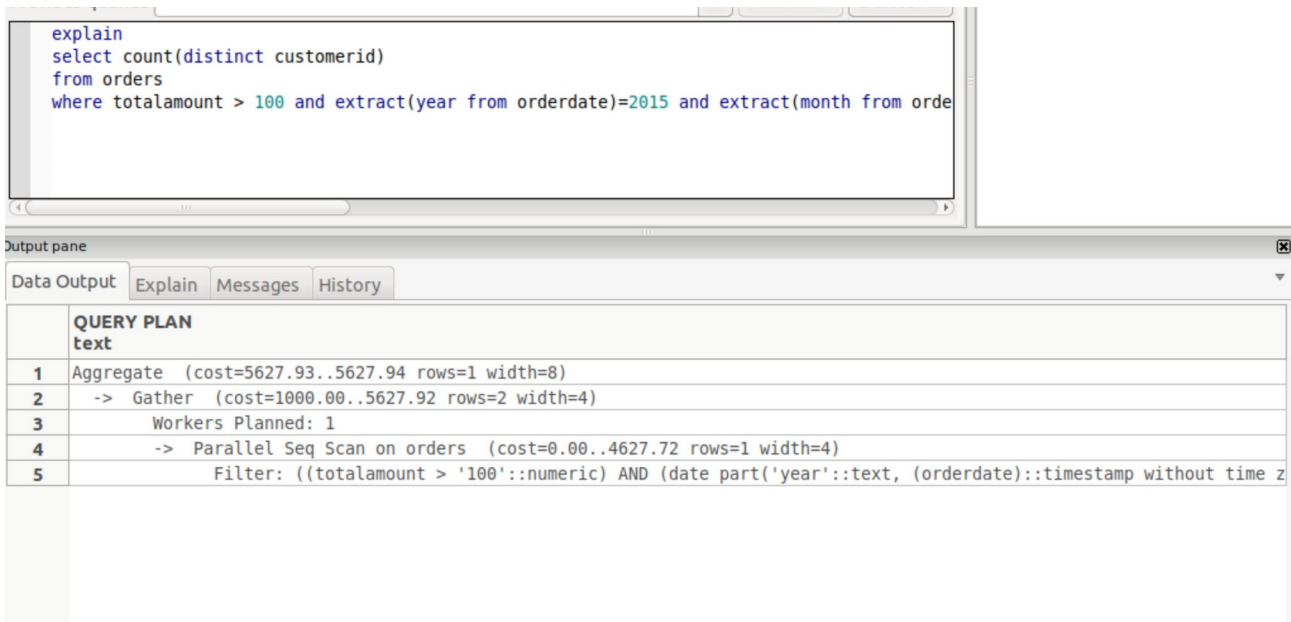


OPTIMIZACION

En este apartado se nos pedía estudiar el impacto de índices sobre determinadas consultas, así como entender el uso de las sentencias EXPLAIN y ANALYZE entre otras.



The screenshot shows a database interface with a SQL query editor and an output pane. The query is:

```
explain
select count(distinct customerid)
from orders
where totalamount > 100 and extract(year from orderdate)=2015 and extract(month from orde
```

The output pane shows the 'EXPLAIN' tab with the following query plan:

	QUERY PLAN
1	Aggregate (cost=5627.93..5627.94 rows=1 width=8)
2	-> Gather (cost=1000.00..5627.92 rows=2 width=4)
3	Workers Planned: 1
4	-> Parallel Seq Scan on orders (cost=0.00..4627.72 rows=1 width=4)
5	Filter: ((totalamount > '100'::numeric) AND (date part('year'::text, (orderdate)::timestamp without time z

Estudio de impacto de índices:

Se observa que la acción más costosa es el 'seq scan' pues este realiza un escaneo secuencial de la tabla de datos almacenada desde la primera fila hasta que la consulta se cumple.

Para mejorar el rendimiento creamos índices sobre esta tabla. Nuestra primera aproximación fue pensar en crear un índice para la selección de pedidos por el coste total, pero dado que más del 50% de los pedidos de la base de datos tenían un coste mayor de 100, el rendimiento no mejoraba mucho pues no descartábamos un gran número de filas, y con la creación de los siguientes índices (mucho más útiles) su mejora de rendimiento era imperceptible. Es por ello que lo descartamos.

Lo siguiente que pensamos fue crear un índice para la operación de extracción del año en cuestión, esto es:

```
create index index_year on orders(extract(year from orderdate));
```

SQL Editor Graphical Query Builder

Previous queries Delete Delete All

```
EXPLAIN
select count(distinct customerid)
from orders
where totalamount > 100 and extract(year from orderdate) = 2015 and extract(month from orderdate)= 4;

create index idx_totalamount on orders(totalamount);
create index index_year on orders(extract(year from orderdate));

create index index_month on orders(extract(month from orderdate));

create index index_yearmonth on orders(extract(year from orderdate), extract(month from orderdate));
```

Output pane

Data Output Explain Messages History

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

Al ejecutar de nuevo la query vimos una notable mejora en el rendimiento, pues el coste se reducía de 5627 a 1509. El cambio se observa en el cambio de 'seq scan' que se realiza en orders por un 'bitmap heap scan'. Este se divide en dos pasos: la búsqueda de las filas que cumplan la condición en el índice creado y luego 'traerlas' del disco para mostrarlo. Traer las filas de una en una es más costoso que leerlas de manera secuencial, el motivo de que esto sea menos costoso es porque gracias al índice no tendremos que leer toda la tabla, pues el subconjunto de filas que tenemos que buscar es notablemente menor que el número total (aproximadamente un 12%).

```
EXPLAIN
select count(distinct customerid)
from orders
where totalamount > 100 and extract(year from orderdate) = 2015 and extract(month from orderdate)= 4;

create index idx_totalamount on orders(totalamount);
create index index_year on orders(extract(year from orderdate));

create index index_month on orders(extract(month from orderdate));

create index index_yearmonth on orders(extract(year from orderdate), extract(month from orderdate));
```

