

SISTEMAS INFORMÁTICOS I PRÁCTICA 3

S. Xiao Fernández Marín

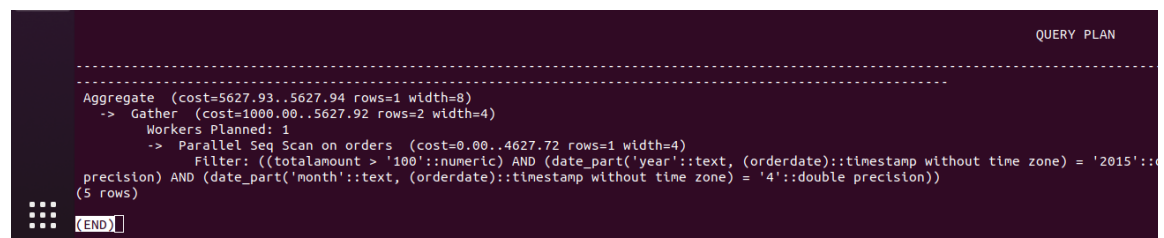
1392

1) NoSQL

En esta sección, hemos creado un script en Python que se encarga de crear una base de datos documental en MongoDB a partir de la base de datos en SQL. Esta base de datos es accesible desde nuestra página web, e integra una nueva opción en el menú llamado “topUSA”, que muestra el resultado de las consultas realizadas a los documentos contenidos en MongoDB.

2) Optimización

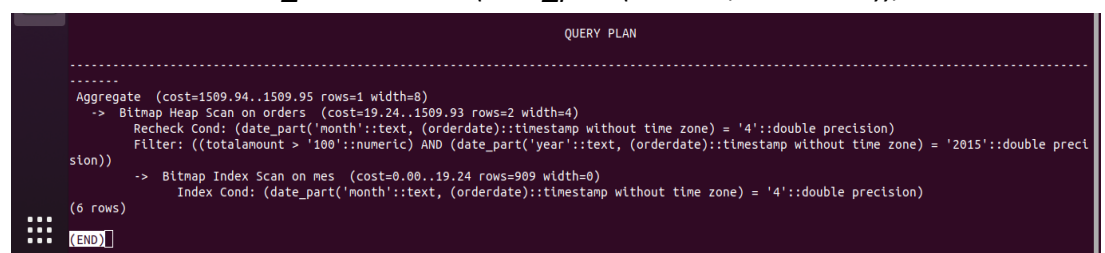
- **Estudio del impacto de un índice:** creamos la consulta *clientesDistintos.sql* y podemos observar que utilizando la sentencia EXPLAIN, la búsqueda es secuencial y tiene gran coste, de 5627 unidades, como podemos ver en la imagen del plan de ejecución:



```
-----
Aggregate (cost=5627.93..5627.94 rows=1 width=8)
-> Gather (cost=1000.00..5627.92 rows=2 width=4)
    Workers Planned: 1
    -> Parallel Seq Scan on orders (cost=0.00..4627.72 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))
(5 rows)
(END)
```

Por tanto, añadimos un índice en la tabla *months* para mejorar el rendimiento de la consulta. De este modo Postgres seleccionará la parte de la tabla a revisar, y la consulta reducirá su coste más de la mitad, a 1509. Este es el resultado:

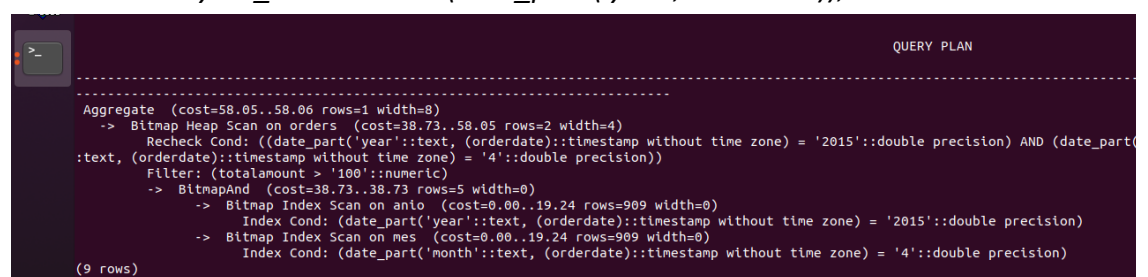
CREATE INDEX month_idx ON orders(date_part ('month', orderdate));



```
-----
Aggregate (cost=1509.94..1509.95 rows=1 width=8)
-> Bitmap Heap Scan on orders (cost=19.24..1509.93 rows=2 width=4)
    Recheck Cond: (date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision)
    Filter: (((totalamount > '100'::numeric) AND (date_part('year'::text, (orderdate)::timestamp without time zone) = '2015'::double precision)
(6 rows)
-> Bitmap Index Scan on mes (cost=0.00..19.24 rows=909 width=0)
    Index Cond: (date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision)
(END)
```

Probamos eliminando este índice y creando uno sobre la tabla *year*. El resultado es exactamente el mismo que en la anterior:

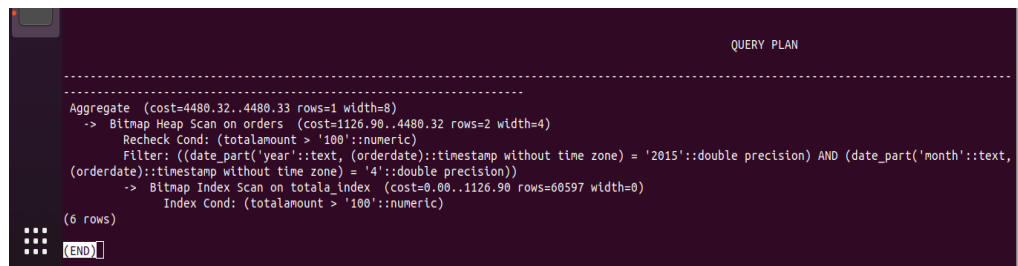
CREATE INDEX year_idx ON orders(date_part ('year', orderdate));



```
-----
Aggregate (cost=58.05..58.06 rows=1 width=8)
-> Bitmap Heap Scan on orders (cost=38.73..58.05 rows=2 width=4)
    Recheck Cond: (((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))
    Filter: (totalamount > '100'::numeric)
-> BitmapAnd (cost=38.73..38.73 rows=5 width=0)
    -> Bitmap Index Scan on anio (cost=0.00..19.24 rows=909 width=0)
        Index Cond: (date_part('year'::text, (orderdate)::timestamp without time zone) = '2015'::double precision)
    -> Bitmap Index Scan on mes (cost=0.00..19.24 rows=909 width=0)
        Index Cond: (date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision)
(9 rows)
```

Finalmente, probamos creando un índice sobre el importe total. Previamente eliminamos los índices anteriormente creados. Obtenemos lo siguiente:

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



```
QUERY PLAN
-----
Aggregate  (cost=4480.32..4480.33 rows=1 width=8)
-> Bitmap Heap Scan on orders  (cost=1126.90..4480.32 rows=2 width=4)
    Recheck 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))
-> Bitmap Index Scan on totala_idx  (cost=0.00..1126.90 rows=60597 width=0)
    Index Cond: (totalamount > '100'::numeric)

(6 rows)

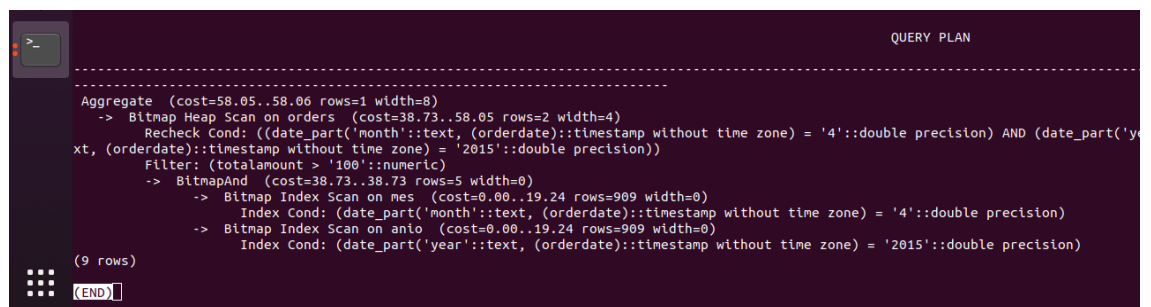
(END)
```

Aunque la planificación es la misma que en los dos casos anteriores, el coste se ha casi triplicado. Por tanto, eliminamos esta posibilidad, ya que el valor de *totalamount* es muy variable.

