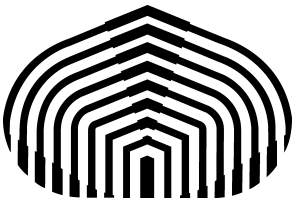

Universidad Simón Bolívar

Departamento de Computación y Tecnología de la Información

CI5313 - Arquitectura y Administración de Bases de Datos

Profs. Marlene Goncalves y Josué Ramírez

Informe sobre Optimización de Consultas: FBV

Ackerman, Moisés	Benzecri, Gustavo	Klie, David
11-10005@usb.ve	11-10097@usb.ve	11-10500@usb.ve

Caracas, junio de 2016

Índice

Introducción

Una Base de datos comienza con un problema en el cual se quieren almacenar datos de forma persistente. De este problema y su universo que se suele llamar «minimundo» surge un modelo lógico que, después de varias iteraciones, representa de la forma mas exacta posible el problema (o al menos lo relacionado con almacenamiento de datos).

Luego del modelado, el siguiente paso es llevar ese modelo lógico a uno mas concreto. En este paso se crean entonces todas las tablas y relaciones entre las mismas, se declaran las resticciónes y se determinan los disparadores que tendrá la base de datos.

Cuando termina toda la fase transformación del modelo lógico a físico (o de más bajo nivel), se podría decir que la Base de datos esta lista para recibir datos. Mientras el volumen de datos sea pequeño, el manejador encargado de realizar las consultas, actualizaciones o transacciones puede manejar con relativa facilidad los datos. Sin embargo, las bases de datos tienen a crecer más que a quedarse estáticas y, cuando esto ocurre, algunas operaciones pueden tornarse costosas.

Afortunadamente, los sistemas manejadores de bases de datos proveen estructuras y facilidades para optimizar estas operaciones.Se tienen, entre las más famosas para optimizar consultas, indices, tablas particionadas o ajuste de parámetros como memoria. Estas estructuras, no obstante, deben ser especificadas por el usuario encargado del buen funcionamiento de la base de datos.

Las estadísticas de cada tabla juegan un papel importante en la elección de un mejor plan de ejecución por parte del manejador de la base de datos y para la elección de una estructura auxiliar. Las estadísticas tambíen le dicen mucho al administrador de la base de datos acerca de qué estructura se debe crear para optimizar un grupo de consultas u operaciones sobre la base de datos.

El presente informe tiene como finalidad describir el comportamiento de estructuras de optimización de consultas ante una Báse de datos de un tamaño considerablemente grande sobre la cuál se deben optimizar ciertas consultas. Para optimizar estas consultas, el úso de los conocimientos teóricos acerca de optimización de consultas, las estadísticas que se puedan obtener del manejador, las que se puedan derivar de estas y serán las herramientas para determinar que estructuras se deberían usar o no en alguna de las consultas planteadas.

Cabe mencionar que todas las pruebas de velocidad, tanto de las consultas originales sin estructuras secundarias, como de las consultas mejoradas por nosotros con estructuras secundarias para facilitar el acceso, se realizaron en una máquina dedicada de Amazon Web Services (AWS), a menos que se indique lo contrario.

Estadísticas sobre la base de datos FBV

El primer paso pasa poder optimizar las consultas dadas es conocer el volumen total de la base de datos, el volumen de cada tabla individual, el tamaño promedio de cada fila, y las estadísticas apropiadas para cada consulta.

Estadísticas generales de la base de datos

En esta subsección se describen las estadísticas relacionadas con la base de datos en su totalidad, independientemente de las consultas que se vayan a optimizar.

Volumen total de los datos

Para determinar el volumen total de tuplas existentes en la base de datos se realizó la siguiente consulta:

```
select sum(n_live_tup)
from pg_stat_user_tables
where schemaname='original';
```

La consulta arrojó un valor de 8660591 tuplas.

Volumen total de cada tabla

Para calcular el volumen total de cada tabla utilizamos la siguiente consulta:

```
select
  c.relname,
  n_live_tup,
  relpages,
  floor(n_live_tup/relpages::float) as tuples_per_page
from
  pg_stat_user_tables b,
  pg_class c
where relnamespace = (select oid from pg_namespace where nspname='original')
  and schemaname = 'original'
  and b.relname= c.relname;
```

De esta consulta se obtuvieron los resultados de la Tabla 1 .

Tabla 1: Tuplas, páginas y tuplas por página de las tablas de FBV.

Tabla	Tuplas	Páginas	Tuplas por pagina
`part`	200000	3715	53
`supplier`	10000	213	46
`partusupp`	800000	17451	45
`lineitem`	6001181	98544	60
`region`	5	1	5
`nation`	25	1	25
`customer`	150000	3566	42
`orders`	1500000	25196	59

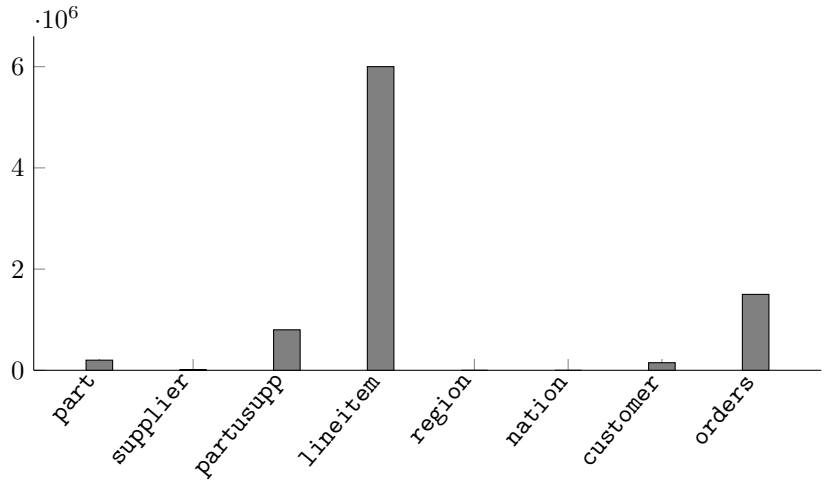

Figura 1: Cantidad de tuplas en cada tabla

En la Figura 1 podemos observar que la tabla de `lineitem` ocupa la mayor parte de los datos almacenados (aproximadamente un 69 % del total de tuplas). Cualquier consulta que requiera acceder a una buena parte de los datos almacenados en esta tabla exigirá mucha atención al momento de ser optimizada.

Para las tablas `nation` y `region`, dado que caben en una sola página, se puede concluir que no necesitan optimización alguna pues un acceso directo es siempre la mejor opción para recuperar sus registros.

Análisis de cada tabla

En el análisis de cada tabla tomarán en cuenta los siguientes aspectos:

- Tamaño promedio de la tupla

- Para cada atributo, su tamaño promedio, cantidad de elementos distintos y factor de reducción.
- Cualquier otra estadística que resulte últil para la optimización de alguna de las consultas propuestas.

Para el cálculo del tamaño promedio de cada tupla, en todas las tablas, se realizó la siguiente consulta:

```
select
  tablename,
  sum(avg_width) as tam_promedio_tuplas
from
  pg_stats
where
  tablename in (
    select
      relname
    from pg_stat_user_tables)
group by
  tablename;
```

De la cual se obtuvieron los siguientes resultados

Tabla 2: Tamaño promedio de cada tupla

Tabla	Tamaño promedio de tupla
part	114
supplier	137
partusupp	144
lineitem	98
region	78
nation	91
customer	158
orders	100

En el análisis de cada tabla, se realizó una consulta diseñada para extraer la siguiente información acerca de las tablas:

- El tamaño medio de cada atributo
- El número de valores distintos que tiene la columna. Si el valor es -1, es un valor único

- La correlación que existe entre el orden lógico y el orden físico (mientras más cercano a 1 o -1 mejor a la hora de que el manejador haga index scan)
- La frecuencia más alta hallada. Esta nos permite tener una cota superior para un determinado valor
- El factor reductor. Para conocer la selectividad de la columna.

La consulta es la siguiente:

```
prepare stats_table as
select
  attname,
  avg_width,
  ( case
      when n_distinct < 0  and n_distinct <> -1
        then -n_distinct * t.reltuples
        else n_distinct
      end) as ndistinct,
  correlation,
  most_common_freqs[1] as upper_bound,
  ( case
      when n_distinct < 0
        then -1 / (n_distinct * t.reltuples)
        else 1 / n_distinct
      end) as FR
from
  pg_stats,
  ( select
      relname,
      reltuples
    from
      pg_class
    where relnamespace = (
      select
        oid
      from
        pg_namespace
      where
        nspname='original')
    ) t
where t.relname = tablename
  and tablename = $1
  and schemaname = 'original'
order by
  fr;
```

Análisis de `lineitem`

Al ejecutar la consulta sobre lineitem se obtuvieron los resultados mostrados en la Tabla 3. De la tabla se puede decir que las columnas con peor selectividad son `l_linestatus`, `l_returnflag` y `l_shipsintruct`. Por otro lado, los atributos con mayor selectividad tenemos `l_comment`, `l_orderkey` y `l_extendedprice`.

Tabla 3: Estadísticas para la relación `lineitem`.

Atributo	Tam. pr.	# vals. dist.	Correlación	Cota sup.	Selectividad
`l_orderkey`	4	1206300	1	0.000016667	0.000000829
`l_partkey`	4	197029	0.00235099	0.00003	5.07539e-06
`l_suppkey`	4	10000	-0.000106049	0.000173333	0.0001
`l_linenumber`	4	7	0.176068	0.250317	0.14285714
`l_quantity`	5	50	0.0195651	0.0205067	0.02
`l_extendedprice`	8	767024	0.000341259	2.33333e-05	1.3037401e-06
`l_discount`	4	11	0.0868008	0.0922367	0.09090909
`l_tax`	4	9	0.109181	0.11199	0.11111111
`l_returnflag`	2	3	0.377041	0.506517	0.33333333
`l_linestatus`	2	2	0.499747	0.500087	0.5
`l_shipdate`	4	2525	-0.00126623	0.000536667	0.00039603
`l_comitdate`	4	2465	-0.00119272	0.000536667	0.00040567
`l_receipdate`	4	2543	-0.00128363	0.000553333	0.00039323
`l_shipinstruct`	13	4	0.250591	0.250767	0.25
`l_shipmode`	5	7	0.145059	0.143523	0.14285714
`l_comment`	27	1763690	0.000151724	0.000193333	0.000000567

Análisis de `orders`

Al ejecutar la consulta sobre orders, se obtuvieron los resultados mostrados en la Tabla 4.

