

MEMORIA P3 - SI

1. NO-SQL
2. Optimización de consultas
3. Transacciones

Iván Fernández París
Javier Fraile Iglesias

1. NO-SQL

A)

Para la parte de NO-SQL, se han realizado las consultas para cada uno de los puntos en funciones de python por separado.

Además en la primera consulta donde se extraen las películas, aprovechamos para traernos el id de la película que será beneficioso para las posteriores consultas.

A continuación, guardamos en una función toda la información necesaria al estilo mongo (notación JSON).

B)

Para la consulta del 100% de coincidencia con los géneros hemos guardado en un cursor las 400 películas, y posteriormente las recorremos una a una, cogiendo los géneros de las películas a comparar en cada momento, si el número de géneros es el mismo y además los géneros de uno excepto “excepts” los del otro son 0, implica que son los mismos.

Para la del 50% en python primero se comprueba que realmente tenga más de un género, y posteriormente en postgres de la misma manera (con cursores), se mira que los de uno menos los del otro sean los de la primera entre 2, de esta manera, sabemos que la mitad de los géneros han tenido que coincidir al menos.

C)

Para crear las colecciones de mongoDB, sabemos que tienen notación JSON.

Primero nos conectamos al cliente de mongoDB, desde ese cliente creamos la base de datos Si, y a continuación creamos la colección topUK.

Finalmente, con el insert_many que pymongo incorpora (instrucción de mongoDB), se insertan los registros que hemos preparado en los anteriores apartados con ese estilo JSON.

D)

La primera consulta nos muestra las películas de ciencia ficción entre 1994 y 1998, usando el “find” de mongo, accediendo al atributo “genres” y al “year”, donde los años se especifican con “\$gte” (mayor o igual) y “\$lte” (menor o igual).

La segunda nos pide las películas de drama en 1998 que empiezan por “The”, para ello, con el “\$regex”, que se encarga de buscar cadenas por el title, en este caso que finalicen con The\$, que es lo que conseguimos con el dólar, porque los títulos tienen esa notación (...., The).

La última nos pide las películas donde coincidan en el reparto Julia y Alec, para ello, poniendo \$and[], especificamos la condición “and” donde añadimos en su interior el nombre de los dos actores.

2. Optimización de consultas

E)

- a) La consulta que se realiza es extraer tanto el año como el mes del “orderdate” además del tipo de tarjeta de crédito “VISA”.

```
-explain select count(distinct c.city)
from customers c inner join orders o on c.customerid = o.customerid where c.creditcardtype = 'VISA'
extract (month from o.orderdate) = 04 and extract (year from o.orderdate) = 2016;
```

- b) En cuanto a la ejecución con explain obtenemos los siguientes resultados:

```
create index index_creditCardType on customers(customerid, creditcardtype);
```

```
-explain select count(distinct c.city)
from customers c inner join orders o on c.customerid = o.customerid where c.creditcardtype = 'VISA'
extract (month from o.orderdate) = 04 and extract (year from o.orderdate) = 2016;
```

```
plain select count(distinct c.city) from customers;
```

```
EXPLAIN PLAN
Aggregate (cost=5385.40..5385.41 rows=1 width=0)
  > Gather (cost=1000.28..5385.40 rows=1 width=0)
    Workers Planned: 1
      > Nested Loop (cost=0.29..4385.30 rows=1 width=0)
        > Parallel Seq Scan on orders o (cost=0.00..4360.38 rows=3 width=4)
          Filter: ((date_part('month'::text, orderdate)::timestamp without time zone) = '4'::double precision) AND (date_part('year'::text, (orderdate)::timestamp without time zone) = '2016'::double precision)
        > Index Scan using customers_pkey on customers c (cost=0.29..8.30 rows=1 width=10)
          Index Cond: (customerid = o.customerid)
          Filter: ((creditcardtype)::text = 'VISA'::text)
```

Como se puede observar la ejecución con explain nos muestra una ejecución en árbol, donde cada parte de la ejecución es un nodo y es el propio “postgresql” el que se encarga de organizar como él prevé que vaya a ser mas optimo para la ejecución.

En nuestro caso, realiza internamente un parallel sec scan, que se encarga de recorrer secuencialmente la tabla orders, filtrando esta por el mes y año que hemos especificado.

Además, al mismo nivel, un Index Scan, que es un escaneo más rápido ya que lo realiza sobre el índice implícito que tiene la propia clave primaria de la tabla customers, filtrando esta última por el tipo ‘VISA’ de tarjeta de crédito.

El nested loop se encarga de juntar estos dos escaneos por individual y agruparlos (se hace debido al join).

A nivel superior, el nodo Gather, el cual se encarga de juntar los resultados obtenidos de los hilos, pero en nuestro caso como aparece en “Workers Planned”, tan solo es 1.

