Sistemas Informáticos

Prácticas 2020

Práctica 3: NoSQL, Optimización y Transacciones.

Realizado por:

Kevin de la Coba Malam

Marcos Bernuy López

1. Material entregado

En el zip entregado podemos encontrar varias carpetas las cuales tienen la resolución de los ejercicios de la práctica 3:

- **SQL**: Dentro de esta carpeta podemos encontrar *eliminarDeleteCascade.sql*, fichero el cual usamos para cambiar las restricciones de las claves externas de las tablas *orders* y *orderdetail*. En esta carpeta también podemos encontrar *updPromo.sql* el cual utilizamos para crear una nueva columna en nuestra base de datos, concretamente en la tabla *customers*. Además, en ella también creamos un trigger que actualiza el precio de las órdenes de un usuario cada vez que la columna promo se actualiza. El script *clientesDistintos.sql* se utiliza para obtener aquellos clientes que tienen pedido en un mes en concreto y de un precio superior a uno dado. Por último en esta carpeta podemos encontrar el script *countStatus.sql*, donde se muestran las consultas utilizadas y sus resultados, para analizar las distintas queries.

- **App**:
    - **Templates**: tenemos todos los ficheros .html los cuales se emplearán para la interfaz de la página web.
    - **static**: Archivos estáticos usados por la página web.
    - **database.py**, **routes.py**, **__init__.py** y **__main__.py**: Archivos necesarios para el funcionamiento de la página web.
    - **createMongoDBFromPostgreSQLDB.py**: Archivo python donde se ejecuta un script para crear una base de datos mongodb dada una base de datos PostgreSQL. En concreto obtiene las 800 películas más actuales de Estados Unidos y las inserta en una nueva base de datos de mongodb.

- **Transacciones**: Carpeta que donde se resuelve parte del ejercicio 3, las transacciones. Esta contiene el esqueleto proporcionado en los ficheros auxiliares.

2. Resultados

a. NoSQL

En el primer ejercicio tuvimos que crear una base de datos mongodb, dado el resultado de una query en la base de datos PostgreSQL.

La consulta que realizamos fue la siguiente:

```
select imdb_movies.movieid, movietitle, year
                                    from imdb_movies,
imdb_moviecountries
                                    where imdb_movies.movieid =
imdb_moviecountries.movieid
                                    and imdb_moviecountries.country =
'USA'
                                    order by imdb_movies.year
                                    desc
                                    limit 800
```

El resultado se guarda en una lista y pasamos a crear diccionarios en base a el resultado, estos diccionarios se introducen en una lista la cual es usada por mongodb. En resumen, creamos una lista de diccionarios en base al resultado de la consulta anterior.

No todos los datos que debemos introducir en la base de datos mongo están en el resultado, hay otros que debemos crear nosotros mismos como las películas **más relacionadas** y las **relacionadas**.

Para crear la lista de películas más relacionadas, dada una película ‘x’, iteramos sobre las 800 películas ya obtenidas, hacemos una consulta en cada una de ellas (llamaremos a estas ‘y’) para obtener los géneros, por último comparamos si los géneros de la película ‘x’ con los géneros de la película ‘y’, si los todos géneros de ‘x’ están en ‘y’, se añade ‘y’ a la lista de películas más relacionadas con ‘x’.

Ahora bien, para las películas relacionadas, comprobamos uno a uno los géneros de la película ‘y’, si el género a comprobar está en ‘x’, consideramos esto como un “match”. Una vez hemos iterado por todos los géneros de ‘y’ hacemos la siguiente operación: (número de géneros de ‘x’) / (matches), si el resultado de la operación está entre 1,5 y 2,

consideraremos a la película ‘y’ como una película relacionada y la añadiremos a la lista de películas relacionadas con ‘x’.

Pasamos ahora a las consultas de mongodb.

```
consulta = {'$and': [] }
    nombre_pelicula = request.form['nombre_pelicula']
    if nombre_pelicula != '':
        consulta['$and'].append({'title': {'$regex': '.*'+nombre_pelicula+'.*'}})
    nombre_actor = request.form['nombre_actor']
    if nombre_actor != '':
        consulta['$and'].append({'actors': {'$regex': '.*'+nombre_actor+'.*'}})
    nombre_director = request.form['nombre_director']
    if nombre_director != '':
        consulta['$and'].append({'directors': {'$regex': '.*'+nombre_director+'.*'}})
    ano_pelicula = request.form['ano_pelicula']
    if ano_pelicula != '':
        consulta['$and'].append({'year': {'$regex': '.*'+ano_pelicula+'.*'}})
    genero_pelicula = request.form['genero_pelicula']
    if genero_pelicula != '':
        consulta['$and'].append({'genres': {'$regex': '.*'+genero_pelicula+'.*'}})
    search_result = mycol.find(consulta)
```

En la página topUSA hemos creado un buscador el cual, dados unos ciertos parametros, ejecuta una query en la base de datos mongodb. Simplemente lo que hacemos es un método POST que envíe estos parámetros, desde el archivo routes.py recogemos esos parámetros y construimos la consulta en formato mongodb, como son varios parámetros añadimos un '$and' al principio, de esta manera especificamos que todos los parámetros que vayamos a introducir deben ser cumplidos. Mientras comprobamos si se han introducido los parámetros hacemos append a la lista que contiene los filtros de la query. Una vez hecha la query la ejecutamos y este es el resultado:

Buscamos todas las películas que tengan en el título “Life” y tengan como género “Comedy”:

Título	Género	Año	Directores	Actores	Películas más relacionadas	Películas relacionadas
After Life (2005)	Comedy Short	2005	Johnson, Bramley	Burns, Bruce (IV) Mantell, Jim		Broken Hearts Club (2006) Big Bang Theory (2006) Heavyweights (2005) Visitors, The (2005) Truce, The (2001) Next Best Thing, The (2000) Flintstones in Viva Rock Vegas, The (2000) Down to You (2000) Opportunists, The (2000) Small Time Crooks (2000)
Life (1999/I)	Comedy Drama	1999	Demme, Ted	Emmerich, Noah Emory Jr., James Ermey, R. Lee Evans, Nate (I) Everett, Todd (I)	Broken Hearts Club (2006) Next Best Thing, The (2000) Down to You (2000) Opportunists, The (2000) Kid, The (2000) Beautiful (2000) Coyote Ugly (2000) Whatever It Takes (2000) Nurse Betty (2000) Where the Money Is (2000)	Night Fliers (2006) Big Bang Theory (2006) After Life (2005) Living Dead Girl (2003) Frequency (2000) Here on Earth (2000) Crime and Punishment in Suburbia (2000) Flintstones in Viva Rock Vegas, The (2000) Crow: Salvation, The (2000) Passion of Mind (2000)
My Life So Far (1999)	Biography Comedy Drama	1999	Hudson, Hugh	Firth, Colin Forrest, Stewart Gleeson, Brendan Anderson, Ross (III) Baird, Daniel	Isn't She Great (2000) Man on the Moon (1999) Permanent Midnight (1998)	Broken Hearts Club (2006) Next Best Thing, The (2000) Down to You (2000) Opportunists, The (2000) Kid, The (2000) Beautiful (2000) Coyote Ugly (2000) Whatever It Takes (2000) Nurse Betty (2000) Where the Money Is (2000)
						Flintstones in Viva Rock Vegas, The (2000)

Este es el resultado (no se ven todas las películas en esta captura).

Se nos pide también que mostremos en la web 3 tablas con los resultados de unas queries:

- *Comedias de 1997 con "life" en el título.*

  Para esta consulta creamos el siguiente código:

```
mycol.find({'$and':
                    [
                      {"year":'1997'},
                      {"genres":{'$elemMatch': {'$regex' : ".*Comedy.*"}}},
                      {"title": {'$regex': ".*Life.*"}}
                    ]
                  })
```

  Con "$and" estamos indicando que todas las condiciones que se muestren a continuación deben ser cumplidas. Con "$elemMatch" indicamos que filtre las películas que tengan el género *comedy*. Por último con "$regex" indicamos que se cumpla la expresión regular.

Comedias de 1997 con "Life" en el título

Título	Género	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	Broken Hearts Club (2006) Big Bang Theory (2006) After Life (2005) Next Best Thing, The (2000) Flintstones in Viva Rock Vegas, The (2000) Down to You (2000) Opportunists, The (2000) Small Time Crooks (2000) Kid, The (2000) Beautiful (2000)	
Life Less Ordinary, A (1997)	Comedy Crime Drama Fantasy Romance	1997	Boyle, Danny	Gorham, Christopher Gowdy, Chuck Hedaya, Dan Holm, Ian Chaykin, Maury		Broken Hearts Club (2006) Next Best Thing, The (2000) Down to You (2000) Opportunists, The (2000) Kid, The (2000) Coyote Ugly (2000) Whatever It Takes (2000) Where the Money Is (2000) Boys and Girls (2000) Hamlet (2000)

- *Películas dirigidas por Woody Allen en los 90*.

  El código es el siguiente:

```
mycol.find({'$and':
                    [
                      {'year': {'$lt':'2000'}},
                      {'year':{'$gt': '1989'}},
                      {'directors': {'$elemMatch': {'$regex': '.*Woody.*'}}},
                      {'directors': {'$elemMatch': {'$regex': '.*Allen.*'}}}
                    ]
                  })
```

  Podemos ver que hay nuevas expresiones como “$lt” que significa “less than” (menor que) y “$gt” que significa “greater than” (mayor que). Con estas dos expresiones filtramos el año de las películas, siendo mayor que 1989 y menor que 2000.

Películas dirigidas por Woody Allen en los 90

Título	Género	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	Coyote Ugly (2000) High Fidelity (2000) Duets (2000) Slaves to the Underground (1997)	Broken Hearts Club (2006) Next Best Thing, The (2000) Down to You (2000) Opportunists, The (2000) Kid, The (2000) Beautiful (2000) Whatever It Takes (2000) Nurse Betty (2000) Where the Money Is (2000) Wonder Boys (2000)
Celebrity (1998)	Comedy	1998	Allen, Woody	Dark, Peter Darrow, Tony DiCaprio, Leonardo Doumanian, John Edelson, Kenneth	Broken Hearts Club (2006) Big Bang Theory (2006) After Life (2005) Next Best Thing, The (2000) Flintstones in Viva Rock Vegas, The (2000) Down to You (2000) Opportunists, The (2000) Small Time Crooks (2000) Kid, The (2000) Beautiful (2000)	
Deconstructing Harry (1997)	Comedy	1997	Allen, Woody	Darrow, Tony Doumanian, John Edelson, Kenneth Frazer, Dan Garvey, Ray	Broken Hearts Club (2006) Big Bang Theory (2006) After Life (2005) Next Best Thing, The (2000) Flintstones in Viva Rock Vegas, The (2000) Down to You (2000) Opportunists, The (2000) Small Time Crooks (2000) Kid, The (2000) Beautiful (2000)	

- *Películas en las que Johnny Galecki y Jim Parsons compartan reparto.*

El código de la consulta es el siguiente:

```
mycol.find({"$and":
                    [
                      {"actors": {'$elemMatch': {'$regex': ".*Parsons, Jim.*"}}},
                      {"actors": {'$elemMatch': {'$regex': ".*Galecki,
Johnny.*"}}}
                    ]
                  })
```

Esta es la consulta más simple de todas, solo ponemos 2 filtros en los que decimos que hayan actores que contengan el nombre “Parsons, Jim” y “Galecki, Johnny”.