Tabla 4: Estadísticas para la relación `orders`.

Atributo	Tam. pr.	# vals. dist.	Correlación	Cota sup.	Selectividad
`o_orderkey`	4	-1	0.999999	-	6.666666667e-07
`o_comment`	49	1461050	0.000667332	1.33333e-05	6.844383362e-07
`o_totalprice`	8	1439660	0.00179892	0.00001	6.946098966e-07
`o_custkey`	4	96824	0.00212814	4.66667e-05	0.000010328
`o_orderdate`	4	2406	0.00254059	0.00056	0.0004156276
`o_clerk`	16	1000	0.000855419	0.00118	0.001
`o_orderpriority`	9	5	0.201246	0.201113	0.2
`o_orderstatus`	2	3	0.476312	0.488067	0.3333333333
`o_shippriority`	4	1	1	1	1

Análisis de `customer`

Al ejecutar la consulta sobre orders, se obtuvieron los resultados mostrados en la Tabla 5.

Tabla 5: Estadísticas para la relación `customer`.

Atributo	Tam. pr.	# vals. dist.	Correlación	Cota sup.	Selectividad
`c_phone`	16	-1	0.000497003		6.666666667e-06
`c_custkey`	4	-1	0.999988		6.666666667e-06
`c_name`	19	-1	0.999988		6.666666667e-06
`c_address`	26	-1	0.0011718		6.666666667e-06
`c_comment`	73	149968	-0.000241386	1.33333e-05	6.668089192e-06
`c_acctbal`	6	140187	0.00410825	2.66667e-05	7.133329053e-06
`c_nationkey`	4	25	0.0405429	0.0410733	0.04
`c_mktsegment`	9	5	0.200573	0.20126	0.2

Análisis de `part`

Al ejecutar la consulta sobre part, se obtuvieron los resultados mostrados en la Tabla 6.

Tabla 6: Estadísticas para la relación `part`.

Atributo	Tam. pr.	# vals. dist.	Correlación	Cota sup.	Selectividad
`p_partkey`	4	-1	0.999905		5e-06
`p_name`	33	199997	-0.000591865	1e-05	5.000075001e-06
`p_comment`	14	131753	-0.00166307	0.000615	7.589960000e-06
`p_retailprice`	6	20899	0.192004		4.784917938e-05
`p_type`	21	150	0.0095101	0.007255	0.0066666666667
`p_size`	4	50	0.0237148	0.020885	0.02
`p_container`	8	40	0.0252164	0.025765	0.025
`p_brand`	9	25	0.0414318	0.041165	0.04
`p_mfgr`	15	5	0.201261	0.20152	0.2

Análisis de `partsupplier`

Al ejecutar la consulta sobre partsupplier, se obtuvieron los resultados mostrados en la Tabla 7.

Tabla 7: Estadísticas para la relación `partsupplier`.

Atributo	Tam. pr.	# vals. dist.	Correlación	Cota sup.	Selectividad
`ps_comment`	126	798569	0.000561844	6.66667e-06	1.252239944e-06
`ps_partkey`	4	200105	0.999993	1.33333e-05	4.997376377e-06

Atributo	Tam. pr.	# vals. dist.	Correlación	Cota sup.	Selectividad
ps_supplycost	6	97978	-0.000937153	4e-05	1.020637285e-05
ps_suppkey	4	10000	0.00181275	0.000156667	0.0001
ps_availqty	4	9999	0.00135311	0.00018	0.0001000100010

Análisis de supplier

Al ejecutar la consulta sobre supplier, se obtuvieron los resultados mostrados en la Tabla 8.

Tabla 8: Estadísticas para la relación supplier.

Atributo	Tam. pr.	# vals. dist.	Correlación	Cota sup.	Selectividad
s_address	25	-1	0.000978475		0.0001
s_suppkey	4	-1	0.999954		0.0001
s_name	19	-1	0.999954		0.0001
s_phone	16	-1	0.00634556		0.0001
s_comment	63	-1	-0.00867227		0.0001
s_acctbal	6	9955	0.0158685	0.0002	0.0001004
s_nationkey	4	25	0.0458266	0.0438	0.04

Análisis de nation

Al ejecutar la consulta sobre nation, se obtuvieron los resultados mostrados en la Tabla 9.

Tabla 9: Estadísticas para la relación nation.

Atributo	Tam. pr.	# vals. dist.	Correlación	Cota sup.	Selectividad
n_comment	75	-1	0.0469231		0.04
n_nationkey	4	-1	1		0.04
n_name	8	-1	0.913077		0.04
n_regionkey	4	5	0.347692	0.2	0.2

Análisis de region

Al ejecutar la consulta sobre region, se obtuvieron los resultados mostrados en la Tabla 10.

Tabla 10: Estadísticas para la relación region.

Atributo	Tam. pr.	# vals. dist.	Correlación	Cota sup.	Selectividad
r_regionkey	4	-1	1		0.2

Atributo	Tam. pr.	# vals. dist.	Correlación	Cota sup.	Selectividad
`r_name`	7	-1	1		0.2
`r_comment`	67	-1	0.6		0.2

Optimización de las consultas

A continuación se presenta un análisis de las consultas a optimizar, el tiempo que tardan y la mejora, de haberla, con respecto a la consulta original en caso de ser una propuesta de optimización de la consulta.

Por razones de espacio no se colocarán los `explain plans` en este documento pero si se desea consultar alguno de estos planes se encuentran almacenados en la carpeta **results** tanto en formato YAML como en el formato estándar de salida de postgres. Sin embargo, se podrán encontrar en este documento los planes de ejecución de la consulta en formato gráfico para su fácil entendimiento.

En alguna consultas se realizaron estadísticas puntuales para ayudar a determinar, junto con las estadísticas de la sección anterior, para proponer una optimización más adecuada.

Es importante destacar que se trabajó con dos copias de la BD suministrada: una sin modificación alguna y una segunda base de datos conde se crearon las estructuras propuestas para la optimización de la consulta.

Consulta Q11: Proveedor con el mínimo costo

La consulta a optimizar es la siguiente:

```
select
  s_acctbal,
  s_name,
  n_name,
  p_partkey,
  p_mfgr,
  s_address,
  s_phone,
  s_comment
from
  part,
  supplier,
  partsupp,
  nation,
  region
where
  p_partkey = ps_partkey
  and s_suppkey = ps_suppkey
  and p_size = $1
  and p_type like $2
  and s_nationkey = n_nationkey
  and n_regionkey = r_regionkey
  and r_name = $3
  and ps_supplycost = (
    select
      min(ps_supplycost)
    from
      partsupp,
      supplier,
      nation,
      region
    where
      p_partkey = ps_partkey
      and s_suppkey = ps_suppkey
      and s_nationkey = n_nationkey
      and n_regionkey = r_regionkey
      and r_name = $3
  )
order by
  s_acctbal desc,
  n_name,
```

```
  s_name,
  p_partkey;
```

Al hacer `Explain Analyze` sobre la consulta sin ningúna estructura adicional se obtuvo el plan de ejecución mostrado en la Figura 2, con una duración total de 1992 ms. Puede verse que el nodo más caro y lento en este grafo fue el *Seq Scan* (906 ms) que se hace sobre `partsupp` como parte del *Hash Join* con la tabla `supplier` sobre la igualdad de los atributos `ps_suppkey` y `s_suppkey`, cuyos nombres indican que se trata de un join sobre una clave foránea entre la entidad `partsupp` y su `supplier` correspondiente.

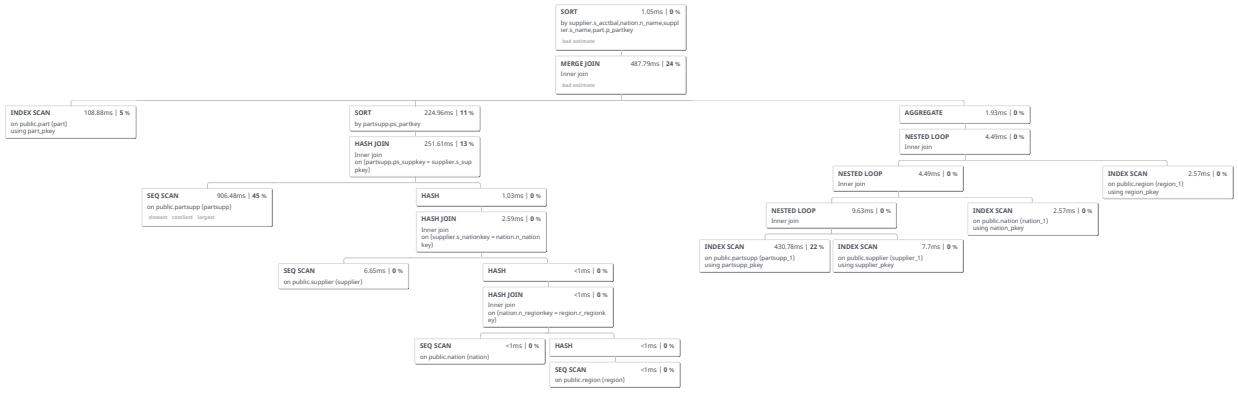

Figura 2: Árbol de ejecucion de la consulta Q11 original

Cabe mencionar que antes de plantear cualquier cambio a la estructura de una base de datos para mejorar el desempeño de una consulta, es buena idea revisar la consulta en sí. En este caso, puede notarse que la consulta tiene una sub-consulta correlacionada, en la cual se busca el costo mínimo para una pieza entre todos los proveedores que la ofrecen. Esta sub-consulta puede entonces reescribirse con una *Common Table Expression* con la palabra clave `with` para evitar que se evalúe más de una vez por cada pieza. Resulta entonces la siguiente consulta con idéntica semántica:

```
with minPerPart as (
  select
    min(ps_supplycost) as mpp_mincost,
    ps_partkey         as mpp_partkey
  from
    part,
    partsupp,
    supplier,
    nation,
    region
  where p_partkey          = ps_partkey
    and ps_suppkey         = s_suppkey
    and s_nationkey        = n_nationkey
```

```
    and n_regionkey        = r_regionkey
    and p_size             = $1
    and reverse(p_type) like reverse($2)
    and r_name             = $3
  group by
    mpp_partkey
  order by
    mpp_partkey
)
select
  s_acctbal,
  s_name,
  n_name,
  p_partkey,
  p_mfgr,
  s_address,
  s_phone,
  s_comment
from
  part,
  supplier,
  partsupp,
  nation,
  region,
  minPerPart
where p_partkey          = ps_partkey
  and ps_suppkey         = s_suppkey
  and s_nationkey        = n_nationkey
  and p_size             = $1
  and reverse(p_type) like reverse($2)
  and r_name             = $3
  and ps_supplycost      = mpp_mincost
  and p_partkey          = mpp_partkey
order by
  s_acctbal desc,
  n_name,
  s_name,
  p_partkey;
```