Ya como cierre y última operación es el Aggregate, el cual se encarga de hacer el count de la solución.

- c) Como vemos, por la parte de los customers, el índice es implícito ya que está creado por la propia clave primaria de la relación customers. Con lo cual por ese lado no podemos optimizar mucho más.

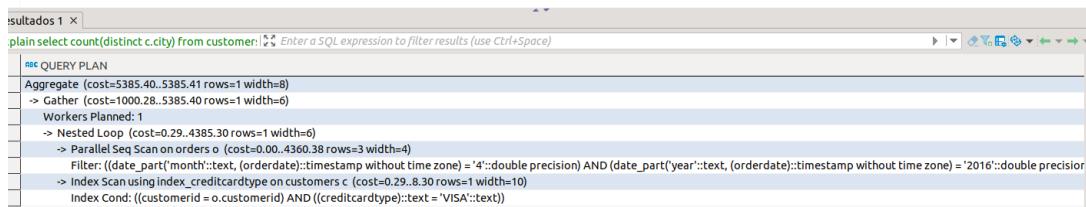
Parece que lo que se puede buscar optimizar es la otra parte de la relación sobre la tabla orders ya que es un escaneo secuencial de la tabla con lo cual es más costoso.

Sin embargo, hemos intentado optimizar ese escaneo secuencial de orders y no hemos sido capaces de crear un índice que sea óptimo, ya que si o si tiene que leer todos los registros de orders para obtener el resultado.

- d) La única forma que hemos conseguido que la planificación cambie es con un índice explícito que abarca “cusotmerid” y “creditcardtype”, pero que tampoco se ha visto mejorado en cuanto al coste.

```
create index index_creditCardType on customers(customerid, creditcardtype);

explain select count(distinct c.city)
  from customers c inner join orders o on c.customerid = o.customerid where c.creditcardtype = 'VISA'
  extract (month from o.orderdate) = 04 and extract (year from o.orderdate) = 2016;
```



F)

En la primera consulta, analizamos el plan de ejecución, y observamos que primeramente, a nivel interno, se está realizando un escaneo secuencial de la tabla orders aplicando al mismo un filtro donde el status sea igual a pagado, que esta consulta tarda 28 segundos aproximadamente, y a continuación una vez obtenidos, se hace otro escaneo secuencial, pero esta vez sobre la tabla customers y sin filtro, por lo cual esta tarda tan solo 5 segundos.

Finalmente, tan solo se aplica el filtro de not in de la primera sobre la segunda.

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

Explain Analyze results:

- Seq Scan on customers (cost=3961.65..4490.81 rows=7046 width=4) (actual time=35.030..40.733 rows=4688 loops=1)
 - Filter: (NOT (hashed SubPlan 1))
 - Rows Removed by Filter: 9405
- SubPlan 1
 - Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.066..28.473 rows=18163 loops=1)
 - Filter: ((status)::text = 'Paid '::text)
 - Rows Removed by Filter: 163627
- Planning time: 0.246 ms
- Execution time: 40.941 ms

En cuanto a la segunda consulta, a nivel interno vemos que se ejecuta un append, que viene a causa de ese union all, y dentro se realizan de nuevo 2 escaneos secuenciales, uno de ellos coge todos los customerid de customers, mientras que el otro coge los de la tabla orders que cumplen que el estado sea ha pagado (que de nuevo es el que más tarda).

Como vemos de las 15000 filas que coge este append, 14093 las ha cogido customers tardando mucho menos (5 segundos aprox), mientras que las restantes las ha cogido del segundo seq scan de la tabla orders, 909 filas, a un tiempo de 26 segundos aproximadamente, por el hecho de tener que haber realizado el filtro.

Eso se debe en gran parte porque el campo de status no está indexado.

Finalmente, cuando se obtienen los registros de esa unión, lo que se hace es agrupar por dichos ids, y una vez unidos se agrupan en un tiempo de 2 segundos aproximadamente mediante “HashAggregate”.

```

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

```

Resultados 1 ×

explain analyze select customerid from (select cus | Enter a SQL expression to filter results (use Ctrl+Space)

	nyc QUERY PLAN
1	HashAggregate (cost=4537.41..4539.41 rows=200 width=4) (actual time=48.976..50.844 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=0.014..33.874 rows=32256 loops=1)
6	-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.014..4.676 rows=14093 loops=1)
7	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.017..26.147 rows=18163 loops=1)
8	Filter: ((status)::text = 'Paid'::text)
9	Rows Removed by Filter: 163627
10	Planning time: 0.215 ms
11	Execution time: 51.202 ms

Para la tercera consulta, realmente está pasando algo muy parecido, porque a nivel interno, dentro de un append, se están realizando los planes de ejecución para dos subconsultas mediante “subquery scan”, pero en el interior de cada una se sigue realizando el escaneo secuencial de la tabla customers y orders con el filtro de pagado respectivamente, tardando lo mismo cada una que el apartado anterior, sin embargo, a nivel exterior del append, se está realizando mediante un “HashSetOp”, (en su operación except).

