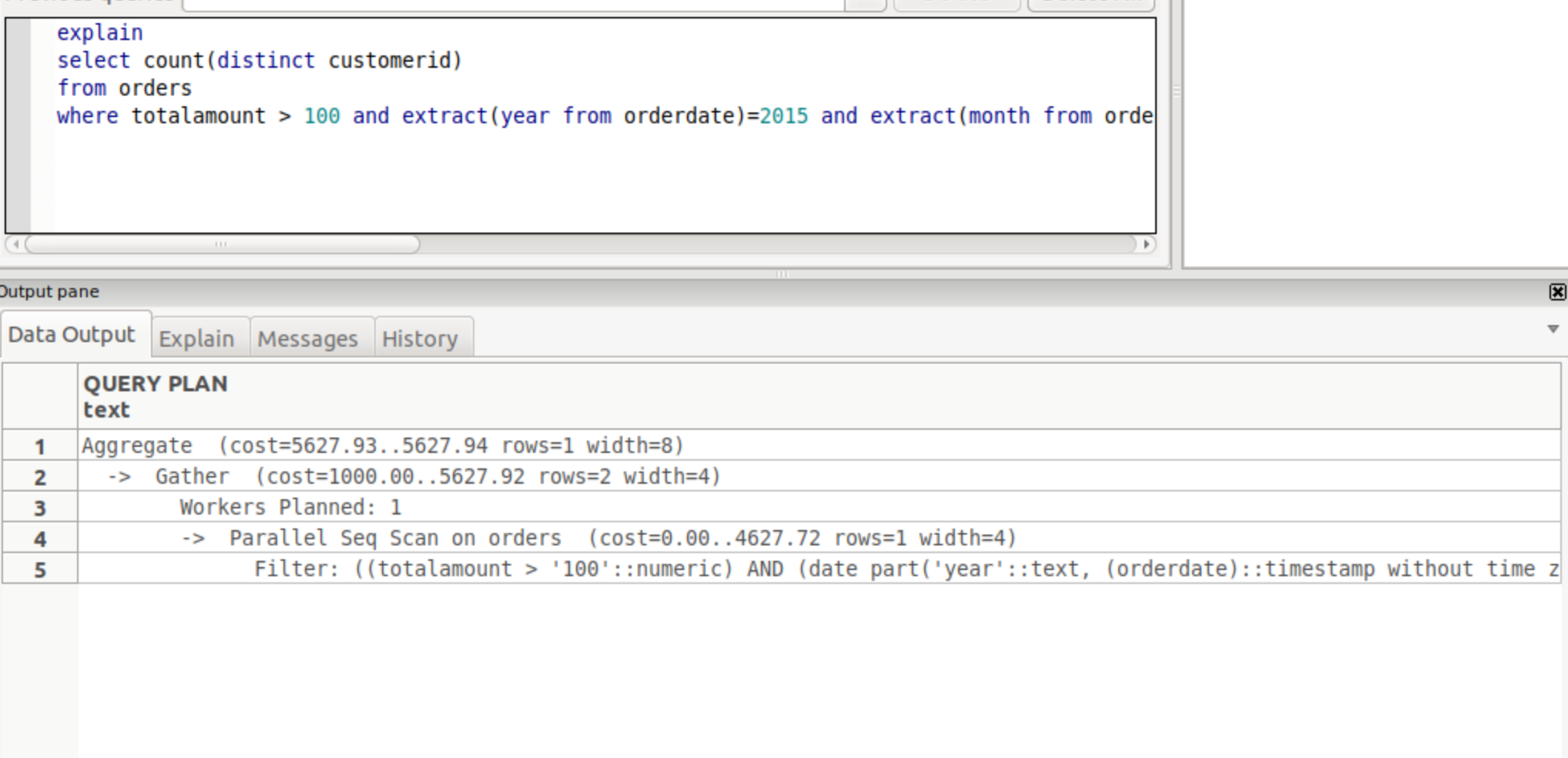
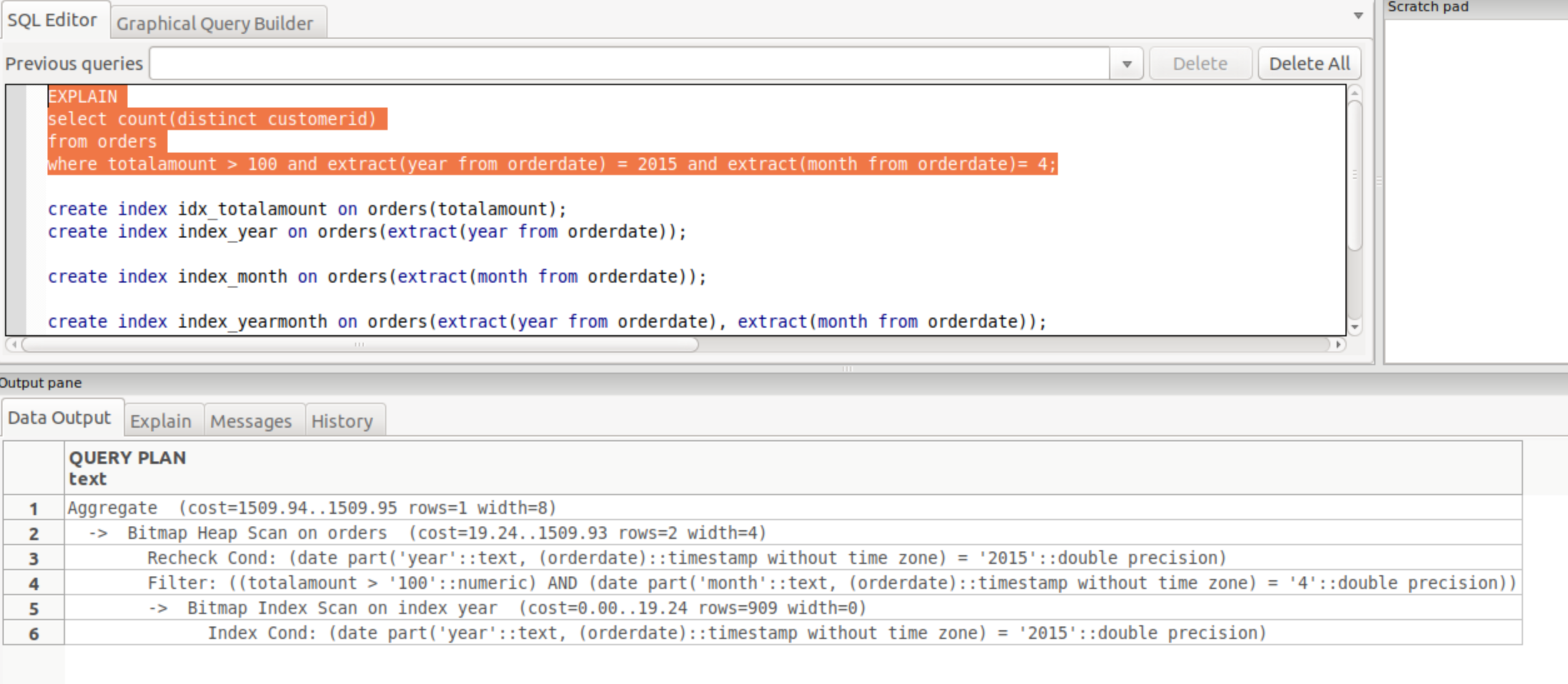
OPTIMIZACION