Películas en las que Johnny Galecki y Jim Parsons compartan reparto

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

b. Optimización

En primer lugar se nos pide una consulta SQL, debemos mostrar el número de clientes los cuales tienen pedidos que en un cierto mes y año, superan un cierto umbral. La consulta es la siguiente:

```
SELECT COUNT(a.customerid) FROM customers AS a, orders as b
WHERE a.customerid = b.customerid AND EXTRACT(YEAR FROM b.orderdate) =
2015 AND EXTRACT(MONTH FROM b.orderdate) = 4 AND b.totalamount > 100
```

En nuestro caso pusimos el mes de abril de 2015 y una cantidad mínima de 100.

Al ejecutar **EXPLAIN** … de nuestra consulta obtuvimos la siguiente salida:

```
Finalize Aggregate  (cost=5636.14..5636.15 rows=1 width=8)
  ->  Gather  (cost=5636.03..5636.14 rows=1 width=8)
        Workers Planned: 1
        ->  Partial Aggregate  (cost=4636.03..4636.04 rows=1 width=8)
              ->  Nested Loop  (cost=0.29..4636.02 rows=1 width=4)
                    ->  Parallel Seq Scan on orders b
(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 preci (...)
                    ->  Index Only Scan using customers_pkey on
customers a  (cost=0.29..8.30 rows=1 width=4)
                          Index Cond: (customerid = b.customerid)
```

Podemos apreciar que la consulta tiene un coste 5636.14, y que la ejecución de la consulta se basa en recorrer mediante un nested loop de manera paralela la tabla *orders*, a la vez se aplica un filtro y se escanean las primary keys de *customers*. Podemos ver que a las primary key de *customers* se las trata como índices, esto es debido a que las primary keys son índices implícitos.

Se nos pide ahora crear índices, nosotros decidimos crearlos en las columnas *totalamout* y *orderdate*, ¿Por Qué? Porque son las que utilizamos y *customerid* ya es un índice.

Primero lo aplicamos en *orderdate*:

```
-- Creamos un index en orderdate
CREATE INDEX index_orderdate ON orders(orderdate)
```

Al hacer EXPLAIN … obtuvimos lo siguiente:

```
Finalize Aggregate  (cost=5636.14..5636.15 rows=1 width=8)
  ->  Gather  (cost=5636.03..5636.14 rows=1 width=8)
        Workers Planned: 1
        ->  Partial Aggregate  (cost=4636.03..4636.04 rows=1 width=8)
              ->  Nested Loop  (cost=0.29..4636.02 rows=1 width=4)
                    ->  Parallel Seq Scan on orders b
(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 preci (...)
                    ->  Index Only Scan using customers_pkey on
customers a  (cost=0.29..8.30 rows=1 width=4)
                          Index Cond: (customerid = b.customerid)
```

Al parecer el plan de ejecución y el coste son los mismos que el anterior. Podemos concluir que el índice no es relevante.

Por último borramos el índice en *orderdate* y lo creamos en *totalamount*:

```
DROP INDEX index_orderdate
-- Creamos un indice en el total amount
CREATE INDEX index_totalamount ON orders(totalamount)
```

Tras ejecutar EXPLAIN … obtuvimos lo siguiente:

```
Aggregate  (cost=4496.93..4496.94 rows=1 width=8)
  ->  Nested Loop  (cost=1127.18..4496.92 rows=2 width=4)
```

```
        ->  Bitmap Heap Scan on orders b  (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 index_totalamount
(cost=0.00..1126.90 rows=60597 width=0)
                    Index Cond: (totalamount > '100'::numeric)
        ->  Index Only Scan using customers_pkey on customers a
(cost=0.29..8.30 rows=1 width=4)
              Index Cond: (customerid = b.customerid)
```

Podemos ver que el coste se ha reducido, y el plan de ejecución ha cambiado. Se ha creado un bitmap sobre el *totalamount*.

Posteriormente se nos hicieron las siguientes preguntas:

¿Qué consulta devuelve algún resultado nada más comenzar su ejecución? Todas las consultas del anexo devuelven una respuesta rápida (en menos de 160 mseg).

¿Qué consulta se puede beneficiar de la ejecución en paralelo? Todas las consultas se benefician de la ejecución en paralelo, ya que si eliminamos la ejecución paralela, tarda aproximadamente 32 veces más (3.2 segundos)

```
SELECT a.customerid
            FROM customers AS a, orders AS b
            WHERE b.status <> 'Paid' AND a.customerid = b.customerid
```

*Consulta sin paralelizar

Al final se nos pide analizar 2 consultas ubicadas en el anexo 2 del enunciado de la práctica.

En primer lugar hacemos **EXPLAIN** … de la consulta y obtuvimos lo siguiente:

```
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)
```

Ahora creamos un índice en status:

```
CREATE INDEX index_status ON orders(status)
```

Volvemos a hacer el `EXPLAIN …` de la consulta:

```
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 index_status  (cost=0.00..19.24
rows=909 width=0)
              Index Cond: (status IS NULL)
```

Podemos ver que tras crear el índice el coste se ha reducido más de la mitad y en vez de hacer un escaneo secuencial, se recorre un bitmap heap.