Luego de reescribir la consulta, agregamos un par de índices a la base de datos. El primero lo creamos sobre la tabla `part`, en las columnas `p_size` y el reverso de `p_type`, en ese órden. Éste índice le permite al manejador conseguir primero las piezas de un tamaño específico, y luego las que tengan un sufijo en particular. Como nos interesa que la comparación sea por sufijo en lugar de por prefijo, sólo tiene sentido usar el reverso de `p_type`, invirtiendo también ambos operandos del operador `like`. Los atributos del índice están en este órden

porque invertirlos no permitiría hacer búsquedas por igualdad en el atributo `p_size`, que es lo que se desea en esta consulta. Este índice fue creado con el siguiente comando:

```
create index idx_q11_p_size_reverse_type
  on part (p_size, reverse(p_type) text_pattern_ops);
```

El segundo índice que agregamos para esta consulta es sobre la tabla `partsupp`, en los atributos `ps_partkey`, `ps_suppkey`, `ps_supplycost`, y permite que dos operaciones *Join* utilicen *Index Only Scan*s en el lado interno, en lugar de la costosa secuencia *Bitmap Index Scan - Bitmap Heap Scan* sobre una tabla del tamaño de `partsupp`. Este índice fue creado con el siguiente comando:

```
create index idx_q11_partsupp
  on partsupp (ps_partkey, ps_suppkey, ps_supplycost);
```

Luego de la reescritura de la consulta y la creación del par de índices, el tiempo de ejecución (con caché fría) de esta consulta baja de 1992 ms a 334 ms, tan sólo el 16.7 % del tiempo de la consulta original. El plan resultante es el mostrado en la Figura 3.

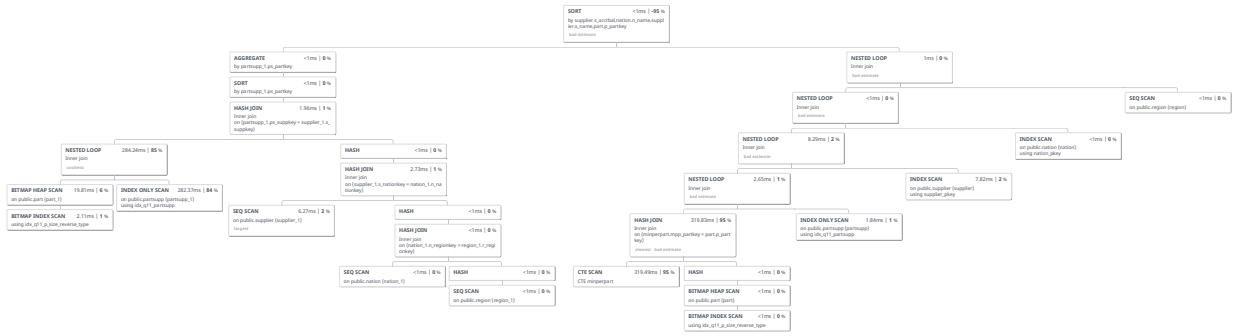

Figura 3: Árbol de ejecucion de la consulta Q11 modificada, con índices

Luego de esta mejora, consideramos cambiar el índice sobre la tabla `part` para que incluyera más columnas, específicamente `p_partkey` y `part.p_mfgr`, ya que estas columnas son usadas en nodos subsecuentes, de forma que los pares de nodos *Bitmap Index Scan - Bitmap Heap Scan* sobre la tabla `part` se pudieran convertir en un *Index Only Scan*. Sin embargo, este no es el caso, debido a que el índice resultante es demasiado grande, y el factor reductor del nodo muy pequeño como para esto, y termina siendo más económico el par de nodos *Bitmap Index Scan - Bitmap Heap Scan* sobre el índice sencillo.

Otra crítica posible a esta propuesta está en el nuevo nodo `CTE Scan`. Este nodo resulta de la evaluación de la consulta **`with`** dentro de la consulta principal, y su tiempo de ejecución es significativo, unos 319 ms. Sin embargo, esta alternativa resulta muy superior a la original, en la cual se evaluaba muchas veces la misma consulta correlacionada, generando un árbol más costoso.

Consulta Q12: Prioridad de envío

La consulta a optimizar es la siguiente:

```
select
  l_orderkey,
  sum(l_extendedprice*(1-l_discount)) as revenue,
  o_orderdate,
  o_shippriority
from
  customer,
  orders,
  lineitem
where
  c_mktsegment = $1
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < $2
  and l_shipdate > $2
group by
  l_orderkey,
  o_orderdate,
  o_shippriority
order by
  revenue desc,
  o_orderdate;
```

Al hacer `Explain Analyze` sobre la consulta sin ningúna estructura adicional se obtuvo el plan de ejecución mostrado en la Figura 4, con una duración total de 17932 ms. El nodo más caro y lento en este grafo es el *Nested Loop* que tomó 16.59 segundos en ejecutarse lo cual equivale a 93 % del tiempo de ejecución y es claramente el cuello de botella de la consulta. Este se debe a que el bucle externo es grande y por cada elemento del bucle externo (que cuenta como una operación *I/O*) y debe recorrer la segunda tabla (el cual cuenta también como un *I/O*) y tiene un tamaño similar al del conjunto del bucle externo.

Como el cuello de botella es el *Nested Loop Join*, se debe buscar reducir el número de registros que deben formar parte de este. Se crearon índices compuestos que incluyan a todos los atributos que se utilizan en la consulta para este fin.

Se estudió la selectividad de cada uno de los conjuntos filtrados por las condiciones en la sección `where` de la consulta. Se debe agarrar la condición más selectiva en el índice compuesto para reducir el conjunto de registros lo más rápido posible y reduciendo el tiempo de las consultas.

El primer índice creado fue sobre `orders`

Figura 4: Arbol de ejecución de la consulta Q12 original

```
create index q12_orders_idx
  on orders(o_custkey, o_orderdate, o_orderkey, o_shippriority);
```

Este índice elimina el *Sequential Scan* utilizado al buscar las ordenes al principio y se cambia por *Index Only Scan*, el cual reduce el tiempo de ejecución de este nodo por aproximadamente la mitad del tiempo original además de reducir el número de registros con los se trabajará por el resto de la consulta.

El segundo índice creado fue sobre `lineitem`

```
create index q12_lineitem_idx
  on lineitem(l_shipdate, l_orderkey, l_extendedprice, l_discount);
```

Este índice elimina el *Index Scan* realizado sobre el nodo de selección de lineitem y lo cambio por un *Index Only Scan*, el cual reduce el tiempo de ejecución de 16.48 segundos a 1 segundo (aunque el anterior fue ejecutado en paralelo con el `join`), el 6 % del tiempo original.

El tercer índice creado fue sobre `customer`

```
create index q12_customer_idx
  on customer(c_mktsegment, c_custkey);
```

Este índice elimina el *Sequential Scan* realizado al seleccionar los compradores de la consulta y lo cambia por un *Index Only Scan*, el cual reduce el tiempo de ejecución de 235 ms a 12 ms, el 5 % del tiempo original.

Además, se crea una tabla de hash para orders y se cambia el *Nested Loop Join* por un *Hash Join*, lo cual reduce el tiempo de ejecución de este `join` de 16.59 segundos a 1.48 segundos, el 9 % del tiempo original.

El tiempo de ejecución (con caché fría) de esta consulta baja de 17932 ms a 2985 ms, tan sólo el 16.5 % del tiempo de la consulta original. El plan resultante es el mostrado en la Figura 5.