Output pane

Data Output Explain Messages History

	QUERY PLAN
1	Aggregate (cost=58.05..58.06 rows=1 width=8)
2	-> Bitmap Heap Scan on orders (cost=38.73..58.05 rows=2 width=4)
3	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))
4	Filter: (totalamount > '100'::numeric)
5	-> BitmapAnd (cost=38.73..38.73 rows=5 width=0)
6	-> Bitmap Index Scan on index_month (cost=0.00..19.24 rows=909 width=0)
7	Index Cond: (date part('month'::text, (orderdate)::timestamp without time zone) = '4'::double precision)
8	-> Bitmap Index Scan on index_year (cost=0.00..19.24 rows=909 width=0)
9	Index Cond: (date part('year'::text, (orderdate)::timestamp without time zone) = '2015'::double precision)

Viendo esto decidimos probar a añadir otro índice en este caso para la extracción del mes, es decir:

```
create index index_month on orders(extract(year from orderdate));
```

Observamos que el rendimiento de la consulta mejora de nuevo, pasando de un coste de 1509 a un coste total de 58. El razonamiento es el mismo que el anterior.

Viendo esto se nos ocurrió crear un índice más específico uniendo las dos situaciones anteriores en lugar de tenerlos por separados, es decir, un índice para la extracción de

la fecha completa. De este modo nos lo imaginamos como un índice de un libro donde cada capítulo es una fecha, y en ellos están los pedidos que se realizaran ese día. Es

```
EXPLAIN
select count(distinct customerid)
from orders
where totalamount > 100 and extract(year from orderdate) = 2015 and extract(month from orderdate)= 4;

create index index_year on orders(extract(year from orderdate));
create index index_month on orders(extract(month from orderdate));
create index index_yearmonth on orders(extract(year from orderdate), extract(month from orderdate));

drop index index_year;
drop index index_month;
```

Output pane

Data Output Explain Messages History

	QUERY PLAN
1	Aggregate (cost=23.80..23.81 rows=1 width=8)
2	-> Bitmap Heap Scan on orders (cost=4.47..23.79 rows=2 width=4)
3	Recheck Cond: ((date part('year'::text, (orderdate)::timestamp without time zone) = '2015'::double precision) AND (date part('month'::text, (orderdate)::t
4	Filter: (totalamount > '100'::numeric)
5	-> Bitmap Index Scan on index_yearmonth (cost=0.00..4.47 rows=5 width=0)
6	Index Cond: ((date part('year'::text, (orderdate)::timestamp without time zone) = '2015'::double precision) AND (date part('month'::text, (orderdate

claro que el número de filas que tendremos que 'traer' será menor, por lo que debería mejorar el rendimiento.

Así obtuvimos nuestro mejor resultado de rendimiento, pasando de 5627 a 23.80:

Haciendo uso de **EXPLAIN ANALYZE** podemos comprobar también una mejora en el tiempo de ejecución (a costa de tardar un poco más en tiempo de planificación pues 'bitmap heap scan' consta de dos pasos en su planificación) entre ejecutar la consulta sin índices, y con el último índice discutido (el de fecha completa):

Sin índices:

8	Planning time: 0.169 ms
9	Execution time: 39.730 ms

Con índice:

9	Planning time: 0.349 ms
10	Execution time: 4.482 ms

F) Vamos ahora a estudiar la forma de las consultas del Anexo 1.

```
EXPLAIN ANALYZE
select customerid
from customers
where customerid not in (
select customerid
from orders
where status='Paid'
```

QUERY PLAN	
text	
1	Seq Scan on customers (cost=3961.65..4490.81 rows=7046 width=4) (actual time=43.959..46.671 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=0.016..36.149 rows=18163 loops=1)
6	Filter: ((status)::text = 'Paid'::text)
7	Rows Removed by Filter: 163627
8	Planning time: 0.240 ms
9	Execution time: 46.906 ms

