

SI P3 Memoria

Por Daniel Varela & Guillermo Martín-Coello

Parte 1: NoSQL

MongoDB e implementación de una base de datos no relacional

A: Base de datos MongoDB a partir de la base de datos postgresql.

Durante el primer apartado de la práctica se pedía pasar los datos de la base dada pgsql a una nueva base de datos no relacional creada en MongoDB. Para alcanzar dicho objetivo hemos utilizado python, accediendo a la base de datos postgresql a partir de sqlalchemy y creando y añadiendo los datos a la nueva base de datos con la librería Pymongo.

B: Script automático.

Creando éste script en python nos dimos cuenta de que íbamos a tener que ejecutarlo frecuentemente y tardaba demasiado, así que reestructuramos y optimizamos todo el código para que fuese mucho más rápido y eficiente. Para hacer esto, las películas relacionadas las obtenemos a través de python en vez de realizar queries para cada una de ellas reduciéndose así el coste.

C: Aplicación web.

Para implementar la base de datos en la aplicación web simplemente utilizamos el esqueleto dado y modificamos la función topUK para que llamase a nuevas funciones que diseñamos para el acceso a la base de datos MongoDB recientemente creada. Con estas funciones podemos obtener el resultado esperado de este apartado mediante comandos ofrecidos por la biblioteca de MongoDB.

D: Tablas web.

Diseñamos tablas en html para que la información proporcionada para los diferentes requisitos esperados se mostrase de manera limpia y ordenada, sólo mostrando los actores en el caso de que el usuario quiera para así obtener una mejor visualización del resto del contenido en caso contrario.

Parte 2: Optimización

Funcionamiento interno de las sentencias sql en postgresSQL

E: Estudio del impacto de un índice

a)

Creamos una consulta ciudadesDistintas.sql la cual muestra el número de ciudades distintas con clientes que tienen pedidos en un mes y un año dados y que a su vez tienen una tarjeta de crédito de tipo visa. La consulta queda del siguiente modo:

```
explain select count(distinct(c.city)) from customers c natural join orders o
      where '08' = date_part('month', o.orderdate)
      and '2016' = date_part('year', o.orderdate)
      and c.creditcardtype = 'VISA'
```

Imagen I a: Código de ejecución de consulta

Primero probamos a hacerla con to_char para poder escribir el mes y el año seguidos, pero para poder hacer uso de los índices como se requiere en los siguientes apartados pasamos a hacer uso de date_part.

b)

Al realizar un explain sobre la consulta obtenemos el siguiente resultado:

```
Aggregate (cost=5385.63..5385.64 rows=1 width=8)
-> Gather (cost=1000.28..5385.62 rows=1 width=118)
Workers Planned: 1
-> Nested Loop (cost=0.29..4385.52 rows=1 width=118)
  -> Parallel Seq Scan on orders o (cost=0.00..4360.38 rows=3 width=4)
    Filter: ('08':double precision = date_part('month':text, (orderdate):timestamp without time zone)) AND ('2016':double precision = date_part('year':text, (orderdate):timestamp without time zone)))
  -> Index Scan using customers_pkey on customers c (cost=0.29..8.30 rows=1 width=122)
    Index Cond: (customerid = o.customerid)
    Filter: ((creditcardtype):text = 'VISA':text)
```

Imagen I b: Plan y tiempo de Ejecución de consulta

Podemos apreciar que la consulta tiene un coste elevado (en torno a 5000).

Este resultado nos provee no solo el coste sino también el plan de ejecución de la consulta el cual está dividido en los siguientes pasos:

1. Hace un gather de todos los datos
2. Para hacer el gather, hace un nested loop

3. Hace un scan secuencial paralelo filtrando los meses y años que no sean los indicados
4. Hace un index scan con la primary key de customerid y filtra todos los creditcardtype que no sean VISA

c)

En nuestro intento por crear un índice que disminuye el coste de la query, creamos un índice en date_part('month', orders.OrderDate) para que así sea más rápido comprobar la condición de la query que comprueba date_part('month', orders.OrderDate)=01:

```
create index ind on orders(date_part('month', orders.orderdate));
```

