UNIVERSIDAD AUTÓNOMA DE MADRID

SISTEMAS INFORMÁTICOS I

(2019-2020)

PRÁCTICA 4

Alba Ramos Pedroviejo
Javier Lozano Almeda

Grupo 1363

Madrid, 19 de Diciembre de 2019

ÍNDICE DE CONTENIDOS

Optimización

Estudio del impacto de un índice

Apartado A

Hemos creado la siguiente consulta:

```
select count (distinct orders.customerid)
from customers inner join orders
on orders.customerid = customers.customerid
where extract(year from orderdate)=2015 and
    extract(month from orderdate)=04 and
    totalamount>100;
```

En ella los valores rojos son los parámetros del enunciado, que se pueden personalizar.

Apartado B

Hemos ejecutado el comando explain sobre la consulta. Podemos observar que el coste total de realizar la consulta va a ser de 5636.24 y va a afectar a una sola fila, por ser un agregado (count). Se observa que primero se hace un nested loop, correspondiente al join entre dos tablas. Después, se hace una búsqueda secuencial sobre toda la tabla orders (seq scan) que es la que más tarda (4627.72), ya que debe ir comprobando todas las orders, y se filtra el resultado según los campos del where, eliminando las filas que no lo cumplan. Finalmente, se usa el índice (creado por defecto en las claves primarias) correspondiente a customerid para filtrar los resultados.

Apartado C

Hemos creado un índice sobre totalamount, que es la parte que más tardaba en ejecutarse. Lo hemos creado con la sentencia: create index idx_totalamount on orders(totalamount);

Apartado D

Si ejecutamos el comando explain tras crear el índice, vemos que efectivamente se ha reducido el tiempo de ejecución a 4496.93. Podemos observar que la parte de seq scan ha cambiado y ahora es heap scan (debido al índice, se crea un árbol y por eso cambia el plan de ejecución), y ha desaparecido la parte de Gather que era la más costosa.

```
Aggregate  (cost=4496.93..4496.94 rows=1 width=8)
  -> Nested Loop  (cost=1127.18..4496.92 rows=2 width=4)
       -> Bitmap Heap Scan on orders  (cost=1126.90..4480.32 rows=2 width=4)
            Recheck Cond: (totalamount > '100'::numeric)
            Filter: ((date_part('year'::text, (orderdate)::timestamp without time zone) = '2015'::double precision) AND (date_part('mo...
            -> Bitmap Index Scan on idx_totalamount  (cost=0.00..1126.90 rows=60597 width=0)
                 Index Cond: (totalamount > '100'::numeric)
       -> Index Only Scan using customers_pkey on customers  (cost=0.29..8.30 rows=1 width=4)
```

Apartado E

Hemos probado también a añadir un índice sobre orderdate, pero hemos observado que no cambia los tiempos de búsqueda. Esto se debe a que realmente nosotros tenemos que ir extrayendo el mes y el año de cada fecha, para compararlos con los argumentos. Esto obliga a tener que ir analizando las fechas, por lo que el índice no tiene ningún efecto.

Estudio del impacto de preparar sentencias SQL

Apartados A, B y C

Hemos añadido una sentencia if-else en la función getListaCliMes, que si use-prepare es True ejecute un prepare de la consulta, el bucle llamando a execute pasándole los parámetros y, finalmente, un deallocate. En caso contrario, en el bucle ejecutará la consulta normal. Ambos bucles paran si break0 es True y se ha llegado a 0 clientes.

Apartado D

En primer lugar hemos realizado varias pruebas sin índices, probando para distintos intervalos para Abril de 2015 entre 300 y 100. Se adjunta una comparativa para intervalos de 5 y que no pare si no hay clientes:

5175	0
5180	0
5185	0
5190	0
5195	0
5200	0
5205	0
5210	0
5215	0
5220	0
5225	0
5230	0
5235	0
5240	0
5245	0
5250	0
5255	0
5260	0
5265	0
5270	0
5275	0
5280	0
5285	0
5290	0
5295	0

Tiempo: 31725 ms

Nueva consulta

5185	0
5190	0
5195	0
5200	0
5205	0
5210	0
5215	0
5220	0
5225	0
5230	0
5235	0
5240	0
5245	0
5250	0
5255	0
5260	0
5265	0
5270	0
5275	0
5280	0
5285	0
5290	0
5295	0

Tiempo: 31491 ms

Usando prepare

Nueva consulta

Podemos ver que la salida sin prepare tarda 31725 ms, mientras que con prepare tarda 31491 ms. En el resto de pruebas también hemos visto una mejora usando prepare.