Como podemos observar, los tiempos son parecidos, ya que los escaneos secuenciales son idénticos a los del ejemplo 2, y la posterior operación de “HashSetOp” tarda 1 segundo tan solo.

```

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

```

Resultados 1 ×

explain analyze select customerid from customers | Enter a SQL expression to filter results (use Ctrl+Space)

	nyc QUERY PLAN
1	HashSetOp Except (cost=0.00..4640.83 rows=14093 width=8) (actual time=50.789..51.791 rows=4688 loops=1)
2	-> Append (cost=0.00..4603.32 rows=15002 width=8) (actual time=0.048..37.602 rows=32256 loops=1)
3	-> Subquery Scan on "SELECT* 1" (cost=0.00..634.86 rows=14093 width=8) (actual time=0.048..6.495 rows=14093 loops=1)
4	-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4) (actual time=0.046..4.317 rows=14093 loops=1)
5	-> Subquery Scan on "SELECT* 2" (cost=0.00..3968.47 rows=909 width=8) (actual time=0.017..28.269 rows=18163 loops=1)
6	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4) (actual time=0.016..26.204 rows=18163 loops=1)
7	Filter: ((status)::text = 'Paid'::text)
8	Rows Removed by Filter: 163627
9	Planning time: 0.141 ms
10	Execution time: 52.403 ms

Dentro de las 3 consultas, se distinguen 2 maneras de ejecutar las consultas:

- La primera consulta que se realiza secuencialmente un scan y posteriormente otro.
- La tercera consulta, que se realiza al mismo nivel de manera paralela.

G)

- Una vez restablecida la base de datos ejecutamos las consultas para analizar los tiempo y la planificación sin índices:

- b) El plan de ejecución para las consultas sin índice lo que se realiza es un escaneo secuencial de toda la tabla orders que lo que hace es recorrer toda la tabla, y le aplica el filtro del status dependiendo de cual sea el status de dicha consulta, esta es la parte que más tarda en ambos casos, y posteriormente, se aplica un aggregate, que lo que hace es contar los registros en uno solo, que apenas tarda tiempo.

```

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

```

```

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

```

Resultados 1 x

explain analyze select count(*) from orders where: Enter a SQL expression to filter results (use Ctrl+Space)

1. **Aggregate** (cost=3507.17..3507.18 rows=1 width=8) (actual time=17.333..17.334 rows=1 loops=1)
 -> Seq Scan on orders (cost=0.00..3504.90 rows=909 width=0) (actual time=17.330..17.330 rows=0 loops=1)
 Filter: (status IS NULL)
 Rows Removed by Filter: 181790
Planning time: 0.039 ms
Execution time: 17.356 ms

Resultados 1 x

explain analyze select count(*) from orders where: Enter a SQL expression to filter results (use Ctrl+Space)

1. **Aggregate** (cost=3961.65..3961.66 rows=1 width=8) (actual time=28.843..28.843 rows=1 loops=1)
 -> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=0) (actual time=0.012..21.974 rows=127323 loops=1)
 Filter: ((status)::text = 'Shipped'::text)
 Rows Removed by Filter: 54467
Planning time: 0.053 ms
Execution time: 28.868 ms

Sin embargo, cuando ejecutamos las consultas, podíamos esperar unos resultados parejos ya que ambos hacen la misma operación, pero no es así.

Esto puede deberse por dos causas:

1. Null como tal no es ningún valor, con lo cual no puede ser comparado, por esa razón no ponemos que status = NULL, e igual es más fácil de identificar en ese aspecto a la hora de ejecutar la query, mientras que en la de shipped, es una cadena.
2. Como podemos ver, además, esta query nos devuelve 0 rows, con lo cual ninguna fila cumple dicha condición y no habrá que emplear muchos recursos mientras que en la de shipped devuelve 127000 filas.

