

UNIVERSIDAD AUTÓNOMA



PRÁCTICAS DE SISTEMAS INFORMÁTICOS I

PRÁCTICA 4

Memoria

Autores:

Adrián FERNÁNDEZ

Santiago GONZÁLEZ-CARVAJAL

Pareja 11

Grupo 1401

5 de diciembre de 2018

Índice

1. Optimización	2
1.1. Índices	2
1.1.1. Query inicial	2
1.1.2. Añadiendo índices	2
1.1.3. Comparación de planes	2
1.2. Preparación de sentencias SQL	4
1.3. Forma de la consulta	6
1.3.1. Primera consulta	6
1.3.2. Segunda consulta	7
1.3.3. Tercera consulta	7
1.3.4. Preguntas	8
1.4. Estadísticas	8
1.4.1. Queries del apéndice 2	8
1.4.2. Estudio del coste de ejecución	9
1.4.3. Índice en status	9
1.4.4. Estudio tras la creación del índice	9
1.4.5. Generación las estadísticas sobre la tabla orders	10
1.4.6. Estudio tras generación de estadísticas	10
1.4.7. Comparación con las otras consultas proporcionadas	11
2. Transacciones y deadlocks	12
2.1. Transacciones	12
2.2. Bloqueos	12
3. Seguridad	12
3.1. Acceso indebido a un sitio web	12
3.2. Acceso indebido a información	12
4. Conclusión y propuestas	12

1. Optimización

1.1. Índices

1.1.1. Query inicial

```
1 — Query inicial
2 SELECT COUNT(DISTINCT customers.customerid)
3 FROM customers INNER JOIN orders ON (customers.customerid=orders.
   customerid)
4 WHERE orders.totalamount > 100 AND TO_CHAR(orders.orderdate, 'YYYYMM')
   = '201504';
```

Tiempo de ejecución: 44.117 msec.

1.1.2. Añadiendo índices

```
1 — Indices en orders.orderdate
2 CREATE INDEX idx_year ON orders(EXTRACT(year from orders.orderdate));
3 CREATE INDEX idx_month ON orders(EXTRACT(month from orders.orderdate));
4
5 — Indice en orders.totalamount
6 CREATE INDEX idx_amount ON orders(totalamount);
```

- Tiempo de ejecución con los índices en orders.orderdate: 22.427 msec.
- Tiempo de ejecución con todos los índices: 20.377 msec.

Observaciones:

Los índices introducidos son de tipo "btree", el cual es usado para consultas genéricas.

1.1.3. Comparación de planes

Planes:

	QUERY PLAN text
1	Aggregate (cost=5636.23..5636.24 rows=1 width=8) (actual time=42.197..42.197 rows=1 loops=1)
2	-> Gather (cost=1000.28..5636.22 rows=2 width=4) (actual time=1.103..43.410 rows=1539 loops=1)
3	Workers Planned: 1
4	Workers Launched: 1
5	-> Nested Loop (cost=0.29..4636.02 rows=1 width=4) (actual time=0.401..36.728 rows=770 loops=2)
6	-> Parallel Seq Scan on orders (cost=0.00..4627.72 rows=1 width=4) (actual time=0.332..33.418 rows=770 loops=2)
7	Filter: ((totalamount > '100'::numeric) AND (date part('year'::text, (orderdate)::timestamp without time zone) = '2015'::double precis
8	Rows Removed by Filter: 90126
9	-> Index Only Scan using customers pkey on customers (cost=0.29..8.30 rows=1 width=4) (actual time=0.004..0.004 rows=1 loops=1539)
10	Index Cond: (customerid = orders.customerid)
11	Heap Fetches: 789
12	Planning time: 0.749 ms
13	Execution time: 44.117 ms

Figura 1: Plan de ejecución para la query inicial.