Por último usamos `ANALYZE orders`, esto analiza la tabla para poder hacer consultas sobre esta de la manera más eficiente.

Ejecutamos `EXPLAIN …`:

```
Aggregate  (cost=7.33..7.34 rows=1 width=8)
  ->  Index Only Scan using index_status on orders  (cost=0.42..7.32
rows=1 width=0)
        Index Cond: (status IS NULL)
```

El coste se ha reducido 500 veces en comparación con la consulta original, y 200 veces sin usar `ANALYZE`, por tanto podemos concluir que el efecto que analize hace en las consultas es más que notable.

c. Transacciones

Este es el último apartado de la práctica.

En primer lugar debemos usar el esqueleto proporcionado, y terminar la función *borraCliente*. Esta función lo que hace borrar un customer de la base de datos, dependiendo de las condiciones que se introduzcan en la página, condiciones como “debe dar error”, “se debe hacer un commit después del begin”.

Pero antes de nada, **¿Qué es una transacción?** “*El punto esencial de una transacción es el de que convertimos varios pasos, en una operación del tipo, todo o nada. Los pasos intermedios no son visibles para otras transacciones concurrentes, y si ocurre*

algún error que impida a la transacción acabar, entonces ninguno de los pasos anteriores afectarán a la base de datos" PostgreSQL: Documentation: 13: 3.4. Transactions. https://www.postgresql.org/docs/current/tutorial-transactions.html.

La ejecución de la función sigue los siguientes pasos:

```
del_1 = "DELETE FROM orderdetail\
                USING orders AS a\
                WHERE a.customerid = "+str(customerid)+" AND a.orderid
= orderdetail.orderid"
    del_2 = "DELETE FROM orders\
                WHERE customerid = "+str(customerid)+""
    del_3 = "DELETE FROM customers WHERE customerid =
"+str(customerid)+""
```

Lo primero que hacemos es dividir las consultas en 3, primero los *orderdetails*, luego las *orders* orders y por último el customer.

```
# Creamos una lista para poder iterar sobre las consultas
    if bFallo:
        lista = [del_2, del_3, del_1]
    else:
        lista = [del_1, del_2, del_3]
```

La ejecución de la transacción se termina cuando se han ejecutado las 3 partes, por lo tanto para ejecutar las tres partes las añadimos a una lista e iteramos sobre esta ejecutando dichas partes. Podemos ver que dependiendo de si debe fallar o no, el orden de las consultas es uno u otro.

```
    for del_ in lista:
      i+=1


      # Ejecutamos begin
      dbr.append("[info] Begin: en ejecución...")
      if bSQL:
        db_conn.execute("BEGIN")
      else:
        session.begin(subtransactions=True)
      dbr.append("[info] Begin: ¡Ejecutado!")
```

Comenzamos el loop y empezamos ejecutando BEGIN. Con esta sentencia marcamos el inicio de una transacción. Podemos ver que dependiendo de si queremos usar sentencias en SQL o funciones de SQLAlchemy seguimos un camino u otro.

```
# Commit intermedio
if bCommit:
    dbr.append("[info] Commit (intermedio): en ejecución...")
    if bSQL:
        db_conn.execute("COMMIT")
    else:
        session.commit()
    dbr.append("[info] Commit (intermedio): ¡Ejecutado!")
```

Aquí tenemos el primer COMMIT. **¿Qué hace un COMMIT?** Al ejecutar un COMMIT en una base de datos lo que estamos haciendo es que los cambios hechos se hagan visibles para otras transacciones.

```
# Hacemos commit intermedio en caso de fallo
if bFallo:
    dbr.append("[info] Commit (fallo): en ejecución...")
    if bSQL:
        db_conn.execute("COMMIT")
    else:
        session.commit()
    dbr.append("[info] Commit (fallo): ¡Ejecutado!")
```

En el caso de que se quisiera forzar un fallo en la base de datos, ejecutamos un segundo COMMIT intermedio.

```
# Ejecutamos la consulta
dbr.append("[info] Consulta "+str(i)+" de 3: en ejecución...")
if bSQL:
    db_conn.execute(del_)
else:
    session.execute(del_)
# Sleep justo después de la ejecución
time.sleep(duerme)
dbr.append("[info] Consulta "+str(i)+" de 3: ¡Ejecutada!")
```

Llegamos al punto donde se ejecuta el borrado correspondiente, simplemente se hace o con sentencias SQL o usando funcionalidades de SQLAlchemy.

Podemos ver también que hay un sleep, este sleep se usa posteriormente para comprobar los bloqueos entre transacciones.

```
        dbr.append("[info] Commit: en ejecución...")
        if bSQL:
            db_conn.execute("COMMIT")
        else:
            session.commit()
        dbr.append("[info] Commit: ¡Ejecutado!")
```

Por último se ejecutaría un último COMMIT tras haber ejecutado el borrado. Aquí solo se llegaría en caso de que el borrado fuese un borrado de éxito.

```
    except Exception as e:
        print("Exception in DB access:")
        print("-"*60)
        traceback.print_exc(file=sys.stderr)
        print("-"*60)
```

En caso de que el borrado fuese incorrecto llegaríamos a este punto del código. Al intentar ejecutar el borrado se produciría una excepción debido a que estamos borrando en un orden incorrecto y provocaría problemas con las foreign keys.

```
        dbr.append("[info] Error")
        dbr.append("[info] Error - Rollback: en ejecución...")
        # Hacemos rollback
        if bSQL:
            db_conn.execute("ROLLBACK")
        else:
            session.rollback()
        dbr.append("[info] Error - Rollback: ¡Ejecutado!")
```

