Javier Gómez Martínez
Carlos Li Hu

Práctica 4

APARTADO A

Sin índices

```
SELECT count(orderid) AS cc
FROM customers NATURAL JOIN orders
WHERE totalamount > 100 AND
EXTRACT(MONTH FROM orderdate)=4 AND
EXTRACT(YEAR FROM orderdate)=2014;
```

Output pane

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Aggregate (cost=6702.86..6702.87 rows=1 width=4)
2	-> Nested Loop (cost=0.29..6702.85 rows=2 width=4)
3	-> Seq Scan on orders (cost=0.00..6686.23 rows=2 width=8)
4	Filter: ((totalamount > '100'::numeric) AND (date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision) AND
5	-> Index Only Scan using customers_pkey on customers (cost=0.29..8.30 rows=1 width=4)
6	Index Cond: (customerid = orders.customerid)

En este apartado hemos hecho un explain de esta consulta, con los datos que dicen como ejemplo, sin haber creado ningún tipo de índice.
Como vemos la consulta se compone de 2 filtros esenciales, uno que consulta que el totalamount y el order date cumplan las restricciones impuestas mediante un Seq Scan.
Y otra que consiste en un “Index Only Scan” para ver que coincidan el primary key de customers (customerid) con el campo del mismo nombre en orders.

Y en la parte más externa del arbol de nodos creado, podemos ver cómo se hace un NestedLoop para encontrar cada una de las filas resultado de la query, y luego usamos el aggregate (count) con esas filas.
Como vemos en la parte más externa, el coste inicial antes de obtener la primera tupla es de 6702,86 (debido al tiempo de filtrado descrito y de agregación) y el coste total estimado es de 6702,87 unidades, pues el tiempo de proceso está esencialmente ubicado en el procesamiento de las filas, no en imprimirlas.
Además se estima que recibirá 1 tupla como resultado (porque es un count) y nos dice que el tamaño estimado de las filas recibidas será de 4B (pues count devuelve un integer).

Con índices

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

Output pane

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Aggregate (cost=6061.33..6061.34 rows=1 width=4)
2	-> Nested Loop (cost=1754.69..6061.32 rows=2 width=4)
3	-> Bitmap Heap Scan on orders (cost=1754.41..6044.70 rows=2 width=8)
4	Recheck Cond: (totalamount > '100'::numeric)
5	Filter: ((date_part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision) AND (date_part('year'::text, (orderdat
6	-> Bitmap Index Scan on idx_totalamount_orders (cost=0.00..1754.41 rows=94665 width=0)
7	Index Cond: (totalamount > '100'::numeric)
8	-> Index Only Scan using customers_pkey on customers (cost=0.29..8.30 rows=1 width=4)
9	Index Cond: (customerid = orders.customerid)

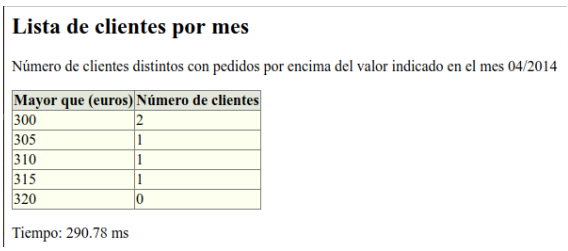
Hemos probado con todos los índices relevantes que hemos encontrado, es decir, orderdate, totalamount y customerid, y hemos notado que el único que ha modificado los costes ha sido el de totalamount, pues, como vemos en la captura, ha disminuído el coste total e inicial a 6061 unidades

Podemos notar como se mantiene el filtro de que coincida el customerid, y el de que se busque por orderdate, pero para la parte del totalamount está haciendo uso del índice que hemos creado, haciendo que el coste para encontrar las filas del resultado disminuya notablemente.
Esto es debido a que con un índice creado, acceder a las filas que tienen un totalamount mayor a uno pedido, le resulta muy cómodo, pues el índice guarda referencias a columnas con un cierto orden.

APARTADO B

Para todos estos apartados, hemos ejecutado la consulta en la pagina web con los valores por defecto que nos daban para los umbrales minimo, intervalos,etc.

Sin Indices no preparada



Lista de clientes por mes

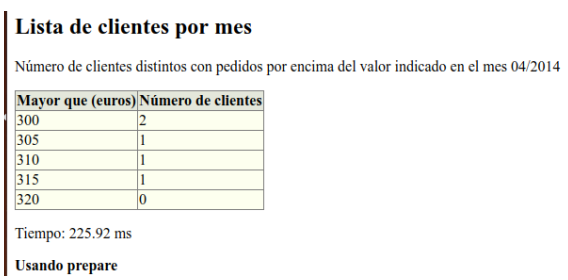
Número de clientes distintos con pedidos por encima del valor indicado en el mes 04/2014

Mayor que (euros)	Número de clientes
300	2
305	1
310	1
315	1
320	0

Tiempo: 290.78 ms

Como podemos ver aquí le cuesta casi 300 ms devolvernos la consulta.