	QUERY PLAN text
1	Aggregate (cost=3049.93..3049.94 rows=1 width=8) (actual time=22.048..22.048 rows=1 loops=1)
2	-> Hash Join (cost=946.34..3046.03 rows=1560 width=4) (actual time=13.938..21.623 rows=1539 loops=1)
3	Hash Cond: (orders.customerid = customers.customerid)
4	-> Bitmap Heap Scan on orders (cost=276.25..2371.84 rows=1560 width=4) (actual time=4.335..11.260 rows=1539 loops=1)
5	Recheck Cond: (date part('month':text, (orderdate)::timestamp without time zone) = '4':double precision)
6	Filter: ((totalamount > '100':numeric) AND (date part('year':text, (orderdate)::timestamp without time zone) = '2015':double precision))
7	Rows Removed by Filter: 13387
8	Heap Blocks: exact=1687
9	-> Bitmap Index Scan on idx_month (cost=0.00..275.86 rows=14858 width=0) (actual time=3.694..3.694 rows=14926 loops=1)
10	Index Cond: (date part('month':text, (orderdate)::timestamp without time zone) = '4':double precision)
11	-> Hash (cost=493.93..493.93 rows=14093 width=4) (actual time=9.574..9.574 rows=14093 loops=1)
12	Buckets: 16384 Batches: 1 Memory Usage: 624kB
13	-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.011..4.705 rows=14093 loops=1)
14	Planning time: 1.527 ms
15	Execution time: 22.427 ms

Figura 2: Plan de ejecución para la query con índices en **orders.orderdate**.

	QUERY PLAN text
1	Aggregate (cost=3039.93..3039.94 rows=1 width=8) (actual time=20.279..20.280 rows=1 loops=1)
2	-> Hash Join (cost=944.24..3036.16 rows=1511 width=4) (actual time=11.660..19.862 rows=1539 loops=1)
3	Hash Cond: (orders.customerid = customers.customerid)
4	-> Bitmap Heap Scan on orders (cost=274.15..2362.10 rows=1511 width=4) (actual time=2.911..10.417 rows=1539 loops=1)
5	Recheck Cond: (date part('month':text, (orderdate)::timestamp without time zone) = '4':double precision)
6	Filter: ((totalamount > '100':numeric) AND (date part('year':text, (orderdate)::timestamp without time zone) = '2015':double precision))
7	Rows Removed by Filter: 13387
8	Heap Blocks: exact=1687
9	-> Bitmap Index Scan on idx_month (cost=0.00..273.77 rows=14580 width=0) (actual time=2.238..2.238 rows=14926 loops=1)
10	Index Cond: (date part('month':text, (orderdate)::timestamp without time zone) = '4':double precision)
11	-> Hash (cost=493.93..493.93 rows=14093 width=4) (actual time=8.715..8.715 rows=14093 loops=1)
12	Buckets: 16384 Batches: 1 Memory Usage: 624kB
13	-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.013..4.398 rows=14093 loops=1)
14	Planning time: 0.614 ms
15	Execution time: 20.377 ms

Figura 3: Plan de ejecución para la query con todos los índices.

Observaciones:

Los planes de ejecución 1 y 2 difieren a la hora de aplicar el filtro. Ya que el primero lo aplica mediante un "Seq Scan" el segundo mediante un "Heap Scan" tras haber realizado Index Scan.^{en} el mes de la fecha. También se diferencian a la hora de unir las tablas, el primero hace "Gather" de las filas de **orders** que cumplen las condiciones del filtro con las filas de **customers** que tengan el mismo **customerid**, el segundo hace "Hash Join" de las filas de **orders** que cumplen las condiciones del filtro con las filas de **customers** y aplicando la condición de que tengan el mismo **customerid**.

Los planes 2 y 3 son idénticos, salvo en la línea 4, donde se aprecia que el índice en **orders.totalamount** permite descartar 49 filas más al aplicar el filtro de precio y fecha.

En resumen, indexar **orders.orderdate** ha aportado una mejora de casi el 100 % de rendimiento, mientras que indexar **orders.totalamount** apenas ha mejorado el rendimiento de la consulta.

1.2. Preparación de sentencias SQL

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	2
305	1
310	1
315	1
320	0

Tiempo: 69 ms

Usando prepare

[Nueva consulta](#)

Figura 4: Usando prepare.

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	2
305	1
310	1
315	1
320	0

Tiempo: 93 ms

[Nueva consulta](#)

Figura 5: Sin usar prepare.

El índice, como podemos ver en los resultados que aparecen en 6 y en 7 mejora el tiempo de las consultas reduciéndolo aproximadamente a la mitad en ambos casos.

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	2
305	1
310	1
315	1
320	0

Tiempo: 29 ms

Usando prepare

[Nueva consulta](#)

Figura 6: Usando prepare con índice en año y mes.

