PRÁCTICA 3
MEMORIA

NoSQL, Optimización y Transacciones

Por Daniel Cerrato y David Garitagoitia

Escuela Politécnica Superior

Universidad Autónoma de Madrid

[1] Si se hace algún COMMIT intermedio, es necesario usar de nuevo la sentencia BEGIN, pues COMMIT acaba la transacción como tal, sin embargo, aún no habremos terminado con el propósito de la transacción general.

[1] Si se hace algún COMMIT intermedio, es necesario usar de nuevo la sentencia BEGIN, pues COMMIT acaba la transacción como tal, sin embargo, aún no habremos terminado con el propósito de la transacción general.

1. Descripción del material entregado

Al inicio de la práctica, se nos facilita un contenedor que hay que descomprimir y ejecutar para poblar la base de datos con la que hemos trabajado durante la práctica.

También se nos facilitan los archivos necesarios para ejecutar una página web. Esta inicia en una página "index" que contiene dos enlaces: uno para ver la página que contiene las tablas del primer apartado y otro para acceder a la página de borrado de ciudades en la base de datos para el apartado 3.

2. Ejercicios y discusiones

A-B. Para esta primera parte hemos generado la base de datos en MongoDB con el archivo "createMongoDBFromPostgreSQLDB.py", el cual esta subdividido en tres partes:

1. Generar listas con los datos de las películas inglesas más actuales.

2. Generar una lista con estos datos organizados en diccionarios preparados para crear la base de datos en MongoDB.

3. Generar la base de datos en MongoDB con la lista de películas obtenida.

C-D. Para esta última parte del apartado 1, en el archivo "routes.py" de la aplicación, se ha codificado la función "topUK", generando listas de películas a través de consultas a la base de datos en MongoDB. Estas serán expuestas en la página web, en el primer enlace denominado con el mismo nombre que la función ("topUK").

E. Para este ejercicio, hemos creado un script de SQL que realiza una consulta con la que se obtiene el número de ciudades distintas con clientes que tienen pedidos en un mes escogido y que tienen una tarjeta VISA.

Para esto generamos primero una función que da formato a la fecha de los pedidos. Necesitamos crearla de forma **inmutable** para poder usarla como índice más tarde.

Acto seguido generamos una lista de sentencias EXPLAIN para comparar los resultados de usar distintos índices.

Empezamos con explain base que indica que tardará alrededor de 50393.92 mseg

Acto seguido, generamos un índice en "customerid" pero no tiene efecto en los tiempos de planificación, esto se debe a que "customerid" es una clave primaria, las cuales ya se usan como índices en las búsquedas.

Después, reemplazamos el índice por otro sobre "country" y por otro sobre "orderdate", sin mejora. Esto se debe a que la consulta no busca sobre el campo "country" ni "orderdate".

[1] Si se hace algún COMMIT intermedio, es necesario usar de nuevo la sentencia BEGIN, pues COMMIT acaba la transacción como tal, sin embargo, aún no habremos terminado con el propósito de la transacción general.

Luego, probamos con el índice sobre "creditcardtype", lo cual mejoró mínimamente el tiempo de la query (50044.71 mseg). Esto se debe a que no es un campo que sea clave primaria y, en este caso, sí que usamos para realizar la búsqueda, pero no usa muchas operaciones.

Seguidamente, utilizamos el índice sobre la función creada anteriormente. Esto hace que los tiempos de la query reduzcan en gran cantidad (2258.61 mseg). Esto es porque es la parte de la query que más operaciones realiza y, por tanto, más tiempo consume.

Finalmente, creamos el índice binario sobre "customerid" y la función en conjunto. Esto hizo que los tiempos de la query se redujeran un poco más (2109.21 mseg). La mejora viene dada porque "customerid" es uno de los campos más buscados y que más operaciones gasta junto a la función.

F. Para este ejercicio vamos a comparar los formatos de unas querys muy simples que nos proporcionan y responder a un par de preguntas sobre los resultados.

QUERY 1

```
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)
```

Como se puede observar, esta query debería devolver los resultados en un tiempo de 3961.65 a 4490.81 mseg. Inicialmente, busca en la tabla "orders" las filas con la columna "status" con el valor "Paid" de forma secuencial, es decir, en toda la tabla. Finalmente, devuelve los ids de los clientes de las tuplas obtenidas.

QUERY 2

```
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=4)
        ->  Seq Scan on orders  (cost=0.00..3959.38 rows=909 width=4)
            Filter: ((status)::text = 'Paid'::text)
```

[1] Si se hace algún COMMIT intermedio, es necesario usar de nuevo la sentencia BEGIN, pues COMMIT acaba la transacción como tal, sin embargo, aún no habremos terminado con el propósito de la transacción general.