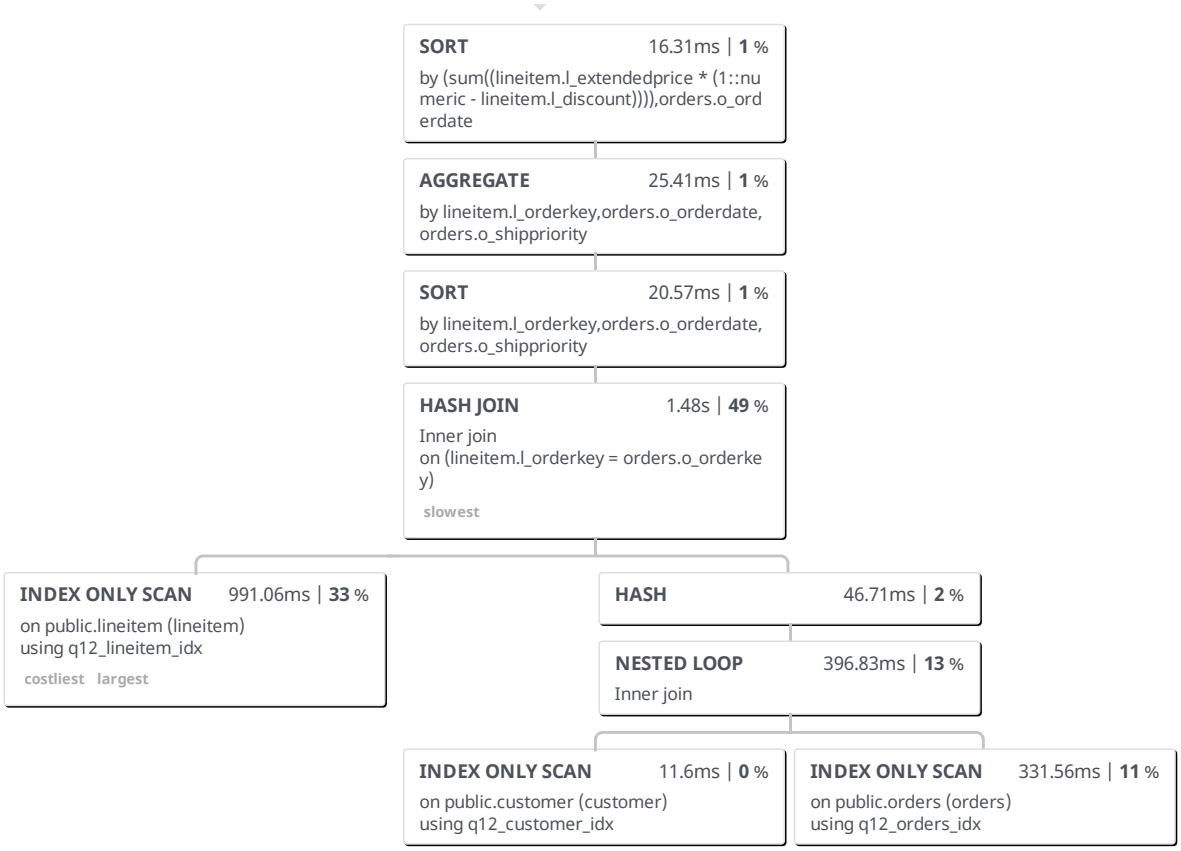

Figura 5: Árbol de ejecución de la consulta Q12 modificada

Consulta Q13: Reporte de ítems devueltos

La consulta que se tuvo que optimizar fue la siguiente:

```
prepare q13 as
select c_custkey,c_name, sum(l_extendedprice * (1 - l_discount)) as revenue,
    c_acctbal, n_name, c_address, c_phone, c_comment
from customer, orders, lineitem, nation
where c_custkey = o_custkey and l_orderkey = o_orderkey
    and o_orderdate >= $1 and o_orderdate < $1 + interval '3 month'
    and l_returnflag = 'r' and c_nationkey = n_nationkey
group by c_custkey, c_name, c_acctbal, c_phone, n_name, c_address, c_comment
order by revenue desc;
```

Al ejecutar la consulta sin ningúna estructura adicional se obtuvo el plan de ejecución mostrado en la la Figura 6 . En ella se puede observar que la operación de *scan* sobre la tabla `lineitem` es la operación mas costosa, lenta y extensa de toda la consulta.

En la Figura se puede observar que en todas las tablas involucradas se ejecuta un *Sequential scan* , siendo la tabla `lineitem` la tabla que consume la mayor parte del tiempo de la consulta. Luego, se hacen las proyecciones sobre las tablas `customer`, `nation` y `orders` para obtener los atributos deseados de cada tabla. Esto no sucede en `lineitem`.

Los siguientes pasos son un *Hash Join* entre las tuplas recuperadas de `lineitem` y `orders`, seguido de otro *Hash Join* entre el resultado previo y la relación `customer` y finalmente un último *Hash Join* sobre el resultado de las dos operaciones previas y la tabla `nation`.

Ya para finalizar se procede a ordenar todas las túplas por los atributos especificados en la cláusula **group by** , se ejecuta la función de agregación `sum(l_extendedprice * (1 - l_discount))` y se ordena de forma descendiente la tabla resultante de acuerdo al resultado de la función de agregación.

El tiempo total de ejecución de la consulta fue de 17826,476 ms de los cuales, 12154,645 ms tomó la recuperación de las tuplas deseadas de la tabla `lineitem`. Desde el sistema operativo, la operación tardó 17940.00 ms , 100 ms extra por el *overhead* de iniciar el proceso psql.

Optimizar la recuperación de datos para la tabla `lineitem` fue el objetivo principal para una mejora notable de esta consulta.

Primera Iteración

Para evitar un escaneo secuencial sobre toda la tabla `lineitem` se propuso crear un índice compuesto sobre los atributos `l_returnflag`, `l_orderkey`, `l_extendedprice` y `l_discount` y el mismo orden en el que aqui se menciona.

Un índice con todos estos atributos tendría tuplas de 18 bytes de clave mas los gastos 'administrativos' del apuntador a la página donde se encuentra la tupla completa. Este índice

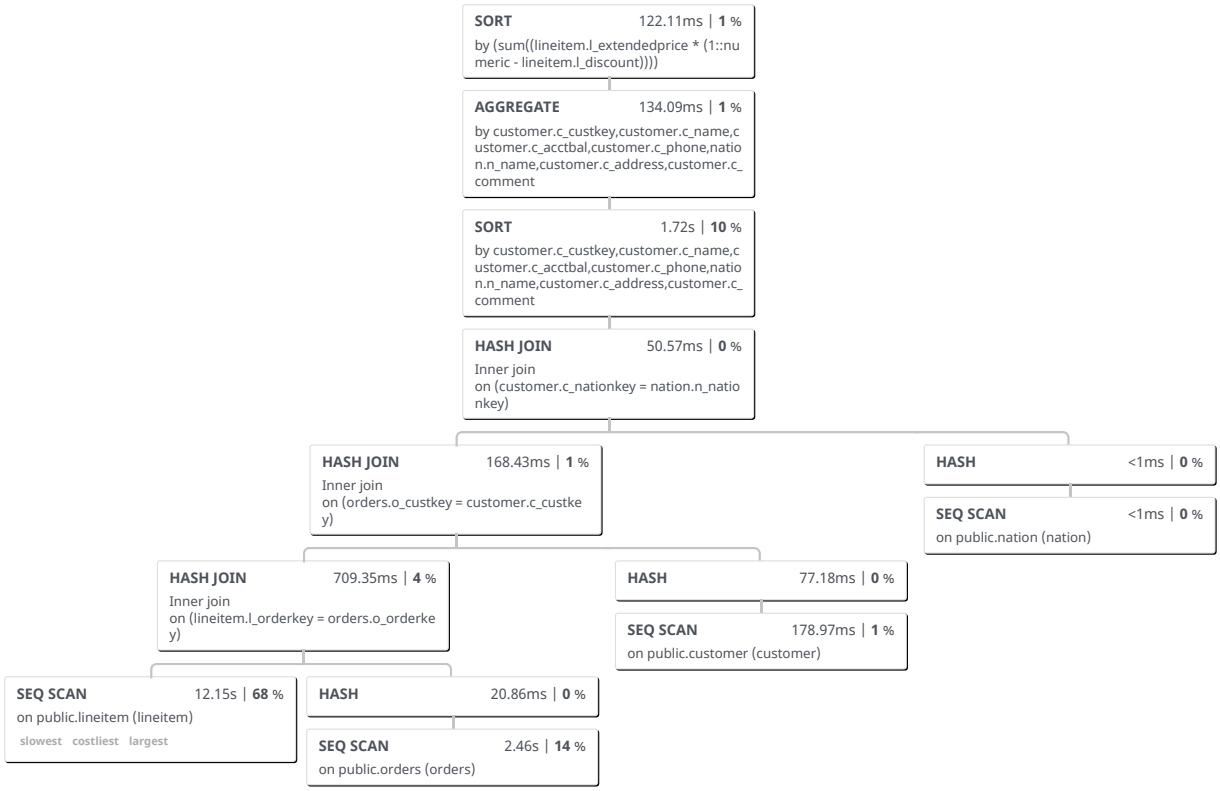

Figura 6: Arbol de ejecucion de la consulta Q13

reduciría el número de operaciones de entrada-salida de 98544 con el escaneo secuencial en `lineitem` a un total estimado de 29563 páginas.

Por otro lado el índice tendría las entradas ordenadas de tal manera que se podría evitar un escaneo completo sobre el índice al solo tener que buscar las entradas que cumplan la condición: `l_returnflag`='R'. Para conocer cual es el porcentaje de la tabla `lineitem` que cumple esta condición se realizó la siguiente consulta sobre las estadísticas:

```
select most_common_vals, most_common_freqs
    from pg_stats
    where attname='l_returnflag';
```

Esta consulta devolvió lo siguiente:

Tabla 11: La tabla muestra que para el valor R, solo un 25 % de las tuplas lo cumplen.

most_common_vals	most_common_freqs
{N,R,A}	{0.5083,0.251133,0.240567}

Dado que el factor de reducción de la columna es 0.33333333... se tiene un mejor pronóstico para el índice al solo tener que recuperar aproximadamente $29563 \times 0.25 = 7390$ páginas.

El mísmo razonamiento se aplicó para la tabla `orders` que, si bien no es tan masiva como la anterior, optimizar el acceso a la misma puede mejorar el desempeño de la consulta.

Para orders se propuso un índice sobre los atributos `o_orderdate`, `o_orderkey` y `o_custkey` en ese orden. Este índice reduciría el número de operaciones de entrada- salida para el escaneo de orders de 25196 a un estimado de 5124 páginas.

Como el factor reductor de este atributo es bajo, 0.00041 para ser exactos, y la cota superior es 0.00056 se estimó obtener un total de $90 \times 0.00056 \times 100 = 5{,}04\,\%$ de las tuplas (259 págnas) del índice en el peor caso y de $90 \times 0.00041 \times 100 = 3{,}69\,\%$.

Luego de la de este análisis se procedió a crear los índices:

```
--elementos para la fecha
create index q13_orders_idx
    on orders(o_orderdate, o_orderkey, o_custkey);

--elementos para lineitem
create index q13_lineitem_idx
    on lineitem(l_returnflag, l_orderkey,
        l_extendedprice, l_discount);
```

Luego se consultó el número de páginas que tenía cada índice con la consulta:

```
select relname,relpages
    from pg_class
    where relname in ('q13_orders_idx','q13_lineitem_idx');
```

obteniendose la Tabla 12 .

Tabla 12: Cantidad de páginas de los índices propuestos

Índice	Paginas
q13_lineitem_idx	29755
q13_orders_idx	5779

Finalmente se procedio a ejecutar la consulta Q13 mejorada para observar que ocurría. El arbol de ejecución con la duración de cada nodo se puede observar en la la Figura 7.

En la Figura 7 se puede observar un cambio en el plan de ejecución de la cosulta con respecto a la original: Ahora se realiza un *Index only scan* sobre los índices definidos sobre tablas `lineitem` y `orders` reduciendo en el caso de `lineitem`, de 12154,645 ms, a 480,31 ms la recuperación de los atributos deseados de esta tabla. Esta reducción representa un 96,04 % de mejoría con respecto al nodo del plan original.

Para la relación `customer` se realiza un *Index scan* y sobre `nation` se realiza un *Sequential scan* seguido de una proyección basada en hash para recuperar los atributos deseados de la tabla.