Imagen II: Código de creación de índice en “month”

Ejecutamos ahora la query con EXPLAIN y obtenemos un coste muy inferior con respecto a la que no tenía indice (1549<5385):

	RBC QUERY PLAN
1	Aggregate (cost=1549.41..1549.42 rows=1 width=8)
2	-> Nested Loop (cost=19.52..1549.41 rows=1 width=118)
3	-> Bitmap Heap Scan on orders o (cost=19.24..1507.66 rows=5 width=4)
4	Recheck Cond: ('8':double precision = date_part('month':text, (orderdate)::timestamp without time zone))
5	Filter: ('2016':double precision = date_part('year':text, (orderdate)::timestamp without time zone))
6	-> Bitmap Index Scan on ind (cost=0.00..19.24 rows=909 width=0)
7	Index Cond: ('8':double precision = date_part('month':text, (orderdate)::timestamp without time zone))
8	-> Index Scan using customers_pkey on customers c (cost=0.29..8.30 rows=1 width=122)
9	Index Cond: (customerid = o.customerid)
10	Filter: ((creditcardtype):text = 'VISA':text)

Imagen III: Plan y tiempo de Ejecución de consulta

d)

Cuando analizamos el nuevo plan de ejecución para la consulta con índice podemos apreciar que la diferencia con respecto a la consulta sin índice radica en que para comprobar la condición del mes, en vez de hacer un sequential scan, hace un Index scan con el índice que hemos creado por lo tanto obtiene un coste inferior ya que el index scan es un escaneo mucho más rápido que el secuencial. También cabe destacar que el gather desaparece reduciendo enormemente el coste.

e)

En este apartado creamos varios índices y analizamos su efecto en el procedimiento y el coste de la consulta.

Primero creamos un índice en la columna creditcardtype de la tabla customers, de la siguiente forma:

```
create index ind on customers(creditcardtype);
```

Imagen IV: Código de creación de índice en “creditcardtype”

Esperando que este índice redujera el coste, ejecutamos EXPLAIN:

```
EXPLAIN QUERY PLAN
1 Aggregate (cost=5385.63..5385.64 rows=1 width=8)
2   > Gather (cost=1000.28..5385.62 rows=1 width=118)
3     Workers Planned: 1
4       > Nested Loop (cost=0.29..4385.52 rows=1 width=118)
5         >> Parallel Seq Scan on orders o (cost=0.00..4360.38 rows=3 width=4)
6           Filter: ((8::double precision = date_part('month'::text, (orderdate)::timestamp without time zone)) AND ('2016'::double precision = date_part('year'::text, (orderdate)::timestamp without time zone)))
7         -> Index Scan using customers_pkey on customers c (cost=0.29..8.30 rows=1 width=122)
8           Index Cond: (customerid = o.customerid)
9             Filter: ((creditcardtype)::text = 'VISA'::text)
```

Imagen V: Plan y tiempo de Ejecución de consulta

Al contrario de nuestra hipótesis inicial , el coste de la consulta se mantiene como estaba, y también lo hace el plan de ejecución, llegamos a la conclusión de que esto se debe a que ya hacía un index scan pero en vez de hacerlo con un índice externo, lo hacía con la primary key de customers (customerid), de esta forma la sentencia no se veía afectada por la creación de un índice externo ya que ya hacía un index scan previamente.

Seguimos con nuestro propósito de probar nuevos índices esta vez con uno similar al creado en el apartado b pero esta vez enfocado a indexar los años en vez de los meses:

```
create index ind on orders(date_part('year', orders.orderdate));
```

Imagen VI: Código de creación de índice en “year”

Con este índice esperábamos obtener un resultado de ejecución bastante similar al del apartado b ya que hacíamos lo mismo solo que en vez de indexar por meses indexamos por años. Con el EXECUTE obtenemos lo siguiente:

	ABC QUERY PLAN
1	Aggregate (cost=1549.41..1549.42 rows=1 width=8)
2	-> Nested Loop (cost=19.52..1549.41 rows=1 width=118)
3	-> Bitmap Heap Scan on orders o (cost=19.24..1507.66 rows=5 width=4)
4	Recheck Cond: ('2016'::double precision = date_part('year'::text, (orderdate)::timestamp without time zone))
5	Filter: ('8'::double precision = date_part('month'::text, (orderdate)::timestamp without time zone))
6	-> Bitmap Index Scan on ind (cost=0.00..19.24 rows=909 width=0)
7	Index Cond: ('2016'::double precision = date_part('year'::text, (orderdate)::timestamp without time zone))
8	-> Index Scan using customers_pkey on customers c (cost=0.29..8.30 rows=1 width=122)
9	Index Cond: (customerid = o.customerid)
10	Filter: ((creditcardtype)::text = 'VISA'::text)

Imagen VII: Plan y tiempo de Ejecución de consulta

Tal y como esperábamos el resultado es el mismo que indexando los meses esto es porque el plan de ejecución planteado es prácticamente el mismo cambiando solo los meses por los años.

Nuestra siguiente comprobación de índice pasaba por fusionar el índice de años y el de meses así que creamos un índice que junta ambas columnas (es decir, un índice compuesto con un atributo para month y un atributo para year) para comprobar el rendimiento de la ejecución en un contexto de búsqueda con índices paralelos:

```
create index ind on orders(date_part('month', orders.orderdate), date_part('year', orders.orderdate));
```

Imagen VIII: Código de creación de índice compuesto

Nuestra hipótesis era que se reduciría enormemente el tiempo ya que el plan de ejecución pasaría a hacer solo un index scan en vez de dos. Tras el explain obtenemos lo siguiente:

	ABC QUERY PLAN
1	Aggregate (cost=65.53..65.54 rows=1 width=8)
2	-> Nested Loop (cost=4.76..65.53 rows=1 width=118)
3	-> Bitmap Heap Scan on orders o (cost=4.47..23.78 rows=5 width=4)
4	Recheck Cond: (('8'::double precision = date_part('month'::text, (orderdate)::timestamp without time zone)) AND ('2016'::double precision = date_part('year'::text, (orderdate)::timestamp without time zone)))
5	-> Bitmap Index Scan on ind (cost=0.00..4.47 rows=5 width=0)
6	Index Cond: ('8'::double precision = date_part('month'::text, (orderdate)::timestamp without time zone)) AND ('2016'::double precision = date_part('year'::text, (orderdate)::timestamp without time zone)))
7	-> Index Scan using customers_pkey on customers c (cost=0.29..8.30 rows=1 width=122)
8	Index Cond: (customerid = o.customerid)
9	Filter: ((creditcardtype)::text = 'VISA'::text)

Imagen XI: Plan y tiempo de Ejecución de consulta

Se puede distinguir que, como era de esperar, el resultado tiene un coste mucho más reducido que todas las anteriores ejecuciones, esto se debe a que como se indica previamente, se ejecuta un index con una doble condición en vez de hacerlas por separado.

Sabiendo que con un indice de doble condición se reducía tanto el tiempo decidimos probar a crear dos índices por separado (primero month y luego year) para ver qué sucedía:

```
create index ind1 on orders(date_part('month', orders.orderdate));
create index ind2 on orders(date_part('year', orders.orderdate));
```

Imagen XII: Código de creación de múltiples índices

Nuestra hipótesis era que sería un poco más lento que el índice de doble condición ya que volvían a haber dos índices pero seria mas rapido que si solo hubiera uno. Ejecutamos el EXPLAIN y obtenemos lo siguiente:

The screenshot shows the pgAdmin Explain plan window. The query plan is as follows:

```
1 Aggregate (cost=99.79..99.80 rows=1 width=8)
  -> Nested Loop (cost=39.01..99.79 rows=1 width=118)
    -> Bitmap Heap Scan on orders o (cost=38.73..58.04 rows=5 width=4)
      Recheck Cond: ((2016::double precision = date_part('year'::text, (orderdate)::timestamp without time zone)) AND ('8'::double precision = date_part('month'::text, (orderdate)::timestamp without time zone)))
      -> BitmapAnd (cost=38.73..38.73 rows=5 width=0)
        -> Bitmap Index Scan on ind2 (cost=0.00..19.24 rows=909 width=0)
          Index Cond: (2016::double precision = date_part('year'::text, (orderdate)::timestamp without time zone))
        -> Bitmap Index Scan on ind1 (cost=0.00..19.24 rows=909 width=0)
          Index Cond: ('8'::double precision = date_part('month'::text, (orderdate)::timestamp without time zone))
    -> Index Scan using customers_pkey on customers c (cost=0.29..8.30 rows=1 width=122)
      Index Cond: (customerid = o.customerid)
      Filter: ((creditcardtype)::text = 'VISA'::text)
```

Imagen XIII: Plan y tiempo de Ejecución de consulta

Tal y como se esperaba, el coste se reduce con respecto a tener un índice pero aumenta con respecto a tener un índice con una doble condición, obteniendo así un coste de 99.

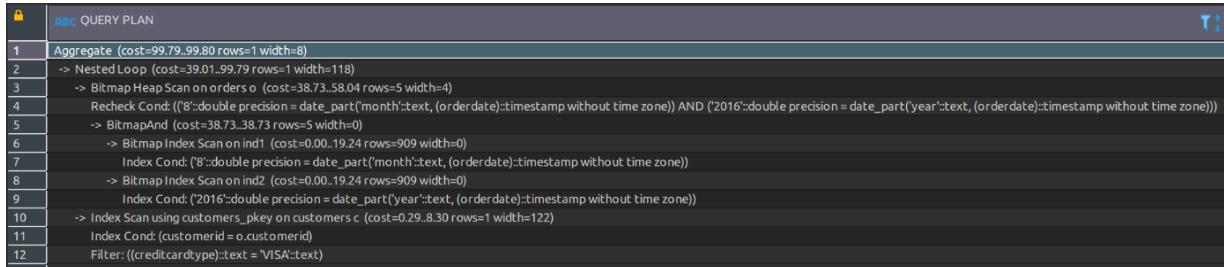
Finalmente, sabíamos que no era lo mismo crear un índice sobre A y luego sobre B que crear un índice sobre B y luego sobre A, que era más rápido indexar aquel índice que tuviera un resultado más pequeño ya que de esta forma, al indexar el más grande, se vería fuertemente reducido en cuanto a coste.

Decidimos entonces cambiar el orden de los últimos índices probados empezando así por indexar year y luego indexar month:

```
create index ind2 on orders(date_part('year', orders.orderdate));
create index ind1 on orders(date_part('month', orders.orderdate));
```

Imagen XIV: Código de creación de múltiples índices II

Nuestra hipótesis era que reduciría el coste ya que creíamos que había más películas en un mes dado que en un año concreto. Ejecutamos EXPLAIN y obtenemos lo siguiente:



```
RBC QUERY PLAN
Aggregate (cost=99.79..99.80 rows=1 width=8)
  -> Nested Loop (cost=39.01..99.79 rows=1 width=118)
    -> Bitmap Heap Scan on orders o (cost=38.73..58.04 rows=5 width=4)
      Recheck Cond: ('8':double precision = date_part('month':text, (orderdate)::timestamp without time zone)) AND ('2016':double precision = date_part('year':text, (orderdate)::timestamp without time zone))
      -> BitmapAnd (cost=38.73..38.73 rows=5 width=0)
        -> Bitmap Index Scan on ind1 (cost=0.00..19.24 rows=909 width=0)
          Index Cond: ('8':double precision = date_part('month':text, (orderdate)::timestamp without time zone))
        -> Bitmap Index Scan on ind2 (cost=0.00..19.24 rows=909 width=0)
          Index Cond: ('2016':double precision = date_part('year':text, (orderdate)::timestamp without time zone))
    -> Index Scan using customers_pkey on customers c (cost=0.29..8.30 rows=1 width=122)
      Index Cond: (customerid = o.customerid)
      Filter: ((creditcardtype):text = 'VISA':text)
```

Imagen XV: Plan y tiempo de Ejecución de consulta

Al contrario de lo que esperábamos, no hay ningún cambio notable en el resultado. Esto puede deberse a que no se han analizado las estadísticas de la base de datos, por lo que la estimación del coste dada por el comando explain no detalla las diferencias que podría haber entre la cantidad de películas con el mismo mes y la cantidad de películas con el mismo año. Si se analiza previamente las estadísticas de la tabla el plan de ejecución sería, probablemente, diferente con una estimación más rápida y eficiente.

