SISTEMAS INFORMÁTICOS

PRÁCTICA 4

Alejandro Pascual[1] y Víctor Yrazusta[2]

Índice

[1] alejandro.pascualp@estudiante.uam.es
[2] victor.yrazusta@estudiante.uam.es

1. OPTIMIZACIÓN

A. Estudio del impacto de un índice

Primero hemos creado la sentencia para obtener contar el número de usuarios.

```
SELECT COUNT(*)
FROM (
    SELECT DISTINCT
        o.customerid AS customer_id
    FROM orders AS o
    WHERE
        o.totalamount>100
        AND EXTRACT(YEAR FROM o.orderdate)='2015'
        AND EXTRACT(MONTH FROM o.orderdate)='04'
) AS filtered_customers;
```

Cuyo EXPLAIN nos da como resultado:

```
Aggregate (cost=6816.97..6816.98 rows=1 width=8)
 -> Unique (cost=6816.93..6816.94 rows=2 width=4)
     -> Sort (cost=6816.93..6816.94 rows=2 width=4)
        Sort Key: o.customerid
         -> Gather (cost=1000.00..6816.92 rows=2 width=4)
            Workers Planned: 1
             -> Parallel Seq Scan on orders o
                (cost=0.00..5816.72 rows=1 width=4)
                Filter: (...)
```

Hemos decidido crear un índice sobre el totalamount de orders, para que el filtro se aplique con mayor eficiencia.

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

El resultado del EXPLAIN tras la creación del índice es:

```
Aggregate (cost=5669.36..5669.37 rows=1 width=8)
 -> Unique (cost=5669.33..5669.34 rows=2 width=4)
     -> Sort (cost=5669.33..5669.33 rows=2 width=4)
        Sort Key: o.customerid
         -> Bitmap Heap Scan on orders o (cost=1126.9..5669.32 rows=2 width=4)
            Recheck Cond: (totalamount > '100'::numeric)
            Filter: (...)
             -> Bitmap Index Scan on index_orders_totalamount
                (cost=0.00..1126.90 rows=60597 width=0)
                Index Cond: (totalamount > '100'::numeric)
```

Podemos observar como el coste es algo menor tras la incorporación de este índice. Se puede apreciar que la reducción en el coste viene del siguiente fragmento:

```
Parallel Seq Scan on orders o (cost=0.00..5816.72 rows=1 width=4)
```

Que tras la creación del índice pasa a ser:

```
Bitmap Index Scan on index_orders_totalamount
(cost=0.00..1126.90 rows=60597 width=0)
Index Cond: (totalamount > '100'::numeric)
```

Este resultado encaja con nuestro objetivo al crear el índice, ya que ya no es necesario realizar un costoso escaneo secuencial y aplicar los filtros a cada uno de los casos, si no que se aprovecha el índice para localizar la sección de pedidos relevantes (con un totalamount mayor que 100).

Con un objetivo similar hemos creado un índice sobre orderdate de orders.

```
CREATE INDEX index_orders_orderdate ON orders(orderdate);
```

Pero, en esta ocasión, no se ha podido aprovechar el índice y se sigue realizando un escaneo secuencial, por lo que el resultado del EXPLAIN es el mismo que sin el índice. Creemos que esto se debe a que resulta más complejo detectar y optimizar el filtro para unos valores concretos como son el mes y el año, que para un rango numérico como el establecido por el filtro por totalamount.

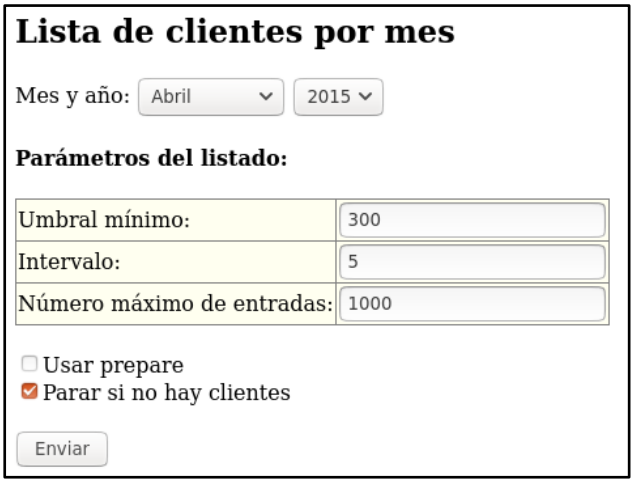
También hemos probado con un índice sobre el customerid de orders, ya que se puede apreciar que se utiliza como clave de ordenación para realizar el filtro DISTINCT.

```
CREATE INDEX index_orders_customerid ON orders(customerid);
```

Sin embargo, este índice tampoco ha tenido ningún impacto y el EXPLAIN es, de nuevo, idéntico al que obtenemos sin el índice.

