Práctica 4. Optimización, Transacciones y Seguridad

Asignatura: Sistemas Informáticos I (Prácticas de Laboratorio)
Grupo: 1363
Pareja: 4
Nombres: Sánchez Redondo, Pablo; Solana Vera, Antonio

Parte A. Optimización.

Ejercicio A: Índices

En primer lugar, mediremos el rendimiento de una consulta SQL y estudiaremos distintos planes de ejecución, comparándolos entre sí, mediante el uso del comando EXPLAIN. Para ello, hemos creado el fichero *"clientesDistintos.sql"* (adjunto a esta memoria de prácticas). En la consulta, hemos mostrados el número de clientes distintos que tienen pedidos en un mes dado y hemos creado distintos índices aplicados a las columnas o atributos en orderdate, totalamount y en ambas. Así, la Tabla 1 muestra los principales resultados obtenidos:

	Consulta Base	Índice en columna orderdate	Índice en columna totalamount	Índice en ambas
Coste total	5636	5636	4496	5636
Número total de operaciones	6	6	6	6

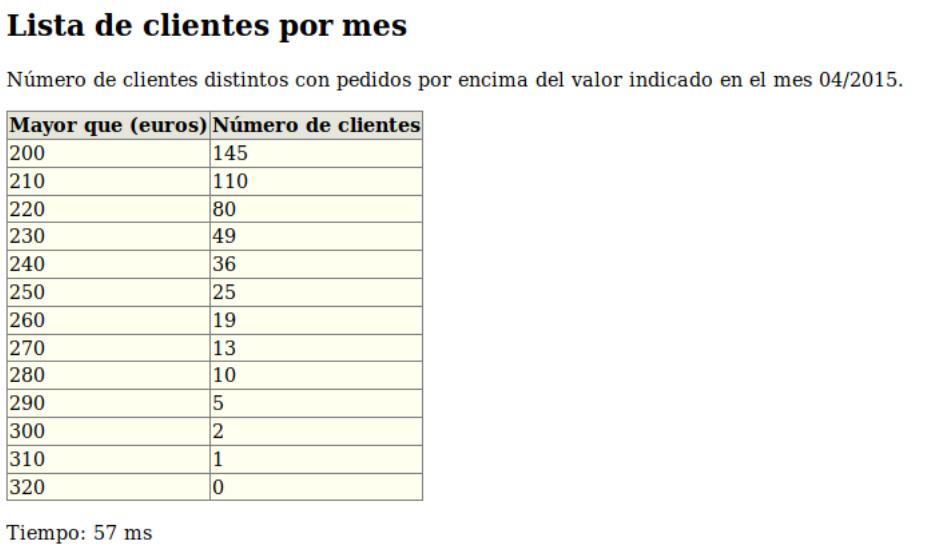
Tabla 1. Rendimiento del uso de un índice sobre una consulta SQL.

En general, los resultados obtenidos muestran que la creación de un índice en la columna totalamount mejora el rendimiento de la consulta en un 20% (en función del coste total). Así, la consulta totalamount tuvo un número total de operaciones de 4496, mientras que la consulta base tuvo un número total de operaciones de 5636 contra la base de datos. Como conclusión, podemos decir que el uso de índices acelera las búsquedas y mejora el rendimiento para los resultados obtenidos.

Ejercicio B: Preparar consultas

	Con índice	Sin índice
Con prepare	57	409
Sin prepare	57	494

Tabla 1. Rendimiento del uso de un índice y/o un prepare sobre una consulta SQL.