En esta segunda query, se devolverán los datos en un tiempo de 4537.41 a 4539.41 mseg. Primero, realiza las dos subquerys, acto seguido, las une y, finalmente, devuelve los ids de los clientes eliminando las repeticiones.

QUERY 3

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)

Esta última query devolverá los resultados en un tiempo de 0.00 a 4640.83 mseg. Realiza la búsqueda pedidos con “status” con valor “Paid” y la búsqueda de todos los clientes a la vez, uniéndolos y añadiéndolos en la lista que realiza el “except”, para ir mostrando según vaya obteniendo los ids de los clientes que cumplan los requisitos.

Según creemos, la segunda query es la que puede beneficiarse si se ejecuta en paralelo, pues puede ejecutar las subconsultas de la unión en paralelo. La tercera query puede llegar a beneficiarse también, pues puede hacer las subquerys del “except” por separado.

G. En este último ejercicio del segundo apartado, vamos a comparar los resultados de distintas querys cuando añadimos un índice o no, y cuando generamos un análisis de la query antes de generar sus planificaciones.

Primero realizamos los “explain” de ambas consultas

QUERY 1

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)

QUERY 2

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)

[1] Si se hace algún COMMIT intermedio, es necesario usar de nuevo la sentencia BEGIN, pues COMMIT acaba la transacción como tal, sin embargo, aún no habremos terminado con el propósito de la transacción general.

Como se puede observar, la primera query es más rápida que la segunda. La primera devuelve los datos en un tiempo aproximado de 3507 mseg, mientras que la segunda tardará unos 3961 mseg.

Ahora creamos el índice sobre la columna “status” en “orders” y realizamos de nuevo los “explain” de ambas consultas.

QUERY 1

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 countstatus (cost=0.00..19.24 rows=909 width=0)

Index Cond: (status IS NULL)

QUERY 2

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 countstatus (cost=0.00..19.24 rows=909 width=0)

Index Cond: ((status)::text = 'Shipped'::text)

Inicialmente las querys buscan de forma secuencial. Al crear el índice, la búsqueda se realiza a través de un bitmap, lo que hace que las búsquedas sean mucho más eficientes.

Eliminamos el índice y ejecutamos el análisis.

QUERY 1

Aggregate (cost=3504.90..3504.91 rows=1 width=8)

-> Seq Scan on orders (cost=0.00..3504.90 rows=1 width=0)

Filter: (status IS NULL)

QUERY 2

Finalize Aggregate (cost=4210.87..4210.88 rows=1 width=8)

-> Gather (cost=4210.76..4210.87 rows=1 width=8)

Workers Planned: 1

-> Partial Aggregate (cost=3210.76..3210.77 rows=1 width=8)

-> Parallel Seq Scan on orders (cost=0.00..3023.69 rows=74826 width=0)

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

[1] Si se hace algún COMMIT intermedio, es necesario usar de nuevo la sentencia BEGIN, pues COMMIT acaba la transacción como tal, sin embargo, aún no habremos terminado con el propósito de la transacción general.

Tras ejecutar la sentencia analyze, sin el índice en juego, el tiempo de ejecución de la primera query no ha variado prácticamente. Sin embargo, la segunda query ha empeorado en rendimiento, pues el análisis ha decidido que es mejor llevar a cabo la query de forma paralela, pero con un solo hilo (un poco absurdo). La gestión de este hilo es lo que ha hecho que la query sea más lenta.

Volvemos a crear el índice sobre la columna "status" de la tabla "orders" y ejecutamos el análisis.

QUERY 1

Aggregate (cost=7.25..7.26 rows=1 width=8)

-> Index Only Scan using countstatus on orders (cost=0.42..7.25 rows=1 width=0)

Index Cond: (status IS NULL)

QUERY 2

Finalize Aggregate (cost=4210.87..4210.88 rows=1 width=8)

-> Gather (cost=4210.76..4210.87 rows=1 width=8)

Workers Planned: 1

-> Partial Aggregate (cost=3210.76..3210.77 rows=1 width=8)

-> Parallel Seq Scan on orders (cost=0.00..3023.69 rows=74826 width=0)

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

Tras recuperar el índice, vemos que esta vez, la primera query sí que tiene una mejora notable para el escaso tiempo de ejecución que ya tenía. Sin embargo, la segunda query sigue sin mejorar, continua exactamente igual que sin el índice y tras el analyze.

Eliminamos de nuevo el índice para comparar con dos querys nuevas, realizando todo el proceso hecho hasta el momento.

QUERY 3

Aggregate (cost=4004.14..4004.15 rows=1 width=8)

-> Seq Scan on orders (cost=0.00..3959.38 rows=17906 width=0)

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

[1] Si se hace algún COMMIT intermedio, es necesario usar de nuevo la sentencia BEGIN, pues COMMIT acaba la transacción como tal, sin embargo, aún no habremos terminado con el propósito de la transacción general.