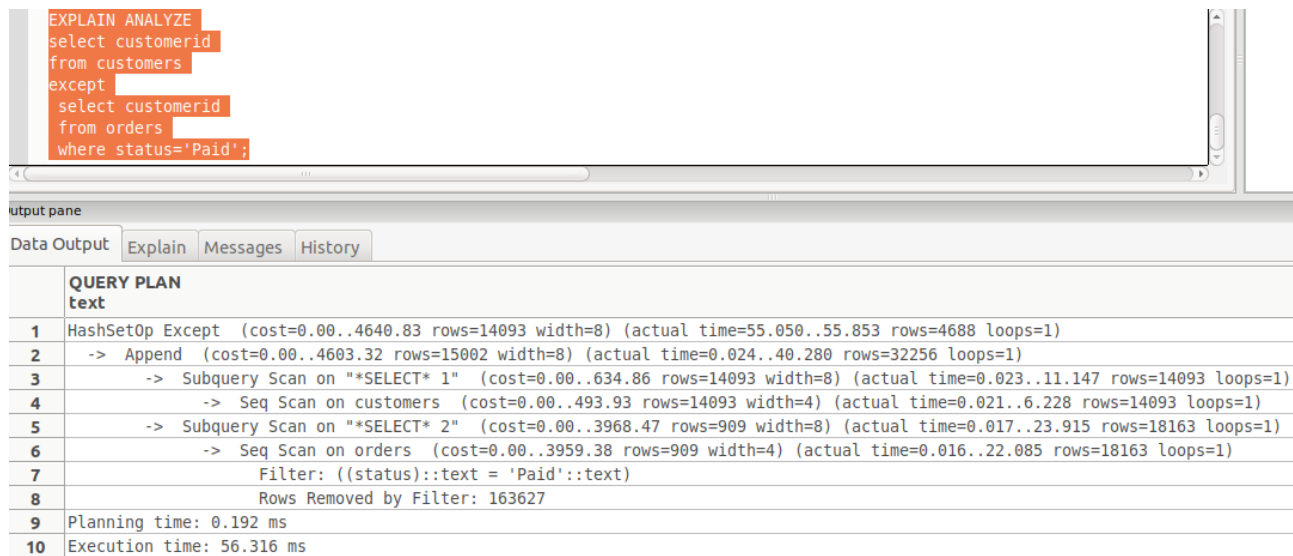
En la primera consulta observamos que se realizan dos *seq scans*, en primer lugar uno de la tabla *orders* y posteriormente uno de la tabla *customers* con la restricción de *status="paid"*.

```
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	
text	
1	HashAggregate (cost=4537.41..4539.41 rows=200 width=4) (actual time=48.834..50.296 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=0.030..30.374 rows=32256 loops=1)
6	-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.029..5.049 rows=14093 loops=1)
7	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.015..21.592 rows=18163 loops=1)
8	Filter: ((status)::text = 'Paid'::text)
9	Rows Removed by Filter: 163627
10	Planning time: 0.438 ms
11	Execution time: 50.586 ms

En la segunda consulta se realiza una operación de *HashAggregate* y posteriormente se realizan en paralelo dos *seq scan*, donde se recorren secuencialmente las tablas de

orders y *costumers*. Por este motivo esta query se beneficia de la ejecución en paralelo.



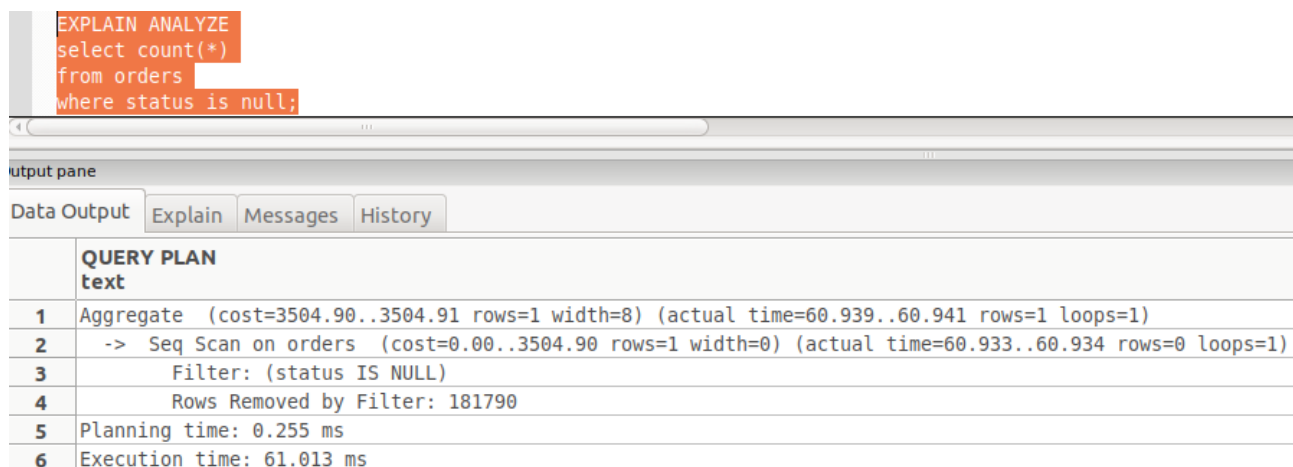
```
EXPLAIN ANALYZE
select customerid
from costumers
except
select customerid
from orders
where status='Paid';
```

	QUERY PLAN text
1	HashSetOp Except (cost=0.00..4640.83 rows=14093 width=8) (actual time=55.050..55.853 rows=4688 loops=1)
2	-> Append (cost=0.00..4603.32 rows=15002 width=8) (actual time=0.024..40.280 rows=32256 loops=1)
3	-> Subquery Scan on "SELECT* 1" (cost=0.00..634.86 rows=14093 width=8) (actual time=0.023..11.147 rows=14093 loops=1)
4	-> Seq Scan on costumers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.021..6.228 rows=14093 loops=1)
5	-> Subquery Scan on "SELECT* 2" (cost=0.00..3968.47 rows=909 width=8) (actual time=0.017..23.915 rows=18163 loops=1)
6	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.016..22.085 rows=18163 loops=1)
7	Filter: ((status)::text = 'Paid')::text
8	Rows Removed by Filter: 163627
9	Planning time: 0.192 ms
10	Execution time: 56.316 ms

En la tercera consulta se realiza una operacion de *HashSetOp* cuyo coste de ejecución a diferencia de las dos anteriores es (cost=0.00..4640.83 rows=14093 width=8) por lo que mostrará resultados nada más comenzar la ejecución. Además, esta consulta realiza un *append*, que tiene dos *subqueries* que, al igual que la anterior consulta, se benefician de la ejecución en paralelo.

G) Estudiamos el coste de ejecución de las dos consultas del anexo 2 con la sentencia EXPLAIN. Para este primer analisis no usamos ningún índice y obtenemos los siguientes resultados:

- **Consulta 1**



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

	QUERY PLAN text
1	Aggregate (cost=3504.90..3504.91 rows=1 width=8) (actual time=60.939..60.941 rows=1 loops=1)
2	-> Seq Scan on orders (cost=0.00..3504.90 rows=1 width=0) (actual time=60.933..60.934 rows=0 loops=1)
3	Filter: (status IS NULL)
4	Rows Removed by Filter: 181790
5	Planning time: 0.255 ms
6	Execution time: 61.013 ms

- **Consulta 2**

<pre>EXPLAIN ANALYZE select count(*) from orders where status = 'Shipped';</pre>	
<p>Output pane</p> <p>Data Output Explain Messages History</p>	
	<p>QUERY PLAN</p> <p>text</p>
1	Finalize Aggregate (cost=4210.86..4210.87 rows=1 width=8) (actual time=56.890..58.611 rows=1 loops=1)
2	-> Gather (cost=4210.75..4210.86 rows=1 width=8) (actual time=56.839..58.605 rows=2 loops=1)
3	Workers Planned: 1
4	Workers Launched: 1
5	-> Partial Aggregate (cost=3210.75..3210.76 rows=1 width=8) (actual time=48.974..48.974 rows=1 loops=2)
6	-> Parallel Seq Scan on orders (cost=0.00..3023.69 rows=74822 width=0) (actual time=0.043..37.913 rows=63662 loops=2)
7	Filter: ((status)::text = 'Shipped'::text)
8	Rows Removed by Filter: 27234
9	Planning time: 0.207 ms
10	Execution time: 58.707 ms

A continuación creamos un índice para la tabla orders y la columna status y volvemos a analizar el coste de ejecución de ambas consultas. Observamos que el coste de ejecución mejora considerablemente. También observamos que el tiempo de planificación se incrementa levemente. En cambio, el tiempo de ejecución se reduce enormemente gracias a la introducción del índice. La razón es exactamente la misma que hemos explicado anteriormente en el apartado E.

- **Consulta 1**

<p>Data Output Explain Messages History</p>	
	<p>QUERY PLAN</p> <p>text</p>
1	Aggregate (cost=7.29..7.30 rows=1 width=8) (actual time=0.079..0.082 rows=1 loops=1)
2	-> Index Only Scan using index status on orders (cost=0.42..7.29 rows=1 width=0) (actual time=0.072..0.072 rows=0 loops=1)
3	Index Cond: (status IS NULL)
4	Heap Fetches: 0
5	Planning time: 0.530 ms
6	Execution time: 0.165 ms

- **Consulta 2**

<p>Data Output Explain Messages History</p>	
	<p>QUERY PLAN</p> <p>text</p>
1	Finalize Aggregate (cost=4210.86..4210.87 rows=1 width=8) (actual time=39.887..42.230 rows=1 loops=1)
2	-> Gather (cost=4210.75..4210.86 rows=1 width=8) (actual time=39.786..42.223 rows=2 loops=1)
3	Workers Planned: 1
4	Workers Launched: 1
5	-> Partial Aggregate (cost=3210.75..3210.76 rows=1 width=8) (actual time=33.446..33.447 rows=1 loops=2)
6	-> Parallel Seq Scan on orders (cost=0.00..3023.69 rows=74822 width=0) (actual time=0.052..25.684 rows=63662 loops=2)
7	Filter: ((status)::text = 'Shipped'::text)
8	Rows Removed by Filter: 27234
9	Planning time: 0.275 ms
10	Execution time: 42.305 ms

A continuación ejecutamos la sentencia ANALYZE sobre la tabla orders y volvemos a analizar los costes, el tiempo y el plan de ejecución de ambas consultas.

En el caso de la consulta 1, observamos que la planificación de la consulta cambia. Se sustituye el *seq scan* de la tabla *orders* por una búsqueda por índice. Por este motivo el tiempo de ejecución, el tiempo y el coste mejora considerablemente.

Sin embargo, en la consulta 2 la mejora de tiempo no es tan notable, aunque sigue siendo considerable. Observamos que la planificación no cambia, y la diferencia del coste es insignificante.

- **Consulta 1**