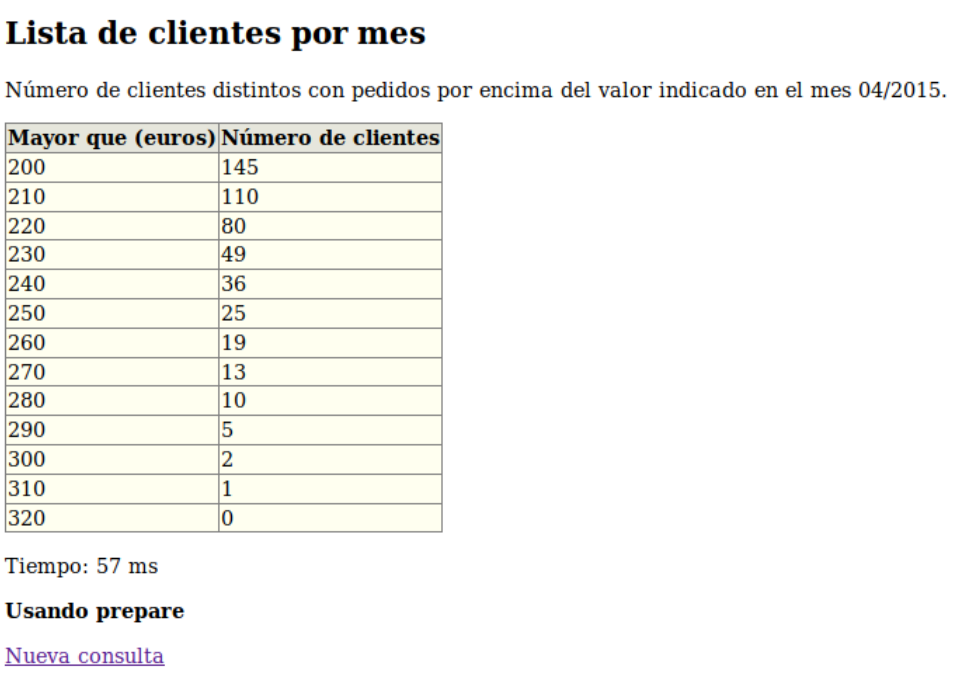
Lista de clientes por mes

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

Mayor que (euros)	Número de clientes
200	145
210	110
220	80
230	49
240	36
250	25
260	19
270	13
280	10
290	5
300	2
310	1
320	0

Tiempo: 57 ms

Imagen 1. Rendimiento de la consulta sin PREPARE y con INDEX en la web.



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
200	145
210	110
220	80
230	49
240	36
250	25
260	19
270	13
280	10
290	5
300	2
310	1
320	0

Tiempo: 57 ms

Usando prepare

Nueva consulta

Imagen 2. Rendimiento de la consulta sin PREPARE y con INDEX en la web.

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
200	145
210	110
220	80
230	49
240	36
250	25
260	19
270	13
280	10
290	5
300	2
310	1
320	0

Tiempo: 494 ms

Nueva consulta

Imagen 3. Rendimiento de la consulta sin PREPARE y sin INDEX en la web.

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
200	145
210	110
220	80
230	49
240	36
250	25
260	19
270	13
280	10
290	5
300	2
310	1
320	0

Tiempo: 409 ms

Usando prepare

Nueva consulta

Imagen 4. Rendimiento de la consulta sin PREPARE y sin INDEX en la web.

Se puede observar que con los prepare no causan ninguna diferencia usando INDEX dado que la velocidad de ejecución de la consulta ya es muy rápida. En cambio, a falta de un INDEX se puede notar una ligera diferencia (una mejora de un 18%).

Lista de ficheros adjuntos

- *database.py*: Contiene las sentencias ejecutadas en las mediciones del ejercicio (a través de la interfaz web)

Ejercicio C: Impacto de la forma de consultas

	Consulta 1	Consulta 2	Consulta 3
Coste total	3961	4537	0
Número total de operaciones	2	4	6

i. La tercera consulta devuelve el resultado de forma inmediata por que el coste de ejecución es cero.
ii. La segunda puede ser ejecutada en paralelo después del agregate. La tercera puede ser paralelizada sin ningún problema porque es una operación de append repetida múltiples veces.

Ejercicio D: Estudio del impacto de las estadísticas

	Consulta 1	Consulta 2	Consulta 3
Coste total	3961	4537	0
Número total de operaciones	2	4	6

i. La tercera consulta devuelve el resultado de forma inmediata por que el coste de ejecución es cero.
ii. La segunda puede ser ejecutada en paralelo después del agregate. La tercera puede ser paralelizada sin ningún problema porque es una operación de append repetida múltiples veces.