Notificamos el error y hacemos un ROLLBACK al punto inicial antes de haber ejecutado cualquier sentencia de la transacción.

```
        # Finalizando la transaccion
        dbr.append("[info] Error - END: en ejecución...")
        if bSQL:
```

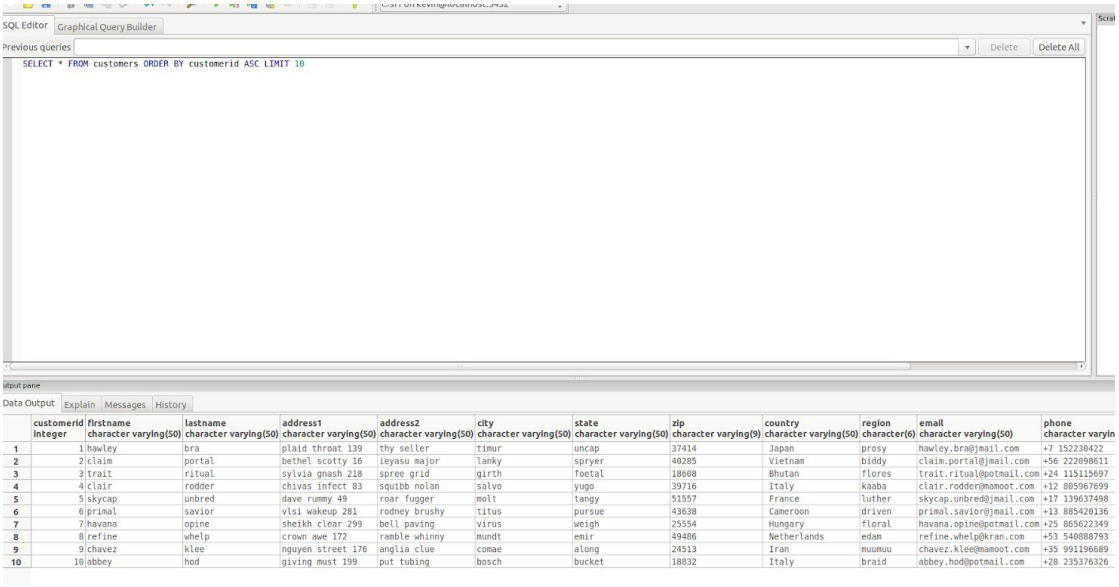
```
        db_conn.execute("END")
    else:
        session.execute("END")
    dbr.append("[info] Error - END: ¡Ejecutado!")
    # Desconexión de la base de datos y sesión sqlalchemy
    dbCloseConnect(db_conn)
    session.close()
```

Finalizamos el código ejecutando la sentencia END, para así marcar el fin de nuestra transacción.

Este sería un ejemplo de ejecución:

Vamos a borrar el customer 1, pero antes vamos a provocar un fallo durante el borrado.

Antes de nada comprobemos la base de datos.



	customerid integer	firstname character varying(50)	lastname character varying(50)	address1 character varying(50)	address2 character varying(50)	city character varying(50)	state character varying(50)	zip character varying(9)	country character varying(50)	region character(6)	email character varying(50)	phone character varyin
1	1	hawley	bra	plaid throat 139	thy seller	timur	uncap	37414	Japan	prosy	hawley.bra@jmail.com	+7 152230422
2	2	claim	portal	bethel scotty 16	ieyasu major	lanky	spryer	40285	Vietnam	biddy	claim.portal@jmail.com	+56 222098611
3	3	trait	ritual	sylvia gnash 218	spree grid	girth	foetal	18608	Bhutan	flores	trait.ritual@potmail.com	+24 115115697
4	4	clair	rodder	chivas infect 83	squibb nolan	salvo	yugo	39716	Italy	kaaba	clair.rodder@mamoot.com	+12 805967699
5	5	skycap	unbred	dave rummy 49	roar fugger	molt	tangy	51557	France	luther	skycap.unbred@jmail.com	+17 139637498
6	6	primal	savior	vlsi wakeup 281	rodney brushy	titus	pursue	43638	Cameroon	driven	primal.savior@jmail.com	+13 885420136
7	7	havana	opine	sheikh clear 299	bell paving	virus	weigh	25554	Hungary	floral	havana.opine@potmail.com	+25 865622349
8	8	refine	whelp	crown awe 172	ramble whinny	mundt	emir	49486	Netherlands	edam	refine.whelp@kran.com	+53 540886793
9	9	chavez	klee	nguyen street 176	anglia clue	comae	along	24513	Iran	muumuu	chavez.klee@mamoot.com	+35 991196689
10	10	abbey	hod	giving must 199	put tubing	bosch	bucket	18832	Italy	braid	abbey.hod@potmail.com	+28 235376326

Si nos fijamos podemos ver que el customer 1 está.

```
SELECT * FROM customers ORDER BY customerid ASC LIMIT 10

SELECT * FROM orders WHERE customerid = 1
```

Output pane

Data Output | Explain | Messages | History

	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying(10)
1	108	2019-01-31	1	41.6088765603328709	15	47.85	Shipped
2	107	2015-10-29	1	130.7443365695792881	15	150.36	Shipped
3	104	2018-03-07	1	26.6296809986130374	15	30.62	Shipped
4	105	2015-11-30	1	12.2052704576976422	15	14.04	Processed
5	106	2018-12-31	1	26.8146093388811836	15	30.84	Shipped
6	103	2015-12-29	1	25.1502542764678688	15	28.92	Shipped

Comprobamos las *orders* del customer 1.

```
SELECT * FROM customers ORDER BY customerid ASC LIMIT 10

SELECT * FROM orders WHERE customerid = 1

SELECT a.* FROM orderdetail AS a, orders AS b WHERE a.orderid = b.orderid AND b.customerid = 1
```