B. Estudio del impacto de preparar sentencias SQL

Primero hemos terminado de programar la web y hemos comprobado que funciona.



Lista de clientes por mes

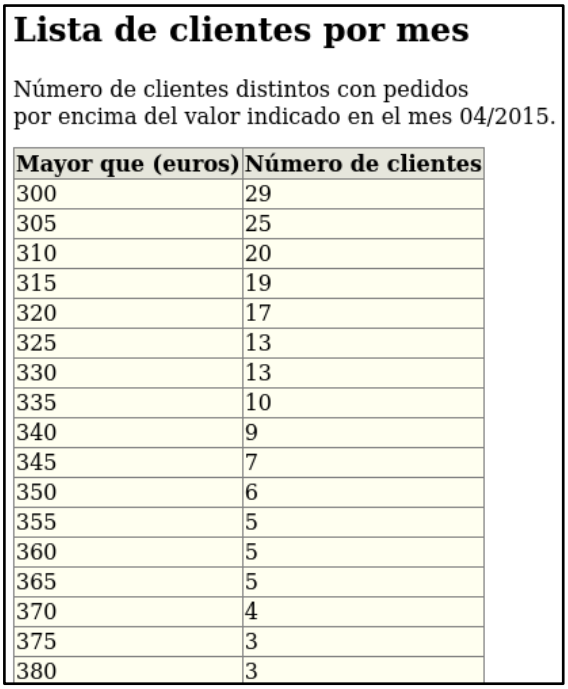
Mes y año: Abril 2015

Parámetros del listado:

Umbral mínimo:	300
Intervalo:	5
Número máximo de entradas:	1000

☐ Usar prepare
☑ Parar si no hay clientes

Enviar

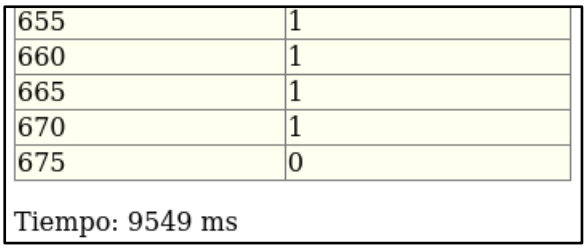


Lista de clientes por mes

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

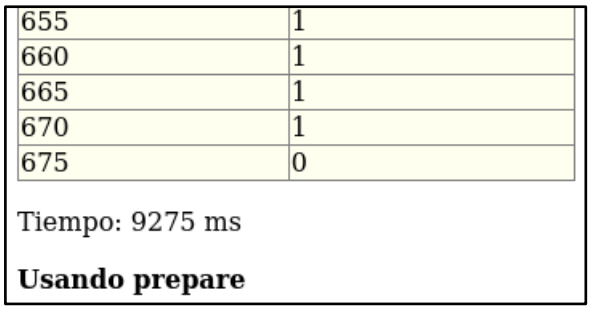
Mayor que (euros)	Número de clientes
300	29
305	25
310	20
315	19
320	17
325	13
330	13
335	10
340	9
345	7
350	6
355	5
360	5
365	5
370	4
375	3
380	3

Después hemos ejecutado las pruebas con la base de datos limpia. Tanto con el prepare como sin él las consultas tienen unos tiempos de ejecución similares.



655	1
660	1
665	1
670	1
675	0

Tiempo: 9549 ms

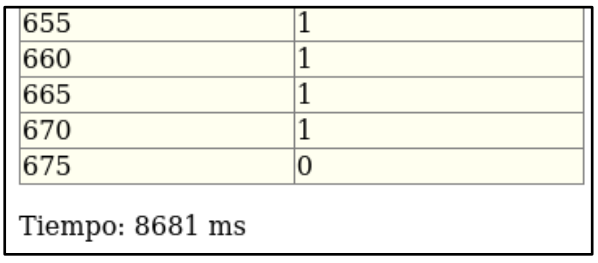


655	1
660	1
665	1
670	1
675	0

Tiempo: 9275 ms

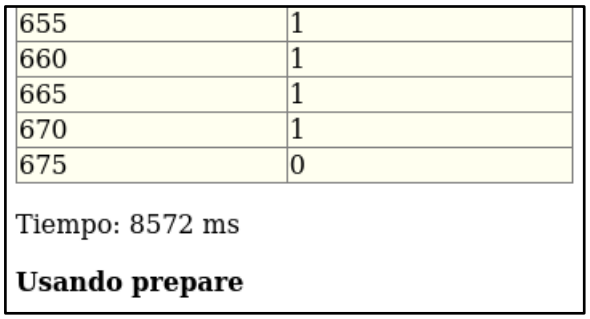
Usando prepare

Tras aplicar el índice y ejecutar ANALYZE, hemos visto una pequeña mejora:



655	1
660	1
665	1
670	1
675	0

Tiempo: 8681 ms



655	1
660	1
665	1
670	1
675	0

Tiempo: 8572 ms

Usando prepare

Hemos comprobado que para intervalos muy grandes (por lo tanto, muy pocas entradas en la tabla y pocas consultas) el uso de prepare ralentiza la operación.

C. Estudio del impacto de cambiar la forma de realizar una consulta

Primero hemos obtenido los planes de ejecución de las tres opciones.

Opción 1:

```
Seq Scan on customers (cost=3639.57..4168.73 rows=7046 width=4)
Filter: (NOT (hashed SubPlan 1))
SubPlan 1
 -> Seq Scan on orders (cost=0.00..3594.38 rows=18076 width=4)
    Filter: ((status)::text = 'Paid'::text)
```

Opción 2:

```
HashAggregate (cost=4429.91..4431.91 rows=200 width=4)
Group Key: customers.customerid
Filter: (count(*) = 1)
 -> Append (cost=0.00..4269.07 rows=32169 width=4)
     -> Seq Scan on customers  (cost=0.00..493.93 rows=14093 width=4)
     -> Seq Scan on orders  (cost=0.00..3594.38 rows=18076 width=4)
        Filter: ((status)::text = 'Paid'::text)
```

Opción 3:

```
HashSetOp Except (cost=0.00..4490.42 rows=14093 width=8)
 -> Append (cost=0.00..4409.99 rows=32169 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..3775.14 rows=18076 width=8)
         -> Seq Scan on orders (cost=0.00..3594.38 rows=18076 width=4)
            Filter: ((status)::text = 'Paid'::text)
```

La segunda y tercera consulta van concatenando resultados en función de los valores obtenidos de las subconsultas, por lo que tiene sentido que devuelvan al inicio la tabla vacía y vayan agregando los resultados. La primera, que se tiene que ejecutar en orden secuencial tras la subconsulta, no parece devolver nada al inicio.

La segunda y tercera consulta (usando UNION y EXCEPT) tiene dos subconsultas completamente independientes entre sí, por lo que se pueden realizar en paralelo. Esto es coherente con los operadores de conjuntos, cuyas partes son independientes entre sí. En la primera consulta, pese a existir una subconsulta, solo se puede ejecutar en serie con la principal, ya que la principal es dependiente de ella.

D. Estudio del impacto de la generación de estadísticas

Resultados de EXPLAIN en ambas alternativas en la base de datos limpia.

Opción 1:

```
Aggregate (cost=3139.91..3139.93 rows=1 width=8)
 -> Seq Scan on orders (cost=0.00..3139.90 rows=6 width=0)
    Filter: (status IS NULL)
```

Opción 2:

```
Finalize Aggregate (cost=3845.90..3845.91 rows=1 width=8)
 -> Gather (cost=3845.78..3845.89 rows=1 width=8)
    Workers Planned: 1
     -> Partial Aggregate (cost=2845.78..2845.79 rows=1 width=8)
         -> Parallel Seq Scan on orders (cost=0.0..2658.69 rows=74837 width=0)
            Filter: ((status)::text = 'Shipped'::text)
```

Añadimos el índice sobre el status de orders:

```
CREATE INDEX index_orders_status ON orders(status);
```

Resultados de EXPLAIN en ambas alternativas tras la creación del índice:

Opción 1:

```
Aggregate (cost=39.35..39.36 rows=1 width=8)
 -> Index Only Scan using index_orders_status on orders
    (cost=0.42..39.32 rows=12 width=0)
    Index Cond: (status IS NULL)
```

Opción 2:

```
Finalize Aggregate (cost=3845.90..3845.91 rows=1 width=8)
 -> Gather (cost=3845.78..3845.89 rows=1 width=8)
    Workers Planned: 1
     -> Partial Aggregate (cost=2845.78..2845.79 rows=1 width=8)
         -> Parallel Seq Scan on orders (cost=0.0..2658.69 rows=74837 width=0)
            Filter: ((status)::text = 'Shipped'::text)
```

Podemos ver cómo, para el filtro IS NULL, PostgreSQL es capaz de aprovechar el índice para optimizar muchísimo la consulta. Sin embargo, en el caso de comparar status con ‘Shipped’, no es capaz de mejorar el resultado. Esto nos hace confiar más en nuestra teoría del apartado A, en la que ya observamos como no se optimizaban los filtros con operaciones más complejas como comparaciones con valores concretos.