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	2
305	1
310	1
315	1
320	0

Tiempo: 40 ms

[Nueva consulta](#)

Figura 7: Sin usar prepare con índice en año y mes.

Observaciones:

El prepare es útil cuando vas a realizar la misma consulta muchas veces variando, simplemente, los valores (es decir, los valores se pueden ver como argumentos en la consulta). Mientras que daría peores resultados que ejecutar solo la consulta si el número de consultas que fuéramos a hacer fuera pequeño. Por ejemplo, para una sola consulta, hacer el prepare no nos proporcionaría ninguna mejora (es más, probablemente, empeoraría el tiempo de ejecución al tener que hacer PREPARE, EXECUTE y DEALLOCATE).

1.3. Forma de la consulta

1.3.1. Primera consulta

Código:

```
1 select
2     customerid
3 from
4     customers
5 where
6     customerid not in (
7         select
8             customerid
9         from
10            orders
11         where
12             status='Paid'
13     );
```

Observaciones:

La planificación de esta consulta es simple, devuelve todos los resultados que no pertenezcan a la subconsulta. Para ello primero ejecuta la subconsulta y finalmente elimina las filas de la consulta principal cuyo customerid está en ellas.

Plan:

	QUERY PLAN text
1	Seq Scan on customers (cost=3961.65..4490.81 rows=7046 width=4) (actual time=23.107..25.803 rows=4688 loops=1)
2	Filter: (NOT (hashed SubPlan 1))
3	Rows Removed by Filter: 9405
4	SubPlan 1
5	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.014..19.140 rows=18163 loops=1)
6	Filter: ((status)::text = 'Paid'::text)
7	Rows Removed by Filter: 163627
8	Planning time: 0.164 ms
9	Execution time: 25.990 ms

Figura 8: Plan de ejecución para la primera consulta.

1.3.2. Segunda consulta

Código:

```
1 select
2     customerid
3 from (
4     select
5         customerid
6     from
7         customers
8     union all
9     select
10        customerid
11    from
12        orders
13    where
14        status='Paid'
15    ) as A
16 group by
17     customerid
18 having
19     count(*) = 1;
```

Observaciones:

Esta consulta realiza la unión de las dos tablas resultantes de ambas subconsultas, después agrupa y comprueba la condición, y, a continuación, muestra los resultados.

Plan:

	QUERY PLAN text
1	HashAggregate (cost=4537.41..4539.41 rows=200 width=4) (actual time=25.880..27.431 rows=4688 loops=1)
2	Group Key: customers.customerid
3	Filter: (count(*) = 1)
4	Rows Removed by Filter: 9405
5	-> Append (cost=0.00..4462.40 rows=15002 width=4) (actual time=0.007..19.273 rows=32256 loops=1)
6	-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.006..1.408 rows=14093 loops=1)
7	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.005..16.146 rows=18163 loops=1)
8	Filter: ((status)::text = 'Paid'::text)
9	Rows Removed by Filter: 163627
10	Planning time: 0.091 ms
11	Execution time: 27.584 ms

Figura 9: Plan de ejecución para la segunda consulta.

1.3.3. Tercera consulta

Código:

```
1 select
2     customerid
3 from
4     customers
5 except
6 select
7     customerid
8 from
9     orders
10 where
11     status='Paid';
```


Observaciones:

Esta consulta primero da un resultado parcial con la primera subconsulta (antes del except) del que luego elimina todos los resultados de la segunda subconsulta (después del except).

Plan:

	QUERY PLAN text
1	HashSetOp Except (cost=0.00..4640.83 rows=14093 width=8) (actual time=26.876..27.554 rows=4688 loops=1)
2	-> Append (cost=0.00..4603.32 rows=15002 width=8) (actual time=0.008..21.478 rows=32256 loops=1)
3	-> Subquery Scan on "**SELECT* 1" (cost=0.00..634.86 rows=14093 width=8) (actual time=0.007..2.591 rows=14093 loops=1)
4	-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.007..1.471 rows=14093 loops=1)
5	-> Subquery Scan on "**SELECT* 2" (cost=0.00..3968.47 rows=909 width=8) (actual time=0.007..17.189 rows=18163 loops=1)
6	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.006..15.876 rows=18163 loops=1)
7	Filter: ((status)::text = 'Paid'::text)
8	Rows Removed by Filter: 163627
9	Planning time: 0.054 ms
10	Execution time: 27.807 ms

Figura 10: Plan de ejecución para la tercera consulta.