Lista de ficheros adjuntos

1. *countStatus.sql*: Contiene las sentencias ejecutadas en las mediciones del ejercicio.

Transacciones y Deadlock.

Ejercicio E: Estudio de Transacciones

Para eliminar a un usuario el orden correcto es: orderdetail, orders y por último customers, porque orderdetail tiene una foreign key a orders y, a su vez orders tiene una foreign key con customers. Para el fallo, eliminamos orderdetails, hacemos el commit o no, y luego eliminamos de la tabla customers, por lo que salta un error.

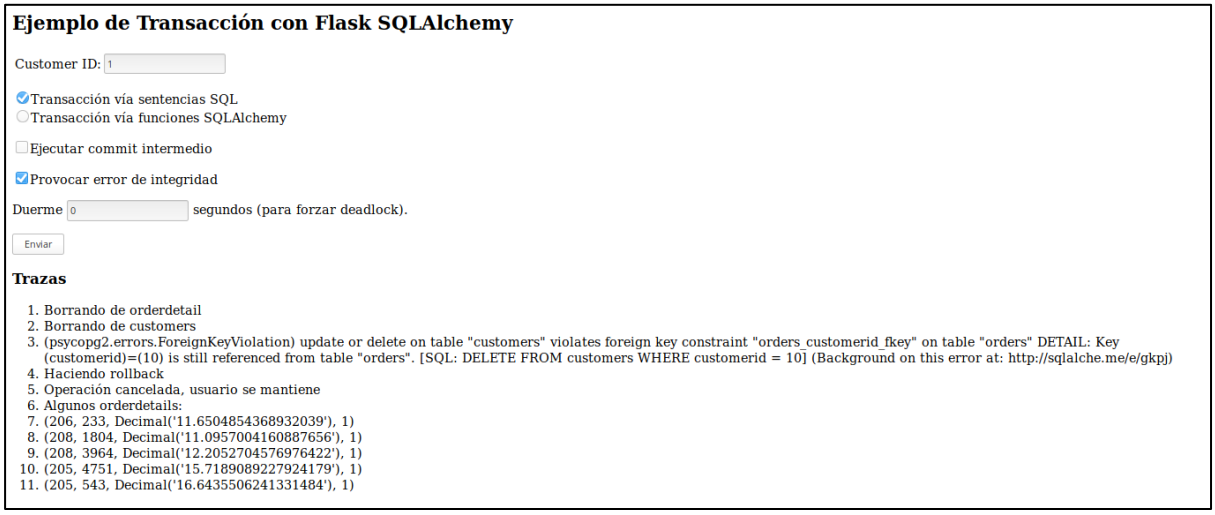


Imagen 5. Resultado de la web sin commit.

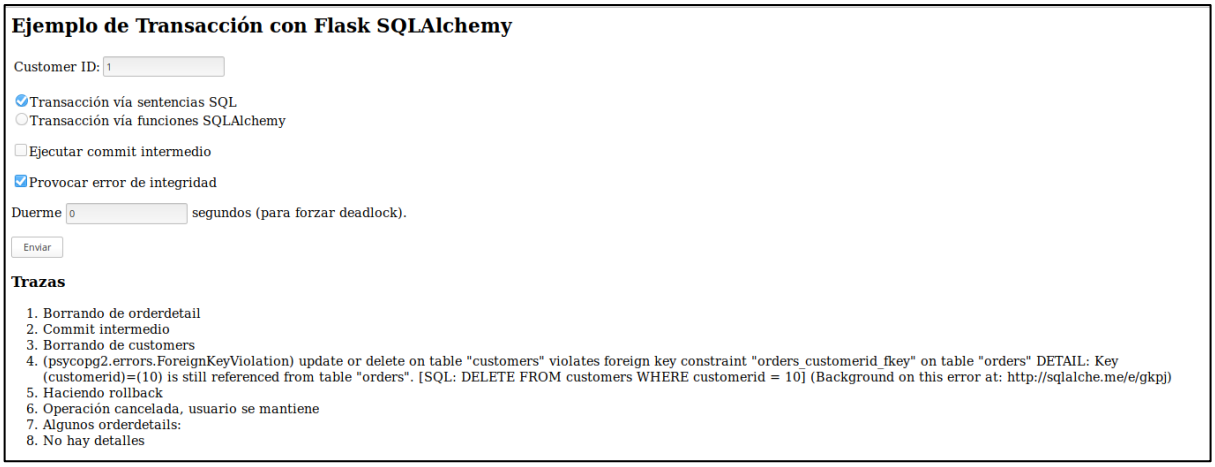


Imagen 6. Resultado con el commit en orderdetail.