Hemos realizado pruebas tras crear el índice usando los mismos datos que en el caso anterior. Para intervalos de 5 obtenemos lo siguiente:

1285	0
1286	0
1287	0
1288	0
1289	0
1290	0
1291	0
1292	0
1293	0
1294	0
1295	0
1296	0
1297	0
1298	0
1299	0

Tiempo: 696 ms

Nueva consulta

1283	0
1284	0
1285	0
1286	0
1287	0
1288	0
1289	0
1290	0
1291	0
1292	0
1293	0
1294	0
1295	0
1296	0
1297	0
1298	0
1299	0

Tiempo: 669 ms

Usando prepare

Comprobamos, de nuevo, que el uso de prepare mejora el rendimiento. Además, al haber usado índices ahora tardan menos las dos consultas respecto a las pruebas anteriores. Finalmente, tras ejecutar analyze y probar de nuevo las consultas, para intervalos de 3 vemos los siguientes resultados:

3261	0
3264	0
3267	0
3270	0
3273	0
3276	0
3279	0
3282	0
3285	0
3288	0
3291	0
3294	0
3297	0

Tiempo: 737 ms

Nueva consulta

3261	0
3264	0
3267	0
3270	0
3273	0
3276	0
3279	0
3282	0
3285	0
3288	0
3291	0
3294	0
3297	0

Tiempo: 684 ms

Usando prepare

Nueva consulta

El prepare, de nuevo, es más óptimo. Sin embargo, para otros intervalos hemos visto que la diferencia de tiempos no era tan grande como en otros casos. Esto se debe a que analyze crea sus optimizaciones para las consultas, como explicaremos más adelante en el apartado dedicado a este tema.

El uso de sentencias preparadas puede empeorar el rendimiento si se trata de consultas que se van a ejecutar pocas veces, ya que el tiempo necesario para preparar la sentencia puede ser mayor en estos casos respecto al tiempo que va a tardar la consulta. Se recomienda su uso cuando vamos a tener que realizar una consulta muchas veces y con diferentes parámetros, además de para consultas complejas que involucren varias tablas (join) y condiciones (and).

Estudio del impacto de cambiar la forma de realizar consultas

Apartado A

Para la primera consulta obtenemos el siguiente análisis. Vemos que utiliza hashed SubPlan para hacer el filtro NOT: primero recorre toda la tabla customers para ver los customerid (por lo tanto devuelve resultados nada más ejecutarse) y después filtra aquellos que no están en la tabla devuelta por la subconsulta. Para la subconsulta recorre toda la tabla orders y obtiene las que tienen un estado Paid.

```
Seq Scan on customers  (cost=3961.65..4490.81 rows=7046 width=4)
  Filter: (NOT (hashed SubPlan 1))
  SubPlan 1
    ->  Seq Scan on orders  (cost=0.00..3959.38 rows=909 width=4)
          Filter: ((status)::text = 'Paid'::text)
```

Para la segunda consulta obtenemos el siguiente análisis. Lo primero que hace es juntar las tablas customers y orders (en la que aplica el filtro del estado) mediante Append, sin eliminar duplicados. Para ello debe recorrer ambas tablas enteras. Finalmente, debe agrupar los resultados y comprobar el valor de count en cada uno para quedarse con los que no estén duplicados. También hemos visto con explain analyze que esta es la que tarda más tiempo real en ejecutarse.

```
HashAggregate  (cost=4537.41..4539.41 rows=200 width=4)
  Group Key: customers.customerid
  Filter: (count(*) = 1)
  ->  Append  (cost=0.00..4462.40 rows=15002 width=4)
        ->  Seq Scan on customers  (cost=0.00..493.93 rows=14093 width=4)
        ->  Seq Scan on orders  (cost=0.00..3959.38 rows=909 width=4)
              Filter: ((status)::text = 'Paid'::text)
```