QUERY 4

Aggregate (cost=4049.95..4049.96 rows=1 width=8)

-> Seq Scan on orders (cost=0.00..3959.38 rows=36231 width=0)

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

Sin índice ni análisis, se puede apreciar que las querys tardan "bastante".

Creamos de nuevo el índice y comparamos

QUERY 3

Aggregate (cost=2310.78..2310.79 rows=1 width=8)

-> Bitmap Heap Scan on orders (cost=355.19..2266.02 rows=17906 width=0)

Recheck Cond: ((status)::text = 'Paid'::text)

-> Bitmap Index Scan on countstatus (cost=0.00..350.71 rows=17906 width=0)

Index Cond: ((status)::text = 'Paid'::text)

QUERY 4

Aggregate (cost=2947.68..2947.69 rows=1 width=8)

-> Bitmap Heap Scan on orders (cost=717.21..2857.10 rows=36231 width=0)

Recheck Cond: ((status)::text = 'Processed'::text)

-> Bitmap Index Scan on countstatus (cost=0.00..708.15 rows=36231 width=0)

Index Cond: ((status)::text = 'Processed'::text)

Tras la creación de nuevo del índice vemos que los tiempos evidentemente mejoran. Son un poco más lentas que las querys 1 y 2, pues van a devolver más filas que estas.

Eliminamos por última vez el índice y ejecutamos el análisis.

QUERY 3

Aggregate (cost=4005.35..4005.36 rows=1 width=8)

-> Seq Scan on orders (cost=0.00..3959.38 rows=18391 width=0)

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

[1] Si se hace algún COMMIT intermedio, es necesario usar de nuevo la sentencia BEGIN, pues COMMIT acaba la transacción como tal, sin embargo, aún no habremos terminado con el propósito de la transacción general.

QUERY 4

```
Aggregate  (cost=4051.95..4051.96 rows=1 width=8)
    ->  Seq Scan on orders  (cost=0.00..3959.38 rows=37031 width=0)
          Filter: ((status)::text = 'Processed'::text)
```

La base de datos ha decidido tratar a estas dos querys como lo hizo con la primera, a pesar de que estas dos últimas son iguales que la QUERY 2. De esta forma, los tiempos de ejecución son exactamente iguales que la ejecución sin análisis y sin índice.

Por último, volvemos a crear el índice y volvemos a analizar.

QUERY3

```
Aggregate  (cost=2325.82..2325.83 rows=1 width=8)
    ->  Bitmap Heap Scan on orders  (cost=362.95..2279.84 rows=18391 width=0)
          Recheck Cond: ((status)::text = 'Paid'::text)
          ->  Bitmap Index Scan on countstatus  (cost=0.00..358.35 rows=18391 width=0)
                Index Cond: ((status)::text = 'Paid'::text)
```

QUERY 4

```
Aggregate  (cost=2973.88..2973.89 rows=1 width=8)
    ->  Bitmap Heap Scan on orders  (cost=731.41..2881.30 rows=37031 width=0)
          Recheck Cond: ((status)::text = 'Processed'::text)
          ->  Bitmap Index Scan on countstatus  (cost=0.00..722.15 rows=37031 width=0)
                Index Cond: ((status)::text = 'Processed'::text)
```

Los tiempos entre el explain con índice con análisis y sin él, no varían mucho. Se puede apreciar una ligera ralentización en los tiempos con el análisis. Sin embargo, a pesar de tratar las querys como lo hizo con la QUERY 1, estas no han mejorado sus tiempos como lo hizo esta.

H. Para este último apartado, en este ejercicio vamos a utilizar la página “borraCiudad” y su función en el archivo “routes.py” para llevar a cabo transacciones que borren paso a paso los registros necesarios al querer eliminar todos los clientes de una cierta ciudad.

Las restricciones "ON DELETE CASCADE" están desactivadas por defecto. Esto lo comprobamos cuando intentamos eliminar un cliente, pero no se permite la ejecución, avisándonos de que el usuario tiene una relación de foreign key desde orders.

Para esto, en el archivo “database.py” vamos a codificar la función “delCity”. Aquí vamos a usar la sentencia “try” para capturar posibles excepciones de la transacción y ejecutar

[1] Si se hace algún COMMIT intermedio, es necesario usar de nuevo la sentencia BEGIN, pues COMMIT acaba la transacción como tal, sin embargo, aún no habremos terminado con el propósito de la transacción general.

"ROLLBACK" en caso de que sea necesario, o en su defecto, realizar un "COMMIT". En ambos casos, cerramos la conexión con la base de datos.

El cuerpo de la función pasa por la conexión con la base de datos y una comprobación del método para realizar la transacción. Nosotros solo hemos codificado la parte de SQL sin SQLAlchemy.

