MEMORIA

PRÁCTICA 4:
Optimización, Transacciones y Seguridad

AUTORES:

Sergio Teso Lorenzo
Alberto Estepa Fernández

1. Optimización

A) Estudio del impacto de un índice

Sabemos que un índice requiere su propio espacio en disco y contiene una copia de los datos de la tabla. Eso significa que un índice es una redundancia. Pero puede ser muy recomendable utilizarlo en la resoluciones de consultas para que se optimice el tiempo de resolución.

Para estudiar el caso de la mejora al utilizar un índice en una consulta primero hemos preparado la consulta pedida, *clientesDistintos.sql (*que muestre el número de clientes distintos que tienen pedidos en un mes dado, por ejemplo 201404, con importe (totalamount) superior a un umbral dado, por ejemplo 100).

La hemos preparado en una función donde le pasamos como argumento el umbral del totalamount que hay que superar y el mes dado. Quedaría de la siguiente forma:

```
-----------------------------------
DROP FUNCTION IF EXISTS clientesDistintos(varchar,integer);
CREATE OR REPLACE FUNCTION clientesDistintos(anyomes varchar, amount integer)
RETURNS TABLE (clientes bigint) AS $$

begin
RETURN query
        SELECT count(DISTINCT customerid)
        FROM orders
        WHERE totalamount > amount
                and TO_CHAR(orderdate, 'YYYYMM') = anyomes;

end; $$ language plpgsql;
-----------------------------------
```

Explicando la consulta, tenemos que seleccionar la suma de los distintos clientes (count(Distinct customerid)) que han realizado pedidos (tabla orders) cuyo totalamount del pedido supere el umbral pasado como argumento y los pedidos estén en un mes concreto (pasado como argumento).

Así, analizando la consulta con la sentencia EXPLAIN en un ejemplo *explain analyze select clientesDistintos('201404',100)* cuyo resultado es 1421 obtenemos un tiempo de resolución de:

	QUERY PLAN text
1	Result (cost=0.00..5.25 rows=1000 width=0) (actual time=66.679..66.679 rows=1 loops=1)
2	Total runtime: 66.687 ms

Sin embargo, añadiendo un índice en aquellos campos donde se realiza alguna comparación ya que éstos serán los únicos que nos dan una mejora en la consulta (en nuestro caso totalamount y orderdate):
create index idx_query ON orders (orderdate); --tiempo de creación de unos 200 ms
create index idx_query2 ON orders (totalamount); --tiempo de creación de unos 320 ms

Obtenemos una mejora en los tiempos de cerca de unos 10 segundos de media:

	QUERY PLAN text
1	Result (cost=0.00..5.25 rows=1000 width=0) (actual time=56.672..56.673 rows=1 loops=1)
2	Total runtime: 56.680 ms

Para estudiar esta mejora y estos índices hemos estudiado la mejora que se produce con cada índice por separado.

Así si solo creamos el índice del orderdate tenemos tiempos muy parecidos a la misma consulta sin índices:

	QUERY PLAN text
1	Result (cost=0.00..5.25 rows=1000 width=0) (actual time=66.251..66.252 rows=1 loops=1)
2	Total runtime: 66.260 ms

Y si solo creamos el índice del totalamount obtenemos una mejora muy parecida a la de la creación de los dos índices:

	QUERY PLAN text
1	Result (cost=0.00..5.25 rows=1000 width=0) (actual time=58.884..58.885 rows=1 loops=1)
2	Total runtime: 58.894 ms

Así, el análisis de estos datos nos permite concluir que el índice que mejora la resolución de la consulta es totalamount.

B) Estudio del impacto de preparar sentencias SQL

En este apartado vamos a analizar la mejora en tiempo que implica preparar sentencias SQL o no. Para esto, hemos creado una página PHP pedida, *listaClientesMes.php* (que nos pedirá un mes a mostrar y sumistrará una tabla con el número de clientes distintos para diferentes umbrales hasta llegar al máximo, usando la consulta del apartado anterior).

Tras probar usando prepare y sin usarlo. Llegamos a la concusión que usando prepare es de media 3-4ms más rápido a la hora de efectuar las consultas.

C) Impacto de la forma de realizar una consulta

En este apartado vamos a estudiar los planes de ejecución de tres consultas alternativas dadas, cuyo resultado y objetivo es el mismo.

Básicamente la consulta quiere obtener los id (campo customerid) de los clientes (tabla clientes) que no tengan pedidos (tabla orders) con estado 'pagado' (campo status=='Paid').