Ahora probamos a crear dos índices independientes sobre *month* y *year*, que fueron los que mejor resultado nos dieron. Obtenemos lo siguiente:

```
CREATE INDEX month_idx ON orders(date_part ('month', orderdate));
```

```
CREATE INDEX year_idx ON orders(date_part ('year', orderdate));
```



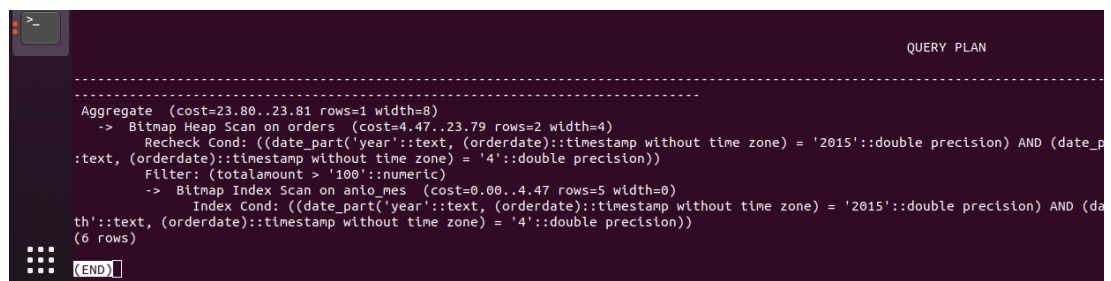
```
QUERY PLAN
-----
Aggregate  (cost=58.05..58.06 rows=1 width=8)
-> Bitmap Heap Scan on orders  (cost=38.73..58.05 rows=2 width=4)
    Recheck Cond: (((date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision) AND (date_part('year'::text, (orderdate)::timestamp without time zone) = '2015'::double precision))
    Filter: (totalamount > '100'::numeric)
-> BitmapAnd  (cost=38.73..38.73 rows=5 width=0)
    -> Bitmap Index Scan on mes  (cost=0.00..19.24 rows=909 width=0)
        Index Cond: (date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision)
    -> Bitmap Index Scan on anio  (cost=0.00..19.24 rows=909 width=0)
        Index Cond: (date_part('year'::text, (orderdate)::timestamp without time zone) = '2015'::double precision)

(9 rows)

(END)
```

A continuación creamos un índice a la vez sobre *year* y *month*:

```
CREATE INDEX year_month_idx ON orders(date_part ('year', orderdate),
date_part('month', orderdate));
```



```
QUERY PLAN
-----
Aggregate  (cost=23.80..23.81 rows=1 width=8)
-> Bitmap Heap Scan on orders  (cost=4.47..23.79 rows=2 width=4)
    Recheck Cond: (((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))
    Filter: (totalamount > '100'::numeric)
-> Bitmap Index Scan on anio_mes  (cost=0.00..4.47 rows=5 width=0)
    Index Cond: (((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))

(6 rows)

(END)
```

Conseguimos mejorar el coste anterior, ya que en este caso solo se realiza un Bitmap Head Scan.

Por último, probamos a realizar un único índice que engloba los tres campos:
`CREATE INDEX myt_idx ON orders(date_part ('year', orderdate), date_part('month', orderdate), totalamount);`

Vemos que se obtiene el mejor rendimiento, por lo que lo añadimos a nuestra consulta.