- c) Ahora crearemos un índice sobre la tabla orders y su campo status y volvemos a ejecutar las consultas:

```

create index index_status on orders(status);

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

```

```

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

```

Resultados 1 x

explain analyze select count(*) from orders where: Enter a SQL expression to filter results (use Ctrl+Space)

1. **Aggregate** (cost=1498.52..1498.53 rows=1 width=8) (actual time=0.011..0.012 rows=1 loops=1)
 -> Bitmap Heap Scan on orders (cost=19.46..1494.25 rows=909 width=0) (actual time=0.010..0.010 rows=0 loops=1)
 Recheck Cond: (status IS NULL)
 -> Bitmap Index Scan on index_status (cost=0.00..19.24 rows=909 width=0) (actual time=0.009..0.009 rows=0 loops=1)
 Index Cond: (status IS NULL)
Planning time: 0.081 ms
Execution time: 0.034 ms

Resultados 1 x

explain analyze select count(*) from orders where: Enter a SQL expression to filter results (use Ctrl+Space)

1. **Aggregate** (cost=1498.79..1498.80 rows=1 width=8) (actual time=29.060..29.067 rows=1 loops=1)
 -> Bitmap Heap Scan on orders (cost=19.46..1496.52 rows=909 width=0) (actual time=5.653..21.417 rows=127323 loops=1)
 Recheck Cond: ((status)::text = 'Shipped'::text)
 -> Bitmap Index Scan on index_status (cost=0.00..19.24 rows=909 width=0) (actual time=5.454..5.456 rows=127323 loops=1)
 Index Cond: ((status)::text = 'Shipped'::text)
Planning time: 0.049 ms
Execution time: 29.128 ms

- d) Como podemos observar ahora la consulta sobre la parte que antes era seq scan, ahora está en dos fases, un “Bitmap Index Scan”, que lo que hace es crear un mapa de bits con aquellos registros que cumplen la condición, y más externamente, el “Bitmap Heap Scan”, que lo que hace es crear otro mapa de bits también pero esta vez es crear la solución con aquellos registros filtrados por la operación anterior. Podemos observar que el del status a NULL es inmediato, mientras que el del status a Shipped, incluso incrementa, esto puede haber sido provocado porque a pesar de que se realiza en 2 fases y es más rápido, al final hay que seguir visitando todas las páginas de la tabla orders, con lo cual sigue habiendo que capturar toda la tabla.

Además, también podemos predecir que este escaneo de índice no es útil si el tamaño de los resultados es muy grande como es el caso, ya que igual no cabe en el mapa de bits que se crea.

- e) Ahora hemos ejecutado un analyze sobre orders para ver los planes de ejecución que efectúa postgres sobre la tabla orders y volvemos a realizar las consultas:
- f)

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

The screenshot shows the pgAdmin interface with the results of the EXPLAIN ANALYZE command. The results are as follows:

Step	Operation	Cost	Time
1	Aggregate	(cost=7.25..7.26 rows=1 width=8)	(actual time=0.016..0.017 rows=1 loops=1)
2	> Index Only Scan using index_status on orders	(cost=0.42..7.25 rows=1 width=0)	(actual time=0.014..0.014 rows=0 loops=1)
3	Index Cond: (status IS NULL)		
4	Heap Fetches: 0		
5	Planning time: 0.087 ms		
6	Execution time: 0.074 ms		

Se efectúa un index only scan, porque toda la información está contenida en el índice, y al ser 0 filas coincidentes, no hace falta más recursos.

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

The screenshot shows the pgAdmin interface with the results of the EXPLAIN ANALYZE command. The results are as follows:

Step	Operation	Cost	Time
1	Finalize Aggregate	(cost=4210.49..4210.50 rows=1 width=8)	(actual time=18.734..20.923 rows=1 loops=1)
2	> Gather	(cost=4210.38..4210.49 rows=1 width=8)	(actual time=18.675..20.920 rows=2 loops=1)
3	Workers Planned: 1		
4	Workers Launched: 1		
5	> Partial Aggregate	(cost=3210.38..3210.39 rows=1 width=8)	(actual time=16.898..16.899 rows=1 loops=2)
6	> Parallel Seq Scan on orders	(cost=0.00..3023.69 rows=74676 width=0)	(actual time=0.009..13.132 rows=63662 loops=2)
7	Filter: ((status)::text = 'Shipped'::text)		
8	Rows Removed by Filter: 27234		
9	Planning time: 0.082 ms		
10	Execution time: 20.947 ms		

Para la de shipped el plan de ejecución decide lanzar dos hilos paralelos sobre la tabla orders, y esto lo sabemos gracias al nodo parallel sec scan, que lo que hace es para 2 hilos, ejecuta en un tiempo de 13 ms cada uno, la consulta, con dicho filtro de shipped, obteniendo cada uno la mitad de resultados 63600, por eso aparece como loops 2, ya que cada uno ejecuta un loop.

Por otro lado, a un nivel mayor, se hace un count parcial con cada nodo, pero no se junta hasta llegar al nodo “grater”, que es el que realiza esa unión y finalmente el “aggregate” del nodo padre es el que hace la suma de los dos.