Data Output	Explain	Messages	History
QUERY PLAN			
text			
1	Aggregate (cost=7.27..7.28 rows=1 width=8) (actual time=0.030..0.032 rows=1 loops=1)		
2	-> Index Only Scan using index status on orders (cost=0.42..7.27 rows=1 width=0) (actual time=0.023..0.024 rows=0 loops=1)		
3	Index Cond: (status IS NULL)		
4	Heap Fetches: 0		
5	Planning time: 0.290 ms		
6	Execution time: 0.115 ms		

- **Consulta 2**

Data Output	Explain	Messages	History
QUERY PLAN			
text			
1	Finalize Aggregate (cost=4209.68..4209.69 rows=1 width=8) (actual time=19.766..21.965 rows=1 loops=1)		
2	-> Gather (cost=4209.56..4209.67 rows=1 width=8) (actual time=19.707..21.961 rows=2 loops=1)		
3	Workers Planned: 1		
4	Workers Launched: 1		
5	-> Partial Aggregate (cost=3209.56..3209.57 rows=1 width=8) (actual time=18.488..18.489 rows=1 loops=2)		
6	-> Parallel Seq Scan on orders (cost=0.00..3023.69 rows=74349 width=0) (actual time=0.009..14.444 rows=63662 loops=2)		
7	Filter: ((status)::text = 'Shipped'::text)		
8	Rows Removed by Filter: 27234		
9	Planning time: 0.092 ms		
10	Execution time: 21.993 ms		

A continuación analizamos las otras dos consultas del anexo 2.

- **Consulta 3**

EXPLAIN ANALYZE select count(*) from orders where status = 'Paid';			
Data Output	Explain	Messages	History
QUERY PLAN			
text			
1	Aggregate (cost=2315.33..2315.34 rows=1 width=8) (actual time=24.386..24.388 rows=1 loops=1)		
2	-> Bitmap Heap Scan on orders (cost=356.74..2270.07 rows=18106 width=0) (actual time=5.567..19.799 rows=18163 loops=1)		
3	Recheck Cond: ((status)::text = 'Paid'::text)		
4	Heap Blocks: exact=1686		
5	-> Bitmap Index Scan on index status (cost=0.00..352.21 rows=18106 width=0) (actual time=4.436..4.436 rows=18163 loops=1)		
6	Index Cond: ((status)::text = 'Paid'::text)		
7	Planning time: 0.244 ms		
8	Execution time: 24.485 ms		

Data Output	Explain	Messages	History
	QUERY PLAN text		
1	Aggregate (cost=2313.69..2313.70 rows=1 width=8) (actual time=5.264..5.265 rows=1 loops=1)		
2	-> Bitmap Heap Scan on orders (cost=356.18..2268.61 rows=18034 width=0) (actual time=1.010..4.328 rows=18163 loops=1)		
3	Recheck Cond: ((status)::text = 'Paid'::text)		
4	Heap Blocks: exact=1686		
5	-> Bitmap Index Scan on index status (cost=0.00..351.68 rows=18034 width=0) (actual time=0.845..0.845 rows=18163 loops=1)		
6	Index Cond: ((status)::text = 'Paid'::text)		
7	Planning time: 0.059 ms		
8	Execution time: 5.285 ms		

Ejecutamos la consulta 3 con y sin la ejecución de la sentencia ANALYZE. Observamos que con la sentencia, la planificación permanece idéntica, el coste de ejecución se mantiene y disminuye considerablemente el tiempo de ejecución al igual que en las otras sentencias.

• Consulta 4

EXPLAIN ANALYZE select count(*) from orders where status ='Processed';			
Data Output	Explain	Messages	History
	QUERY PLAN text		
1	Aggregate (cost=2950.84..2950.85 rows=1 width=8) (actual time=28.137..28.138 rows=1 loops=1)		
2	-> Bitmap Heap Scan on orders (cost=718.29..2859.91 rows=36370 width=0) (actual time=9.571..22.537 rows=36304 loops=1)		
3	Recheck Cond: ((status)::text = 'Processed'::text)		
4	Heap Blocks: exact=1685		
5	-> Bitmap Index Scan on index status (cost=0.00..709.19 rows=36370 width=0) (actual time=8.857..8.858 rows=36304 loops=1)		
6	Index Cond: ((status)::text = 'Processed'::text)		
7	Planning time: 0.371 ms		
8	Execution time: 28.236 ms		

Data Output	Explain	Messages	History
	QUERY PLAN text		
1	Aggregate (cost=2939.67..2939.68 rows=1 width=8) (actual time=8.864..8.865 rows=1 loops=1)		
2	-> Bitmap Heap Scan on orders (cost=711.85..2849.53 rows=36055 width=0) (actual time=1.735..6.955 rows=36304 loops=1)		
3	Recheck Cond: ((status)::text = 'Processed'::text)		
4	Heap Blocks: exact=1685		
5	-> Bitmap Index Scan on index status (cost=0.00..702.83 rows=36055 width=0) (actual time=1.569..1.570 rows=36304 loops=1)		
6	Index Cond: ((status)::text = 'Processed'::text)		
7	Planning time: 0.049 ms		
8	Execution time: 8.886 ms		

Ejecutamos la consulta 4 igual que con la 3. La planificación se mantiene idéntica una vez más, y el coste prácticamente invariable. El tiempo de ejecución en cambio vuelve a bajar considerablemente gracias a la sentencia ANALYZE.

TRANSACCIONES

APARTADO H

Para el apartado H usando el esqueleto suministrado, hemos completado la función correspondiente a *borraCliente* de *database.py* para cumplir todos los requisitos de las transacciones de este apartado. Además cambiamos *borraCliente.html* para que el form sea de tipo POST.

Empezamos realizando la sentencia *BEGIN* para comenzar la transacción y la dividimos en tres etapas:

En primer lugar se borran los datos de la tabla *orderdetail* de ese cliente (query1). A continuación, se borran los datos de la tabla *orders* para ese cliente (query2). Por último borramos al cliente de la tabla *customers* (query3).

Si no ha habido ningún error, se realiza la sentencia *COMMIT* para que se agreguen los nuevos cambios a nuestra base de datos. Si hay algún error, se realiza la sentencia *ROLLBACK* para dejar los datos como estaban antes del *BEGIN* y evitar inconsistencias.

Para el caso que realicemos un commit intermedio, lo que hacemos es dividir la transacción en dos sub-transacciones. Por lo que después de ejecutar query1 realizamos un *COMMIT* para guardar los cambios en la base de datos, y un *BEGIN* para continuar con el resto de la transacción.

A continuación vamos a mostrar los resultados obtenidos para cada una de las opciones que podíamos escoger para la transacción:

- **Transacción sin errores**

Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- ☒ Transacción vía sentencias SQL
☐ Transacción vía funciones SQLAlchemy

☐ Ejecutar commit intermedio

☐ Provocar error de integridad

Duerme segundos (para forzar deadlock).

Trazas

1. Hacemos BEGIN
2. Borramos datos del cliente de Orderdetail
3. Borramos datos del cliente de Orders
4. Borramos datos del cliente de Costumers
5. Borrado completado con éxito
6. COMMIT realizado