- Estudio del impacto de cambiar la forma de realizar una consulta:

Estudio plan de ejecución Query1:

```
dump_v1.2-P3=# explain select customerid
dump_v1.2-P3=# from customers
dump_v1.2-P3=# where customerid not in (
dump_v1.2-P3(# select customerid
dump_v1.2-P3(# from orders
dump_v1.2-P3(# where status='Paid'
dump_v1.2-P3(# );
QUERY PLAN
-----
Seq Scan on customers (cost=3961.65..4490.81 rows=7046 width=4)
  Filter: (NOT (hashed SubPlan 1))
  SubPlan 1
    -> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)
        Filter: ((status)::text = 'Paid'::text)
(5 rows)
```

Estudio plan de ejecución Query2:

```
dump_v1.2-P3=# explain select customerid
dump_v1.2-P3=# from (
dump_v1.2-P3(# select customerid
dump_v1.2-P3(# from customers
dump_v1.2-P3(# union all
dump_v1.2-P3(# select customerid
dump_v1.2-P3(# from orders
dump_v1.2-P3(# where status='Paid'
dump_v1.2-P3(# ) as A
dump_v1.2-P3=# group by customerid
dump_v1.2-P3=# having count(*) =1;
QUERY PLAN
-----
Finalize GroupAggregate (cost=4543.51..4594.68 rows=1 width=4)
  Group Key: orders.customerid
  Filter: (count(*) = 1)
  -> Gather Merge (cost=4543.51..4590.18 rows=400 width=12)
        Workers Planned: 2
        -> Sort (cost=3543.49..3543.99 rows=200 width=12)
              Sort Key: orders.customerid
              -> Partial HashAggregate (cost=3533.84..3535.84 rows=200 width=12)
                    Group Key: orders.customerid
                    -> Parallel Append (cost=0.00..3501.26 rows=6516 width=4)
                          -> Parallel Seq Scan on orders (cost=0.00..3023.69 rows=535 width=4)
                              Filter: ((status)::text = 'Paid'::text)
                          -> Parallel Seq Scan on customers (cost=0.00..435.90 rows=8290 width=4)
(13 rows)
```

Estudio plan de ejecución Query3:

```
dump_v1.2-P3=# explain select customerid
dump_v1.2-P3=# from customers
dump_v1.2-P3=# except
dump_v1.2-P3=# select customerid
dump_v1.2-P3=# from orders
dump_v1.2-P3=# where status='Paid';
QUERY PLAN
-----
HashSetOp Except (cost=0.00..4715.84 rows=14093 width=8)
  -> Append (cost=0.00..4678.34 rows=15002 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..3968.47 rows=909 width=8)
              -> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)
                  Filter: ((status)::text = 'Paid'::text)
(7 rows)
```

Observamos que la última consulta es la que mayor coste tiene, pero también es la única que puede empezar a trabajar nada más ejecutarse, ya que el coste de obtención de la primera tupla es 0.

La segunda consulta es la que se puede beneficiar del trabajo en paralelo, como indica el plan de ejecución (Workers Planned: 2...Parallel Seq Scan on orders...)

Como conclusión podemos decir que si lo que nos piden es empezar cuanto antes el análisis de datos, la mejor forma de realizar la consulta es la tercera. Pero si lo que necesitamos es trabajar lo antes posible con el resultado completo y tenemos la opción de trabajar en paralelo, en este caso nos quedamos con la segunda manera de hacer la consulta, la cual a la larga será mejor que la primera ya que aunque la primera tenga menor coste estimado, la ejecución en paralelo la supera en términos de eficiencia.

- **Estudio del impacto de la generación de estadísticas:**

En este apartado hemos tenido que hacer un estudio del impacto de la generación de estadísticas con un script llamado countStatus.sql

‘ANALYZE’ sirve para recoger estadísticas sobre una tabla dada. La diferencia entre las consultas antes de hacer el ‘ANALYZE’, incluyen una lista de valores comunes de cada columna y un histograma que muestra la distribución de los datos.

3) Transacciones:

En esta sección hemos creado la página web borraCliente.html para practicar el uso de transacciones y estudiar su comportamiento.

También hemos implementado el script updPromo.sql tratando de conseguir un deadlock.