Luego comienza la fase de operaciones join. Primero se realiza un *Merge Join* sobre las tuplas recuperadas del *Index only scan* sobre los índices de `lineitem` y `orders`, seguido de un ordenamiento del resultado para luego realizar un segundo *Merge Join* entre el resultado previo y las tuplas de `customer`. Finalmente, se realiza un *Hash Join* sobre el resultado previo y la relación `nation`.

Luejo de realizarse los Joins se procede a ordenar la tabla resultane de acuerdo a los atributos mencionados en la cláusula **group by**, siendo esta la operación más lenta ahora; se realiza el calculo de la función de agregación: `sum(l_extendedprice * (1 - l_discount))` y finalmente se ordena el resultado por este resultado de acuerdo al comando **group by**.

Se pudo observar que de 17826,476 ms que duraba la ejecución de la consulta, se paso a 2923,460 ms y con una lectura desde el sistema operativo de 3020,00 ms . Esto supone una mejora de un 83,60 % con respecto a la consulta original. Cualitativamente se puede considerar la mejora como **muy buena**.

De 98544 páginas leídas en `lineitem` sin soporte de ningún índice se pasó a 7291 páginas leídas con soporte del índice `q13_lineitem_idx` , estos dátos son consistentes con lo calculado previamente. Sin embargo recuperar los datos deseados de lineitem representa un 17 % de la ejecución de la consulta y sigue siendo la más costosa y la mas larga.

Para ‘orders tenemos un caso similar: de 25196 páginas leídas sin asistencia de ningún índice, se pasó a leer 223 páginas solamente, lo cual es un poco mas bajo del estimado de 255 páginas.

Figura 7: Arbol de ejecución de Q13 optimizado

Segunda iteración

En la Figura 2 se puede observar que antes de realizar la operación de la función de agregación, el manejador debe ordenar mediante un *external sort* las tuplas para luego ejecutar la función de agregación. Este paso es costoso pues debe ordenar en disco.

Dado este problema se propuso aumentar la variable **work_mem** del manejador para ampliar el espacio de la memoria principal donde se ejecuta la consulta. El valor asignado a **work_mem** paso de 4MB a 64MB. En la Figura 3 se puede observar el nuevo árbol de ejecución.

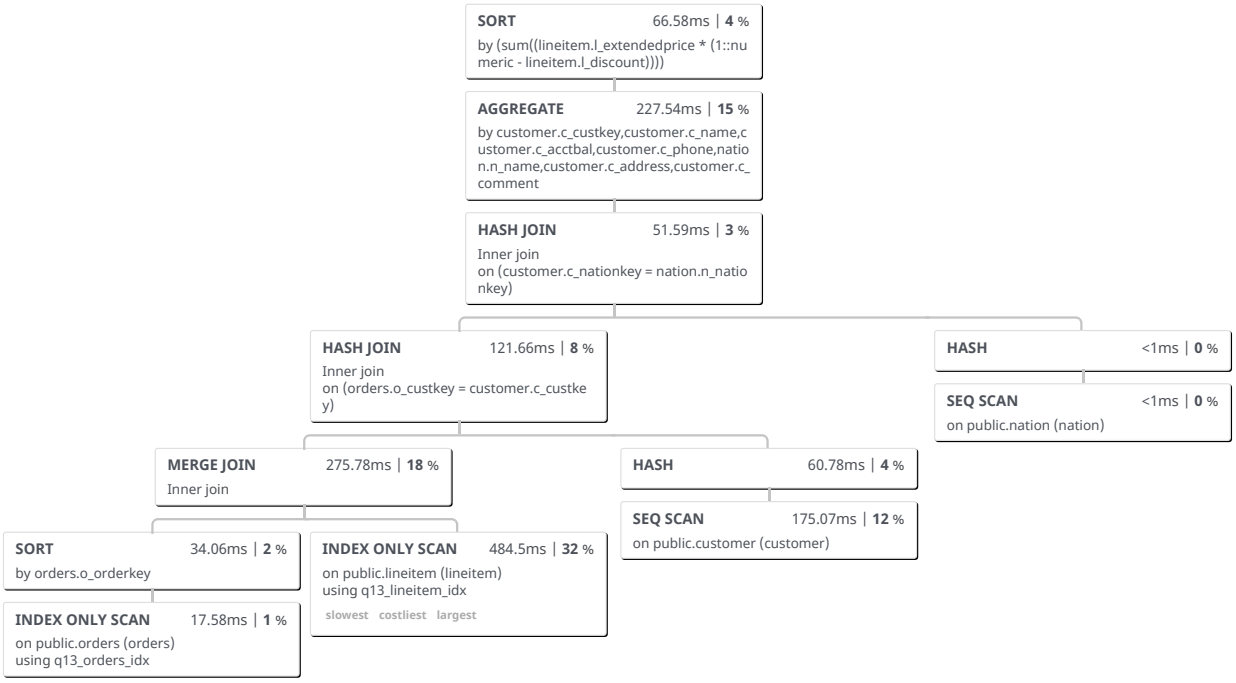

Figura 8: Arbol de ejecución de Q13 optimizada con work_mem=64MB

El arbol de la Figura 8 es similar al de la Figura 6 salvo por el hecho de que se hace un *Index only scan* sobre `lineitem` y `orders` y un *Sequetial scan* sobre las tablas `customer` y `nation` seguido de una proyección basada en hash para extraer los atributos deseados.

En la fase de *joins* se realiza un *Merge Join* entre `customer` y `lineitem` luego se realiza un *Hash Join* entre el resultado anterior y la tabla `customer` para finalmente efectuar un *Hash Join* entre el resultado previo y la tabla `nation`.

Se puede notar que ha desaparecido el nodo de ordenamiento previo que era un *external sort* o un ordenamiento en disco que, dada la lentitud del disco duro con respecto a la memoria principal, es una operación costosa. Se asume que la operación de ordenamiento del **group by** no hizo falta o se realizó *on the fly*.

Con el aumento de la variable `work_mem` de 4MB a 64MB se redujo el tiempo de la consulta de 2923,460 ms a 1466,647 ms lo cual es una mejora en el tiempo de ejecución del 49,83 %

con respecto a la primera optimización. Sin embargo, desde el sistema operativo se reistró una lectura de 1730,00 ms.

Resumen de mejorías

Se pudo observar una notable mejoría en desempeño de la consulta Q13. En la primera iteración se crearon índices que evitaron el acceso a las tablas más grandes de la base de datos reduciendo así la cantidad de accesos a memoria secundaria. Finalmente, en la segunda iteración se aumentó el espacio de memoria de la consulta para eliminar ordenamientos en disco.

Con estas dos iteraciones se logró una reducción del 91,18 % del tiempo de ejecución de la consulta. Sin embargo, no se pudo reducir a menos de un segundo. En la Figura 9 se puede apreciar los cambios.

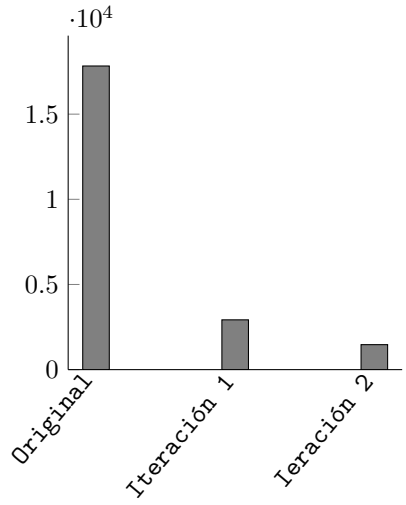

Figura 9: Comparación de tiempos en cada iteración

Consulta Q21: Modos de envío y orden de prioridad

La consulta que se tuvo que optimizar fue la siguiente:

```
prepare q21o as
select l_shipmode,
    sum (case
        when o_orderpriority = '1-URGENT'
        or o_orderpriority = '2-HIGH'
            then 1
            else 0
        end) as high_line_count,
    sum(case
        when o_orderpriority <> '1-URGENT'
        and o_orderpriority <> '2-HIGH'
            then 1
            else 0
        end) as low_line_count
from orders,lineitem
where o_orderkey = l_orderkey and l_shipmode in ($1, $2)
    and l_commitdate < l_receiptdate and l_shipdate < l_commitdate
    and l_receiptdate >= $3 and l_receiptdate < $3 + interval '1 year'
group by l_shipmode
order by l_shipmode;
```

Al ejecutar la consulta sin ningúna estructura adicional se obtuvo el plan de ejecución mostrado en la la Figura 10 . En ella se puede observar que la operación de *scan* sobre la tabla `lineitem` es la operación mas costosa y lenta de toda la consulta, mientras que la operación mas larga fue el *Index scan* sobre la tabla `orders` y se recupera la tupla completa ya ordenada.

Justo despues de recuperar las tuplas, para el caso de `lineitem` se realiza un ordenamiento en memoria de las tuplas obtenidas.

En la fase de *joins* tenemos que el manejador escoge un *Merge Join* como opción más barata, dado que ya venía una tabla ordenada, utilizando la condición `o_orderkey = l_orderkey` como criterio.

Finalmente se efectúa la función de agregación y se ordena el resultado por el atributo `l_shipmode` cuyo costo es despreciable.

El tiempo que se tardó el manejador en producir los resultados fue de 15858,726 ms siendo el *Sequential scan* sobre `lineitem` y el *Index scan* sobre `orders` las operaciones más costosas con 11911,128 ms y 3622,825 ms de duración respectivamente.

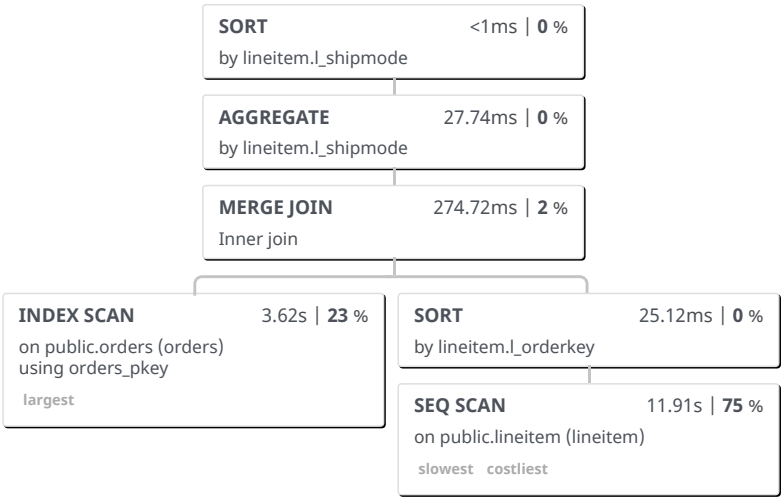

Figura 10: Arbol de ejecucion de la consulta Q21

Primera iteración:

Dado que los *bottle necks* , o cuellos de botella, de la ejecución de la consulta fueron la obtención de las tuplas correspondientes se enfocó en la primera iteración en la obtención eficiente de los atributos deseados de cada tabla evitando el acceso, si es posible, al archivo que contiene las tablas.