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

The screenshot shows the pgAdmin interface with the results of the EXPLAIN ANALYZE command. The results are as follows:

Step	Operation	Cost	Time
1	Aggregate	(cost=2325.00..2325.01 rows=1 width=8)	(actual time=8.580..8.581 rows=1 loops=1)
2	> Bitmap Heap Scan on orders	(cost=362.67..2279.11 rows=18355 width=0)	(actual time=1.197..7.548 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..358.08 rows=18355 width=0)	(actual time=0.970..0.970 rows=18163 loops=1)
6	Index Cond: (status)::text = 'Paid'::text		
7	Planning time: 0.118 ms		
8	Execution time: 8.623 ms		

Para esta tercera consulta sobre estatus en “pagado”, se realiza exactamente lo que se explica en el apartado anterior cuando se crea el índice, y en este caso sí que es eficiente, y esto nos hace pensar que efectivamente cuando antes se realizaba con

shipped, no es capaz de optimizar la consulta por ese modo porque el resultado era demasiado grande.

Sin embargo aquí, cómo obtiene 18000 filas, si emplea el plan adecuado y la optimización da sus frutos.

The screenshot shows the PostgreSQL Explain Analyze interface. At the top, there is a code editor window containing the SQL query:explain analyze select count(*)
from orders
where status = 'Processed';Below it is a results window titled "Resultados 1". The results show the query plan:explain analyze select count(*) from orders where
QUERY PLAN
1 Aggregate (cost=2953.45..2953.46 rows=1 width=8) (actual time=13.776..13.777 rows=1 loops=1)
 > Bitmap Heap Scan on orders (cost=719.18..2862.24 rows=36485 width=0) (actual time=2.290..11.371 rows=36304 loops=1)
 Recheck Cond: (status::text = 'Processed'::text)
 Heap Blocks: exact=1685
 > Bitmap Index Scan on index_status (cost=0.00..710.06 rows=36485 width=0) (actual time=2.102..2.102 rows=36304 loops=1)
 Index Cond: (status::text = 'Processed'::text)
Planning time: 0.073 ms
Execution time: 13.809 msThe plan indicates an aggregate step followed by a bitmap heap scan on the "orders" table, which then uses a bitmap index scan on the "index_status" index. The execution time is shown as 13.809 ms.

En el último caso de estatus a “procesado”, se hace exactamente lo mismo con 36000 filas, que sigue siendo un tamaño aceptable para este plan de ejecución.

3. Transacciones

H)

Primero, comentar que no hace falta borrar las restricciones de borrado en “CASCADE” porque las claves foráneas no tienen el tipo de borrado establecido.

En primer lugar, para la primera parte de este ejercicio, hemos construido las consultas de borrado de orders, customers y orders details dado una ciudad que es lo que se pide.

Para entender lo que está pasando hay que tener en cuenta que una transacciones, son procesos atómicos, con lo cual se deben realizar como un todo; a la mínima que se produzca un fallo en cualquier línea de la transacción, esta se deshace.

Por otro lado las transacciones comienzan por begin, y finalizan por commit, y solo cuando llegan a esta segunda fase es cuando los cambios se producen. Si se da algún caso de error y la transacción ejecuta rollback, esta vuelve a su estado inicial(antes del begin).

Ahora ya estamos dispuestos a analizar el resultado:

Si el parámetro de “bFallo” está a True, significa que vamos a provocar un fallo. Para causar, en primer lugar debemos borrar las dependencias de la tabla order details sobre esa ciudad elegida, si a continuación borramos los customers sobre esa ciudad, saldrá error, porque todavía quedan registros de la tabla orders que son referenciados a esos customers que se van a eliminar, en el except se procede a hacer el mecanismo de rollback que lo que hace es establecer cómo estaba el order details que fue modificado.

Trazas

1. Begin;
2. Delete from orderdetail od where od.orderid in (Select o.orderid from orders o join customers c on c.customerid = o.customerid where c.city = 'peyton');
3. -----Borrado en orderdetail => 0 registros
4. Delete from customers c where c.city = 'peyton';
5. Rollback;
6. -----Rollback en orderdetail => 134 registros
7. -----Rollback en customers => 2 registros
8. -----Rollback en orders => 26 registros

Por el contrario, una ejecución correcta (“bFallo” = False), la transacción se va a realizar en el orden correcto => order detail, orders y por último customers, ya que order details tiene dependencias con orders y a su vez orders tiene dependencias con customers.

De esta manera la transacción finaliza con un commit guardando los cambios.