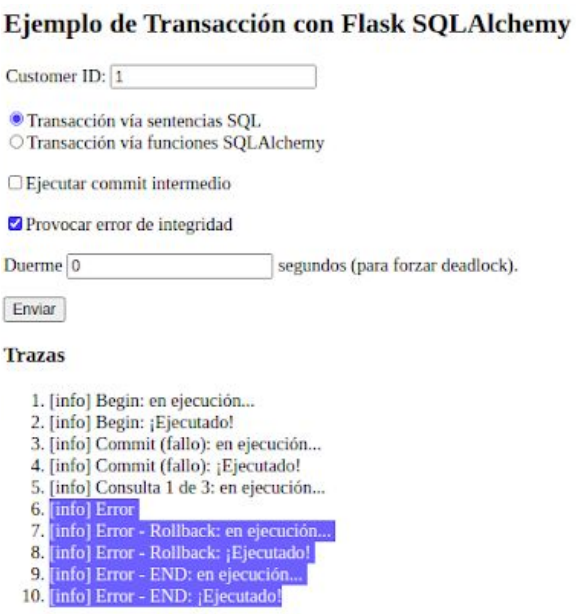
Output pane

Data Output | Explain | Messages | History

	orderid integer	prod_id integer	price numeric	quantity integer
1	108	1256	14.7942672214516874	1
2	108	6125	10.1710587147480351	1
3	108	5873	16.6435506241331484	1
4	105	1609	12.2052704576976422	1
5	107	2648	16.6435506241331484	1
6	107	1204	17.5681923254738789	1
7	107	5825	15.7189089227924179	1
8	107	6422	26.3522884882108183	1
9	107	5066	12.0203421174294961	1
10	107	6088	18.0305131761442441	1
11	107	2597	13.3148404993065188	1
12	107	448	11.0957004160887656	1
13	103	3500	12.2052704576976422	1
14	103	911	12.9449838187702266	1
15	106	4669	9.2464170134073047	1
16	106	724	17.5681923254738789	1
17	104	2105	15.5339805825242718	1
18	104	4709	11.0957004160887656	1

Por último comprobamos los *orderdeatils* del customer 1.

Ahora pasaremos a ejecutar el borrado erróneo.



Una vez ejecutado, podemos ver en la traza que se produce un error mientras se ejecuta el borrado.

Pasamos ahora a comprobar si se ha borrado algo en la base de datos. Solo nos haría falta comprobar las *orders* ya que estas son las que se intentan borrar primero.

```
SELECT * FROM customers ORDER BY customerid ASC LIMIT 10

SELECT * FROM orders WHERE customerid = 1

SELECT a.* FROM orderdetail AS a, orders AS b WHERE a.orderid = b.orderid AND b.customerid = 1
```

tput pane

ata Output | Explain | Messages | History

	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying(10)
1	108	2019-01-31	1	41.6088765603328709	15	47.85	Shipped
2	107	2015-10-29	1	130.7443365695792881	15	150.36	Shipped
3	104	2018-03-07	1	26.6296809986130374	15	30.62	Shipped
4	105	2015-11-30	1	12.2052704576976422	15	14.04	Processed
5	106	2018-12-31	1	26.8146093388811836	15	30.84	Shipped
6	103	2015-12-29	1	25.1502542764678688	15	28.92	Shipped

Vemos que las orders están ahí todas, no ha habido ningún cambio.

Ahora pasamos a hacer una ejecución correcta.

Ejemplo de Transacción con Flask SQLAlchemy

Customer ID: 1

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

[ ] Ejecutar commit intermedio

[x] Provocar error de integridad

Duerme 0 segundos (para forzar deadlock).