Para la tabla `lineitem` se propone la utilización de un índice compuesto que contenga los siguientes atributos: `l_receiptdate`, `l_commitdate`, `l_shipdate`, `l_shipmode` y `l_orderkey` en el orden especificado.

El tamaño de una tupla con todos estos atributos es, aproximadamente, 21 bytes. Esta información se puede constatar en la Tabla 3. Entonces, el número de páginas estimado del índice sería $98544 \times (21/60) = 34491$.

Para la tabla `orders` se propuso un índice con los siguientes atributos: `o_orderkey` y `o_orderpriority` en ese orden. Este índice permite obtener solo los atributos que se requieren para la consulta sin tener que acceder al archivo principal de la tabla con menos operaciones de entrada salida al solo tener que realizar $25196 \times (13/59) = 5552$ en comparación con 25196 que tendría que hacer sin soporte alguno.

Luego de la de este análisis se procedió a crear los índices:

```
--Elementos para orders
create index q21_orders_idx ON orders(o_orderkey, o_orderpriority);

--Elementos para lineitem
create index q21_lineitem_idx ON lineitem(l_receiptdate, l_commitdate,
l_shipdate, l_shipmode, l_orderkey);
```

Luego se consultó el número de páginas que tenía cada índice con la consulta:

```
select relname,relpages
    from pg_class
    where relname in ('q21_orders_idx','q21_lineitem_idx');
```

obteniéndose lo mostrado en la Tabla 13.

Tabla 13: Cantidad de páginas de los índices propuestos

Índice	Páginas
q21_orders_idx	6108
q21_lineitem_idx	29755

A parte, se reescribió la consulta de la siguiente manera equivalente:

```
prepare q21 as
select l_shipmode,
    count(o_orderpriority) filter(where o_orderpriority in
        ('1-URGENT','2-HIGH') )as high_line_count,
    count(o_orderpriority) filter(where o_orderpriority not in
        ('1-URGENT','2-HIGH') )as low_line_count
from orders,lineitem
where o_orderkey = l_orderkey and l_shipmode in ($1, $2)
    and l_commitdate < l_receiptdate and l_shipdate < l_commitdate
    and l_receiptdate >= $3 and l_receiptdate < $3 + interval '1 year'
group by l_shipmode
order by l_shipmode;
```

Finalmente se procedio a ejecutar la consulta Q21 mejorada para observar que ocurría. El arbol de ejecución con la duración de cada nodo se puede observar en la la Figura 11.

En ella se puede observar que la operación de *Index only scan* sobre la tabla `lineitem`, que paso de durar 11911,128 ms en la consulta original a 348,811 ms.

En la fase de *joins* se utiliza un *Index nested loop join* entre los resultados obtenidos del filtrado de `lineitem` con la tabla `orders` con la condición `o_orderkey = l_orderkey`. El índice utilizado para el join fue el propuesto, en el arbol de ejecución de la Figura 11 se puede observar un *Index only scan* sobre `orders`. Sin embargo, este recorrido secuencial se realiza a la misma vez que se realiza el join.

El resto del plan de ejecución se mantiene intacto.

El tiempo de ejecución de la consulta paso de 15858,726 ms a 3124,659 ms, lo que significa una mejora de 80,3 % con respecto a la consulta original. Cualitativamente, esta mejora se puede considerar **muy buena**.

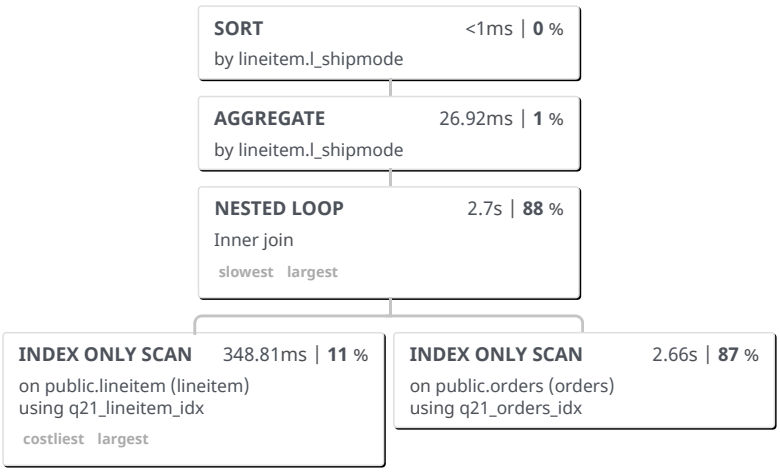

Figura 11: Arbol de ejecucion de la consulta Q21 optimizada

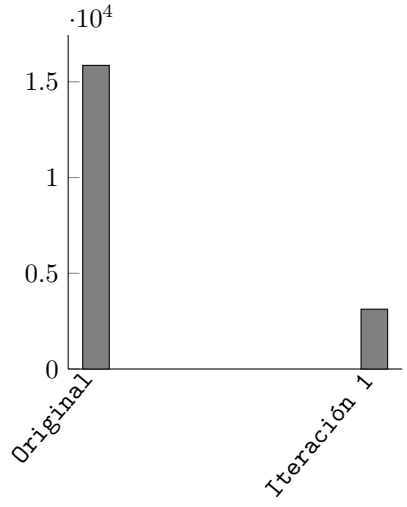

Figura 12: Comparación de tiempos en cada iteración

Adicionalmente, se realizaron pruebas con esta consulta aumentanto el tamaño de la variable `work_mem` per no hubo mejoría notable en la misma. En la Figura 12 se puede apreciar la mejora en tiempo de ejecución de la consulta

Consulta Q22: Relación parte/proveedor

La consulta a optimizar es la siguiente:

```
select
  p_brand,
  p_type,
  p_size,
  count(distinct ps_suppkey) as supplier_cnt
from
  partsupp,
  part
where
  p_partkey = ps_partkey
  and p_brand <> $1
  and p_type not like $2
  and p_size in ($3, $4, $5, $6, $7, $8, $9, $10)
  and ps_suppkey not in (
    select
      s_suppkey
    from
      supplier
    where
      s_comment like '%Customer%Complaints%'
  )
group by
  p_brand,
  p_type,
  p_size
order by
  supplier_cnt desc,
  p_brand,
  p_type,
  p_size;
```

Al hacer `Explain Analyze` sobre la consulta sin ningúna estructura adicional se obtuvo el plan de ejecución mostrado en la Figura 13, con una duración total de 4392 ms. El nodo más lento en este grafo es el *Sort* de la tabla part por la clave (p_brand, p_type, p_size) y representa el 67 % del tiempo de ejución total de la consulta. Otros ineficiencias son las condiciones de desigualdad en el `where`, los cuales son más difíciles de mejorar por el optimizador.

Se reescribió la consulta para eliminar la mayor cantidad de desigualdades posibles. Se logró eliminar el operador `not in` de la consulta, cambiando el operador de condición de la subconsulta de `like` a `not like`.

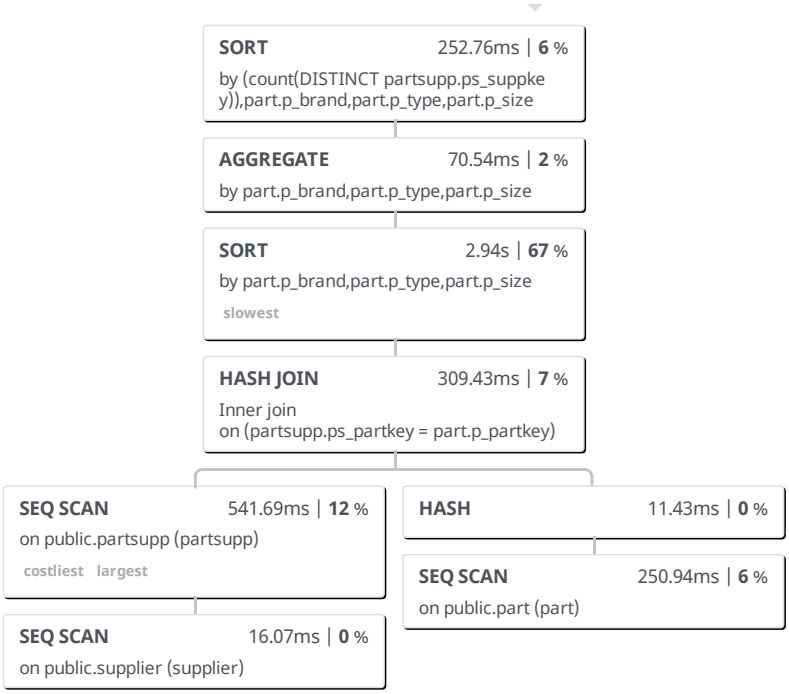

Figura 13: Arbol de ejecución de la consulta Q22 original

```
select
  s_suppkey
from
  supplier
where
  s_comment not like '%Customer%Complaints%';
```

Este resulta en una mejora significativa ya que la tabla `supplier` es mucho más pequeña que `partsupp`, como se ve en la Tabla 14, y no es afectada tanto por el operador `not like` como `partsupp` es afectada por el `not in`.

Tabla 14: Tamaños de las tablas relevantes.