Analizamos las consultas una a una:

Consulta 1:

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

La secuencia *'Seq Scan on customer'* indica que lee toda la tabla customers, con un coste estipulado entre 3961.65 y 4490.81, con 7046 filas de resultado y 4 bytes por fila.
'Filter: (NOT (hashed SubPlan 1))' Indica que filtra por la subconsulta.
Y la subconsulta, al igual que la consulta padre, consiste en leer toda la tabla orders con el filtro status='Paid' y tiene una estimación inicial de coste 0 (si no hay ninguna que cumpla el filtro) o 3959.38, con un resultado de 909 filas de 4 bytes casa una.

Consulta 2:

```
------------------------------------
explain select customerid
from(
        select customerid
        from customers union all
                select customerid
                from orders
                where status='Paid'
        )as A
group by customerid
having count(*) =1;
------------------------------------
"HashAggregate  (cost=4537.41..4539.91 rows=200 width=4)"
"  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)"
------------------------------------
```

La secuencia *' HashAggregate'* es la suma del '*having count(*) =1;'*. Así, agrupa y añade los resultado de, al igual que la consulta 1, recorrer toda la tabla costumer y toda la tabla orders (sabido por los seq scan) y filtrar por status='paid'.

Consulta 3:

```
------------------------------------
explain select customerid
from customers except
        select customerid
        from orders
```

```
        where status='Paid';
-----------------------------------
"HashSetOp Except  (cost=0.00..4640.83 rows=14093 width=4)"
"  ->  Append  (cost=0.00..4603.32 rows=15002 width=4)"
"        ->  Subquery Scan on "*SELECT* 1"  (cost=0.00..634.86 rows=14093 width=4)"
"              ->  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=4)"
"              ->  Seq Scan on orders  (cost=0.00..3959.38 rows=909 width=4)"
"                    Filter: ((status)::text = 'Paid'::text)"
-----------------------------------
```

Esta ejecución trata las consultas como dos subconsultas '*Subquery Scan on "*SELECT* 1" y Subquery Scan on "*SELECT* 2"'*. Al igual que las anteriores recorre toda la tabla costumer y toda la tabla orders (sabido por los seq scan) y filtra por status='paid'. Además la primera línea de la planificación de la ejecución *'HashSetOp Except…'* indica como utiliza el comando SQL except, la base de la consulta.

D) Impacto de la generación de estadísticas

Vamos a analizar dos consultas sencillas:

La consulta 1 cuenta los pedidos con campo status a NULL:

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

La consulta 2 cuenta los pedidos con el campo status a Enviado (Shipped).

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

De la consulta 1 obtenemos de resultado la cantidad 0 y de la consulta 1 obtenermos de resultado 127323.

Vamos a analizarlas con la sentencia Explain:

Consulta 1

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

Vemos que se recorre toda la tabla orders filtrando por el campo status y hace la operación de aggregate con un coste de 3507 (el coste es por recorrer la tabla y filtrar).

Consulta 2

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

Vemos que se recorre toda la tabla orders filtrando por el campo status y hace la operación de

aggregate con un coste de 3961 (el coste es por recorrer la tabla y filtrar).

Vamos a crear un índice en la tabla orders por la columna status (la columna que nos servirá en estas dos consultas).

CREATE INDEX idx_status ON orders(status);

Estudiamos de nuevo la planificación de las consultas anteriores, ahora con el índice:

Consulta 1

"Aggregate (cost=1496.52..1496.53 rows=1 width=0)"
" -> 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)"

Vemos que utiliza el índice filtrándolo con la condición de que status sea NULL, el coste de ésto es mucho más pequeño (unos 19 como máximo). Después recorre el montón del el mapa de bits con esa condición con un coste como máximo de 1494 y lo suma.

Ésto es mucho más eficiente que sin el índice ya que sin el índice era un coste de ~4000 y ahora es ~1500 con el índice.

Consulta 2

"Aggregate (cost=1498.79..1498.80 rows=1 width=0)"
" -> 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)"

Vemos que utiliza el índice filtrándolo con la condición de que status sea Shipped, el coste de ésto es mucho más pequeño (unos 19 como máximo). Después recorre el montón del el mapa de bits con esa condición con un coste como máximo de 1496 y lo suma.

Al igual que el anterior, es mucho más eficiente que sin el índice ya que sin el índice era un coste de ~4000 y ahora es ~1500 con el índice.

Ejecutamos ahora la sentencia ANALYZE sobre la tabla orders para ayudar a determinar los planes de ejecución más eficientes para las consultas.