1.3.4. Preguntas

i. ¿Qué consulta devuelve algún resultado nada más comenzar su ejecución?

La tercera consulta, ya que primero devuelve los resultados y luego va excluyendo los que pertenecen al except.

ii. ¿Qué consulta se puede beneficiar de la ejecución en paralelo?

La segunda, ya que realiza union all de dos tablas que podrían ser calculadas en paralelo.

1.4. Estadísticas

1.4.1. Queries del apéndice 2

```
1 — Primera query
2 SELECT COUNT(*)
3 FROM orders
4 WHERE status IS NULL;
5
6 — Segunda query
7 SELECT COUNT(*)
8 FROM orders
9 WHERE status = 'Shipped';
```

1.4.2. Estudio del coste de ejecución

	QUERY PLAN text
1	Aggregate (cost=3507.17..3507.18 rows=1 width=8)
2	-> Seq Scan on orders (cost=0.00..3504.90 rows=909 width=0)
3	Filter: (status IS NULL)

Figura 11: Plan de ejecución para la primera query.

	QUERY PLAN text
1	Aggregate (cost=3961.65..3961.66 rows=1 width=8)
2	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=0)
3	Filter: ((status)::text = 'Shipped'::text)

Figura 12: Plan de ejecución para la segunda query.

Podemos observar que la primera query tiene un coste inferior a la segunda a pesar de acceder a la tabla de forma idéntica. Esto se debe a que aplican operaciones diferentes al filtrar las filas, la primera comprueba si **orders.status** es "NULL", mientras que la segunda lo compara con la cadena "Shipped". Deducimos pues, que compara cadenas de caracteres es una operación más costosa que comprobar si un atributo es "NULL".

1.4.3. Índice en status

```
1 — Indice en orders.status  
2 CREATE INDEX idx_status ON orders(status);
```

1.4.4. Estudio tras la creación del índice

	QUERY PLAN text
1	Aggregate (cost=1496.52..1496.53 rows=1 width=8)
2	-> Bitmap Heap Scan on orders (cost=19.46..1494.25 rows=909 width=0)
3	Recheck Cond: (status IS NULL)
4	-> Bitmap Index Scan on idx_status (cost=0.00..19.24 rows=909 width=0)
5	Index Cond: (status IS NULL)

Figura 13: Plan de ejecución para la primera query con índice en **orders.status**.

	QUERY PLAN text
1	Aggregate (cost=1498.79..1498.80 rows=1 width=8)
2	-> Bitmap Heap Scan on orders (cost=19.46..1496.52 rows=909 width=0)
3	Recheck Cond: ((status)::text = 'Shipped'::text)
4	-> Bitmap Index Scan on idx status (cost=0.00..19.24 rows=909 width=0)
5	Index Cond: ((status)::text = 'Shipped'::text)

Figura 14: Plan de ejecución para la segunda query con índice en **orders.status**.

Igual que en el resultado anterior, la segunda query es más costosa que la primera a pesar de que tienen el mismo plan de acceso.

La diferencia es que ahora, ambas queries realizan un **Index Scan**.^{en} **orders.status** para aplicar el filtro y recogen las filas seleccionadas con un mapeo con "Heap Scan" sobre la misma condición que el filtro. Esto resulta en un incremento del rendimiento en más de un 100 %.

1.4.5. Generación las estadísticas sobre la tabla orders

```
1 — Generacion de estadisticas
2 ANALYZE VERBOSE orders;
```

Output:

- 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

1.4.6. Estudio tras generación de estadísticas

	QUERY PLAN text
1	Aggregate (cost=7.27..7.28 rows=1 width=8)
2	-> Index Only Scan using idx status on orders (cost=0.42..7.27 rows=1 width=0)
3	Index Cond: (status IS NULL)

Figura 15: Plan de ejecución para la primera query tras la generación de estadísticas.