Tabla	Tuplas
`supplier`	10000
`partsupp`	800000

La consulta reescrita es la siguiente:

```
select
  p_brand,
  p_type,
  p_size,
  count(distinct ps_suppkey) as supplier_cnt
from
  partsupp,
  part
where
  p_partkey = ps_partkey
  and p_brand <> $1
  and p_type not like $2
  and p_size in ($3, $4, $5, $6, $7, $8, $9, $10)
  and ps_suppkey not in (
    select
      s_suppkey
    from
      supplier
    where
      s_comment like '%Customer%Complaints%'
  )
group by
  p_brand,
  p_type,
  p_size
order by
  supplier_cnt desc,
  p_brand,
  p_type,
  p_size;
```

Además, aprovechando el índice sobre part creado para la consulta q11

```
create index idx_q11_p_size_reverse_type
  on part (p_size, reverse(p_type) text_pattern_ops);
```

La reescritura permite a la consulta utilizar los índices disponibles, y resulta en que el nodo que hacía un *Sequential Scan* sobre `part` cambie a un *Bitmap Index Scan*, lo cual reduce el tiempo de ejecución de 251 ms a 124 ms (incluyendo los accesos al *heap*), y que el nodo de selección de `partsupp` pase de un *Sequential Scan* a un *Index Only Scan*, lo cual reduce el tiempo de ejecución de este nodo de 542 ms a 118 ms. Se añade un tiempo adicional por un

nuevo nodo de *Nested Loop Join* que se ejecuta en 173 ms, sin embargo el tiempo total sigue siendo menor que el de la consulta original.

El plan de ejecución resultante puede verse en la Figura 14 y muestra que el tiempo de ejecución disminuyó de 4392 ms a 3618 ms, una reducción del 18 % del tiempo de la consulta original.

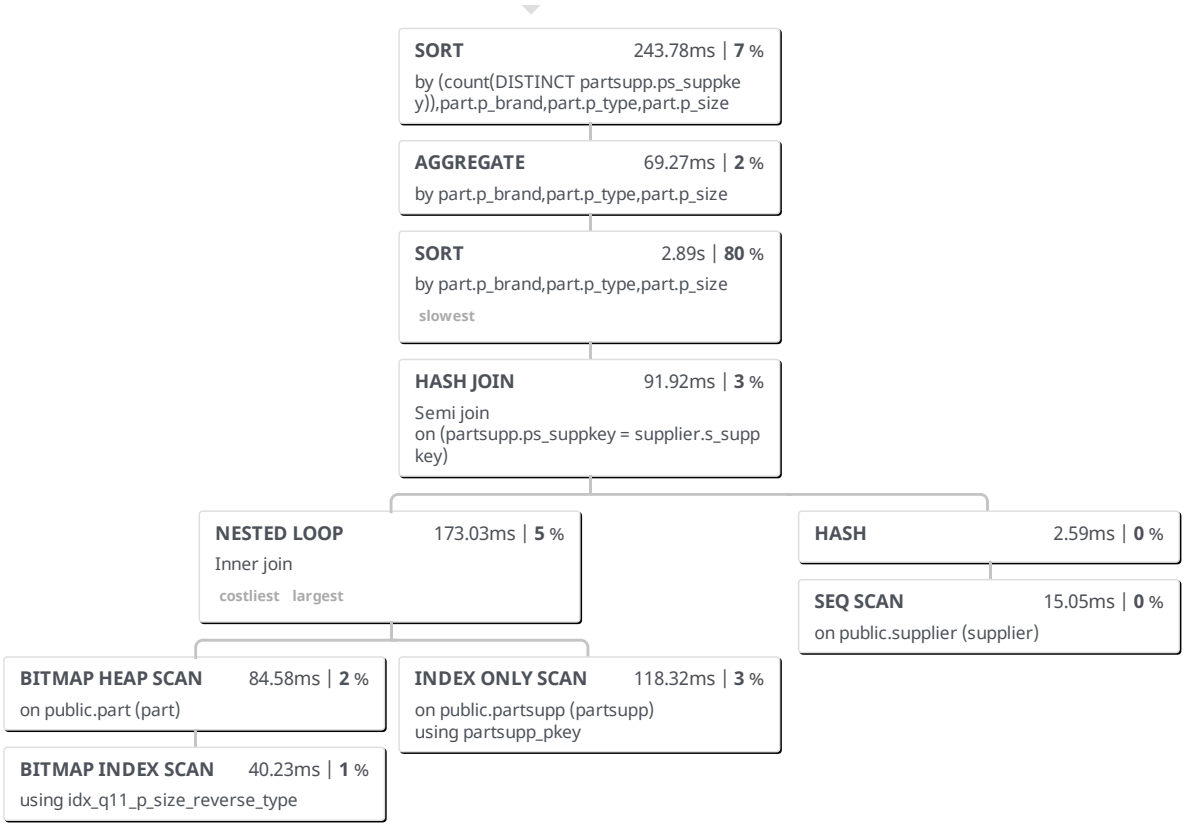

Figura 14: Árbol de ejecución de la consulta Q22 reescrita

Por último, se puede ver que el nodo de Sort de `part` utiliza un *External Merge Sort*, lo cual implica accesos a disco adicionales. Esto se debe a que la memoria de trabajo de la consulta (que por defecto es 4MB) es insuficiente, el sort necesita aproximadamente unos 10MB de memoria para poder hacer un ordernamiento interno.

Se colocó la memoria de trabajo para mejorar esto:

```
SET work_mem='32MB';
```

Y el nodo de sort cambia de *External Merge Sort* a un *quicksort*, ejecutada sobre los datos cargados en memoria. Se logró reducir el costo de la consulta de 3618 ms a 2700 ms. Una reducción del 25 % del tiempo de ejecución de la consulta mejorada. Este cambio fue probado en la máquina de AWS que está equipada con un SSD donde la diferencia de velocidad entre la memoria y el disco es menor que la de un HDD. Se probó en una máquina con HDD y la reducción fue mucho mayor (un 65 % de mejora sobre el *external sort*).

El tiempo de ejecución (con caché fría) de esta consulta baja de 4392 ms a 2700 ms, el 65 % de la consulta original. El plan resultante es el mostrado en la Figura 15.

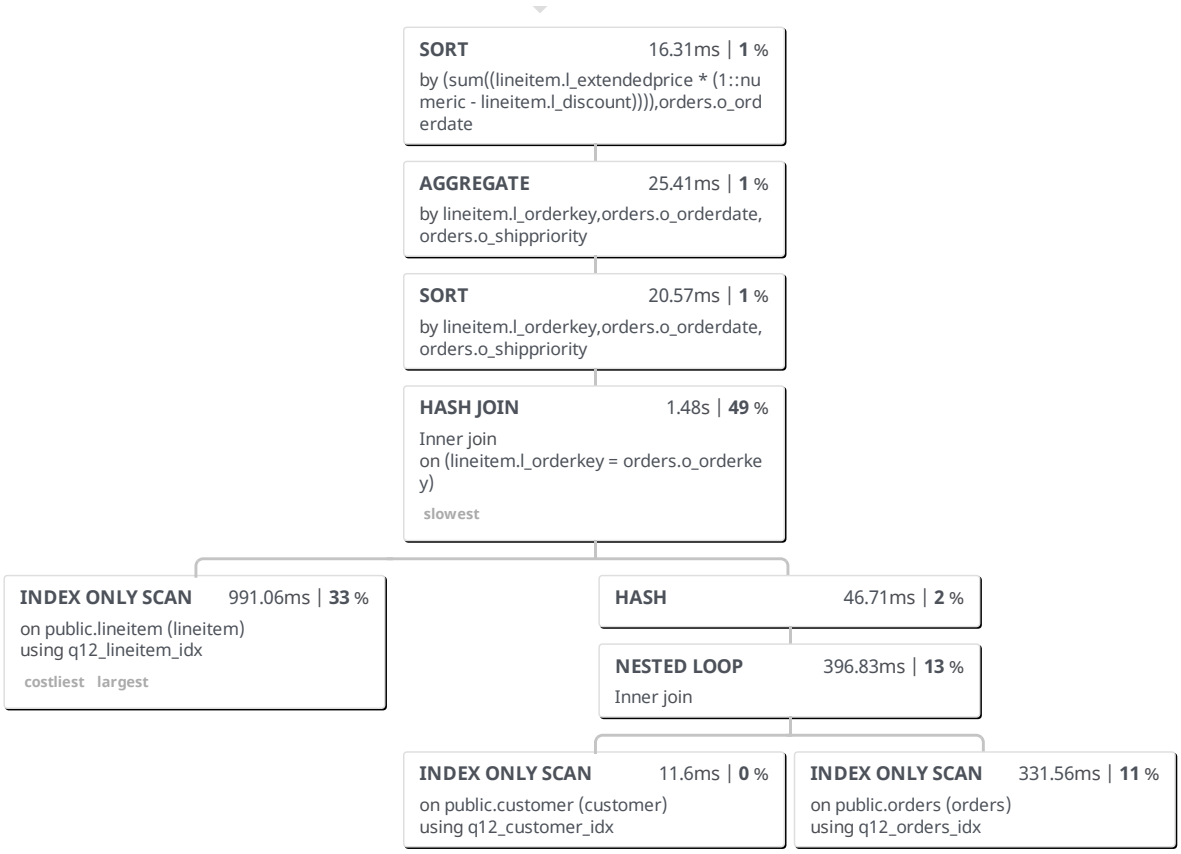

Figura 15: Árbol de ejecución de la consulta Q22 modificada

Consulta Q23: Oportunidad de ventas globales

La consulta a optimizar es la siguiente:

```
select
  cntrycode,
  count(*) as numcust,
  sum(c_acctbal) as totacctbal
from (
  select
    substring(c_phone from 1 for 2) as cntrycode,
    c_acctbal
  from
    customer
  where
    substring(c_phone from 1 for 2) in
      ($1, $2, $3, $4, $5, $6, $7)
    and c_acctbal > (
      select
        avg(c_acctbal)
      from
        customer
      where
        c_acctbal > 0.00
        and substring(c_phone from 1 for 2) in
          ($1, $2, $3, $4, $5, $6, $7)
    )
    and not exists (
      select
        *
      from
        orders
      where
        o_custkey = c_custkey
    )
  ) as custsale
group by
  cntrycode
order by
  cntrycode;
```

Al hacer `Explain Analyze` sobre la consulta sin ningúna estructura adicional se obtuvo el plan de ejecución mostrado en la Figura 16, con una duración total de 3018 ms. Puede verse que el nodo más caro y lento en este grafo fue el *Seq Scan* (2167 ms) que se hace sobre `orders`

como parte del *Hash Anti Join* con la tabla `customer` sobre la igualdad de los atributos `c_custkey` y `o_custkey`, cuyos nombres indican que se trata de un join sobre una clave foránea entre la entidad `order` y su `customer` correspondiente. Como se trata de un *Anti Join*, sólo sobreviven las filas de `customer` que ***no*** tienen filas correspondientes en `orders`, pero el algoritmo para calcular esta operación es muy similar al usual del *Hash Join*.