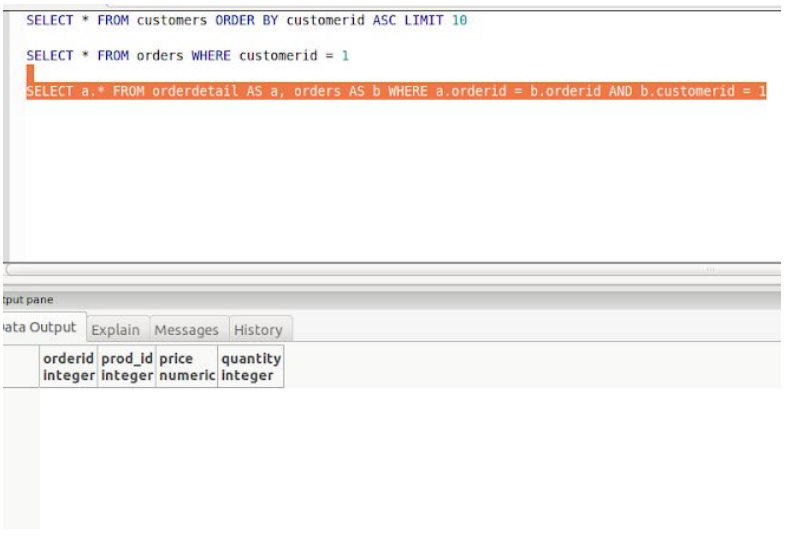
Enviar

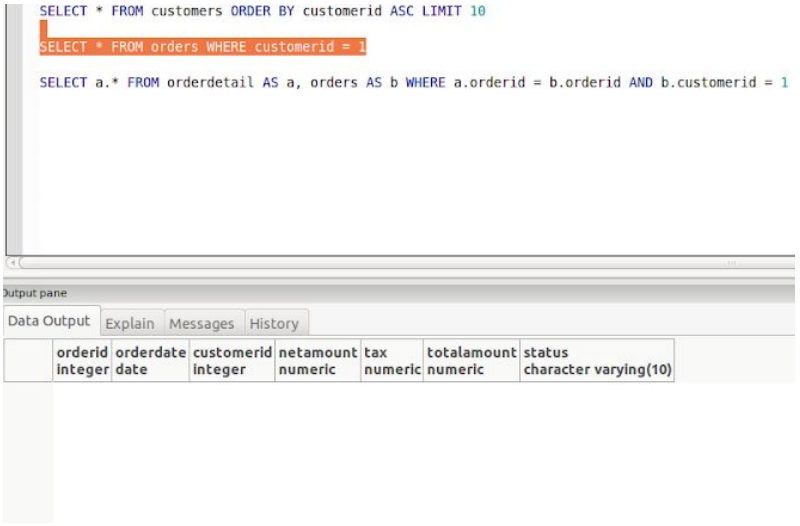
Trazas

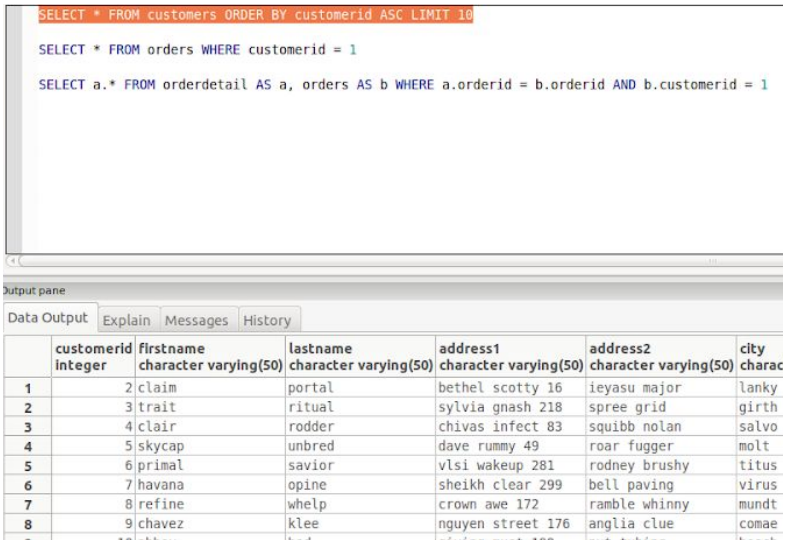
1. [info] Begin: en ejecución...
2. [info] Begin: ¡Ejecutado!
3. [info] Consulta 1 de 3: en ejecución...
4. [info] Consulta 1 de 3: ¡Ejecutada!
5. [info] Commit: en ejecución...
6. [info] Commit: ¡Ejecutado!
7. [info] Begin: en ejecución...
8. [info] Begin: ¡Ejecutado!
9. [info] Consulta 2 de 3: en ejecución...
10. [info] Consulta 2 de 3: ¡Ejecutada!
11. [info] Commit: en ejecución...
12. [info] Commit: ¡Ejecutado!
13. [info] Begin: en ejecución...
14. [info] Begin: ¡Ejecutado!
15. [info] Consulta 3 de 3: en ejecución...
16. [info] Consulta 3 de 3: ¡Ejecutada!
17. [info] Commit: en ejecución...
18. [info] Commit: ¡Ejecutado!
19. [info] Ejecución correcta.
20. [info] Finalizando transaccion.
21. [info] END: en ejecución...
22. [info] END: ¡Ejecutado!

Podemos ver en la traza que en ningún momento se produce un error.

Pasamos a comprobar la base de datos.







	customerid integer	firstname character varying(50)	lastname character varying(50)	address1 character varying(50)	address2 character varying(50)	city charac
1	2	claim	portal	bethel scotty 16	ieyasu major	lanky
2	3	trait	ritual	sylvia gnash 218	spree grid	girth
3	4	clair	rodder	chivas infect 83	squibb nolan	salvo
4	5	skycap	unbred	dave rummy 49	roar fugger	molt
5	6	primal	savior	vlsi wakeup 281	rodney brushy	titus
6	7	havana	opine	sheikh clear 299	bell paving	virus
7	8	refine	whelp	crown awe 172	ramble whinny	mundt
8	9	chavez	klee	nguyen street 176	anglia clue	comae

Podemos ver que al ejecutar las consultas, las *orders* y las *orderdetail* están vacías. En la última consulta vemos que el customer 1 ya no existe en la base de datos.