ANALYZE VERBOSE orders;

INFO: analyzing "public.orders"
INFO: "orders": scanned 1687 of 1687 pages, containing 181790 live rows and 0 dead rows; 30000 rows in sample, 181790 estimated total rows

Ahora volvemos a obtener la planificación de las consultas anteriores:

Consulta 1

"Aggregate (cost=4.44..4.45 rows=1 width=0)"
" -> Index Only Scan using idx_status on orders (cost=0.42..4.44 rows=1 width=0)"
"   Index Cond: (status IS NULL)"

Vemos que ha cambiado la planificación. Ahora solo utiliza el índice con la condición impuesta sin recorrer el mónton del mapa de bits. Tiene un coste como máximo de 4 ya que sabe donde buscar. Hemos obtenido una gran mejora (de coste inicial ~4000 a 4).

Consulta 2

"Aggregate (cost=4057.96..4057.97 rows=1 width=0)"
" -> Index Only Scan using idx_status on orders (cost=0.42..3740.27 rows=127077 width=0)"
"   Index Cond: (status = 'Shipped'::text)"

Vemos que ha cambiado la planificación. Ahora solo utiliza el índice con la condición impuesta sin recorrer el mónton del mapa de bits. Sin embargo esta consulta tiene un coste más elevado debido a que la condición implica a muchas filas de la tabla (127077). Por lo tanto no hemos conseguido una mejora (incluso empeora un poco).

Vamos a analizar otras dos consultas:

La consulta 3 cuenta los pedidos con el campo status a Pagado (Paid).

Consulta 3

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

La consulta 4 cuenta los pedidos con el campo status a Procesado (Processed).

Consulta 4

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

La consulta 3 tiene un resultado de 18163 y de la 4 obtenemos un resultado de 36304

Aplicando la sentencia Explain obtenemos lo siguiente:

Consulta 3

"Aggregate (cost=576.48..576.49 rows=1 width=0)"
" -> Index Only Scan using idx_status on orders (cost=0.42..531.47 rows=18003 width=0)"
"   Index Cond: (status = 'Paid'::text)"

Pero si borramos el índice

"Aggregate (cost=4004.70..4004.71 rows=1 width=0)"
" -> Seq Scan on orders (cost=0.00..3959.38 rows=18131 width=0)"
"   Filter: ((status)::text = 'Paid'::text)"

Vemos así que obtenemos una gran mejora con el índice que sin él. Por otro lado la consulta 3 tiene un mejor tiempo que la consulta 2. Esto puede ser debido a que el filtro es mucho más restrictivo que en la consulta 3 y se obtienen menos resultados.

Consulta 4

"Aggregate (cost=1174.60..1174.61 rows=1 width=0)"

" -> Index Only Scan using idx_status on orders (cost=0.42..1082.83 rows=36709 width=0)"
" Index Cond: (status = 'Processed'::text)"

Pero si borramos el índice

"Aggregate (cost=4049.24..4049.25 rows=1 width=0)"
" -> Seq Scan on orders (cost=0.00..3959.38 rows=35946 width=0)"
" Filter: ((status)::text = 'Processed'::text)"

Ocurre lo mismo que con la consulta 3. Es mucho más eficiente con índice del orden de 4 veces mejor.

Todo este procedimiento se encuentra en el fichero *'countStatus.sql'*.

2. Transacciones

En este apartado de la práctica vamos a trabajar con transacciones.

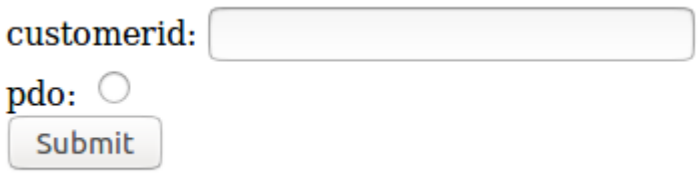
Lo primero que vamos a hacer es realizar un programa php 'borraCliente.php' que borre un cliente de la base de datos, con todos sus respectivos datos (Esto implica borrarlo de la tabla de clientes, de la tabla de pedidos y de la tabla que relaciona los pedidos con los clientes).

Esto no se puede hacer de cualquier forma ya que si intentamos borrar un cliente de la tabla clientes sin borrarlo de la tabla que lo relaciona con los pedidos el sistema dará un error ya que dejaría la base de datos inconsistente. Lo mismo ocurre si no borramos los pedidos a la vez que borramos los datos del pedido. Podríamos dejar pedidos realizados por un cliente inexistente.

Para lograr una consistencia y una buena actualización de la base de datos necesitamos hacerlo mediante transacciones.

Para borrar un cliente lo borraremos por su identificador (customerid). Primero lo borraremos de la tabla del detalle del pedido (orderdetail), luego de la tabla de pedidos (orders) y luego de la tabla de clientes (customers).