Para la tercera consulta obtenemos el siguiente análisis. En este caso, el uso de except (HashSetOp) elimina duplicados, no como en el caso anterior que había que hacerlo a mano. La consulta crea dos subconsultas, una para la tabla customers y otra para orders, las combina e internamente decide cuáles debe eliminar por el except. Esto podría tardar menos si se paraleliza al tratarse de dos subqueries.

```
HashSetOp Except  (cost=0.00..4640.83 rows=14093 width=8)
  ->  Append  (cost=0.00..4603.32 rows=15002 width=8)
        ->  Subquery Scan on "*SELECT* 1"  (cost=0.00..634.86 rows=14093 width=8)
              ->  Seq Scan on customers  (cost=0.00..493.93 rows=14093 width=4)
        ->  Subquery Scan on "*SELECT* 2"  (cost=0.00..3968.47 rows=909 width=8)
              ->  Seq Scan on orders  (cost=0.00..3959.38 rows=909 width=4)
                    Filter: ((status)::text = 'Paid'::text)
```

Estudio del impacto de la generación de estadísticas

Apartado A y B

Primero hemos vuelto a crear la base de datos y hemos estudiado las dos consultas. Para la primera, obtenemos la siguiente salida:

```
Aggregate  (cost=3507.17..3507.18 rows=1 width=8) (actual time=22.245..22.245 rows=1 loops=1)
  ->  Seq Scan on orders  (cost=0.00..3504.90 rows=909 width=0) (actual time=22.229..22.229 rows=0 loops=1)
        Filter: (status IS NULL)
        Rows Removed by Filter: 181790
Planning time: 0.062 ms
Execution time: 22.275 ms
```

Y para la segunda, esta otra:

```
Aggregate  (cost=3961.65..3961.66 rows=1 width=8) (actual time=45.864..45.864 rows=1 loops=1)
  ->  Seq Scan on orders  (cost=0.00..3959.38 rows=909 width=0) (actual time=0.013..35.115 rows=127323 loops=1)
        Filter: ((status)::text = 'Shipped'::text)
        Rows Removed by Filter: 54467
Planning time: 0.070 ms
Execution time: 45.905 ms
```

Podemos observar que ambas consultas parecen realizar la misma secuencia de pasos para obtener el resultado (recorrido de tabla entera, filtrado de resultados y agregación), teniendo como única diferencia el filtro que aplican sobre la tabla orders (la primera se queda los que tengan el status a NULL y la segunda, a Shipped). La segunda, debido a esta condición, tarda más que la primera en ejecutarse. Esto se debe a que en la primera consulta simplemente comprueba que una columna no tenga valor, es decir, esté a NULL, mientras que la segunda consulta debe comprobar que todo el texto de la columna coincida con “Shipped”.
Además, vemos que la primera consulta no devuelve en el recorrido secuencial ninguna fila, porque en la tabla no hay ninguna order con estado NULL.

Apartado C y D

Tras crear un índice en la tabla orders en la columna status (create index idx_status on orders(status);), podemos ver cómo han cambiado ambas salidas. En la primera consulta obtenemos:

```
Aggregate  (cost=1496.52..1496.53 rows=1 width=8) (actual time=0.032..0.033 rows=1 loops=1)
 ->  Bitmap Heap Scan on orders  (cost=19.46..1494.25 rows=909 width=0) (actual time=0.030..0.030 rows=0 loops=1)
       Recheck Cond: (status IS NULL)
       ->  Bitmap Index Scan on idx_status  (cost=0.00..19.24 rows=909 width=0) (actual time=0.029..0.029 rows=0 loops=1)
             Index Cond: (status IS NULL)
Planning time: 0.162 ms
Execution time: 0.063 ms
```

Y en la segunda:

```
Aggregate  (cost=1498.79..1498.80 rows=1 width=8) (actual time=48.398..48.398 rows=1 loops=1)
 ->  Bitmap Heap Scan on orders  (cost=19.46..1496.52 rows=909 width=0) (actual time=15.381..37.499 rows=127323 loops=1)
       Recheck Cond: ((status)::text = 'Shipped'::text)
       Heap Blocks: exact=1687
       ->  Bitmap Index Scan on idx_status  (cost=0.00..19.24 rows=909 width=0) (actual time=15.072..15.072 rows=127323 loops=1)
             Index Cond: ((status)::text = 'Shipped'::text)
Planning time: 0.115 ms
Execution time: 48.449 ms
```

Vemos una gran reducción en el coste de la primera, que era lo esperable tras crear un índice en esa columna. Sin embargo, la segunda consulta predice una mejoría pero realmente empeora. Esta diferencia puede ser porque en el caso de la primera consulta no hay ningún valor en la tabla que sea NULL, pero en la segunda sí que hay, y además observamos que con el índice la condición de status se debe comprobar dos veces.