Sin Indices preparada



Lista de clientes por mes

Número de clientes distintos con pedidos por encima del valor indicado en el mes 04/2014

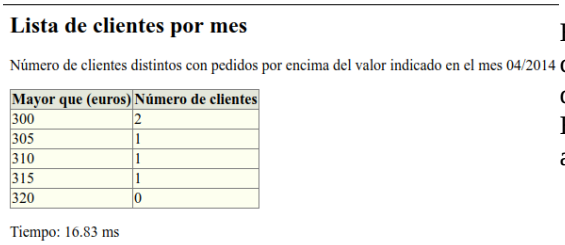
Mayor que (euros)	Número de clientes
300	2
305	1
310	1
315	1
320	0

Tiempo: 225.92 ms

Usando prepare

Aquí podemos notar como el tiempo con respecto a no preparar la consulta, no es muy diferente, es más, dependiendo de la ejecución, hay veces que puede ser menor o mayor a hacerlo sin prepararla (en este caso es menor)

Con Indices No preparada



Lista de clientes por mes

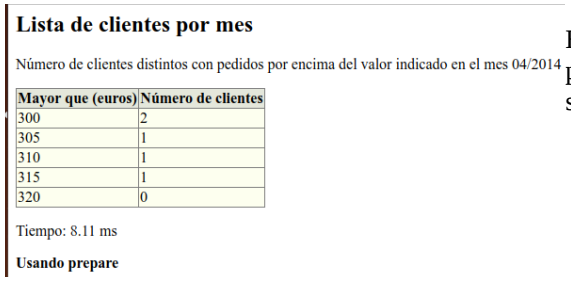
Número de clientes distintos con pedidos por encima del valor indicado en el mes 04/2014

Mayor que (euros)	Número de clientes
300	2
305	1
310	1
315	1
320	0

Tiempo: 16.83 ms

En este apartado, podemos notar como el tiempo para ejecutar la consulta ha disminuído muy considerablemente, más de un factor 10 de reducción.
Esto es debido al índice para totalamount que creamos en el apartado anterior.

Con Indices Preparada



Lista de clientes por mes

Número de clientes distintos con pedidos por encima del valor indicado en el mes 04/2014

Mayor que (euros)	Número de clientes
300	2
305	1
310	1
315	1
320	0

Tiempo: 8.11 ms

Usando prepare

En este apartado hacemos uso de todo, los índices, y que esté preparada la consulta, y el resultado es que el tiempo de ejecución sea el menor de todas las que hemos hecho.

Tras ejecutar ANALYZE

Cuando generamos las estadísticas, no notamos un cambio considerable en el tiempo de respuesta con respecto a las consultas anteriores.
Esto puede ser debido a que aún sabiendo las estadísticas sobre las tablas customers y orders, nuestro planificador considere que la forma de procesar la consulta no se pueda mejorar más en este caso.

APARTADO C
Hemos analizado estas consultas sin ningún índice creado
Consulta 1

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

Output pane

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Seq Scan on customers (cost=3961.65..4490.81 rows=7046 width=4)
2	Filter: (NOT (hashed SubPlan 1))
3	SubPlan 1
4	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)
5	Filter: ((status)::text = 'Paid'::text)

En esta consulta la forma en que se opera internamente consiste en realizar la subconsulta dentro del not in (es decir, mirar los que tienen status=paid), y después aplicarle el filtro de coger todos los que no están en esa subquery

Consulta 2

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

Output pane

Data Output | Explain | Messages | History

	QUERY PLAN text
1	HashAggregate (cost=4537.41..4539.91 rows=200 width=4)
2	Group Key: customers.customerid
3	Filter: (count(*) = 1)
4	-> Append (cost=0.00..4462.40 rows=15002 width=4)
5	-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4)
6	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)
7	Filter: ((status)::text = 'Paid'::text)

En esta consulta vemos como primero hace SeqScan para hacer las 2 consultas más internas a unir, acto seguido, une las consultas con el “union all”, después filtra las que tienen count(*)=1 para una agrupación por customerid, y después muestra el resultado

Consulta 3

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

Output pane

Data Output | Explain | Messages | History

	QUERY PLAN text
1	HashSetOp Except (cost=0.00..4640.83 rows=14093 width=4)
2	-> Append (cost=0.00..4603.32 rows=15002 width=4)
3	-> Subquery Scan on "*SELECT* 1" (cost=0.00..634.86 rows=14093 width=4)
4	-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4)
5	-> Subquery Scan on "*SELECT* 2" (cost=0.00..3968.47 rows=909 width=4)
6	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)
7	Filter: ((status)::text = 'Paid'::text)

Aquí vemos, como lo primero que hace es realizar las 2 consultas internas por separado, y después las resta usando “except”.