F: Estudio de impacto de la forma de la consulta

Consulta I

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

Seq Scan on customers (cost=3961.65..4490.81 rows=7046 width=4)

Filter: (NOT (hashed SubPlan 1))

SubPlan 1

-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)

Filter: ((status)::text = 'Paid'::text)

Imágenes XVI y XVII

Consulta II

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

HashAggregate (cost=4537.41..4539.41 rows=200 width=4)

Group Key: customers.customerid

Filter: (count(*) = 1)

-> Append (cost=0.00..4462.40 rows=15002 width=4)

-> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=

-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)

Filter: ((status)::text = 'Paid'::text)

Imágenes XVIII y XIX

Consulta III

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

```
HashSetOp Except (cost=0.00..4640.83 rows=14093 width=8)
-> Append (cost=0.00..4603.32 rows=15002 width=8)
  -> Subquery Scan on "*SELECT* 1" (cost=0.00..634.86 rows=14093 width=8)
    -> Seq Scan on customers (cost=0.00..493.93 rows=14093 width=4)
  -> Subquery Scan on "*SELECT* 2" (cost=0.00..3968.47 rows=909 width=8)
    -> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=4)
      Filter: ((status)::text = 'Paid'::text)
```

Imágenes XX y XXI

a)

Es la tercera query la que empieza a devolver resultados nada más ser ejecutada ya que el coste va desde 0. Esto podría deberse a que al usarse un except, a cada comprobación de si un resultado está duplicado en ambas subqueries, al hacerlo secuencialmente es capaz de determinar si tiene que añadir el registro al resultado o no uno a uno así que devuelve resultados justo después de ser ejecutada.

b)

Las consultas que se podrían beneficiar de una ejecución en paralelo sería la primera ya que tiene un subplan el cual podría ser ejecutado por un procesador mientras el select sería ejecutado por el otro y después solo habría que juntarlos por la sentencia not in, y también la tercera ya que tiene dos subqueries diferentes las que se podrían hacer también al mismo tiempo

G: Estudio de impacto de generación de estadísticas.

Consulta 1:

```
-- Consulta 1
explain select count(*)
    from orders where status is null;
```

Imágenes XXII: Consulta 1

Como se puede observar esta consulta debe desplazarse por toda la tabla por lo que el coste de ejecución es elevado.

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

Imágenes XXIII: Plan consulta 1

Consulta 2:

```
-- Consulta 2
explain select count(*) from orders where status = 'Shipped' ;
```

Imágenes XXIV: Consulta 2

En el caso de la segunda consulta es muy similar a la primera (sigue escaneando toda la tabla) con la diferencia de que no solo tiene que hacer la comprobación de si “STATUS” existe sino que debe comprobar si es igual al proporcionado, resultando en un tiempo de ejecución aún más lento.

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

Imagenes XXV: Plan consulta 2

A continuación repetiremos las consultas anteriores con un índice:

```
-- Creación índice --
-----
drop index if exists con1_index;
create index con1_index on orders(status)
```

Imagen XXVI: Creación índice

Consulta 1 con índice:

```
Aggregate (cost=1496.52..1496.53 rows=1 width=8)
  -> Bitmap Heap Scan on orders (cost=19.46..1494.25 rows=909 width=0)
      Recheck Cond: (status IS NULL)
      -> Bitmap Index Scan on con1_index (cost=0.00..19.24 rows=909 width=0)
          Index Cond: (status IS NULL)
```