Apartado E y F

Tras realizar la sentencia analyze sobre la tabla orders, obtenemos los siguientes cambios en la primera consulta:

```
Aggregate  (cost=7.26..7.27 rows=1 width=8) (actual time=0.014..0.015 rows=1 loops=1)
 ->  Index Only Scan using idx_status on orders  (cost=0.42..7.26 rows=1 width=0) (actual time=0.010..0.010 rows=0 loops=1)
       Index Cond: (status IS NULL)
       Heap Fetches: 0
Planning time: 0.139 ms
Execution time: 0.055 ms
```

Y en la segunda:

```
Finalize Aggregate  (cost=4210.21..4210.22 rows=1 width=8) (actual time=34.264..34.265 rows=1 loops=1)
 -> Gather  (cost=4210.10..4210.21 rows=1 width=8) (actual time=34.168..38.208 rows=2 loops=1)
      Workers Planned: 1
      Workers Launched: 1
      -> Partial Aggregate  (cost=3210.10..3210.11 rows=1 width=8) (actual time=29.448..29.448 rows=1 loops=2)
           -> Parallel Seq Scan on orders  (cost=0.00..3023.69 rows=74562 width=0) (actual time=0.013..23.133 rows=63662 loops=2)
                Filter: ((status)::text = 'Shipped'::text)
                Rows Removed by Filter: 27234
Planning time: 0.080 ms
Execution time: 38.247 ms
```

La sentencia analyze genera estadísticas sobre la tabla orders para optimizar los planes de ejecución de las consultas que se hagan sobre ella. De esta forma, obtiene una muestra de la tabla orders (un conjunto de filas) sobre la que trabajar. En esta muestra puede no haber ciertos datos que necesitemos, así que el gestor de la base de datos decidirá si usar la muestra o usar la tabla real en función de si el dato solicitado está en la muestra o no.
Por este motivo, el plan de ejecución de ambas consultas ha cambiado según la información que ha recogido el gestor sobre la tabla. En la primera, vemos que se ha utilizado el índice que habíamos creado (Index Only Scan), por lo que está trabajando sobre la tabla real (la muestra no contiene el índice y tampoco contiene valores NULL).
En cambio, para la segunda consulta vemos que no está usando el índice, lo que quiere decir que está trabajando sobre la muestra, cosa que tiene sentido ya que según las consultas que hemos hecho hasta ahora, sólo hemos estado obteniendo valores Shipped. Además, el plan de ejecución para ella es distinto a los casos anteriores y se ha conseguido mejorar bastante el tiempo que tarda la consulta. Aun así, tiene sentido que tarde más que la primera (y que las otras dos que analizaremos en el siguiente apartado) ya que se trata del estado más común en la tabla, y por tanto afecta a un gran número de filas.

Apartado G

Analizando ahora la tercera consulta obtenemos:

```
Aggregate  (cost=2336.98..2336.99 rows=1 width=8) (actual time=8.886..8.887 rows=1 loops=1)
 -> Bitmap Heap Scan on orders  (cost=369.39..2290.22 rows=18706 width=0) (actual time=2.405..7.445 rows=18163 loops=1)
      Recheck Cond: ((status)::text = 'Paid'::text)
      Heap Blocks: exact=1687
      -> Bitmap Index Scan on idx_status  (cost=0.00..364.71 rows=18706 width=0) (actual time=2.066..2.067 rows=18163 loops=1)
           Index Cond: ((status)::text = 'Paid'::text)
Planning time: 0.082 ms
Execution time: 8.973 ms
```

Y para la cuarta consulta obtenemos:

```
Aggregate  (cost=2949.88..2949.89 rows=1 width=8) (actual time=22.969..22.972 rows=1 loops=1)
 -> Bitmap Heap Scan on orders  (cost=717.96..2859.06 rows=36328 width=0) (actual time=4.380..18.332 rows=36304 loops=1)
      Recheck Cond: ((status)::text = 'Processed'::text)
      Heap Blocks: exact=1687
      -> Bitmap Index Scan on idx_status  (cost=0.00..708.88 rows=36328 width=0) (actual time=4.113..4.113 rows=36304 loops=1)
           Index Cond: ((status)::text = 'Processed'::text)
Planning time: 0.083 ms
Execution time: 23.055 ms
```

El plan de ejecución para ambas consultas es el mismo, que coincide con el del status NULL. La cuarta consulta tarda más debido a que afecta a más filas que la tercera. Se observa que ambas consultas están ejecutándose sobre la tabla real y no sobre la muestra, ya que esta no contiene estos valores por ser la primera vez que ejecutamos las consultas y las estadísticas se generaron previamente.