Lo primero que obtendremos de nuestra página será un formulario que nos que cliente borrar (por su id).



Además hemos añadido un segundo campo, atendiendo a los consejos del enunciado para indicar si realizar la transacción via exec() o por la interfaz PDO.

Veamos una prueba de dicho programa:

Supongamos que queremos borrar el cliente con customerid=7. Vemos que está en nuestra base de datos:

```
Select * FROM orderdetail, orders WHERE orderdetail.orderid = orders.orderid AND orders.customerid= 7;
--Select * from orders where customerid = 7;
--Select * from customers where customerid = 7;
```

Output pane

Data Output | Explain | Messages | History

	orderid integer	prod_id integer	price numeric	quantity integer	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying(10)
1	174	3003	17.5681923254738789	1	174	2015-01-10	7	95.2380952380952381	15	109.52	Shipped
2	174	2476	11.0957004160887656	1	174	2015-01-10	7	95.2380952380952381	15	109.52	Shipped
3	174	5560	24.9653259361997226	1	174	2015-01-10	7	95.2380952380952381	15	109.52	Shipped
4	174	75	15.7189089227924179	1	174	2015-01-10	7	95.2380952380952381	15	109.52	Shipped
5	174	3160	13.8696255201109570	1	174	2015-01-10	7	95.2380952380952381	15	109.52	Shipped
6	174	505	12.0203421174294961	1	174	2015-01-10	7	95.2380952380952381	15	109.52	Shipped

```
--Select * FROM orderdetail, orders WHERE orderdetail.orderid = orders.orderid AND orders.cu
Select * from orders where customerid = 7;
--Select * from customers where customerid = 7;
```

Output pane

Data Output | Explain | Messages | History

	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying(10)
1	171	2014-01-06	7	60.1017105871474803	15	69.12	Shipped
2	168	2014-06-18	7	34.9514563106796117	15	40.19	Processed
3	172	2015-12-06	7	145.1687471104946832	15	166.94	Shipped
4	173	2012-12-14	7	123.7170596393897365	15	142.27	Processed
5	174	2015-01-10	7	95.2380952380952381	15	109.52	Shipped
6	167	2012-09-16	7	61.7660656495607953	15	71.03	Processed
7	170	2015-01-17	7	41.0540915395284328	15	47.21	Shipped
8	169	2014-03-19	7	34.2117429496070273	15	39.34	Shipped

```
--Select * FROM orderdetail, orders WHERE orderdetail.orderid = orders.orderid AND orders.customerid= 7;
--Select * from orders where customerid = 7;
Select * from customers where customerid = 7;
```

Output pane

Data Output | Explain | Messages | History

	customerid integer	firstname character varying(50)	lastname character varying(50)	address1 character varying(50)	address2 character varying(50)	city character varying(50)	state character vary
1	7	havana	opine	sheikh clear 299	bell paving	virus	weigh

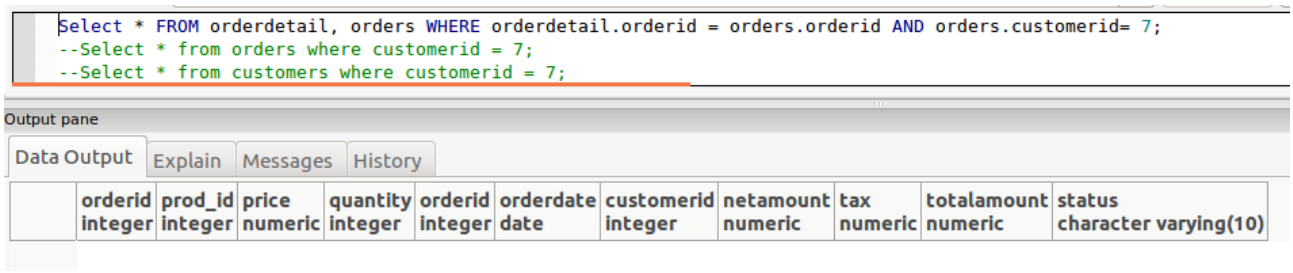
Pues bien, ejecutamos nuestro programa pasándole como valor el 7 al formulario pedido:

customerid: 7

pdo:

Submit

Preguntamos en nuestra base de datos si hay rastro de él y vemos que no:

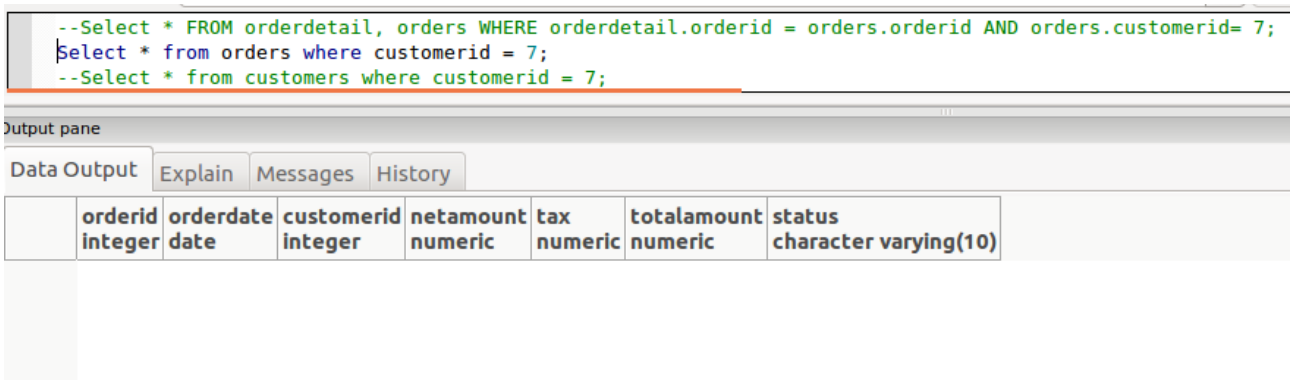


```
Select * FROM orderdetail, orders WHERE orderdetail.orderid = orders.orderid AND orders.customerid= 7;
--Select * from orders where customerid = 7;
--Select * from customers where customerid = 7;
```

Output pane

Data Output | Explain | Messages | History

	orderid integer	prod_id integer	price numeric	quantity integer	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying(10)

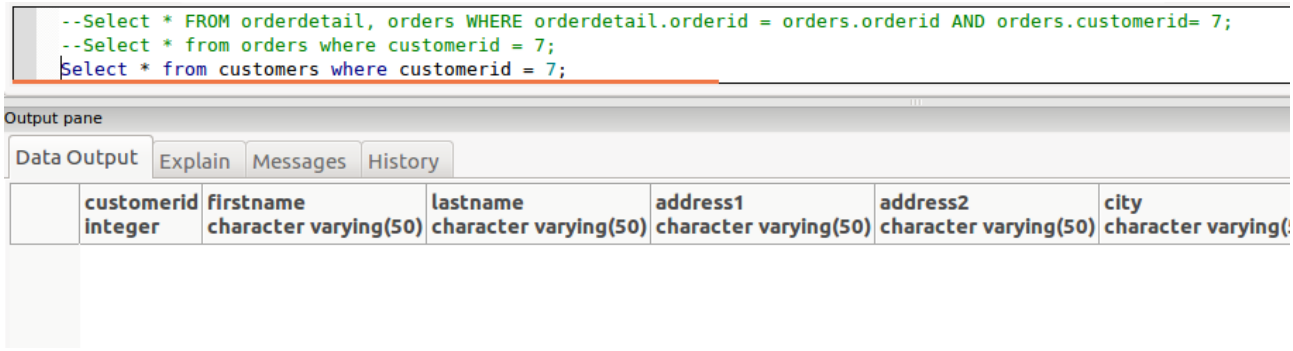


```
--Select * FROM orderdetail, orders WHERE orderdetail.orderid = orders.orderid AND orders.customerid= 7;
Select * from orders where customerid = 7;
--Select * from customers where customerid = 7;
```

Output pane

Data Output | Explain | Messages | History

	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying(10)



```
--Select * FROM orderdetail, orders WHERE orderdetail.orderid = orders.orderid AND orders.customerid= 7;
--Select * from orders where customerid = 7;
Select * from customers where customerid = 7;
```

Output pane

Data Output | Explain | Messages | History

	customerid integer	firstname character varying(50)	lastname character varying(50)	address1 character varying(50)	address2 character varying(50)	city character varying(

Vemos que el resultado ha sido satisfactorio.

Trataremos ahora una versión incorrecta de este programa para estudiar que ocurre:

Por un lado hemos añadido lo que se nos pedía aparte del formulario inicial en borraCliente.php que es un campo del formulario para añadir un commit en mitad antes del fallo.

Hemos añadido varios print de las tablas en mitad del código:

a) Al principio de la transacción.
b) Después de ejecutar el primer borrado (tabla orderdetail, sin fallo).
c) Después de ejecutar el segundo borrado (tabla customerid, con fallo (debido a que borramos la tabla donde se encuentra alojada la foreing key de otra tabla, orders), lo que provocaría excepción y nunca se ejecutaría este print de las tablas).
d) Después del rollback al saltar la excepción (si salta la excepción debería rehacer todos los borrados y mostrar la tabla como al principio de la transacción).