Ejecutamos ANALYZE sobre orders:

```
ANALYZE orders;
```

Resultados de EXPLAIN en las cuatro alternativas tras ejecutar ANALYZE:

Opción 1:

```
Aggregate (cost=22.63..22.64 rows=1 width=8)
 -> Index Only Scan using index_orders_status on orders
    (cost=0.42..22.62 rows=6 width=0)
    Index Cond: (status IS NULL)
```

Opción 2:

```
Finalize Aggregate (cost=3845.60..3845.61 rows=1 width=8)
 -> Gather (cost=3845.49..3845.60 rows=1 width=8)
    Workers Planned: 1
     -> Partial Aggregate (cost=2845.49..2845.50 rows=1 width=8)
         -> Parallel Seq Scan on orders (cost=0.0..2658.69 rows=74719 width=0)
            Filter: ((status)::text = 'Shipped'::text)
```

Opción 3:

```
Aggregate (cost=1967.30..1967.31 rows=1 width=8)
 -> Bitmap Heap Scan on orders (cost=367.80..1921.05 rows=18500 width=0)
    Recheck Cond: ((status)::text = 'Paid'::text)
     -> Bitmap Index Scan on index_orders_status
        (cost=0.00..363.17 rows=18500 width=0)
        Index Cond: ((status)::text = 'Paid'::text)
```

Opción 4:

```
Finalize Aggregate (cost=3845.60..3845.61 rows=1 width=8)
 -> Gather (cost=3845.49..3845.60 rows=1 width=8)
    Workers Planned: 1
     -> Partial Aggregate (cost=2845.49..2845.50 rows=1 width=8)
         -> Parallel Seq Scan on orders (cost=0.0..2658.69 rows=74719 width=0)
            Filter: ((status)::text = 'Shipped'::text)
```

En la línea de los resultados anteriores, podemos seguir viendo como ninguna de las consultas que comparan status con un valor concreto hace uso del índice. Creemos que es especialmente destacable el hecho de que la opción 3 utiliza un acercamiento distinto a la hora de realizar la consulta a los de las opciones 2 y 4. Esto se debe a las optimizaciones de ANALYZE, que generan diferentes tipos de consultas para mejorar los tiempos de ejecución.

2. TRANSACCIONES Y DEADLOCKS

E. Estudio de transacciones

Primero, hemos completado la funcionalidad de la web y hemos comprobado que elimine correctamente la información en el caso sin errores.

Trazas

1. Se elimina orderdetail.
2. Se elimina orders.
3. Se elimina customers.
4. Se hace commit.

A continuación, hemos revisado que hace ROLLBACK en caso de encontrar un error de integridad:

Trazas

1. Se elimina orderdetail.
2. Se elimina orders.
3. Fallo en la eliminación. Se hace rollback.

Por último, hemos comprobado que, si se realiza un COMMIT intermedio antes de provocar el fallo, el ROLLBACK no deshace los cambios previos a dicho COMMIT.

Trazas

1. Se elimina orderdetail.
2. Se elimina orders.
3. Se elimina customers.
4. Se hace commit.

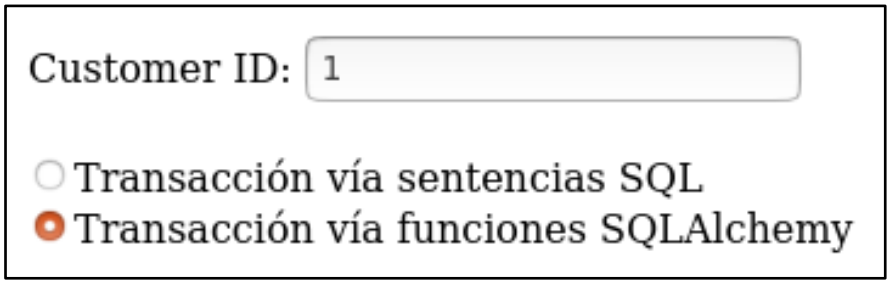
Hemos comprobado, mediante el uso de pgAdmin, que los resultados son correctos y coinciden con el estado real de la base de datos. Todos los COMMIT y ROLLBACK se ven reflejados correctamente.