Trazas

```
1. Begin;
2. Delete from orderdetail od where od.orderid in ( Select o.orderid from orders o join customers c on c.customerid = o.customerid where c.city = 'peyton');
3. ----Borrado en orderdetail => 0 registros
4. Delete from orders o where o.customerid in (select c.customerid from customers c where c.city = 'peyton')
5. ----Borrado en orders => 0 registros
6. Delete from customers c where c.city = 'peyton';
7. ----Borrado en customers => 0 registros
8. Commit;
9. ----Commit en orderdetail => 0 registros
10. ----Commit en customers => 0 registros
11. ----Commit en orders => 0 registros
```

Adicionalmente, se pide que si una flag “bCommit” está a true, que se realiza un commit intermedio en el caso de la transacción que da error.

Ese commit intermedio debe ir justo antes de realizar el borrado de los customers en el caso de error, ya que en ese momento ya se han borrado los registros sobre order_details.

Como ya habíamos mencionado, una vez se realiza el commit, el estado de la tabla queda modificado, prevaleciendo sobre posteriores rollbacks.

Además hay que añadir, que después de hacer un commit, la transacción da por finalizada, con lo cual si se quiere continuar con la ejecución de otra transacción se debe empezar de nuevo con el begin.

Trazas

```
1. Begin;
2. Delete from orderdetail od where od.orderid in ( Select o.orderid from orders o join customers c on c.customerid = o.customerid where c.city = 'earp');
3. ----Borrado en orderdetail => 0 registros
4. Commit;
5. ----Commit en orderdetail => 0 registros
6. ----Commit en customers => 1 registros
7. ----Commit en orders => 21 registros
8. Begin;
9. Delete from customers c where c.city = 'earp';
10. Rollback;
11. ----Rollback en orderdetail => 0 registros
12. ----Rollback en customers => 1 registros
13. ----Rollback en orders => 21 registros
```

Como se puede observar, en el paso 4 el commit se ejecuta con éxito, eliminando sin problemas los registros de order_detail.

Aunque posteriormente salta el mismo error que el caso de fallo al intentar borrar customers, ese commit ya se ha realizado luego “no hay marcha atrás”.

Finalmente hay un caso extra, y se produce cuando el país a borrar no existe, esto postgre lo devuelve como un NOT FOUND, sin embargo, como tal que una consulta no devuelve valores no significa que la consulta sea incorrecta, con lo cual entrará en la transacción de la misma manera y se hará commit sobre las 0 coincidencias que se han producido.

Trazas

1. Begin;
2. Delete from orderdetail od where od.orderid in (Select o.orderid from orders o join customers c on c.customerid = o.customerid where c.city = 'noexiste');
3. ----Borrado en orderdetail => 0 registros
4. Delete from customers c where c.city = 'noexiste';
5. ----Borrado en customers => 0 registros
6. Delete from orders o where o.customerid in (select c.customerid from customers c where c.city = 'noexiste')
7. ----Borrado en orders => 0 registros
8. Commit;
9. ----Commit en orderdetail => 0 registros
10. ----Commit en customers => 0 registros
11. ----Commit en orders => 0 registros

I)

Para la segunda parte de este ejercicio se nos pide crear un interbloqueo, pero en primer lugar, creamos una columna en la tabla customers con el campo de la promoción.

Posteriormente, se va a crear el trigger, que se encarga de hacer el update del precio del carrito en base a esa promoción, que se ejecuta justo en el momento de actualizar ese campo de promoción.

Vamos a trabajar sobre un customer cuya ciudad es "herman" y su id es el 1144, le creamos un carrito y así quedan sus orders:

The screenshot shows a MySQL database interface with a query window and a results window. The query is:

```
select * from orders where customerid = 1144;
```

The results window displays the 'orders' table with 16 rows, labeled 'orders 1 X'. The columns are: orderid, orderdate, customerid, netamount, tax, totalamount, and status. The data includes various order details such as orderids 15.231 through 15.236, dates ranging from 2019-10-09 to 2017-09-26, and statuses like Processed, Shipped, and Paid.

	orderid	orderdate	customerid	netamount	tax	totalamount	status
1	15.231	2019-10-09	1.144	39,9445214979	15	45,94	Processed
2	15.224	2020-10-26	1.144	175,9223300971	18	207,59	Shipped
3	15.222	2020-12-18	1.144	33,3980582524	18	39,41	Processed
4	15.234	2019-08-03	1.144	98,1969486824	15	112,93	Shipped
5	15.227	2016-07-09	1.144	121,4979195562	15	139,72	Shipped
6	15.228	2018-05-29	1.144	68,4234858992	15	78,69	Shipped
7	15.237	2016-07-29	1.144	113,7309292649	15	130,79	Paid
8	15.225	2018-07-06	1.144	34,2117429496	15	39,34	Shipped
9	15.229	2018-07-29	1.144	71,012482663	15	81,66	Processed
10	15.233	2017-07-25	1.144	169,671752196	15	195,12	Processed
11	15.232	2019-12-06	1.144	33,7494220989	15	38,81	Processed
12	15.226	2018-10-08	1.144	51,9648636153	15	59,76	Shipped
13	15.230	2020-07-12	1.144	49,0291262136	15	56,38	Shipped
14	15.223	2020-03-07	1.144	31,067961165	15	35,73	Shipped
15	15.235	2017-05-23	1.144	40,7766990291	15	46,89	Shipped
16	15.236	2017-09-26	1.144	54,5538603791	15	62,74	[NULL]