Imagen XXVII: Plan de ejecución de consulta I con índice

Consulta 2 con índice

```
Aggregate (cost=1498.79..1498.80 rows=1 width=8)
  -> Bitmap Heap Scan on orders (cost=19.46..1496.52 rows=909 width=0)
      Recheck Cond: ((status)::text = 'Shipped'::text)
      -> Bitmap Index Scan on con1_index (cost=0.00..19.24 rows=909 width=0)
          Index Cond: ((status)::text = 'Shipped'::text)
```

Imagen XXVIII: Plan de ejecución de consulta II con índice

Como podemos observar el índice reduce en ambas consultas el tiempo de ejecución. En ambos casos ahora ya no se hace una búsqueda secuencial sino que se hace la búsqueda en un bitmap. El plan de ejecución es similar en ambas sentencias, produciendo unos resultados muy parecidos entre sí. Tener un índice permite que la búsqueda descarte rápidamente gran parte de las filas que de otra manera hubiesen sido comprobadas.

Tras esto utilizamos el comando analyze en la tabla orders:

```
-- Analyze --
-----
analyze verbose orders
```

Imagen XXIX: Comando analyze

Que nos produce el siguiente resultado:

```
"orders": scanned 1687 of 1687 pages, containing 181790 live rows and 0 dead rows;
30000 rows in sample, 181790 estimated total rows
```

Ahora ejecutamos de nuevo las consultas anteriores para apreciar las mejoras al plan de consulta que ha conseguido el comando analyze:

Consulta 1:

```
Aggregate (cost=7.27..7.28 rows=1 width=8)
  -> Index Only Scan using con1_index on orders (cost=0.42..7.26 rows=1 width=0)
      Index Cond: (status IS NULL)
```

Imagen XXX: Plan de ejecución de consulta I con índice y analyze

Consulta 2:

```
Finalize Aggregate (cost=4211.17..4211.18 rows=1 width=8)
  -> Gather (cost=4211.06..4211.17 rows=1 width=8)
      Workers Planned: 1
        -> Partial Aggregate (cost=3211.06..3211.07 rows=1 width=8)
          -> Parallel Seq Scan on orders (cost=0.00..3023.69 rows=74948 width=0)
              Filter: ((status)::text = 'Shipped'::text)
```

Imagen XXXI: Plan de ejecución de consulta II con índice y analyze

Como podemos observar la sentencia 1 ha mejorado su tiempo de ejecución notablemente. Esto se debe a que gracias al comando analyze ha podido mejorar su plan de ejecución utilizando el índice ya que no existen registros vacíos, por lo que la búsqueda es prácticamente instantánea. Sin embargo podemos observar que la segunda sentencia ahora muestra un resultado más lento que antes de la optimización del comando analyze. Esto podría ser ya que tras el comando analyze se utilizan criterios de cálculo más realistas respecto a la cantidad y orden de entradas que se pretende encontrar, y el resultado es superior a la predicción anterior, sin tener en cuenta dichos datos.

A continuación ejecutamos otras dos consultas para comprobar sus planes de ejecución:

```
-- Consulta 3
select count(*) from orders where status ='Paid';
```

```
Aggregate (cost=2322.08..2322.09 rows=1 width=8)
-> Bitmap Heap Scan on orders (cost=361.68..2276.52 rows=18227 width=0)
  Recheck Cond: ((status)::text = 'Paid'::text)
-> Bitmap Index Scan on con1_index (cost=0.00..357.12 rows=18227 width=0)
  Index Cond: ((status)::text = 'Paid'::text)
```

Imágenes XXXII y XXXIII: Código y plan de ejecución de consulta III con índice y analyze

```
-- Consulta 4
select count(*) from orders
where status ='Processed';
```

```
Aggregate (cost=2941.88..2941.89 rows=1 width=8)
-> Bitmap Heap Scan on orders (cost=712.60..2851.50 rows=36152 width=0)
  Recheck Cond: ((status)::text = 'Processed'::text)
-> Bitmap Index Scan on con1_index (cost=0.00..703.56 rows=36152 width=0)
  Index Cond: ((status)::text = 'Processed'::text)
```

Imágenes XXXIV y XXXV: Código y plan de ejecución de consulta IV con índice y analyze

Al ejecutar estas dos nuevas consultas podemos comprobar que sus planes de ejecución son similares a los de las consultas previas antes de ejecutar el comando analyze. Esto se debe a que al ejecutar el comando analyze se generan estadísticas que ayudan a que se creen planes de ejecución más eficientes. En el caso de estas las estadísticas obtenidas han debido sugerir que el plan más eficiente era éste.