	QUERY PLAN text
1	Finalize Aggregate (cost=4211.65..4211.66 rows=1 width=8)
2	-> Gather (cost=4211.54..4211.65 rows=1 width=8)
3	Workers Planned: 1
4	-> Partial Aggregate (cost=3211.54..3211.55 rows=1 width=8)
5	-> Parallel Seq Scan on orders (cost=0.00..3023.69 rows=75140 width=0)
6	Filter: ((status)::text = 'Shipped'::text)

Figura 16: Plan de ejecución para la segunda query tras la generación de estadísticas.

Coste:

El coste de la primera query se ha reducido considerablemente (de 1496 a 7), mientras que el de la segunda ha aumentado (de 1498 a 4211) hasta superar el coste inicial, ya que ahora PostgreSQL es consciente del número aproximado de columnas que van a satisfacer la condición.

Plan:

El plan para la primera consulta ha cambiado drásticamente debido a que el número esperado de resultados es muy pequeño (por ello hace un Index Only Scan comprobando que el valor sea "NULL"). Mientras que el plan de la segunda consulta ha cambiado bastante también, debido, en este caso a que, como hemos comentado anteriormente, el número de resultados esperados ha crecido considerablemente (de ahí que ahora planifique abordar la consulta con un escaneado secuencial ejecutado en paralelo).

1.4.7. Comparación con las otras consultas proporcionadas

Queries:

```

1 — Tercera query
2 SELECT COUNT(*)
3 FROM orders
4 WHERE status = 'Paid';
5
6 — Cuarta query
7 SELECT COUNT(*)
8 FROM orders
9 WHERE status = 'Processed';

```

Planes:

	QUERY PLAN text
1	Aggregate (cost=2310.10..2310.11 rows=1 width=8)
2	-> Bitmap Heap Scan on orders (cost=354.96..2265.41 rows=17876 width=0)
3	Recheck Cond: ((status)::text = 'Paid'::text)
4	-> Bitmap Index Scan on idx_status (cost=0.00..350.49 rows=17876 width=0)
5	Index Cond: ((status)::text = 'Paid'::text)

Figura 17: Plan de ejecución para la tercera query tras la generación de estadísticas.

	QUERY PLAN text
1	Aggregate (cost=2942.42..2942.43 rows=1 width=8)
2	-> Bitmap Heap Scan on orders (cost=712.78..2851.98 rows=36176 width=0)
3	Recheck Cond: ((status)::text = 'Processed'::text)
4	-> Bitmap Index Scan on idx_status (cost=0.00..703.74 rows=36176 width=0)
5	Index Cond: ((status)::text = 'Processed'::text)

Figura 18: Plan de ejecución para la cuarta query tras la generación de estadísticas.

Observaciones:

Como podemos observar la forma de abordar las consultas tercera y cuarta de PostgreSQL, tras generar las estadísticas, es prácticamente la misma, es decir, un escaneado combinando varios índices (bitmap scan). Esto se debe a que el número de resultados que estima obtener se encuentra aproximadamente en el mismo rango en ambas consultas. Esta forma difiere de la de la primera, ya que en ella solo comprueba una condición en el índice (si es "NULL") debido al bajo número de resultados esperados; difiere también de la segunda, ya que en ella se esperan aproximadamente el doble de resultados que en la cuarta y casi el cuádruple que en la tercera. Por lo que para la segunda la forma de abordar la consulta sería un escaneado secuencial ejecutado en paralelo (debido a este gran número de resultados esperados) con el objetivo de tratar de reducir el tiempo de ejecución.

2. Transacciones y deadlocks

2.1. Transacciones

2.2. Bloqueos

3. Seguridad

3.1. Acceso indebido a un sitio web

3.2. Acceso indebido a información

4. Conclusión y propuestas

Referencias

- [1] "Documentación de postgresql." <https://www.postgresql.org/docs/9.3/static/>.
- [2] "Documentación de comandos postgresql." <https://www.postgresql.org/docs/9.3/static/sql-commands.html>.
- [3] "Creación de índices en postgresql." <https://www.postgresql.org/docs/9.1/sql-createindex.html>.
- [4] "Uso de prepare y execute." <https://www.postgresql.org/docs/9.3/sql-prepare.html>.
- [5] "Uso de deallocate." <https://www.postgresql.org/docs/8.1/sql-deallocate.html>.

Además de todas las referencias anteriores, también hemos usado como referencia las diapositivas de teoría y algunos otros sitios web para algunas consultas no destacables.