A continuación, vamos a colocar los sleeps en los puntos para crear este interbloqueo:

En la parte de database.py, se va a colocar una vez se hace el borrado de las tablas justo antes de realizar el commit para confirmar los cambios, de esta manera, nos aseguramos que podemos iniciar otra transacción en el transcurso de esta transacción:

```

else:
    """
    Orden Correcto = orderdetail, orders y customers
    """

    dbr.append(orderdetail)
    db_conn.execute(orderdetail)
    comp = list(db_conn.execute(comprueba_orderdetails))[0]
    dbr.append(f"-----Borrado en orderdetail => {comp[0]} registros")

    dbr.append(orders)
    db_conn.execute(orders)
    comp = list(db_conn.execute(comprueba_orders))[0]
    dbr.append(f"-----Borrado en orders => {comp[0]} registros")

    dbr.append(customers)
    db_conn.execute(customers)
    comp = list(db_conn.execute(comprueba_customers))[0]
    dbr.append(f"-----Borrado en customers => {comp[0]} registros")

    time.sleep(int(duerme))

# TODO: confirmar cambios si todo va bien
dbr.append("Commit;")
db_conn.execute("Commit;")

```

En cuanto al trigger, el sleep lo vamos a realizar antes de hacer el update dentro del trigger, y eso es para que por esta parte nos aseguremos de que los datos han sido borrados en la anterior transacción, pero a su vez, que no haya hecho commit antes de que hagamos el update en el trigger:

```

create or replace function updatePrice() returns trigger as $$ 
begin
    perform pg_sleep(5);
    update orders o set
        totalamount = (totalamount * ((100 - new.promo)/100))
    where new.customerid = o.customerid and o.status is null;

    return new;
end;
$$
language 'plpgsql';

```

Una vez tenemos todo preparado vamos a explicar paso a paso lo que sucede:

1. Vamos en primer lugar a lanzar la página de borrar ciudades con la ciudad base y un tiempo de 15 segundos de sleep:

Ejemplo de Transacción con SQLAlchemy

Ciudad: herman

Transacción vía sentencias SQL
Transacción vía funciones SQLAlchemy

Ejecutar commit intermedio
Provocar error de integridad

Duerme segundos (para forzar deadlock).

2. Ahora rápidamente, vamos a una conexión de base de datos (en nuestro casodbeaver-ce) y vamos a ejecutar la segunda transacción que actualiza el campo promo del customer 1144:

```

begin;
update customers set promo=15 where customerid=1144;
commit;

```

3. A continuación, abrimos también de manera rápida una tercera conexión, en nuestro caso pgAdmin3 y vamos a observar los bloqueos que se producen en el sistema con el comando “pg_locks”:

```
select * from pg_locks
```

Y observamos los dos últimos registros de la tabla:

	locktype text	database oid	relation oid	page integer	tuple smallint	virtualxid text	transactionid xid	classid oid	objid oid	objsubid smallint	virtualtransaction text	pid integer	mode text	granted boolean	fastpath boolean
1	relation	29343	11577								8/731	13919	AccessShareLock	t	t
2	virtualxid					8/731					8/731	13919	ExclusiveLock	t	t
3	relation	29343	29447								5/126	13857	RowExclusiveLock	t	t
4	relation	29343	29344								5/126	13857	RowExclusiveLock	t	t
5	virtualxid										5/126	13857	ExclusiveLock	t	t
6	relation	29343	29447								9/152	13960	AccessShareLock	t	t
7	relation	29343	29467								9/152	13960	AccessShareLock	t	t
8	relation	29343	29467								9/152	13960	RowExclusiveLock	t	t
9	relation	29343	29344								9/152	13960	AccessShareLock	t	t
10	relation	29343	29419								9/152	13960	AccessShareLock	t	t
11	relation	29343	29419								9/152	13960	RowShareLock	t	t
12	relation	29343	29419								9/152	13960	RowExclusiveLock	t	t
13	relation	29343	29413								9/152	13960	AccessShareLock	t	t
14	relation	29343	29413								9/152	13960	RowShareLock	t	t
15	relation	29343	29413								9/152	13960	RowExclusiveLock	t	t
16	virtualxid					9/152					9/152	13960	ExclusiveLock	t	t
17	transactionid						26512				9/152	13960	ExclusiveLock	t	f
18	transactionid							26513			5/126	13857	ExclusiveLock	t	f

Como podemos observar, hay dos operaciones de carácter “transactionid”, una con id 26512 y otra con id 26513 (la posterior), cuyos procesos establecidos por el sistema son el 13960 y 13857 respectivamente y el tipo de bloqueo es “XLOCK”, o también llamado bloqueo exclusivo.