```
select customerid
from orders
where customerid = 12
```

output pane

Data Output Explain Message

customerid
integer

```
select orderid
from orderdetail
where orderdetail
```

output pane

Data Output Explain Message

orderid
integer

```
select customerid
from customers
where customerid = 12
```

output pane

Data Output Explain Message

customerid
integer

Observamos que se han borrado los datos de las tablas correspondientes.

- **Transacción con errores**

Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- ☒ Transacción vía sentencias SQL
☐ Transacción vía funciones SQLAlchemy

☐ Ejecutar commit intermedio

☒ Provocar error de integridad

Duerme segundos (para forzar deadlock).

Trazas

1. Hacemos BEGIN
2. Borramos datos del cliente de Orderdetail
3. Se ha producido algun error al borrar el cliente
4. Realizamos ROLLBACK

Obervamos que al hacer el *ROLLBACK*, la base de datos se mantiene idéntica a como estaba antes del *BEGIN*, no se efectúa ninguno de los cambios.

- Transacción con errores y COMMIT intermedio

Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- ☒ Transacción vía sentencias SQL
☐ Transacción vía funciones SQLAlchemy

☒ Ejecutar commit intermedio

☒ Provocar error de integridad

Duerme segundos (para forzar deadlock).

Trazas

1. Hacemos BEGIN
2. Borramos datos del cliente de Orderdetail
3. Realizamos COMMIT intermedio
4. Se ha producido algun error al borrar el cliente
5. Realizamos ROLLBACK

Al realizar un commit intermedio, los cambios efectuados en la tabla orderdetail se aplicaran a nuestra base de datos, mientras que los de las tablas orders y customers, al producirse un error, se realiza un *ROLLBACK* y no se efectúan. Realizamos un *BEGIN* después de hacer *COMMIT* debido a que si hay un error, como en este caso, al hacer *ROLLBACK*, volverá al estado antes del *BEGIN*, manteniendo así los cambios que se hayan realizado en ese *COMMIT*.