Nada más empezar ejecutamos un "BEGIN" para comenzar la transacción. Y acto seguido comprobamos si nos han pedido que hagamos la transacción correctamente o que generemos un fallo para hacer "ROLLBACK". Los códigos de ambas partes los hemos abstraído a funciones separadas, pero el código es exactamente el mismo, solo que cambiamos de orden algunas querys para generar el error en la función correspondiente.

En ambos empezamos obteniendo el id de todos los clientes de la ciudad especificada. Luego recogemos los ids de los pedidos que hayan realizado. Los metemos en dos listas.

En todos los intentos de borrado comprobamos que las listas no estén vacías. Esto lo hacemos para que no haya excepciones por NOT FOUND.

En ambas transacciones borramos los registros con los productos escogidos en cada pedido de la lista.

Seguidamente, si nos lo piden ejecutaremos un COMMIT y un BEGIN[1] intermedios y/o un SLEEP con el tiempo que nos indiquen.

A continuación, vienen los cambios. En la transacción correcta, borraremos primero los pedidos de la lista, seguido de un COMMIT intermedio en caso de ser necesario y, finalmente, borramos los clientes de la lista. En la transacción incorrecta, intentamos borrar primero los clientes y luego los pedidos de las listas. Esto generará un error en la base de datos puesto que está desactivada la función "ON DELETE CASCADE", pero no se han eliminado las referencias entre las tablas. Como la tabla "orders" tiene una referencia de foreign key sobre la tabla "customers", impide que se borren registros de la tabla "customers" sin haber borrado primero los registros de la tabla "orders" donde se referencien a los clientes que se quieren eliminar.

I. En este último ejercicio hemos creado un script SQL llamado "updPromo.sql", donde realizamos las siguientes tareas:

Creamos una columna "promo" en la tabla "customers".

Creamos un trigger y su función para aplicar el descuento de la columna "promo" a los productos del carrito del cliente que actualice esta columna.

Este trigger lleva una modificación extra que es la adición de un SLEEP arbitrario en principio entre las actualizaciones de las tablas "orderdetail" y "orders", que se quiere usar para generar más tarde un "deadlock" con la página web.

Finalmente, hemos creado cuatro carritos en pedidos ya existentes.

Una vez creado el script, lo ejecutamos y comenzamos las pruebas y discusiones.

[1] Si se hace algún COMMIT intermedio, es necesario usar de nuevo la sentencia BEGIN, pues COMMIT acaba la transacción como tal, sin embargo, aún no habremos terminado con el propósito de la transacción general.

Si comprobamos los cambios esperados tras la actualización de la columna promo de algún cliente, durante el proceso del trigger, desde otra terminal no se ven los cambios que esté haciendo e incluso poniendo el sleep no podemos ver los cambios que hace en la tabla orderdetail, y por supuesto, tampoco los de la tabla orders.

Esto sucede porque el proceso que está actualizando los registros de las tablas, las bloquea y hasta que no llegue al END, no confirma los datos ni los hace visibles.

Pasa lo mismo cuando intentamos ver los datos que están siendo eliminados al eliminar una ciudad de la base de datos. Hasta que no se haga el COMMIT, no se podrán ver los cambios.

Si ajustamos los tiempos de los SLEEP, podemos comprobar que se llega a un "deadlock" con los bloqueos de las tablas "orderdetail" y "orders". Esto sucede porque la transacción de la página web bloquea una de las tablas y se duerme, el trigger mientras tanto bloquea la otra tabla y se duerme. Cuando ambos se despierten, continuarán con sus cometidos y lo siguiente que les toca hacer es bloquear la tabla que tiene bloqueado el otro proceso. De manera que se esperan uno a otro sin fin.

Hemos discutido sobre como resolver este problema y hemos llegado a las siguiente conclusiones:

Evidentemente, si eliminamos los SLEEP se hace más improbable que pase esto, pero no imposible, así que como primera opción sería intentar cambiar el orden de bloqueos en caso de que se pueda, haciendo que ambos procesos bloqueen las tablas en el mismo orden, de manera que, si uno ha bloqueado una tabla, el otro intente bloquear primero la misma que este y no la otra tabla.

Otra idea que hemos tenido es que se puede ejecutar un contador de espera a la hora de bloquear una tabla, y en caso de que el contador llegue a 0, por ejemplo, si están interbloqueados, salgan del bloqueo y vuelvan a intentar bloquear, volviendo a activar el contador. Es muy difícil que se llegue a interbloqueo todo el tiempo una vez hayan llegado al mismo punto.

[1] Si se hace algún COMMIT intermedio, es necesario usar de nuevo la sentencia BEGIN, pues COMMIT acaba la transacción como tal, sin embargo, aún no habremos terminado con el propósito de la transacción general.