Parte 3: Transacciones

Atomicidad, Consistencia, Aislamiento y Durabilidad

Explicación COMMIT:

La instrucción commit es la herramienta cuya función es validar la transacción efectuada. Ya que la transacción es atómica, sólo puede o producirse completamente o no producirse en absoluto. Al realizar el commit se liberan los bloqueos producidos por la transacción y se aplican los cambios a la base de datos. Por ello, es importante que la instrucción commit sólo se realice al final de la transacción, ya que si no podría dejar a ésta en un estado intermedio violando así su propiedad de consistencia.

H. Borrar Ciudad :

Hemos diseñado la función para ejecutar la transacción objetivo del ejercicio. En ella definimos las diferentes opciones que se pueden seleccionar mediante parámetros para modificar el funcionamiento de la misma y más adelante la ejecutamos dentro de una sentencia de tipo “try-except-finally” para controlar los errores fácilmente y terminar la ejecución de manera limpia. Además, todo movimiento dentro de la transacción está documentado mediante trazas para obtener una respuesta visual en la página web esqueleto. Los parámetros para definir el comportamiento de la transacción son dados a través del formulario rellenable en la página web. En caso de error la transacción debe ejecutar un rollback para volver hacia atrás (en caso de que hubiese savepoints se podría volver a ellos pero en esta transacción, al ser pequeña, no son necesarios). Toda la función consta de control de errores para que todo funcione correctamente y no se pierda la propiedad de atomicidad de la función. Se incluye un parámetro para hacer commit en medio de la transacción, lo que violaría las propiedades ACID de la transacción, ya que violaría la atomicidad y consistencia de la función al guardar parte de su progreso pudiendo fallar la otra mitad. Tras este commit le sigue un begin ya que se considera una nueva transacción (pues la anterior ha acabado en el commit).

Trazas

1. Seleccionada ejecución correcta de las consultas
2. Ejecutando : DELETE FROM orderdetail WHERE orderid IN (SELECT orderid FROM orders WHERE customerid IN (SELECT customerid FROM customers WHERE city = 'fuck'))
3. Ejecutando : DELETE FROM orders WHERE customerid IN (SELECT customerid FROM customers WHERE city = 'fuck')
4. Ejecutando : DELETE FROM customers WHERE city = 'fuck'
5. Ejecutando : DUERME 20 segundos
6. Ejecutando : COMMIT
7. Se ha ejecutado correctamente las consultas

Imagen de trazas de una transacción satisfactoria

Trazas

1. Seleccionado fallo en la ejecución de las consultas
2. Ejecutando : DELETE FROM orderdetail WHERE orderid IN (SELECT orderid FROM orders WHERE customerid IN (SELECT customerid FROM customers WHERE city = 'sex'))
3. Ejecutando : DELETE FROM customers WHERE city = 'sex'
4. Se ha producido un error en la ejecución de las consultas
5. Error: (psycopg2.errors.ForeignKeyViolation) update or delete on table "customers" violates foreign key constraint "orders_customerid_fkey" on table "orders" DETAIL: Key (customerid)=(3448) is still referenced from table "orders". [SQL: DELETE FROM customers WHERE city = 'sex'] (Background on this error at: <https://sqlalche.me/e/14/gkpj>)
6. Ejecutando : ROLLBACK
7. Deshaciendo transacción