Ejercicio F: Estudio de Bloqueos y Deadlocks

```
(si1pyenv) vosem@Rex-2077:~/Code/SI1/practica4$ psql -h localhost -U alumnodb si1 -c "SELECT * FROM ORDERDETAIL WHERE orderid=111"
 orderid | prod_id |        price        | quantity
---------+---------+---------------------+----------
     111 |    3260 | 14.7942672214516874 |        1
     111 |     125 | 10.1710587147480351 |        1
     111 |    3331 | 14.7942672214516874 |        1
     111 |    2087 | 15.7189089227924179 |        1
(4 rows)

(si1pyenv) vosem@Rex-2077:~/Code/SI1/practica4$ psql -h localhost -U alumnodb si1 -c "UPDATE orders SET status = NULL WHERE orderid=111"
UPDATE 1
```

Imagen 7. Creación de carritos.

```
(si1pyenv) vosem@Rex-2077:~/Code/SI1/practica4$ psql -h localhost -U alumnodb si1 -c "UPDATE customers SET promo = 0.10  WHERE customerid=2"
UPDATE 1
```

Imagen 8. Eliminar usuario dos y actualizar promo al mismo tiempo con el deadlock.

Trazas

1. Borrando de orderdetail
2. Borrando de orders
3. Borrando de customers
4. Todo bien, commit

Si abrimos otra sesión y intentamos realizar una operación sobre la fila que se esta modificando, no nos mostrará los resultados por que tenemos que esperar a que la operación entera se complete.

Cuando borramos un usuario desde la interfaz web, usando en este ejemplo un tiempo de espera de veinte segundos y a la vez ejecutamos una query que lance el trigger escrito anteriormente, se producen bloqueos en una sola fila (RowExclusiveLock) como se puede ver en la imagen nueve.

Schema	Table name	Virtual Transaction ID	Transaction ID	Process ID	Lock mode	Is lock held?
public	customers	6/690	727	6742	RowExclusiveLock	Yes
public	customers_pkey	6/690	727	6742	RowExclusiveLock	Yes
public	orderdetail	6/690	727	6742	RowExclusiveLock	Yes
public	orders	6/690	727	6742	AccessShareLock	Yes
public	orders_pkey	6/690	727	6742	AccessShareLock	Yes
public	products	6/690	727	6742	AccessShareLock	Yes
public	products_pkey	6/690	727	6742	AccessShareLock	Yes

Imagen 9. phpPgAdmin nos muestra los locks.

No hemos conseguido que se produzca un deadlock ni cambiando los tiempos ni ejecutando en distinto orden las queries. Hemos observado otra situación en la que también se producen bloqueos (imagen 10) pero no un deadlock.

Schema	Table name	Virtual Transaction ID	Transaction ID	Process ID	Lock mode	Is lock held?
public	customers	5/3357	736	6350	AccessExclusiveLock	Yes
public	customers	5/3357	736	6350	RowExclusiveLock	Yes
public	customers_pkey	5/3357	736	6350	RowExclusiveLock	Yes
public	orderdetail	5/3357	736	6350	RowExclusiveLock	Yes
public	orders	5/3357	736	6350	AccessShareLock	Yes
public	orders_pkey	5/3357	736	6350	AccessShareLock	Yes
public	customers	6/1698	735	6742	RowExclusiveLock	Yes
public	customers_pkey	6/1698	735	6742	RowExclusiveLock	Yes
public	orderdetail	6/1698	735	6742	RowExclusiveLock	Yes
public	orders	6/1698	735	6742	AccessShareLock	Yes
public	orders_pkey	6/1698	735	6742	AccessShareLock	Yes
public	products	6/1698	735	6742	AccessShareLock	Yes
public	products_pkey	6/1698	735	6742	AccessShareLock	Yes

Imagen 10. Bloqueo encontrado mientras buscábamos un deadlock.