Ejecutamos dicho código, por ejemplo para el customerid 180 y obtenemos esta salida:

Primer print:

Antes de ejecutar

TABLA ORDERDETAIL

2436-4733-180
2436-3124-180
2436-3492-180

TABLA CUSTOMERS

180-eating

TABLA ORDERS

2439-180
2438-180
2441-180
2437-180
2436-180
2440-180
2446-180
2443-180
2445-180
2442-180
2444-180

Segundo print: Sin problemas como debería, ha borrado la tabla orderdetail.

Despues de ejecutar

TABLA ORDERDETAIL

TABLA CUSTOMERS

180-eating

TABLA ORDERS

2439-180
2438-180
2441-180
2437-180
2436-180
2440-180
2446-180
2443-180
2445-180
2442-180
2444-180

Tercer print: No se ejecuta, como es esperado. Salta la excepción y hace el rollback.

SQLSTATE[23503]: Foreign key violation: 7 ERROR: update or delete on table "customers" violates foreign key constraint "orders_customerid_fkey" on table "orders" DETAIL: Key (customerid)=(180) is still referenced from table "orders".

Cuarto print: Se ejecuta como al principio de la transacción. Como es esperado.

Despues de rollback

TABLA ORDERDETAIL

2436-4733-180
2436-3124-180
2436-3492-180

TABLA CUSTOMERS

180-eating

TABLA ORDERS

2439-180
2438-180
2441-180
2437-180
2436-180
2440-180
2446-180
2443-180
2445-180
2442-180
2444-180

Veamos ahora que ocurre si activamos la opción del commit en el formulario donde activamos un

commit en mitad del código, antes de llegar al error:

Primer print:

Antes de ejecutar

TABLA ORDERDETAIL

2848-2990-207
2848-6615-207
2848-4974-207

TABLA CUSTOMERS

207-iliac

TABLA ORDERS

2847-207
2849-207
2848-207

Segundo print: Sin problemas como debería, ha borrado la tabla orderdetail. La diferencia es que hace un commit en mitad de la transacción original, luego esto se actualiza y empieza otra transacción.

Despues de ejecutar

TABLA ORDERDETAIL

TABLA CUSTOMERS

207-iliac

TABLA ORDERS

2847-207
2849-207
2848-207

Tercer print: No se ejecuta, como es esperado. Salta la excepción y hace el rollback.

SQLSTATE[23503]: Foreign key violation: 7 ERROR: update or delete on table "customers" violates foreign key constraint "orders_customerid_fkey" on table "orders" DETAIL: Key (customerid)=(207) is still referenced from table "orders".

Cuarto print: Al haber hecho el commit, hemos actualizado la tabla orderdetail y por eso no hay nada en ella.

Despues de rollback

TABLA ORDERDETAIL

TABLA CUSTOMERS

207-iliac

TABLA ORDERS

2847-207
2849-207
2848-207

Vamos a estudiar ahora los bloqueos y deadlocks

Para ello vamos a crear un script, *updPromo.sql*, que cree una nueva columna, promo, en la tabla customers. Esta columna contendrá un descuento (en porcentaje) promocional.

ALTER TABLE customers ADD promo INTEGER;

Además añadimos al script 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. También le hemos añadido un sleep.

3. Seguridad

G) Acceso indebido a un sitio web

Dada una pagina PHP en el enunciado intentaremos realizar SQL-Inyection para acceder a su base de datos y obtener información a la cual no deberíamos poder acceder. Nos es dado un nombre de usuario para intentar validar el login, pero lo cierto es que la página es tan vulnerable que no es ni necesario este campo para obtener los datos que queremos.

Hemos usado esta sentencia de SQL inyection:

a' OR 1=1 --

Como esta comparando una sentencia de texto introducimos la a’ al final para indicar que acaba ahi la cadena que espera que introduzca el usuario. Luego añadimos el operador OR y comparamos con la sentencia 1=1 que es siempre TRUE para que la consulta siempre sea exitosa. Finalmente añadir el ‘--’ para indicar que todo lo que viene detrás es un comentario SQL y no lo tenga en cuenta.

Para evitar estos casos y que accedan a nuestra base de datos podemos tomar diferentes medidas. Una opción es limpiar la entrada dada por el usuario para que no pueda introducir strings raros con los que hacer SQL inyection. Otro método es evitar conectarse a una base de datos como root para evitar que puedan alterar o eliminar nuestras tablas. Finalmente tener cuidado con nuestra propia base de datos. Un usuario puede registrar con éxito un nombre como O’connor, y luego al sacar el nombre de nuestra base de datos y usarlo para otra cosa, usar la comilla para hacer SQL inyection. Por tanto siempre hay que hacer un check de los datos variables que vamos a hacer en una consulta aunque provengan de nuestra propia base de datos para evitar vulnerabilidades.

