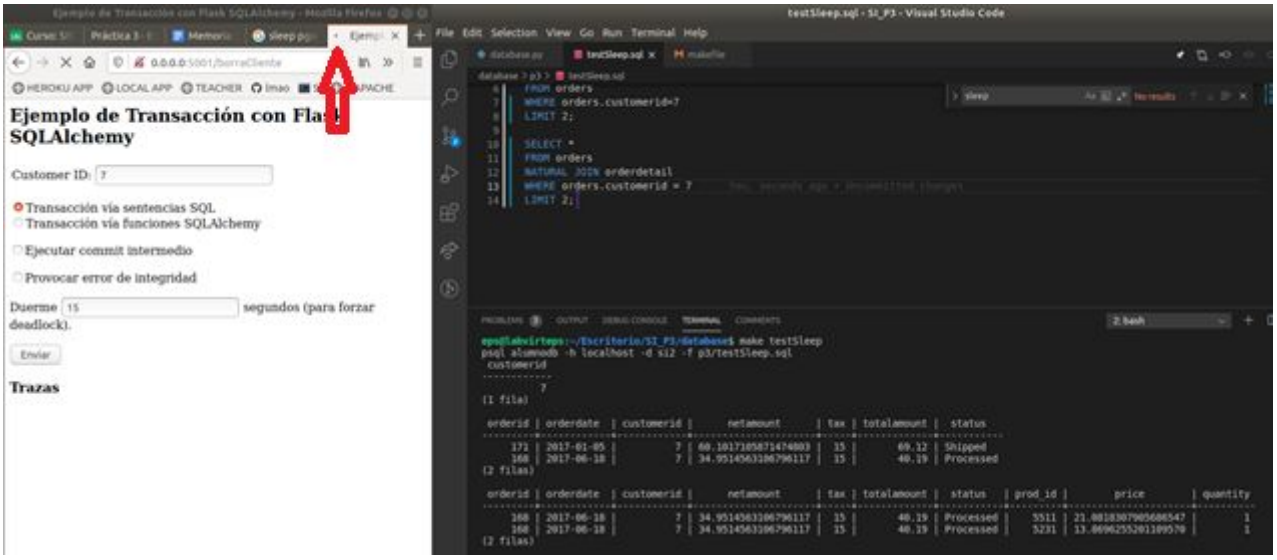
```
psql alumnodb -h localhost -d si2 -f p3/testupdPromo.sql
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1


Precios antes de la promo:
 orderid | prod_id | price
---------+---------+-------
  999999 |       1 |    13
  999999 |       2 |    15
  999999 |       3 |    18
(3 filas)



Promo de 40%
UPDATE 1
 orderid | prod_id | price
---------+---------+-------
  999999 |       1 |   7.8
  999999 |       2 |     9
  999999 |       3 |  10.8
(3 filas)
```