Se nos pide ahora, primero crear una columna en *customers* (promo), después hacer un trigger el cual cuando esta columna sea modificada, cambie los valores de las *orders* y los *orderdetails* del usuario modificado.

```
-- Modificamos la tabla customers
ALTER TABLE customers ADD COLUMN promo DECIMAL(4,2);

-- Creamos el trigger
DROP TRIGGER IF EXISTS updPromo ON customers;

CREATE OR REPLACE FUNCTION updPromo()
RETURNS TRIGGER AS
$$
```

```
        BEGIN
         IF(TG_OP = 'UPDATE') THEN
                                -- Actualizamos los orderdetails
                                UPDATE orderdetail AS b
                                SET price = price - price * (NEW.promo/100)
                                FROM orders AS a
                                WHERE a.customerid = NEW.customerid AND
a.orderid = b.orderid AND a.status IS NULL;

                                -- Actualizamos las orders
                                UPDATE orders AS a
                                       SET netamount = t.t_price,
                                              totalamount = t.t_price +
(a.tax/100) * t.t_price
                                       FROM   (
                                              -- Precio total tras aplicar
el descuento después de
                                              SELECT c.orderid AS t_id,
SUM(c.price) AS t_price
                                              FROM orderdetail AS c, orders
AS d
                                              WHERE c.orderid = d.orderid
AND d.customerid = NEW.customerid AND d.status IS NULL
                                              GROUP BY c.orderid
                                              ) AS t
                                       WHERE a.customerid = NEW.customerid
AND a.status is NULL AND t.t_id = a.orderid;

                                -- Sleep tras haber modificado los datos
                                PERFORM pg_sleep(20);

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

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

*Consultar código en el código fuente, se podrá ver mejor.

Lo que básicamente hacemos en el trigger es que cuando la columna promo se actualice en un customer, cogemos la promoción y la descontamos de todos los *orderdetails* y por último calculamos de nuevo el *totalamount* y *netamount* de *orders*.

Para el estudio de los bloqueos se ha puesto un sleep de 20 segundos justo después de actualizar los valores.

Para consultar los bloqueos en pgadmin 3 ejecutamos la siguiente consulta:

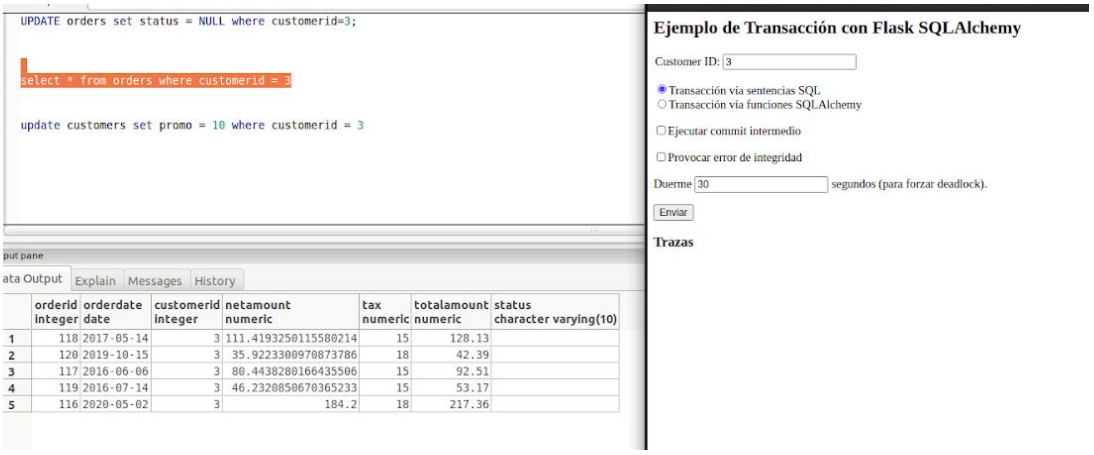
```
select relation::regclass, * from pg_locks where not granted;
```

Con esta consulta vemos los bloqueos de las transacciones.

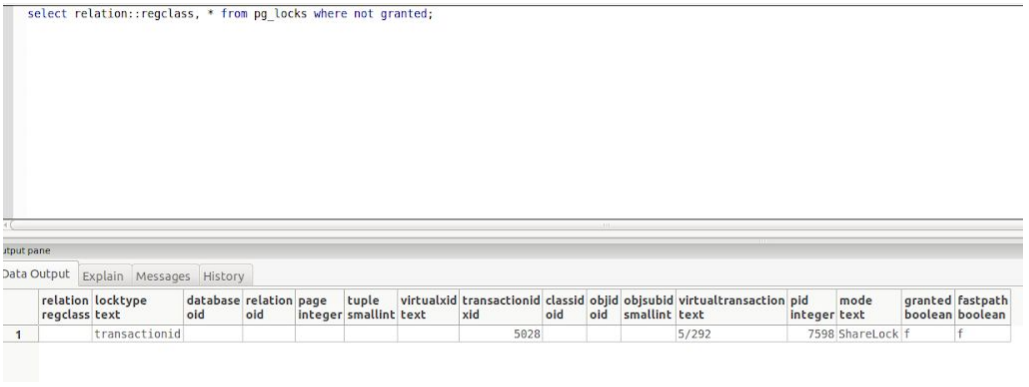
Pasemos ahora a generar un bloqueo. Para generar un bloqueo lo que haremos será:

1. Crear varios carritos.
2. Borrar un customer con los carritos borrados pero hacer un sleep durante el borrado.
3. Actualizar la columna *promo* del customer que se está eliminando.
4. Por último consultar los bloqueos.

Una vez los pasos 1, 2, 3 se han hecho pasamos a comprobar los bloqueos de la base de datos.



	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying(10)
1	118	2017-05-14	3	111.4193250115580214	15	128.13	
2	120	2019-10-15	3	35.9223300970873786	18	42.39	
3	117	2016-06-06	3	80.4438280166435506	15	92.51	
4	119	2016-07-14	3	46.2320850670365233	15	53.17	
5	116	2020-05-02	3	184.2	18	217.36	



	relation regclass	locktype text	database oid	relation oid	page integer	tuple smallint	virtualxid text	transactionid xid	classid oid	objid oid	objsubid smallint	virtualtransaction text	pid integer	mode text	granted boolean	fastpath boolean
1		transactionid						5028				5/292	7598	ShareLock	f	f

Al ejecutar la consulta para ver bloqueos podemos ver que hay uno, el update está esperando a que la transacción que borre finalice para poder ejecutarse.

Este es un bloqueo pero, **¿es un deadlock?** No, el update depende de que la transacción finalice pero la transacción no depende de nadie. Si la transacción dependiera del update entonces sí podríamos decir que es un deadlock.