Memoria SI
(Práctica 4)

Sergio Galán Martín
sergio.galanm@estudiante.uam.es

David Cabornero Pascual
david.cabornero@estudiante.uam.es

11 de diciembre de 2019

1. Optimización

1.1. Apartado A

Hemos observado con la consulta *explain* que el *count* realiza una búsqueda secuencial, lo cual tiene un gran coste.

```
Aggregate (cost=5627.93..5627.94 rows=1 width=8)
  -> Gather (cost=1000.00..5627.92 rows=2 width=4)
       Workers Planned: 1
       -> Parallel Seq Scan on orders (cost=0.00..4627.72 rows=1 width=4)
            Filter: ...
```

Vamos a probar a añadir índices sobre *totalamount*, *year* y *month*. Nuestro primer acercamiento fue poniendo un índice sobre *totalamount*, incrementando la eficiencia (aunque era bastante mejorable). A continuación se deja el resultado:

```
Aggregate (cost=4480.32..4480.33 rows=1 width=8)
  -> Bitmap Heap Scan on orders (cost=1126.90..4480.32 rows=2 width=4)
       Recheck Cond: (totalamount > '100'::numeric)
       Filter: ...
       -> Bitmap Index Scan on totalamount_idx
            (cost=0.00..1126.90 rows=60597 width=0)
          Index Cond: (totalamount > '100'::numeric)
```

Podemos pasar que ahora se realiza un *heap scan*, lo cual hace que no nos veamos examinados a examinar todas las filas de la tabla, mejorando claramente la eficiencia.

Después de probar numerosas combinaciones, hemos concluido que la mejor opción posible a nivel de eficiencia consiste en insertar un índice combinado, donde primero lo aplicamos sobre el año y el mes indiferentemente y después sobre el *totalamount*. Esto es debido a que hay muchas entradas con totalamount ¿100 y bastantes menos que se hayan realizado en una pareja mes/año determinada. Probablemente si pusiésemos un umbral muy alto el índice que indexa primero por totalamount y luego por mes/año nos diera un resultado mejor. A continuación dejamos unos ejemplos significativos:

```
-- Con indice mezclado ymt
Aggregate (cost=12.30..12.31 rows=1 width=8)
  -> Bitmap Heap Scan on orders (cost=4.45..12.29 rows=2 width=4)
       Recheck Cond: ...
```

```
      ->  Bitmap Index Scan on ymt_idx (cost=0.00..4.45 rows=2 width=0)
            Index Cond: ...

-- Con indice mezclado tym
Aggregate (cost=1973.91..1973.92 rows=1 width=8)
  ->  Index Scan using tym_idx on orders (cost=0.42..1973.90 rows=2 width=4)
      Index Cond: ...

-- Con indice totalamount y monthyear
Aggregate (cost=23.80..23.81 rows=1 width=8)
  ->  Bitmap Heap Scan on orders (cost=4.47..23.79 rows=2 width=4)
      Recheck Cond: ...
      Filter: (totalamount > '100'::numeric)
      ->  Bitmap Index Scan on monthyear_idx (cost=0.00..4.47 rows=5 width=0)
            Index Cond: ...
```

Como podemos observar, lo peor que podemos hacer es indexar primero por *totalamount* (como podríamos esperar, ya que si el totalamount es bajo, hay una gran cantidad de entradas que cumplen ese requisito). Después de esto, lo más eficiente es un índice combinado frente a tres índice independientes.

Anotaciones: Da igual poner primero el year y luego el month (posiblemente porque la diferencia de número de meses y de años es despreciable, no como con el totalamount) Lo más eficiente es hacer un index primero con la date y después con el totalamount Con el que más se nota cualquier arreglo de eficiencia es con el mes y el año. Poner tres índices es puta mierda

1.2. Apartado C

```
-- Apendice 1 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)
-- Apendice 1 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)
-- Apendice 1 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)
```

La única *query* que empieza a mostrar resultados tras ejecutarla es la tercera, ya que es la única que tiene coste 0.00 como inicial en su operación principal (HashSetOp). A parte de esta razón numérica, se puede ver también en que HashSetOp usa un conjunto, y con sus dos *subqueries*

cuando se encuentra un cliente (*query1*) que no tenga algún pedido con status *Paid* (*query2*) ya lo "sube.ªl conjunto y podemos mostrarlo.