El funcionamiento de todos estos casos es coherente con los mecanismos de las transacciones. Todas las comenzamos con un BEGIN. A continuación, realizamos las distintas consultas a la base de datos y, si alguna provoca un fallo, utilizamos ROLLBACK para deshacer las consultas previas a dicho fallo. Por último, realizamos COMMIT al final de la transacción, para que los cambios sean aplicados de manera definitiva.

El caso en el que realizamos un COMMIT intermedio, supone dividir el proceso en dos transacciones distintas. Por lo que, si falla la segunda, no se revierte la primera.

Respondiendo a la pregunta del enunciado, es necesario realizar un BEGIN tras el COMMIT ya que este comando cierra la transacción, por lo que es necesario abrir una nueva.

También hemos implementado estos mismos ejemplos con SQLAlchemy y hemos comprobado que se comporta de la misma forma.



F. Estudio de bloqueos y deadlocks

Primero, hemos creado el script solicitado.

```
-- Crear nueva columna promo
ALTER TABLE customers
ADD promo numeric;

-- Crear la función de actualización
CREATE OR REPLACE FUNCTION updPromoFunction()
RETURNS trigger AS $$
DECLARE
BEGIN
    IF NEW.promo=OLD.promo THEN
        return NEW;
    END IF;

    UPDATE orders AS o
    SET totalamount=ROUND(
        CAST(
            o.netamount*(100+o.tax)/100*(100-NEW.promo)/100
            AS numeric
        ),
        2
    )
    WHERE
        o.customerid=NEW.customerid
        AND o.status IS NULL
    ;

    RETURN NEW;
END $$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS updPromo
ON customers;

CREATE TRIGGER updPromo
AFTER UPDATE
ON customers
FOR EACH ROW
EXECUTE PROCEDURE updPromoFunction();
```

Después, hemos comprobado el correcto funcionamiento de este.

```
1  UPDATE orders SET status=NULL WHERE customerid=3;
2  UPDATE customers SET promo=50 WHERE customerid=3;
3  SELECT * FROM orders WHERE customerid=3;
```

Data Output

	orderid [PK] integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying (10)
1	116	2019-04-09	3	184.20	18	108.68	[null]
2	118	2016-04-19	3	113.55	15	65.29	[null]
3	119	2015-06-21	3	46.19	15	26.56	[null]
4	117	2015-05-14	3	80.37	15	46.21	[null]
5	120	2018-09-21	3	36.28	18	21.41	[null]

Se puede observar como el precio final (totalamount) es inferior al precio base (netamount). Esto se debe a que, pese a existir un impuesto del 15% o 18%, el usuario cuenta con una promoción del 50%.

Para generar el deadlock, hemos situado la espera entre los recursos presentes en ambos procesos. Es decir, entre el uso de las tablas customers y orders. En la eliminación en necesario borrar los elementos de orders antes de eliminar a un cliente. Por el contrario, en la actualización de la promoción de un cliente, el proceso comienza en la tabla customers y después el trigger trata de actualizar la tabla orders.

Aprovechando este cruce de sentidos, podemos generar el deadlock si logramos que coincidan en los puntos mencionados. Ajustando los tiempos hemos conseguido obtener el error por deadlock en pgAdmin:

```
ERROR:  deadlock detected
DETAIL:  Process 43738 waits for ShareLock on transaction 9130; blocked by process 56940.
Process 56940 waits for RowExclusiveLock on relation 33102 of database 33101; blocked by process 43738.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (51,32) in relation "orders"
SQL state: 40P01
```

Y en la interfaz web para borrar un cliente:

Trazas

1. Se elimina orderdetail.
2. Se elimina orders.
3. Se elimina customers.
4. Fallo en la eliminación. Se hace rollback.

Hay diferentes maneras de abordar el riesgo de generar deadlocks. Se pueden dar soluciones por manejo de errores si es aceptable hacer ROLLBACK, se pueden reducir mucho las probabilidades de que sucedan mediante el uso de consultas cortas y optimizadas y se pueden evitar por diseño, manteniendo un mismo flujo de acceso a los datos en todas las transacciones.

3. SEGURIDAD

G. Acceso indebido a un sitio web

H. Acceso indebido a la información