Transacciones y deadlocks

Estudio de transacciones

Hemos modificado routes.py para que obtenga los valores de los parámetros mediante request.args en vez de request.form, por ser un método GET. Hemos implementado la funcionalidad pedida en database.py. También hemos modificado el template proporcionado para que no muestre si ejecutar las sentencias con SQL o con SQLAlchemy, ya que al ser una parte opcional que iba a llevar mucho trabajo (al tener que mapear toda la base de datos para utilizar la interfaz SQLAlchemy) hemos decidido no realizarla.

Para mostrar el progreso que va teniendo la ejecución de las consultas se ha usado el array de mensajes dbr. En este array vamos añadiendo cuántas filas se ven afectadas por los borrados durante la transacción y sobre qué tablas (esto se ha realizado mediante el atributo rowcount del objeto ResultProxy que se devuelve tras hacer una query en SQLAlchemy). También mostramos el resultado de realizar selects para ese customer id cuando, tras hacer el rollback, queremos ver que las tablas han vuelto a su estado original, y también durante el proceso de borrado. De esta forma, vemos tanto el número de filas afectadas como el borrado efectivo en la tabla.
Se han realizado las consultas en órdenes distintos según el parámetro bFallo fuera True o False. Para que no haya error de integridad, habiendo eliminado las restricciones CASCADE de la base de datos, es necesario ejecutar los borrados en un orden incorrecto. Primero ejecutamos el borrado sobre orderdetail, que no va a dar problemas porque no contiene valores que vayan a ser referenciados mediante claves externas desde otras tablas (el borrado se ejecutará correctamente, y en la traza veremos que, efectivamente, se han borrado los datos). Después intentaremos borrar de la tabla customers, y esto va a producir el error de integridad, ya que en la tabla orders se está referenciando a esta tabla. En ese momento, ocurrirá el rollback y los borrados ejecutados hasta entonces se revertirán (las trazas mostrarán que las filas borradas ahora se pueden recuperar de la tabla, como si no hubiese ocurrido nada). En este punto, si se realizó un commit intermedio, el borrado de orderdetail no se revierte. Esto es correcto y se debe a que se está cumpliendo la propiedad de aislamiento de las bases de datos: las operaciones dentro de una transacción son independientes y no afectan a las de otras transacciones (tras hacer un commit, se crea una transacción nueva, o sino el rollback o commit posterior daría un error). Si no queremos provocar el error, simplemente borraremos primero los datos de orders y después los de customers, y ya se podrá realizar commit correctamente al no haber más tablas que referencian a customers.

Cuando aparece una situación NOT FOUND (es decir, no hay datos para ese customerid), no hacemos nada para evitarlo, ya que cuando intentemos borrar de la base de datos registros relativos a un usuario que no tiene compras realizadas simplemente no se va a borrar nada. En este caso, en dbr se mostrará que las filas afectadas han sido 0.

Estudio de bloqueos y deadlocks

Apartados A-C

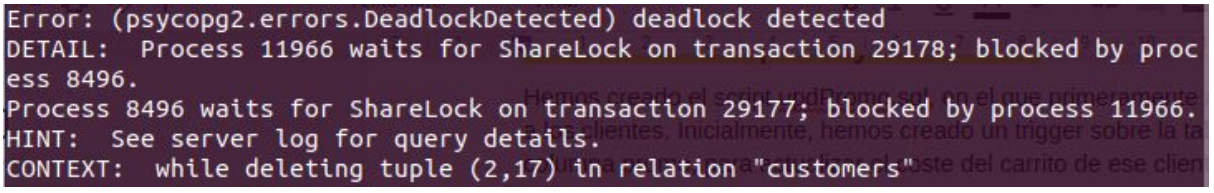
Hemos creado el script updPromo.sql, en el que primeramente añadimos la columna promo a los clientes. Inicialmente, hemos creado un trigger sobre la tabla customers sobre la columna promo, para actualizar el coste del carrito de ese cliente cada vez que se actualice su promo (para ello, calcularemos cuál era el precio inicial usando old.promo, pues no queremos actualizar sobre un precio que ya tiene un descuento, y calcularemos el precio con el nuevo descuento usando new.promo).

