SISTEMAS INFORMÁTICOS

PRÁCTICA 4

Alejandro Pascual1 y Víctor Yrazusta2

Índice

[1. OPTIMIZACIÓN 2](#_Toc27566420)

[A. Estudio del impacto de un índice 2](#_Toc27566421)

[B. Estudio del impacto de preparar sentencias SQL 5](#_Toc27566422)

[C. Estudio del impacto de cambiar la forma de realizar una consulta 6](#_Toc27566423)

[D. Estudio del impacto de la generación de estadísticas 7](#_Toc27566424)

[2. TRANSACCIONES Y DEADLOCKS 9](#_Toc27566425)

[E. Estudio de transacciones 9](#_Toc27566426)

[F. Estudio de bloqueos y deadlocks 10](#_Toc27566427)

# 1. OPTIMIZACIÓN

## A. Estudio del impacto de un índice

Primero hemos ejecutado el script de la práctica anterior para inicializar los valores de los costes en la base de datos.

-- ESTABLECER PRECIOS

UPDATE orderdetail AS od

SET price=p.price

FROM products AS p

WHERE od.prod\_id=p.prod\_id;

UPDATE orderdetail AS od

SET price=ROUND(

    CAST(

        od.price/POW(

            1.02,

            2019-EXTRACT(year from o.orderdate)

        ) AS numeric

    ),

    2

)

FROM orders AS o

WHERE od.orderid=o.orderid;

-- UPDATE orders SET net\_amount=NULL, total\_amount=NULL;

CREATE OR REPLACE VIEW orders\_details\_net\_amount AS

SELECT

    order\_id,

    SUM(price\*quantity) AS net\_amount

FROM orders\_details

GROUP BY order\_id;

UPDATE orders AS o

SET

    net\_amount=odn.net\_amount,

    total\_amount=ROUND(CAST(odn.net\_amount\*(100+o.tax)/100 AS numeric), 2)

FROM orders\_details\_net\_amount AS odn

WHERE

    o.net\_amount IS NULL

    AND o.order\_id=odn.order\_id

;

Después hemos creado la siguiente sentencia.

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=5262.85..5262.86 rows=1 width=8)

 -> Unique (cost=5262.83..5262.84 rows=1 width=4)

     -> Sort (cost=5262.83..5262.84 rows=1 width=4)

        Sort Key: o.customerid

         -> Gather (cost=1000.00..5262.82 rows=1 width=4)

            Workers Planned: 1

             -> Parallel Seq Scan on orders o

(cost=0.00..4262.72 rows=1 width=4)

                Filter: ((totalamount > '100'::numeric) AND (date\_part('year'::text, (orderdate)::timestamp without time zone) = '2015'::double precision) AND (date\_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision))

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=8.48..8.49 rows=1 width=8)

 -> Unique (cost=8.46..8.47 rows=1 width=4)

     -> Sort (cost=8.46..8.47 rows=1 width=4)

        Sort Key: o.customerid

         -> Index Scan using index\_orders\_totalamount on orders o

(cost=0.42..8.45 rows=1 width=4)

            Index Cond: (totalamount > '100'::numeric)

            Filter: ((date\_part('year'::text, (orderdate)::timestamp without time zone) = '2015'::double precision) AND (date\_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision))

Podemos observar como el coste es 620 veces 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..4262.72 rows=1 width=4)

Que tras la creación del índice pasa a ser:

Index Scan using index\_orders\_totalamount on orders o

(cost=0.42..8.45 rows=1 width=4)

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 le 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.

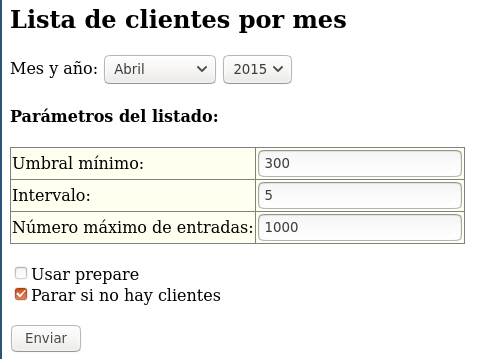
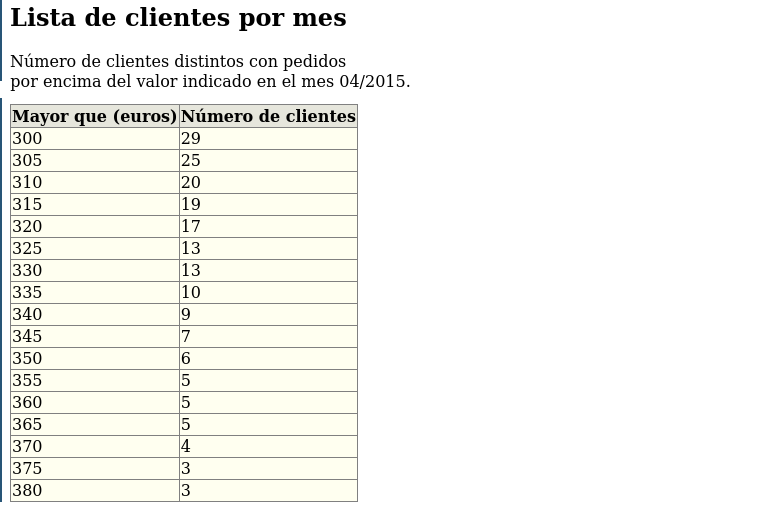
También hemos probado con un índice sobre el customerid de orders, ya que se puede apreciar que se utiliza como clave de ordenación para realizar el filtro DISTINCT.

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.



Después hemos ejecutado las pruebas con la base de datos limpia.

## 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)

PENDIENTE RESPONDER A CUAL DEVUELVE ALGO INSTANTANEAMENTE.

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

## F. Estudio de bloqueos y deadlocks