4. A continuación, vamos otra vez adbeaver donde habíamos ejecutado la transacción 2 y observamos lo siguiente:

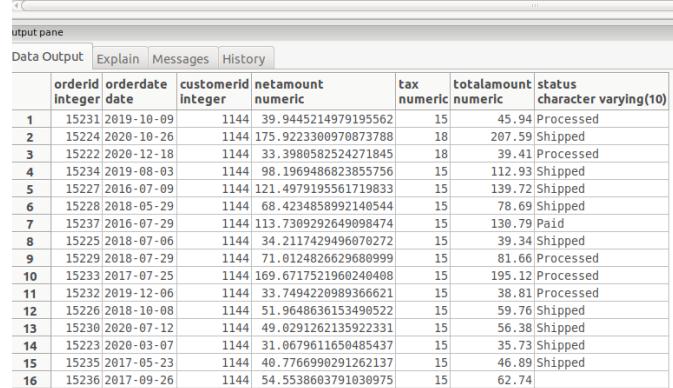
SQL Error [40P01]: ERROR: se ha detectado un deadlock.
 Detail: El proceso 13857 espera ShareLock en transacción 26512; bloqueado por proceso 13960.
 El proceso 13960 espera ShareLock en transacción 26513; bloqueado por proceso 13857.
 Hint: Vea el registro del servidor para obtener detalles de las consultas.
 Where: mientras se actualizaba la tupla (1086,4) en la relación `orders$`
 sentencia SQL: «update orders o set
 totalamount = (totalamount * ((100 - new.promo)/100))
 where new.customerid = o.customerid and o.status is null»
 función PL/pgSQL `updateprice()` en la línea 4 en sentencia SQL

Efectivamente, vemos que se ha producido un deadlock, ya que el pid 13857 espera SLOCK de la transacción 26512 (correspondiente al proceso con la transacción 1), y el proceso con pid 13860 espera un SLOCK de la transacción 26513, que corresponde precisamente a la transacción 2.

Por eso, podemos observar y llevar a la conclusión de que se ha producido un deadlock, ya que produce una situación de bloqueo recíproco entre las dos transacciones.

Ahora en una nueva ejecución para la misma situación, vamos a ejecutar desde esa tercera conexión (pgAdmin3), si los cambios son visibles:

```
select * from orders where customerid = 1144;
```



	orderid	orderdate	customerid	netamount	tax	totalamount	status
	integer	date	integer	numeric	numeric	numeric	character varying(10)
1	15231	2019-10-09	1144	39.9445214979195562	15	45.94	Processed
2	15224	2020-10-26	1144	175.922330970873788	18	207.59	Shipped
3	15222	2020-12-18	1144	33.3980582524271845	18	39.41	Processed
4	15234	2019-08-03	1144	98.1969486823855756	15	112.93	Shipped
5	15227	2016-07-09	1144	121.4979195561719833	15	139.72	Shipped
6	15228	2018-05-29	1144	68.4234858992140544	15	78.69	Shipped
7	15237	2016-07-29	1144	113.7309292649098474	15	130.79	Paid
8	15225	2018-07-06	1144	34.2117429496070272	15	39.34	Shipped
9	15229	2018-07-29	1144	71.0124826629680999	15	81.66	Processed
10	15233	2017-07-25	1144	169.6717521960240408	15	195.12	Processed
11	15232	2019-12-06	1144	33.7494220989366621	15	38.81	Processed
12	15226	2018-10-08	1144	51.9648636153490522	15	59.76	Shipped
13	15230	2020-07-12	1144	49.0291262135922331	15	56.38	Shipped
14	15223	2020-03-07	1144	31.0679611650485437	15	35.73	Shipped
15	15235	2017-05-23	1144	40.7766990291262137	15	46.89	Shipped
16	15236	2017-09-26	1144	54.5538603791030975	15	62.74	

Como podemos ver, en el momento que se están realizando las transacciones, estas no son visibles desde otra conexión, y esto se debe a la propiedad de **aislamiento** de las propiedades ACID de los procesos transaccionales.

Esta propiedad lo que dice es que durante el transcurso de una transacción, los cambios que produce dicha transacción no son visibles por otros hasta que esta finaliza (ya sea con commit, o rollback en caso de fallo).

Por ello, hasta que no acabe dicha transacción, si ejecutamos cualquier consulta sobre cualquiera de las tablas que pueden ser modificadas, no son visibles.