Apartados A-J

Este script lo hemos modificado para añadir un sleep al trigger. También hemos añadido un sleep en el código de la transacción que estamos haciendo en database.py. Hemos ido modificando tanto el tiempo que dura el sleep como el lugar en el que colocarlo, probando con sleeps de 0 y hasta 30 segundos. Finalmente, hemos decidido usar 5 segundos.

La transacción de borrado del cliente accede primero a orderdetail, luego a orders y luego a customers. La transacción de actualización accede primero a customers (pues ocurre cuando hay un update en esta tabla) y luego a orders. Por tanto, las tablas orders y customers son las que intervendrán en el deadlock.

Hemos probado a poner el sleep del trigger al final del mismo, tras el update, y el sleep del código antes de borrar un cliente. Se ha generado un deadlock en la página que borra el cliente, y lo hemos podido detectar porque la transacción hacía un rollback y hemos imprimido la excepción que lo causaba en la consola:



En este caso, la transacción de borrado tiene locks sobre orderdetail y orders, y espera. Mientras, la transacción de actualización tiene un lock sobre customers, y al entrar al trigger espera a que la tabla orders se libere. Cuando acabe el primer sleep, la primera transacción tiene el lock sobre orders y quiere un lock sobre customers, y la segunda tiene un lock sobre customers y quiere uno sobre orders. Esto produce el deadlock, y la primera transacción se aborta.

Si el sleep, en cambio, lo colocamos tras el delete, y en el trigger al principio, entonces el deadlock saltaría en la segunda transacción:

```
ERROR:  deadlock detected
DETALLE:  Process 9696 waits for ShareLock on transaction 28978; blocked by process 13634.
Process 13634 waits for ShareLock on transaction 28977; blocked by process 9696.
SUGERENCIA:  See server log for query details.
CONTEXTO:  while updating tuple (217,42) in relation "orders"
SQL statement "update orders
        set totalamount=(totalamount/((100-old.promo)/100))*((100-new.promo)/100)
        where customerid = new.customerid"
PL/pgSQL function aplicarpromo() line 4 at SQL statement
SQL state: 40P01
Detail: Process 9696 waits for ShareLock on transaction 28978; blocked by process 13634.
```

Apartado H

Los datos alterados y, por tanto, los mensajes que pongamos entre medias de la transacción no son visibles hasta que esta acabe debido a la propiedad de atomicidad de las bases de datos. Esta exige que, en una transacción, o se ejecutan todas las operaciones o ninguna. Los mensajes que ponemos entre medias realmente se muestran una vez acabada la transacción. Ahora, por haber añadido los sleeps, se tarda más en ejecutar la transacción, por eso hasta que no se completa no vemos los mensajes.

Apartado K

Para evitar la aparición de deadlocks, lo mejor será que hagamos las transacciones lo más cortas posible, ya que una transacción larga retiene los recursos por más tiempo.
En ocasiones en las que esto no sea posible, se debe tratar de ejecutar las operaciones de cada transacción en un orden consistente respecto a las otras, para anticiparnos a la situación de deadlock.
El uso de índices también reduce el riesgo, ya que gracias a ellos la transacción recorrerá menos filas y, por tanto, creará menos locks.

Seguridad

Acceso indebido a un sitio web

Apartado A

Para hacer login conociendo solamente el nombre de usuario, inyectando SQL en el campo contraseña podemos invalidar la query que realiza el gestor y hacer que realice la que nosotros queramos. Si en el campo contraseña escribimos **'; select * from customers where username='gatsby**, la consulta del gestor para buscar usuario y contraseña va a fallar porque no hay usuarios con la clave vacía (contraseña = ‘ ‘). Sin embargo, como después estamos ejecutando una query que nos devuelve todos los campos asociados a ese nombre de usuario, no nos ha hecho falta la contraseña para hacer login.

Apartado B

Para hacer login sin conocer ni usuario ni contraseña, podemos dejar en blanco el campo de nombre e insertar lo siguiente en el campo de contraseña: **' or 0=0--**

Esta sentencia deja vacío el campo de contraseña y pone una condición que siempre es verdadera, comentando todo lo que venga detrás para evitar el error de cierre de query (el gestor cierra con comilla el texto del campo de contraseña, y como nosotros ya lo hemos cerrado a mano no queremos que lo vuelva a hacer). Así, mientras que la comprobación de usuario y contraseña da error, 0=0 da siempre True y, por precedencia de operadores (AND tiene prioridad sobre OR, por tanto nuestra condición va a ser siempre verdadera, independientemente de lo que ocurra antes en la consulta), se va a mostrar los datos de todos los usuarios del sistema y a coger la primera fila, logueando al primer usuario que aparezca.