Imagen de trazas de una transacción no satisfactoria vía sentencias

Trazas

1. Seleccionado fallo en la ejecución de las consultas
2. Ejecutando : DELETE FROM orderdetail WHERE orderid IN (SELECT orderid FROM orders WHERE customerid IN (SELECT customerid FROM customers WHERE city = 'horny'))
3. Ejecutando : DELETE FROM customers WHERE city = 'horny'
4. Se ha producido un error en la ejecución de las consultas
5. Error: (psycopg2.errors.ForeignKeyViolation) update or delete on table "customers" violates foreign key constraint "orders_customerid_fkey" on table "orders" DETAIL: Key (customerid)=(3743) is still referenced from table "orders". [SQL: DELETE FROM customers WHERE city = 'horny'] (Background on this error at: <https://sqlalche.me/e/14/gkpj>)
6. Ejecutando : ROLLBACK
7. Deshaciendo transacción

Imagen de trazas de una transacción no satisfactoria vía funciones de sqlalchemy

Trazas

1. Seleccionado fallo en la ejecución de las consultas
2. Ejecutando : DELETE FROM orderdetail WHERE orderid IN (SELECT orderid FROM orders WHERE customerid IN (SELECT customerid FROM customers WHERE city = 'ass'))
3. Ejecutando : DELETE FROM customers WHERE city = 'ass'
4. Ejecutando : COMMIT
5. Ejecutando : BEGIN
6. Se ha producido un error en la ejecución de las consultas
7. Error: (psycopg2.errors.ForeignKeyViolation) update or delete on table "customers" violates foreign key constraint "orders_customerid_fkey" on table "orders" DETAIL: Key (customerid)=(8475) is still referenced from table "orders". [SQL: DELETE FROM customers WHERE city = 'ass'] (Background on this error at: <https://sqlalche.me/e/14/gkpj>)
8. Ejecutando : ROLLBACK
9. Deshaciendo transacción

Imagen de trazas de una transacción no satisfactoria con commit intermedio

I. Bloqueos y deadlocks:

Con tal de probar los bloqueos hemos creado un script llamado updPromo.sql el cual creaba un trigger para la tabla customers que llama a una función llamada updPromo(). Esta función actualiza el totalprice de las orders en estado NULL del customer aplicando un cupón de descuento de porcentaje promo (promo es una nueva columna de la tabla customer que hemos creado donde se guarda un valor decimal del 0 al 100). Posteriormente hemos modificado las orders de un customer específico poniendo dos de ellas a NULL para poder añadirles un descuento.

Se nos pedía que usáramos “**PERFORM PG_SLEEP(10);**” para poner un tiempo de espera en la función y que también le pusieramos un tiempo de espera en la transacción de database.py. Seguidamente hemos ido modificando el sitio donde poner la espera hasta conseguir un deadlock.

Finalmente hemos puesto el sleep de la función de SQL antes de hacer el update y el de la transacción de database.py después de eliminar la tabla orders y antes de eliminar customers. Lo hemos hecho de esta forma ya que mientras que la primera solicita el recurso de orders para poder hacer el update y la segunda solicita el recurso de customers y no se pueden liberar ni terminar hasta que la otra les haya cedido ese registro formando así un deadlock. Cuando ejecutamos como se nos propone en el apartado g de esta pregunta podemos observar que la transacción hace un rollback. En cuanto a la pregunta del apartado h, podemos concluir que los resultados no son visibles debido a las propiedades ACID de las transacciones,

en concreto a la propiedad de atomicidad la cual especifica que una transacción tiene que realizarse al completo o no realizarse siendo esta una unidad indivisible.

Para evitar los deadlocks es recomendable la brevedad de las transacciones, es decir reducirlas a lo mínimo posible permitiendo una más fácil atomicidad. Otro enfoque sería estudiar los posibles bloqueos y llegar a una forma de solventarlos todos mediante esperas o un orden diferente de las transacciones. Y por último, la forma más efectiva de afrontar los deadlocks sería hacer uso de savepoints. Estos no evitan la aparición de deadlocks sino que más bien reaccionan a ellos haciendo que cuando se ejecute un rollback a causa de un deadlock no se pierda toda la transacción sino que se retome desde el último savepoint (algo así como el checkpoint a mitad de partida en los juegos de Mario Bros).