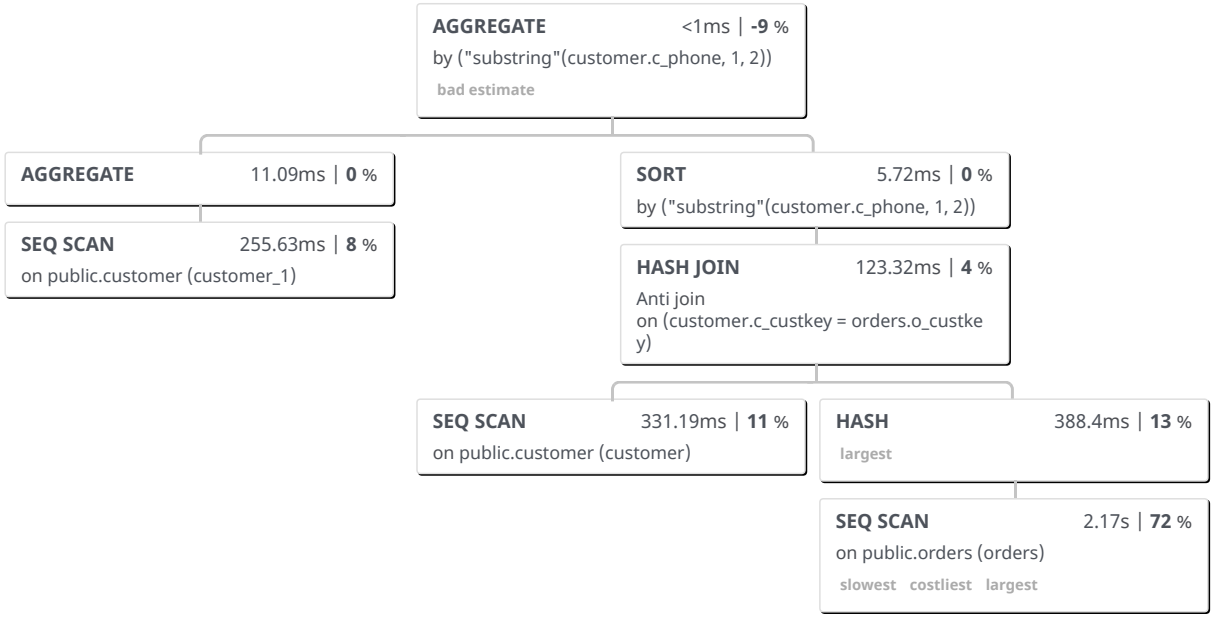

Figura 16: Arbol de ejecucion de la consulta Q23 original

Cabe mencionar que la consulta fue reescrita para facilitar su interpretación, sin que esto cambiara su semántica ni su eficiencia. Esta reescritura produjo la siguiente consulta:

```
select
  substring(c_phone from 1 for 2) as cntrycode,
  count(1)                        as numcust,
  sum(c_acctbal)                  as totacctbal
from
  customer
where substring(c_phone from 1 for 2) in
    ($1, $2, $3, $4, $5, $6, $7)
  and c_acctbal > (
    select
      avg(c_acctbal)
    from
      customer
    where substring(c_phone from 1 for 2) in
        ($1, $2, $3, $4, $5, $6, $7)
      and c_acctbal > 0.00)
```

```
  and not exists (
    select 1
    from
      orders
    where
      o_custkey = c_custkey
  )
group by
  cntrycode
order by
  cntrycode;
```

Básicamente, se simplificó la estructura de la consulta, dejando la consulta interna con las selecciones, agrupaciones y ordenamientos de la externa. Cabe mencionar que, a pesar de que la parte

```
and not exists (
  select 1
from
  orders
where
  o_custkey = c_custkey
)
```

de la consulta podría sugerir ineficiencia, pues tiene la estructura de una consulta correlacionada, este no es el caso, y de hecho reescribir esto como una *Common Table Expression* resultaría en una consulta menos eficiente, pues el *Anti Join* se convertiría en una costosa diferencia de conjuntos.

Posterior a esto, se crearon dos índices para mejorar el rendimiento de esta consulta. El primero, creado sobre la tabla `customer`, sobre los primeros 2 dígitos del número de teléfono del cliente y sobre su balance de cuentas, permite al manejador conseguir primero los clientes con un prefijo telefónico específico y luego los que tengan un balance de cuentas en un rango particular. Los atributos del índice están en este órden porque invertirlos no permitiría hacer búsquedas por igualdad en los primeros 2 dígitos del número de teléfono, que es lo que se desea en esta consulta. Este índice fue creado con el siguiente comando:

```
create index idx_q23_c_ctrycode_acctbal
  on customer (substring(c_phone from 1 for 2), c_acctbal);
```

El segundo índice que agregamos para esta consulta es sobre la tabla `orders`, en el atributo `o_custkey`. Parece un índice muy sencillo y su impacto no necesariamente es obvio, pero permite conseguir rápidamente las órdenes correspondientes a un cliente en particular. En el caso de esta consulta, interesa conocer los clientes *sin* órdenes, para lo cual este índice resulta

aún más conveniente: basta una búsqueda *Index Only* sobre este índice para saber cuáles son los clientes que no están representados en esta tabla, y se hace innecesario recuperar información directamente de la tabla `orders`. Este índice fue creado con el siguiente comando:

```
create index idx_q23_o_custkey
  on orders (o_custkey);
```

Luego de la creación del par de índices, el tiempo de ejecución (con caché fría) de esta consulta baja de 3018 ms a 379 ms, tan sólo el 12.5 % del tiempo de la consulta original. El plan resultante es el mostrado en la Figura 17.

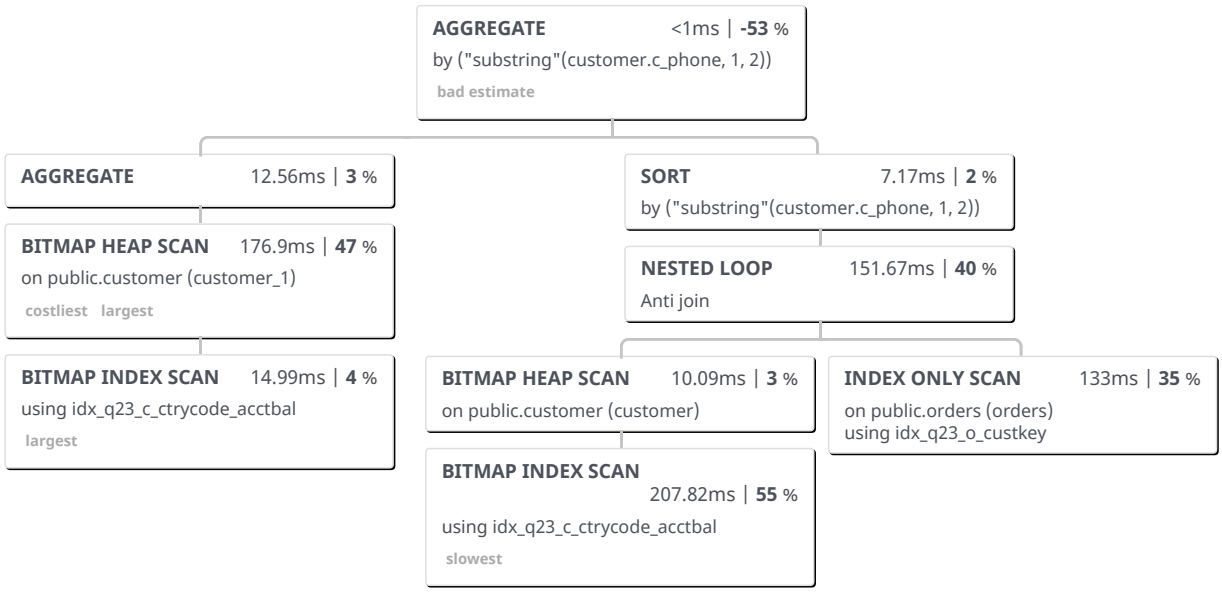

Figura 17: Arbol de ejecucion de la consulta Q23 modificada (a)

Conclusiones

El proyecto sirvió para reforzar y poner en práctica lo aprendido en clase sobre el diseño físico y la optimización de consultas. Se experimentaron y utilizaron técnicas de optimización de consultas como la creación de índices compuestos, reescritura de consultas, clustering, particionamiento y gestión de memoria de trabajo y *shared buffers*.

Se empezó observando el esquema lógico de la base de datos y analizando los planes generados por las consultas originales. Se calcularon estadísticas que informaron sobre el tamaño y características de los datos. Para cada tabla se calculó el número de valores distintos, la correlación, la cota superior y la selectividad de cada uno de sus atributos. Estos datos fueron particularmente útiles para determinar la manera en que se debían definir los índices, reescribir las consultas y particionar las tablas.

Cada integrante del equipo tomó una consulta de la primera parte y una de la segunda, y se analizaron todas las 6 consultas del proyecto. En general, todas consultas mostraron una mejora significativa luego del proceso de optimización. A continuación se muestran los porcentajes de mejora de cada una de las consultas:

Consulta	Tiempo original (ms)	Tiempo optimizado (ms)	Porcentaje de mejora
Q11	1997	414	79.26 %
Q12	17932	2986	83.34 %
Q13	17875	2904	83.75 %
Q21	15859	3125	80.30 %
Q22	4393	2700	38.53 %
Q23	3019	379	87.45 %

Puede verse lo mucho que mejoraron las consultas, algunas mejorando hasta más del 80 % sobre la consulta original. La mejora principal buscada en todas las consultas es crear índices para eliminar los *sequential scans* y sustituirlos por *index only scans*, los cuales son mucho más eficientes ya que solo consulta el índice y minimizan el número de I/Os necesarios, la causa principal del aumento de los tiempos de ejecución de las consultas. Al lograr que el plan de trabajo generera *index only scans*, se redujeron los tamaños de las selecciones y el optimizador seleccionó mejores algoritmos para realizar las operaciones `join`, y los únicos nodos que afectaban un poco los tiempos de ejecución fueron los de ordenamiento donde se utilizaron técnicas como el aumento de la memoria de trabajo para disminuir su impacto en el rendimiento de la consulta. Estos resultados muestran la importancia de la optimización de consultas entre las tareas esenciales para los administradores de bases de datos.