Apartado C

La vulnerabilidad del sistema reside en que se pasa al gestor de bases de datos directamente el texto que haya introducido el usuario en los campos del formulario, sin hacer ningún tipo de comprobación. Según la query introducida por el usuario, se podría hasta borrar la base de datos.

Para evitar esta vulnerabilidad, en vez de querys embebidas tal cual en la aplicación podríamos usar sentencias precompiladas, las cuales aceptan parámetros que podrían ser el nombre y la contraseña, y ya evita que el usuario pueda inyectar SQL. También podríamos usar el propio Python para evitar este problema, pues permite que en vez de recibir strings como parámetros para una consulta usemos “query parameters”, que analizan el tipo de parámetro esperado y escapan caracteres del argumento, para evitar que venga directamente del cliente y pueda tener malas intenciones.

Acceso indebido a información

Apartado B

Conociendo únicamente que la base de datos utiliza PostgreSQL, sabemos que habrá una tabla pg_class que almacena todas las tablas, índices y estructuras similares a tablas de la base. Sabemos también que esta tabla tiene un campo relname con el nombre de la tabla. Si intentamos hacer como en el apartado anterior, ; select relname from pg_class;--, vemos que nos devuelve una lista con tantos elementos como la consulta nos devolvería, pero sin su nombre. Esto se debe a que el servidor está teniendo en cuenta el nombre de la columna que espera recibir de la tabla movies. Para evitar esto, podemos unir al resultado de la búsqueda por año el resultado de nuestra consulta. Union utiliza el nombre de la columna del primer conjunto, por lo tanto ahora sí se nos muestran los resultados deseados. La cadena a introducir sería **' union select relname from pg_class;--**

Apartado C

Si queremos filtrar todos estos resultados para obtener los que pertenezcan al esquema public, podemos usar la cadena **' union select relname from pg_class where relnamespace = (select oid from pg_namespace where nspname = 'public');--**

Apartados D y E

Dado el resultado de la consulta anterior, podemos ver que la tabla que contiene la información deseada es customers. Para obtener su oid, usaremos la consulta **' union select cast(oid as varchar) as oid from pg_class where relname='customers';--** porque union exige que las columnas de ambas consultas sean del mismo tipo.

Apartados F y G

Conociendo la tabla que se debe consultar y que el campo que tiene el nombre de cada columna es attname, ejecutando la consulta **' union select attname from pg_attribute where attrelid=(select oid from pg_class where relnamespace = (select oid from pg_namespace where nspname = 'public') and relname='customers');--** podemos obtener todas las columnas de esa tabla. Las columnas que necesitamos son username y password.

Apartado H

Sacaremos el nombre de usuario y la contraseña de todos los clientes, ambos concatenados y separados por |_| para mejor visualización: **' union select username || '|_|' || password as name_pass from customers;--**

Apartado I

Utilizar Combobox no soluciona el problema, ya que este valor puede modificarse también una vez seleccionado. Lo que sí se podría hacer es, antes de ejecutar ninguna consulta, en el servidor, comprobar que el valor que se ha recibido es uno de los que pertenecían a las opciones del Combobox.

Utilizar POST en lugar de GET no cambia el hecho de que se pueda inyectar código, como hemos visto en el caso de xLoginInjection. Sí que proporciona algo más de seguridad POST respecto a GET, ya que los parámetros no van en la URL, pero no es una mejora significativa, puesto que hay muchas otras formas de seguir inyectando código.

Otra forma de mejorar esta vulnerabilidad sería realizar validaciones en el cliente para permitir sólo campos numéricos o de una longitud máxima mediante expresiones regulares (aunque no debemos fiarnos de esto, pues el cliente puede introducir a mano una URL fraudulenta). Además, podríamos limpiar el input del usuario en el servidor, escapando ciertos caracteres. Finalmente, lo mejor que podemos hacer es utilizar sentencias precompiladas, pues estas permiten establecer que los parámetros se ajusten a un tipo de datos concreto.