FALTA CONCLUIR ALGO

APARTADO D
consultas sin indices

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

Output pane

Data Output | Explain | Messages | History

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

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

Output pane

Data Output | Explain | Messages | History

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

En estas consultas simplemente añadimos una condición de filtrado en la tabla, a la hora de buscar los elementos, esperando encontrar el mismo número de elementos. Aunque la diferencia radica en que al buscar secuencialmente, el que busca en status un NULL tarda un poco menos, porque no tiene que comparar cadenas como hace segunda consulta, que está obligada a procesar la cadena ‘Shipped’ entera para asegurarse de que encuentra las filas correctas

consultas con indice status

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

Output pane

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Aggregate (cost=1496.52..1496.53 rows=1 width=0)
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_orders (cost=0.00..19.24 rows=909 width=0)
5	Index Cond: (status IS NULL)

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

Output pane

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Aggregate (cost=1498.79..1498.80 rows=1 width=0)
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_orders (cost=0.00..19.24 rows=909 width=0)
5	Index Cond: ((status)::text = 'Shipped'::text)

Una vez creado el índice, la diferencia entre costes disminuye, pues los índices guardan referencias a las filas donde sabemos que tenemos el valor de status con un cierto valor, y por tanto, no necesitamos comparar la cadena ‘Shipped’ con cada uno de los campos status de las filas que nos da el índice.

Consultas con índice status y ANALYZE ejecutado

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

Output pane

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Aggregate (cost=7.25..7.26 rows=1 width=0)
2	-> Index Only Scan using idx_status_orders on orders (cost=0.42..7.24 rows=1 width=0)
3	Index Cond: (status IS NULL)

Aquí podemos ver como el coste ha disminuido mucho.

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

Output pane

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Aggregate (cost=4278.63..4278.64 rows=1 width=0)
2	-> Seq Scan on orders (cost=0.00..3959.38 rows=127701 width=0)
3	Filter: ((status)::text = 'Shipped'::text)

Aquí podemos ver que el coste ha aumentado considerablemente

La razón de estos cambios tras ejecutar ANALYZE radica en la naturaleza de ANALYZE, la cual genera estadísticas sobre la tabla orders, que luego utiliza para determinar los planes más eficientes en base a ellas.
En este caso, el comando ha detectado que el número de filas con STATUS a NULL es muy pequeño (concretamente no las hay), y el número de filas con status a 'Shipped' es más grande de lo estimado sin tener las estadísticas (si vemos el resultado de la consulta, veremos que conforman alrededor del 60% de la tabla).
Por estas razones, los EXPLAIN nos han ofrecido unos números tan dispares, pues se han generado estadísticas que nos dan unas aproximaciones más fiables para nuestras consultas.

Consultas adicionales apartado D

Sin indices ni analyze

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

Output pane

Data Output | Explain | Messages | History

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

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

Output pane

Data Output | Explain | Messages | History

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

Con indice para status

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

Output pane

Data Output | Explain | Messages | History

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

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

Output pane

Data Output | Explain | Messages | History

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

Con indice y estadisticas

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

Output pane

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Aggregate (cost=2311.49..2311.50 rows=1 width=0)
2	-> Bitmap Heap Scan on orders (cost=355.43..2266.64 rows=17937 width=0)
3	Recheck Cond: ((status)::text = 'Paid'::text)
4	-> Bitmap Index Scan on idx_status_orders (cost=0.00..350.95 rows=17937 width=0)
5	Index Cond: ((status)::text = 'Paid'::text)

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

Output pane

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Aggregate (cost=2954.00..2954.01 rows=1 width=0)
2	-> Bitmap Heap Scan on orders (cost=719.36..2862.73 rows=36509 width=0)
3	Recheck Cond: ((status)::text = 'Processed'::text)
4	-> Bitmap Index Scan on idx_status_orders (cost=0.00..710.24 rows=36509 width=0)
5	Index Cond: ((status)::text = 'Processed'::text)

Para ambas consultas notamos los mismos resultados, es decir:
Tras crear el indice para status, notamos como el coste estimado dismimuye, como era de esperarse, pero al generar las estadísticas, sufre un efecto similar a la segunda consulta de este apartado, donde vemos que aumenta la estimación del coste, pues ahora se tienen unas estadísticas que nos ayudan a aproximarlo mejor.