La segunda *query* no nos muestra nada hasta que no finaliza, ya que, pese a que se podrían ir obtieniendo resultados parciales, la condición de agrupación imposibilita el hecho de poder mostrar algo hasta que no se ha terminado completamente el Append interno.

La primera *query* es la más evidente, ya que hasta que no termina el SubPlan 1 no puede hacer la filtración de tomar los elementos que no están en el resultado de ese SubPlan.

La primera *query* no es capaz de ejecutar nada al comenzar la ejecución, ya que tiene que realizar un subplan que se lo impide completamente.

La segunda *query* sí que puede ir mostrando los resultados según se inicia la ejecución, ya que la función *HashAggregate* permite ir creando una tabla Hash desordenada gracias a una gran cantidad de memoria que nos permite ver el resultado parcialmente antes de su finalización.

La tercera *query* no puede mostrarse en tiempo de ejecución, dado que *HashSetOp* requiere ordenación en sus resultados (por lo tanto tendremos que esperar al final de la ejecución).

1.3. Apartado D

```
-- Apendice 2 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)
-- Apendice 2 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)
```

Parece que se nota una cierta mejora al intentar contar los campos *NULL*, a diferencia de si se intenta contar un cierto campo. Esto se debe a que la *query* gana en eficiencia si solo es necesario intentar acceder al campo, ya que en caso contrario deberá acceder al valor y realizar una comprobación de *Strings*.

```
--Apendice 2 con index en Status 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 id (cost=0.00..19.24 rows=909 width=0)
                    Index Cond: (status IS NULL)

--Apendice 2 con index en Status 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 id (cost=0.00..19.24 rows=909 width=0)
                    Index Cond: ((status)::text = 'Shipped'::text)
```

Ahora, como podemos ver, la diferencia de costes se ha reducido a un nivel despreciable. Esto es debido a que, al haber indexado el valor *Status* todos los valores son idénticos a ojos de la *query*. Sabemos que hay un *árbol B* detrás que ha 'ordenado' los resultados, así que el mayor coste consigue en encontrar el primer resultado (que no conlleva gran trabajo).

```
--Apendice 2 con index en Status y ANALIZE Query 1
 Aggregate (cost=7.26..7.27 rows=1 width=8)
   ->  Index Only Scan using id on orders (cost=0.42..7.25 rows=1 width=0)
         Index Cond: (status IS NULL)

--Apendice 2 con index en Status y ANALIZE Query 2
Finalize Aggregate (cost=4209.52..4209.53 rows=1 width=8)
  ->  Gather (cost=4209.41..4209.52 rows=1 width=8)
        Workers Planned: 1
        ->  Partial Aggregate (cost=3209.41..3209.42 rows=1 width=8)
              ->  Parallel Seq Scan on orders (cost=0.00..3023.69 rows=74288 width=0)
                    Filter: ((status)::text = 'Shipped'::text)
```

La primera *query* se ha visto claramente beneficiada por el uso de *ANALYZE* (el coste se ha reducido en un factor de 200). Esto es debido a que no hay ningún caso en el que *status* sea *NULL*. Con las estadísticas obtenidas, *ANALYZE* consigue la frecuencia con la que aparece cada atributo. En este caso, como aparece 0 veces, la cantidad de información dada es bastante clarificadora. En cambio, el plan realizado para la segunda Query ha cambiado según las estadísticas conseguidas, y paralelizando conseguiría un plan muy óptimo (de hecho, con **tan solo** 4 *workers* conseguiríamos un rendimiento similar al anterior). Sin embargo, el paralelismo no se puede aplicar una vez se ha indexado, acabando así con la mejora obtenida, ya que las estadísticas han quedado completamente obsoletas.

2. Transacciones y Deadlocks

2.1. Apartado E