Para evitar los deadlocks podemos minimizar tiempos de espera debidos a I/O, en nuestro caso los sleep. Pero la solución más efectiva es acceder a los recursos siempre en el mismo orden. Igual que en C usamos semáforos siempre en el mismo orden para evitar un interbloqueo, en SQL deberíamos hacer lo mismo.

Lista de ficheros adjuntos

- *updPromo.sql:* Creación de la columna promo y trigger que la actualiza

Seguridad.

Ejercicio G: Login

La parte del código que es vulnerable es la siguiente:

```
84  def getCustomer(username, password):
85      # conexion a la base de datos
86      db_conn = db_engine.connect()
87
88      query="select * from customers where username='" + username + "' and password='" + password + "'"
89      res=db_conn.execute(query).first()
90
91      db_conn.close()
92
93      if res is None:
94          return None
95      else:
96          return {'firstname': res['firstname'], 'lastname': res['lastname']}
```

Imagen 11. Código vulnerable a inyección SQL.

De forma más precisa, la línea vulnerable es la 88, cuando guardamos en query un texto con input de usuario sin sanear.

Para saltarnos el proceso de login, podemos introducir lo siguiente:

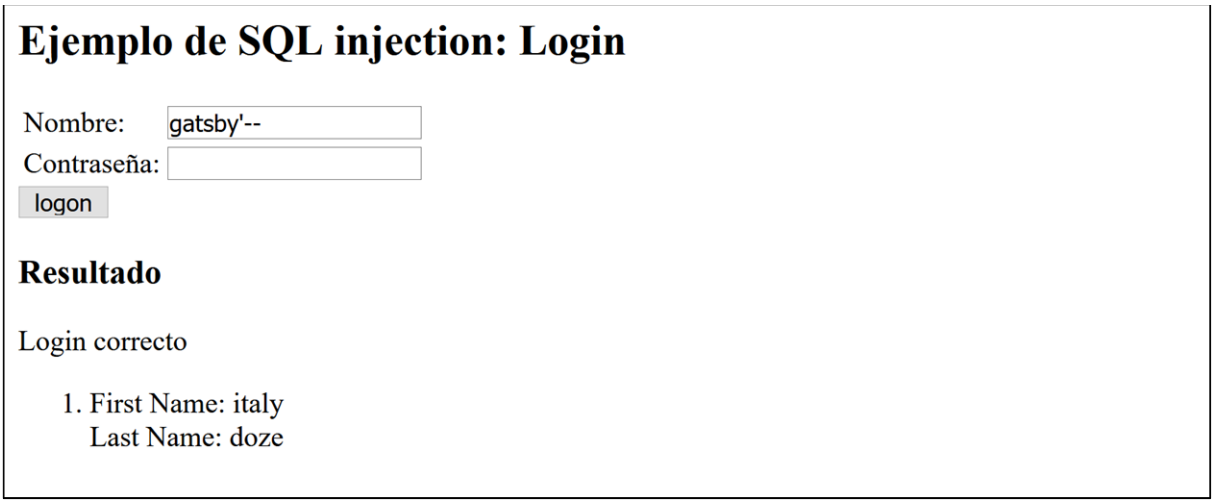


Imagen 12. Demostración de vulnerabilidad en la web con nombre de usuario.

Este texto funciona porque el doble guión del final deja comentada la comprobación de la contraseña.