```
select *  
from orderdetail  
where orderdetail.orderid in  
(select orderid from orders)
```

Output pane

Data Output	Explain	Messages	History												
<table><thead><tr><th>orderid</th><th>prod_id</th><th>price</th><th>quantity</th></tr><tr><th>integer</th><th>integer</th><th>numeric</th><th>integer</th></tr></thead><tbody><tr><td></td><td></td><td></td><td></td></tr></tbody></table>	orderid	prod_id	price	quantity	integer	integer	numeric	integer							
orderid	prod_id	price	quantity												
integer	integer	numeric	integer												

Observamos que el borrado sobre realizados en la tabla orderdetail si que se ha efectuado, a diferencia de los otros dos.

APARTADO I

En este apartado estudiamos los bloqueos y deadlocks que se pueden producir. Para ello partimos de nuevo de la base de datos que se nos proporciona limpia. A continuación, se nos pedía implementar en un script la creación de una nueva columna *promo* en *customers*.

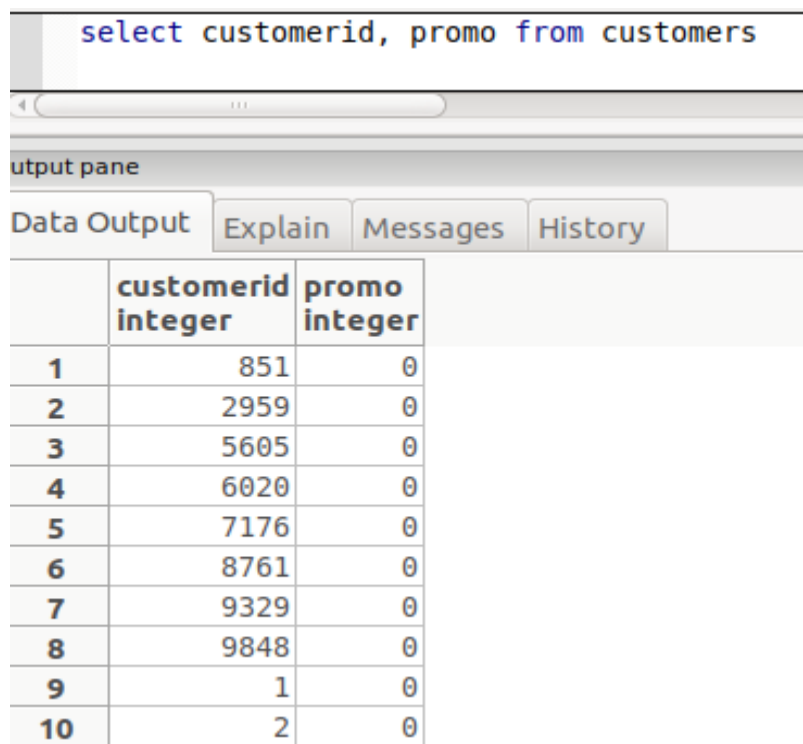
Esto lo llevamos a cabo con la siguiente sentencia:

```
alter table customers add column promo integer default 0;
```

También se nos pedía crear un *trigger* sobre esta tabla *customers* que hiciese un descuento en los artículos de un carrito cuando se alterase la columna recién creada *promo* de un cliente, del porcentaje indicado por esta. Añadimos también un *sleep* al comienzo de nuestro *trigger*, con el fin de crear el deadlock que se nos pide más adelante.

Todo esto se lleva a cabo en el archivo llamado **updPromo.sql**.

Mostramos a continuación capturas del funcionamiento correcto tanto de la creación de la columna *promo*, como del *trigger*.



The screenshot shows a database client interface. At the top, a SQL query is entered: `select customerid, promo from customers`. Below the query, there is a tabbed interface with 'Data Output' selected. The results are displayed in a table with two columns: 'customerid' and 'promo'. The data shows 10 rows of customer records with their IDs and promo values (mostly 0, with some non-zero values like 1 and 2).

	customerid integer	promo integer
1	851	0
2	2959	0
3	5605	0
4	6020	0
5	7176	0
6	8761	0
7	9329	0
8	9848	0
9	1	0
10	2	0

Para poder comprobar el funcionamiento de nuestro *trigger* necesitábamos un customer con el carrito a *null*, por lo que buscamos en nuestra base de datos mediante la consulta:

```
select * from orders where status is NULL; (1)
```

Observamos entonces que no había ningún caso en la base de datos con el status a *null*, por lo que procedimos a crearlo. Pusimos la cesta de algunos *customerid* a *null* para poder trabajar con ellos, mediante la sentencia:

update customers set status = null where customerid = x (2)

con x ciertos *customerids* presentes en la base de datos.

Tras obtener el resultado cambiamos la columna promo al 50% para este cliente mediante la sentencia:

update customers set promo = 50 where customerid = x (3)

	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying(10)
1	207	2018-03-24	10	131.1141932501155802	15	150.78	
2	204	2016-12-14	10	116.5973185390661120	15	134.09	
3	205	2015-11-25	10	83.0328247803975958	15	95.49	
4	203	2016-07-03	10	43.2732316227461858	15	49.76	
5	208	2015-08-07	10	23.3009708737864078	15	26.80	
6	206	2019-03-29	10	11.6504854368932039	15	13.40	
7	202	2017-02-14	10	95.6079519186315303	15	109.95	

Volvemos a ejecutar la sentencia (1) para comprobar que, efectivamente, netamount y total amount se han actualizado, y en netamount tenemos la mitad del precio anterior, pues el descuento se ha aplicado.

	orderid integer	orderdate date	customerid integer	netamount numeric(10,2)	tax numeric	totalamount numeric(10,2)	status character varying(10)
1	207	2018-03-24	10	72.11	15	82.93	
2	204	2016-12-14	10	64.13	15	73.75	
3	205	2015-11-25	10	45.67	15	52.52	
4	203	2016-07-03	10	23.80	15	27.37	
5	208	2015-08-07	10	12.82	15	14.74	
6	206	2019-03-29	10	6.41	15	7.37	
7	202	2017-02-14	10	52.58	15	60.47	

Para el apartado G, tomamos diez segundos de sleep para borraCliente y cinco segundos para el trigger y observamos que el trigger espera a que finalice la transacción, que acabará con error (*ROLLBACK*) y ya se actualizará la tabla orders de forma correcta. Explicaremos los bloqueos más adelante.

Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- ☒ Transacción vía sentencias SQL
☐ Transacción vía funciones SQLAlchemy

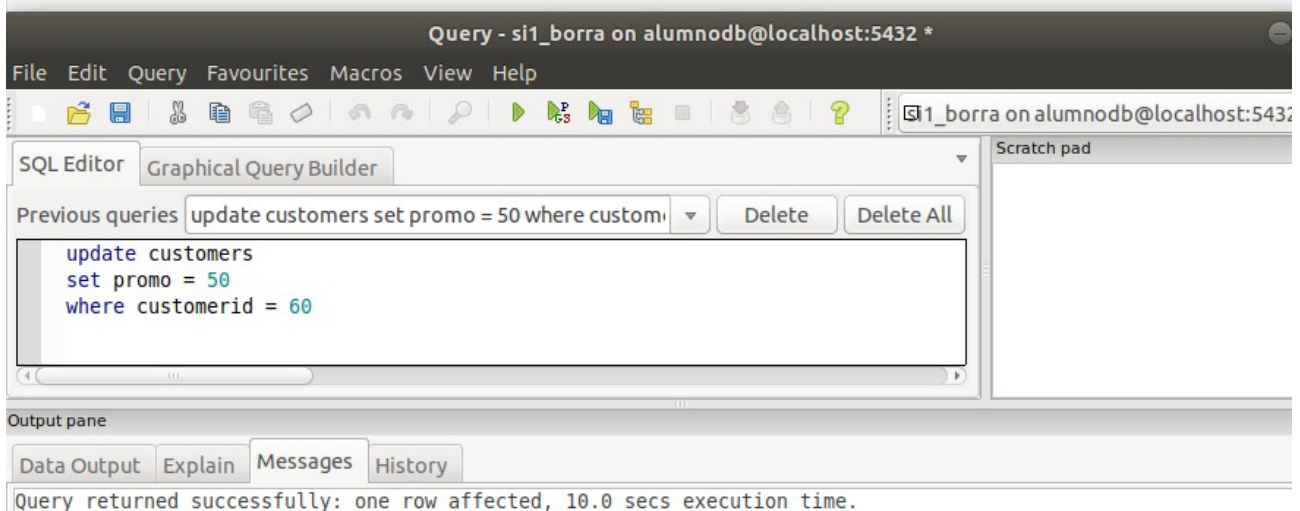
☐ Ejecutar commit intermedio

☒ Provocar error de integridad

Duerme segundos (para forzar deadlock).

Trazas

1. Hacemos BEGIN
2. Borramos datos del cliente de Orderdetail
3. Borramos datos del cliente de Orders
4. Se ha producido algun error al borrar el cliente
5. Realizamos ROLLBACK



Por otro lado, se nos pide comprobar que durante el *sleep* en la página de borrado (que nosotros insertamos dentro del *database.py*, en la función de borrar cliente del-Customer, antes de borrar los datos del cliente de *customers* cuando no hay fallo) o el contenido en el trigger, los datos alterados por la página o por el *trigger* no son visibles.

Esto se debe en el caso de *delCostumer()* a que los cambios efectuados no se producen hasta que se realiza la sentencia *COMMIT* (que es después del *sleep*). En el caso del *trigger*, como realizamos el *sleep* antes de modifica la tabla orders, no se mostrarán las modificaciones durante el *sleep*.

- **Caso *delCostumer()***

1. Antes de iniciar la transacción:

Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- ☒ Transacción vía sentencias SQL
☐ Transacción vía funciones SQLAlchemy

☐ Ejecutar commit intermedio

☐ Provocar error de integridad

Duerme segundos (para forzar deadlock).

Trazas

Previous queries `update customers set promo = 100 where custom`

`select * from customers where customerid = 8`

Output pane

Data Output Explain Messages History

	customerid	firstname	lastname
	integer	character varying(50)	character varying(50)
1	8	refine	whelp

2. Mientras se realiza la transacción:

Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- ☒ Transacción vía sentencias SQL
☐ Transacción vía funciones SQLAlchemy

☐ Ejecutar commit intermedio

☐ Provocar error de integridad

Duerme segundos (para forzar deadlock).

Trazas

Previous queries `update customers set promo = 100 where custom`

`select * from customers where customerid = 8`

Output pane

Data Output Explain Messages History

	customerid	firstname	lastname
	integer	character varying(50)	character varying(50)
1	8	refine	whelp

3. Al finalizar la transacción

Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- ☒ Transacción vía sentencias SQL.
☐ Transacción vía funciones SQLAlchemy

☐ Ejecutar commit intermedio

☒ Provocar error de integridad

Duerme segundos (para forzar deadlock).

Trazas

1. Hacemos BEGIN
2. Borramos datos del cliente de Orderdetail
3. Borramos datos del cliente de Orders
4. Borramos datos del cliente de Costumers
5. Borrado completado con exito
6. COMMIT realizado

Previous queries | `update customers set promo = 100 where custon`

`select * from customers where customerid = 8`

Output pane

Data Output | Explain | Messages | History

	customerid	firstname	lastname
	integer	character varying(50)	character varying(50)

- **Caso trigger**

1. Antes de actualizar promo:

`select * from orders where customerid = 10`

Output pane

Data Output | Explain | Messages | History

	orderid	orderdate	customerid	netamount	tax	totalamount	status
	integer	date	integer	numeric	numeric	numeric	character varying(10)
1	207	2018-03-24	10	131.1141932501155802	15	150.78	
2	204	2016-12-14	10	116.5973185390661120	15	134.09	
3	205	2015-11-25	10	83.0328247803975958	15	95.49	
4	203	2016-07-03	10	43.2732316227461858	15	49.76	
5	208	2015-08-07	10	23.3009708737864078	15	26.80	
6	206	2019-03-29	10	11.6504854368932039	15	13.40	
7	202	2017-02-14	10	95.6079519186315303	15	109.95	

2. Nada mas actualizar promo