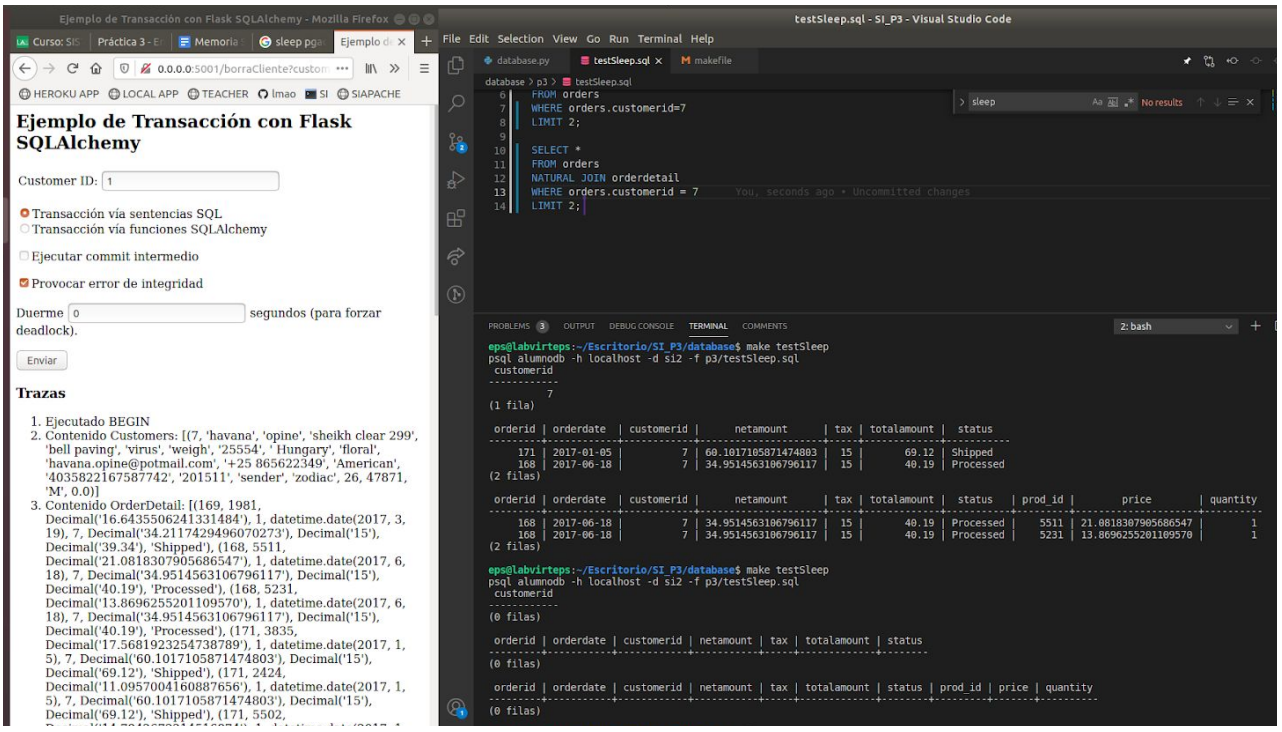
f) El testupdPromo.sql no solo nos sirvió para probar el correcto funcionamiento si no también para hacer los updates de los siguientes ejercicios y forzar el deadlock:

```
INSERT INTO orders
    (orderid, orderdate, customerid,
    netamount, tax, totalamount, status)
VALUES
    (999999, NOW(), 653, 10, 21, 12.1, NULL);
INSERT INTO orderdetail
    (orderid, prod_id, price, quantity)
VALUES
    (999999, 1, 13, 1);
INSERT INTO orderdetail
    (orderid, prod_id, price, quantity)
VALUES
    (999999, 2, 15, 1);
INSERT INTO orderdetail
    (orderid, prod_id, price, quantity)
VALUES
    (999999, 3, 18, 1);
\echo '\n\nPrecios antes de la promo:'
SELECT orderid, prod_id, price
FROM orderdetail
WHERE
    orderid = 999999;
\echo '\n\nPromo de 40%'
UPDATE customers
SET promo = 40
WHERE customerid = 653;
SELECT orderid, prod_id, price
FROM orderdetail
WHERE
    orderid = 999999;
```

h) Mandamos borrar un cliente en la página web y a la vez ejecutamos un script que nos daba la información de ese cliente, como se puede observar mientras la página está cargando (que tarda mucho porque está en el sleep) los contenidos no son actualizados, y una vez ha terminado y ha hecho el commit final se pueden ver los cambios. (La flecha roja apunta a la pestaña para que se observe que está en el sleep (cargando)).

Este comportamiento es debido a que aunque ya se han eliminado partes de los datos como estamos en una transacción la base de datos no ha aceptado los cambios todavía, viene bien por si hay errores durante la ejecución, por ejemplo, en el siguiente apartado provocaremos un *deadlock* pero debido a la gestión hacemos el Rollback cuando lo detectamos.





j) El sleep esta puesto despues de eliminar orderdetail y el order, en un primer lugar habíamos pensado que sería entre el orders y orderdetail, sin embargo después de mucho probar donde ocurría el deadlock se nos bloqueaba en ese punto. Esto puede ser debido a que updPromo modifica orderdetail pero como la página está en medio de una transacción que también lo modifica y se queda en sleep, esto provoca el deadlock como se puede observar en la imagen de abajo.

k) Se puede evitar haciendo rollback como hacemos nosotros, sin embargo no es lo ideal, otras soluciones son acceder de manera similar en todas las consultas y transacciones o haciéndolas más cortas. Se podría solucionar también con alguna medida de software tipo semáforo (contenido de SOPER), que evitase el acceso concurrente al recurso.