H) Acceso indebido a información

Este caso es parecido al que ocurre en el ejemplo anterior. La pagina no tiene ninguna protección contra SQL-Inyection, ni de roles y podemos acceder a todos sus datos sin problemas. En el punto a

se nos informa que la consulta que efectúa la nueva pagina PHP dada es un **SELECT x FROM y WHERE z =** pero que podría ser algo más complicada. Esto no nos importa debido a la técnica usada anteriormente de poner '--' para omitir todo lo que venga después de la consulta.

Hemos usado esta cadena de SQL-Inyection:

1' AND 1=0 UNION SELECT relname as movietitle FROM pg_class --

Es un poco diferente a la anterior. Como no hay ningún tipo de filtro usamos un AND 1=0 en el where para invalidar la consulta anterior y posteriormente hacer un UNION con la consulta que queremos efectuar. Remarcar el uso de poner '--' al final para omitir cualquier cadena que venga después y por tanto que solo se efectúe la sentencia SQL que deseamos. Como en este caso queremos saber todas las tablas del sistema hemos usado la tabla pg_class y el atributo relname para obtener todos los nombres de las tablas en el sistema atacado.

Ahora queremos sacar las tablas relevantes de la pagina y no junto a las del sistema.

Para ello usaremos esta nueva cadena SQL:

1' AND 1=0 UNION SELECT cast(oid as text) FROM pg_namespace WHERE nspname = 'public' –

Usado el truco de 1' AND 1=0 <consulta> -- Obtenemos primero el oid de las tablas public. Destacar el cast a text del oid ya que la pagina imprime chars. Una vez hecho esto obtenemos que el oid de public es = 2200.

Sabiendo esto realizamos la consulta anterior con un where:

1' AND 1=0 UNION SELECT relname FROM pg_class WHERE relnamespace = 2200 –

Obtenemos un listado grande con primaryKeys y seq facilmente distinguibles de las tablas que nos interesan.

Entre todas las tablas vemos una que se llama customers que es la que nos interesa ya que es la mas probable que tenga la información de los clientes. Ahora hallamos su oid con:

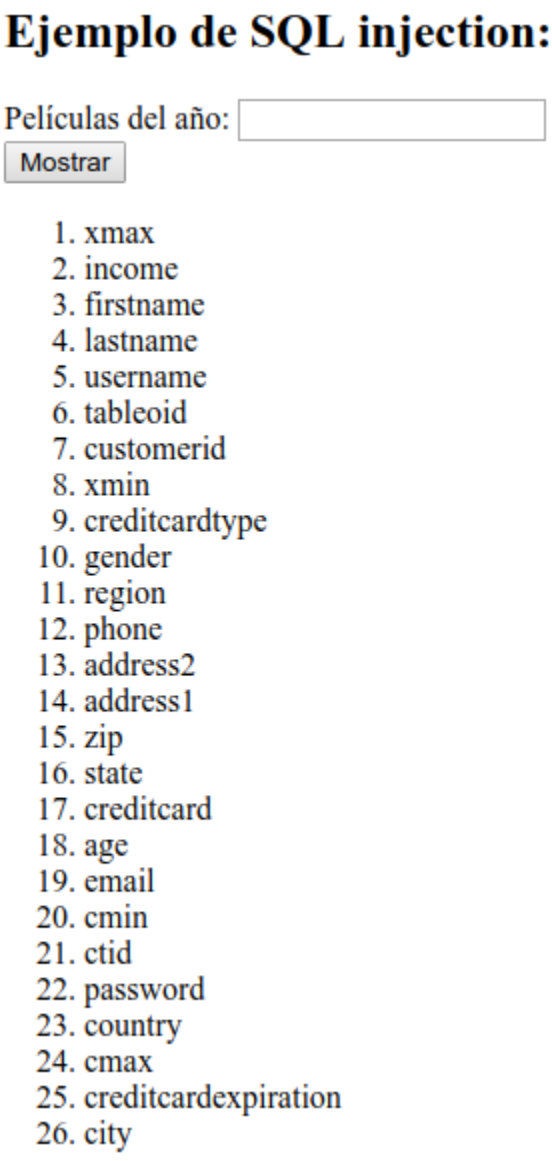
1' AND 1=0 UNION SELECT cast(oid as text) FROM pg_class WHERE relname = 'customers' –

Que es: 22175.