También podemos realizar un login correcto sin conocer un nombre de usuario empleando la siguiente técnica:

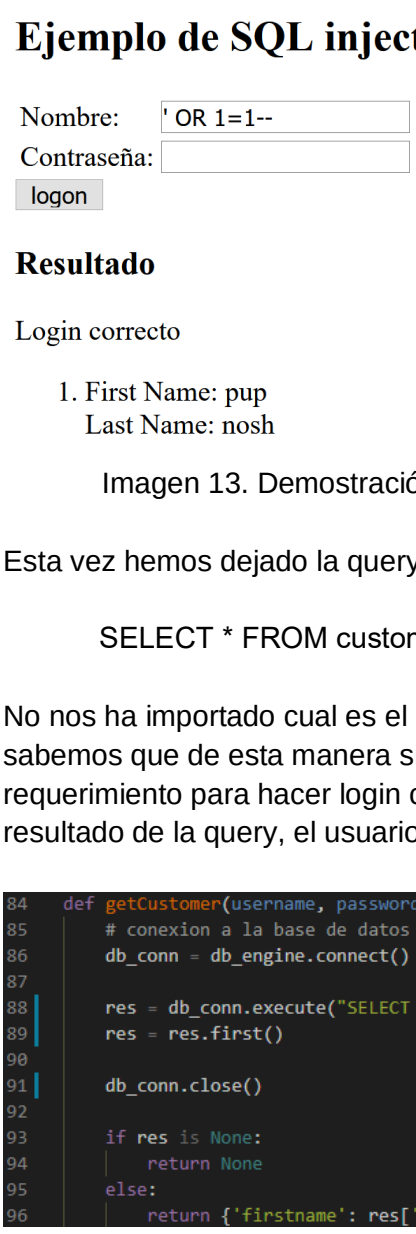


Imagen 13. Demostración de vulnerabilidad en la web sin nombre de usuario.

Esta vez hemos dejado la query de la siguiente manera:

SELECT * FROM customers WHERE username=" OR 1=1

No nos ha importado cual es el usuario y hemos puesto un 1=1 al final de la query porque sabemos que de esta manera siempre nos va a devolver un usuario (que es el único requerimiento para hacer login correcto). Como en el código solo seleccionamos el primer resultado de la query, el usuario que nos muestra la web es el primero que ha encontrado.

```
84  def getCustomer(username, password):
85      # conexion a la base de datos
86      db_conn = db_engine.connect()
87
88      res = db_conn.execute("SELECT * FROM customers WHERE username=%s AND password=%s", (username, password))
89      res = res.first()
90
91      db_conn.close()
92
93      if res is None:
94          return None
95      else:
96          return {'firstname': res['firstname'], 'lastname': res['lastname']}
```

Imagen 14. Código funcional no vulnerable.

En este ejemplo no vulnerable en vez de insertar directamente el input del usuario en nuestra query, le pasamos por un lado la query y por otro los valores a insertar a la función "execute" de SQLAlchemy. Esta función se ocupa por nosotros de sanear el input del usuario, previniendo así una inyección SQL.

Otra forma más correcta de realizar la consulta hubiera sido no usar "raw" SQL, y utilizar las funciones de SQLAlchemy.

Ejercicio H: Acceso a información.

De nuevo tenemos el mismo problema que en el apartado anterior, estamos ejecutando una query con el input del usuario sin sanear.

```
62  def getMovies(anio):
63      # conexion a la base de datos
64      db_conn = db_engine.connect()
65
66      query="select movietitle from imdb_movies where year = '" + anio + "'"
67      resultproxy=db_conn.execute(query)
```

Imagen 15. Código vulnerable para buscar películas.

Podemos ver en la imagen 16, como utilizando el mismo truco de 1=1 para que siempre se cumpla la query, podemos seleccionar todas las películas de la base de datos.

Imagen 16. Demostración de la vulnerabilidad.

Para filtrar información de la base de datos nos hemos con un problema: no conocemos el nombre de la variable que muestra las películas por pantalla, o si son varias. Si lo supiéramos o acertáramos probando podríamos ejecutar la siguiente query:

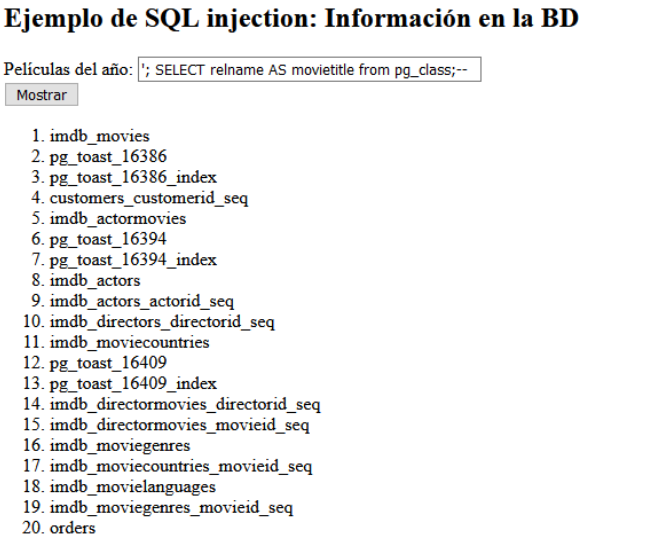


Imagen 17. Mostramos todas las tablas del sistema.

No solo eso, si queremos ver solo las del usuario podemos obtener su OID con la siguiente query:

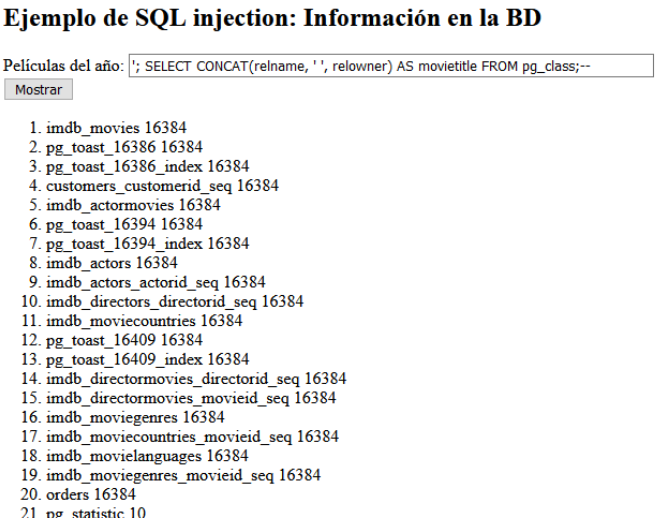


Imagen 18. Obtenemos el OID del usuario.

Y una vez conocido el “relowner” de las tablas que nos interesan podemos ejecutar esta para ver todas las tablas del usuario:

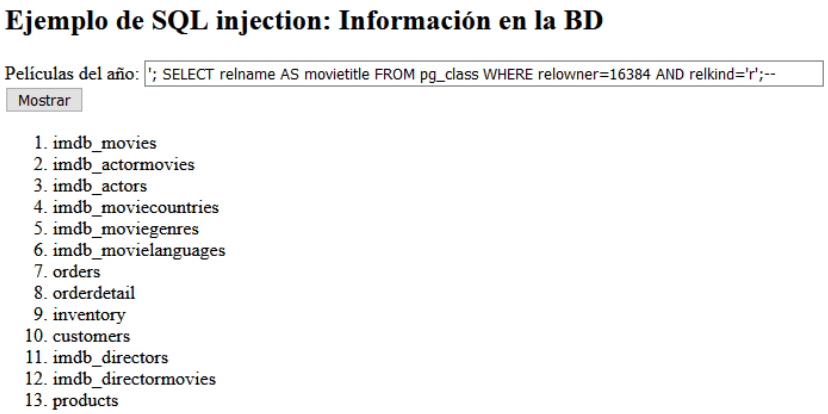


Imagen 19. Obtenemos las tablas del usuario con su OID.

Viendo estas tablas podemos asumir que lo que nos interesa es la tabla “customers” que probablemente tenga la información confidencial que buscamos.
Ahora necesitamos los nombres de las columnas de la tabla. Podemos acceder fácilmente a través de “information_schema.columns” como se ve en la imagen:

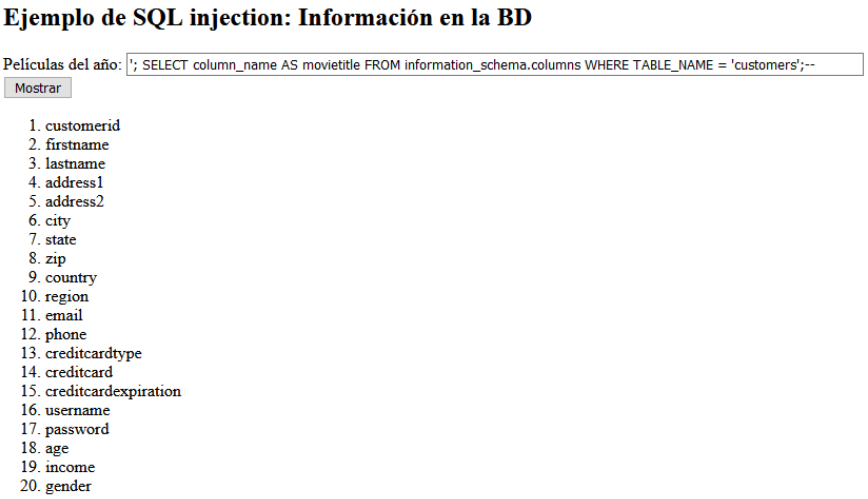


Imagen 20. Acceso al nombre de las columnas de una tabla.

Una vez obtenido el nombre de las columnas, usando “AS movietitle” y “CONCAT()” podemos obtener cualquier información de los clientes de la base de datos. En la imagen 15 se puede observar el nombre, apellido y tarjeta de crédito de todos los clientes.

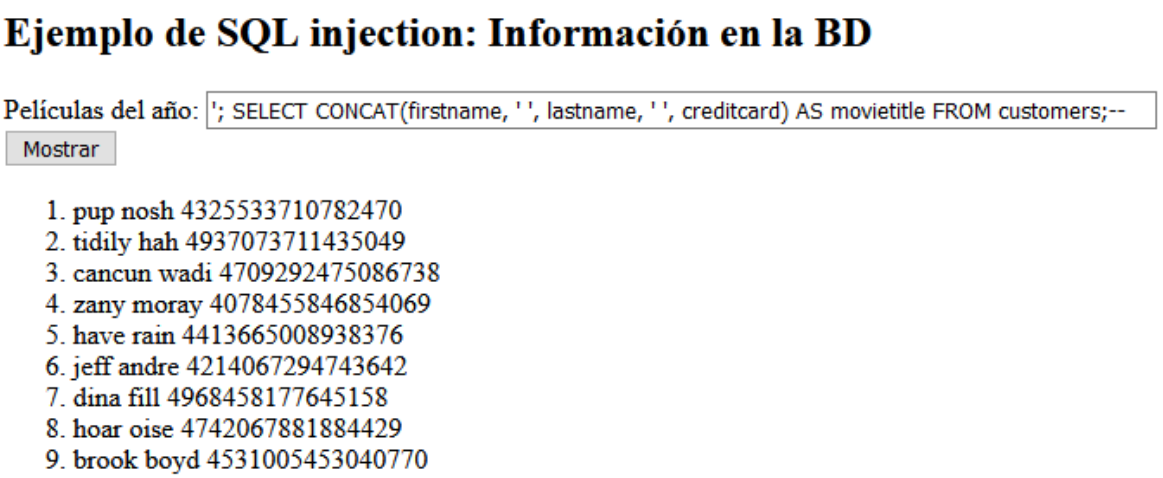


Imagen 21. Obtenemos información de los clientes de la base de datos.

Para resolver este problema no podemos usar un “combobox” porque podemos seguir haciendo las mismas peticiones, aunque ya no de forma manual desde el navegador. Usar post nos deja en la misma posición, no cambia nada ya que el usuario puede seguir usando el navegador u otras herramientas para hacer una inyección SQL.

Un atacante con un proxy como burpsuite puede realizar estas operaciones sin importar cual sea la interfaz mostrada al usuario o el método HTTP utilizado para mandar los datos al servidor.

Para resolver el problema debemos realizar los mismos cambios que en el apartado G (Imagen 8).

Nosotros proponemos la siguiente solución para evitar este tipo de problemas en el futuro. En lugar de utilizar las funciones de acceso a la base de datos de forma directa, podemos crear un nuevo archivo wrapper que nos obligue a pasarle por un lado la query y por otro los parámetros, de manera que sea imposible que aparezcan nuevas vulnerabilidades en el futuro.