```
select * from orders where customerid = 10
```

	orderid integer	orderdate date	customerid integer	netamount numeric	tax numeric	totalamount numeric	status character varying(10)
1	207	2018-03-24	10	131.1141932501155802	15	150.78	
2	204	2016-12-14	10	116.5973185390661120	15	134.09	
3	205	2015-11-25	10	83.0328247803975958	15	95.49	
4	203	2016-07-03	10	43.2732316227461858	15	49.76	
5	208	2015-08-07	10	23.3009708737864078	15	26.80	
6	206	2019-03-29	10	11.6504854368932039	15	13.40	
7	202	2017-02-14	10	95.6079519186315303	15	109.95	

3. Pasado el tiempo del *sleep* del trigger

	orderid integer	orderdate date	customerid integer	netamount numeric(10,2)	tax numeric	totalamount numeric(10,2)	status character varying(10)
1	207	2018-03-24	10	72.11	15	82.93	
2	204	2016-12-14	10	64.13	15	73.75	
3	205	2015-11-25	10	45.67	15	52.52	
4	203	2016-07-03	10	23.80	15	27.37	
5	208	2015-08-07	10	12.82	15	14.74	
6	206	2019-03-29	10	6.41	15	7.37	
7	202	2017-02-14	10	52.58	15	60.47	

OK. Unix Ln 1, Col 43, Ch 43

Query - sl1_borra on alumnodb@localhost:5432 *

File Edit Query Favourites Macros View Help

SQL Editor Graphical Query Builder

Previous queries Delete Delete All

```
update customers set promo = 50 where customerid = 10
```

Output pane

Data Output Explain Messages History

Query returned successfully: one row affected, 5.0 secs execution time.

Se producen dos bloqueos. El trigger se activa cuando se produce un cambio en la tabla customers y se bloquean las filas que son alteradas de dicha tabla. Por otro lado, la transacción al borrar los datos de la tabla orderdetail y de la tabla orders, se produce un bloqueo en las filas de dichas tablas.

Además, al acabar los *sleep* de ambas partes el trigger quiere acceder a una fila bloqueada por la transacción de *orders* mientras que la transacción quiere acceder a una fila de customers bloqueada por el trigger. Por esta razón, se produce un interbloqueo entre ambas partes.

A continuación, ajustamos los tiempos de ambos sleeps (ambos a cinco segundos) y observamos que se produce un *deadlock* y PostgreSQL automáticamente finaliza el trigger para resolverlo.

Ejemplo de Transacción con Flask SQLAlchemy

Customer ID:

- ☒ Transacción vía sentencias SQL
☐ Transacción vía funciones SQLAlchemy

☐ Ejecutar commit intermedio

☒ Provocar error de integridad

Duerme segundos (para forzar deadlock).

Trazas

1. Hacemos BEGIN
2. Borramos datos del cliente de Orderdetail
3. Borramos datos del cliente de Orders
4. Borramos datos del cliente de Customers
5. Borrado completado con éxito
6. COMMIT realizado

```
Previous queries update customers set promo = 100 where custon Delete Delete All
update customers
set promo = 50
where customerid = 6
```

```
Output pane
Data Output Explain Messages History
ERROR: deadlock detected
DETAIL: Process 30961 waits for ShareLock on transaction 158411; blocked by process 31230.
Process 31230 waits for ShareLock on transaction 158412; blocked by process 30961.
HINT: See server log for query details.
CONTEXT: while updating tuple (1686,37) in relation "orders"
SQL statement "update orders
set netamount = (netamount * (1 + (old.promo / 100.0))) * (1 - ( new.promo / 100.0))
where orders.customerid = new.customerid and
orders.status is NULL"
PL/pgSQL function update_promos() line 6 at SQL statement
***** Error *****
ERROR: deadlock detected
```

Para solucionar el problema de los deadlocks tenemos dos aproximaciones: prevención de estos antes de que ocurran, o detección y recuperación del sistema una vez se han producido.

En el caso de la prevención de deadlocks tenemos varias opciones:

- **Prevención por retroceso:** Ante una transacción que solicita un bloqueo y que puede provocar un interbloqueo, esta es retrocedida.
- **Wait-die:** Si tenemos dos transacciones T1 y T2, T1 esperará si es más antigua que T2, es decir, si $ts(T1) < ts(T2)$. De lo contrario, T1 será retrocedida.

En el caso de la detección y recuperación del sistema ante un deadlock tenemos también varias opciones:

- **Detección por timeout:** Cancelaríamos la transacción tras un tiempo de espera.

Otra opción es llevar a cabo las siguientes acciones:

- **Seleccionar una víctima:** dado un conjunto de transacciones en deadlock se determinan qué transacciones deben retroceder. Este es el mecanismo que postgres lleva a cabo y que podemos observar en la captura. Se decide que la consulta de update no se realice para que los recursos estén disponibles para la transacción.
- **Retroceso:** se determina si la transacción debe ser retrocedida por completo (deshaciendo todos los cambios que hizo) o parcialmente, lo cual requiere información adicional del sistema.
- **Inanición:** Además se deberá elegir como víctima a la transacción un número finito de veces para evitar que la misma entre en estado de inanición, es decir, que nunca se complete su ejecución.