Sabiendo esto efectuamos la siguiente consulta para sacar los atributos de esta tabla.

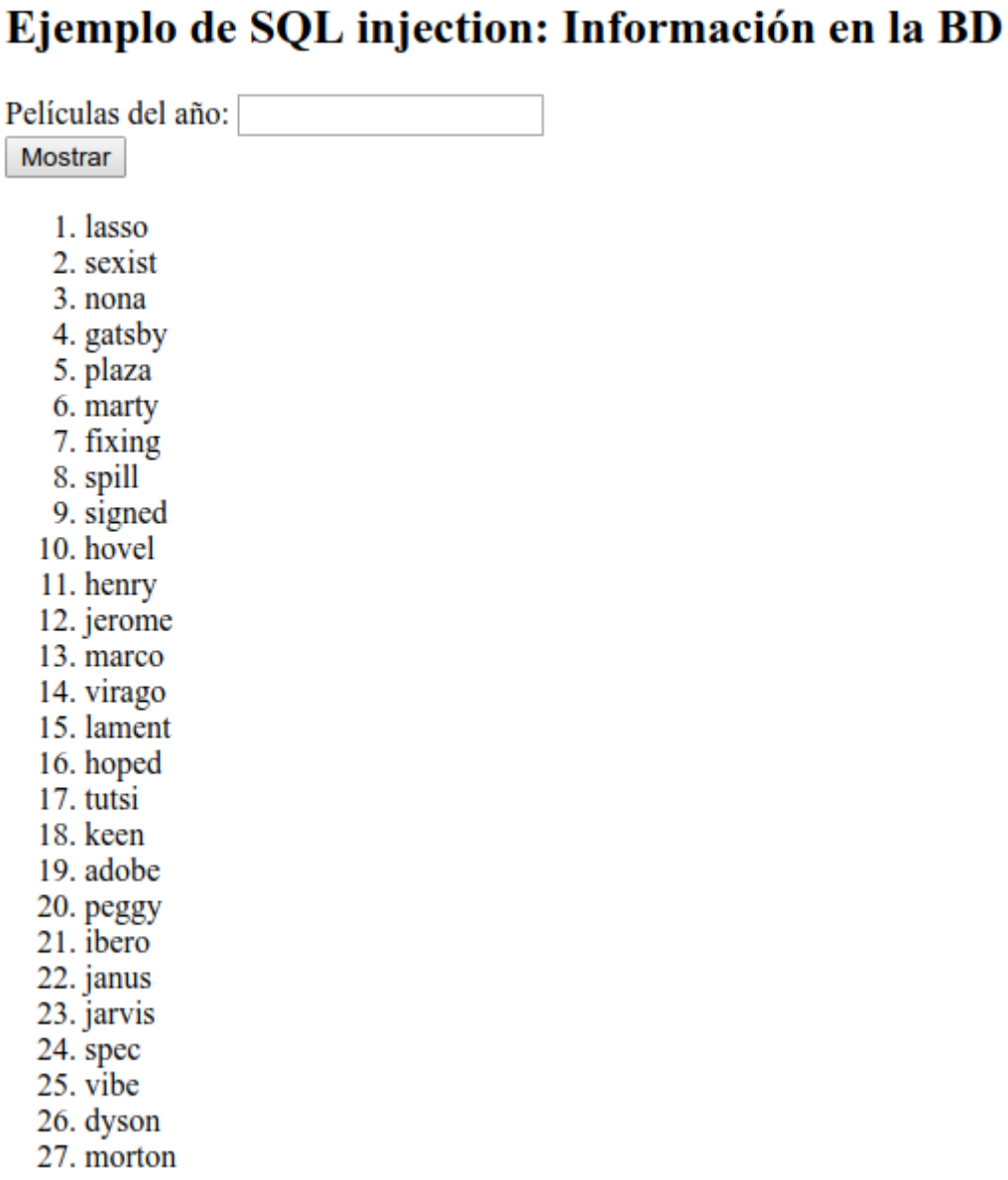
1' AND 1=0 UNION SELECT attname FROM pg_attribute WHERE attrelid = 22175 –

lo cual nos devuelve:



Como podemos ver ya sabemos todos los atributos de la tabla. Y sabiendo el nombre y los atributos de la tabla ya podemos sacar toda la información que queramos de estos. Por ejemplo todos los username's:

1' AND 1=0 UNION SELECT username FROM customers –



En el caso de haber usado una lista desplegable o el método POST no se habría conseguido nada ya que hay diversas maneras de crear o modificar una petición html por parte del cliente. Lo que hay que hacer es proteger en el lado del servidor con los métodos explicados anteriormente.