Se observa que la acción más costosa es el ‘seq scan’ pues este realiza un escaneo secuencial de la tabla de datos almacenada desde la primera fila hasta que la consulta se cumple.

Para mejorar el rendimiento creamos índices sobre esta tabla. Nuestra primera aproximación fue pensar en crear un índice para la selección de pedidos por el coste total, pero dado que más del 50% de los pedidos de la base de datos tenían un coste mayor de 100, el rendimiento no mejoraba mucho pues no descartábamos un gran número de filas, y con la creación de los siguientes índices (mucho más útiles) su mejora de rendimiento era imperceptible. Es por ello que lo descartamos.

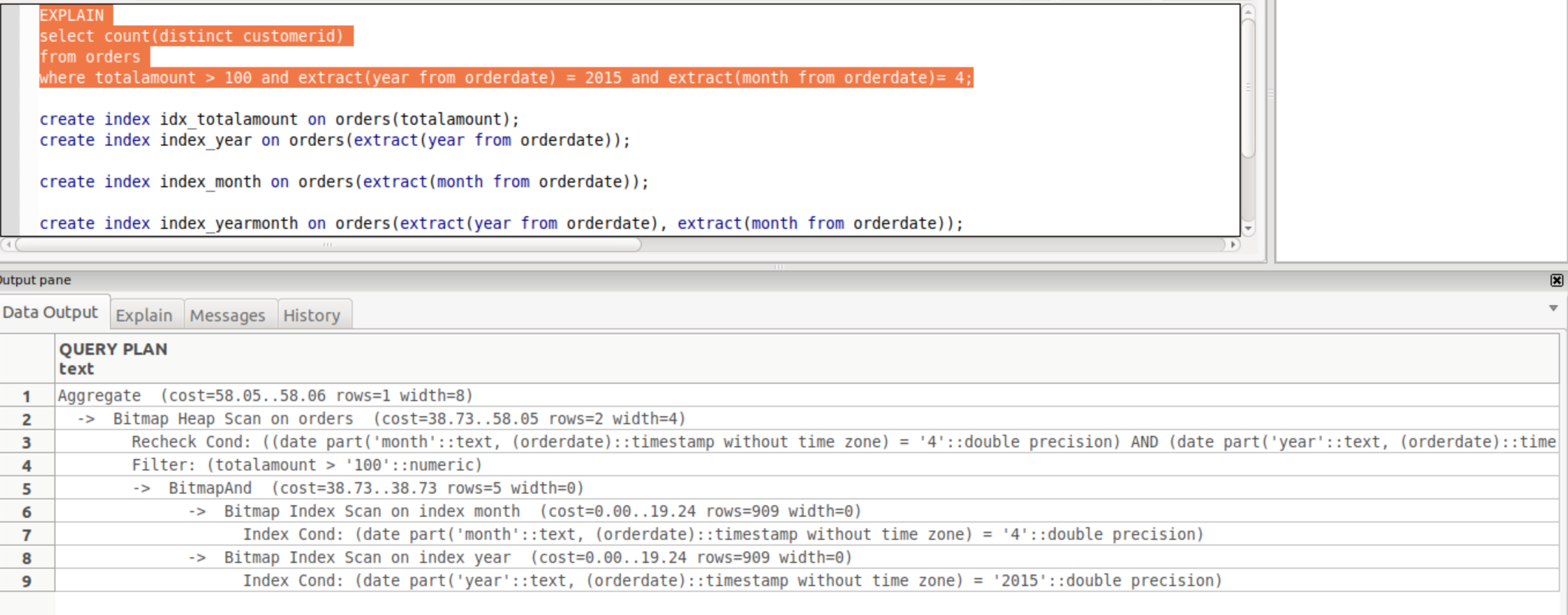
Lo siguiente que pensamos fue crear un índice para la operación de extracción del año en cuestión, esto es:

*create index index\_year on orders(extract(year from orderdate));*

**

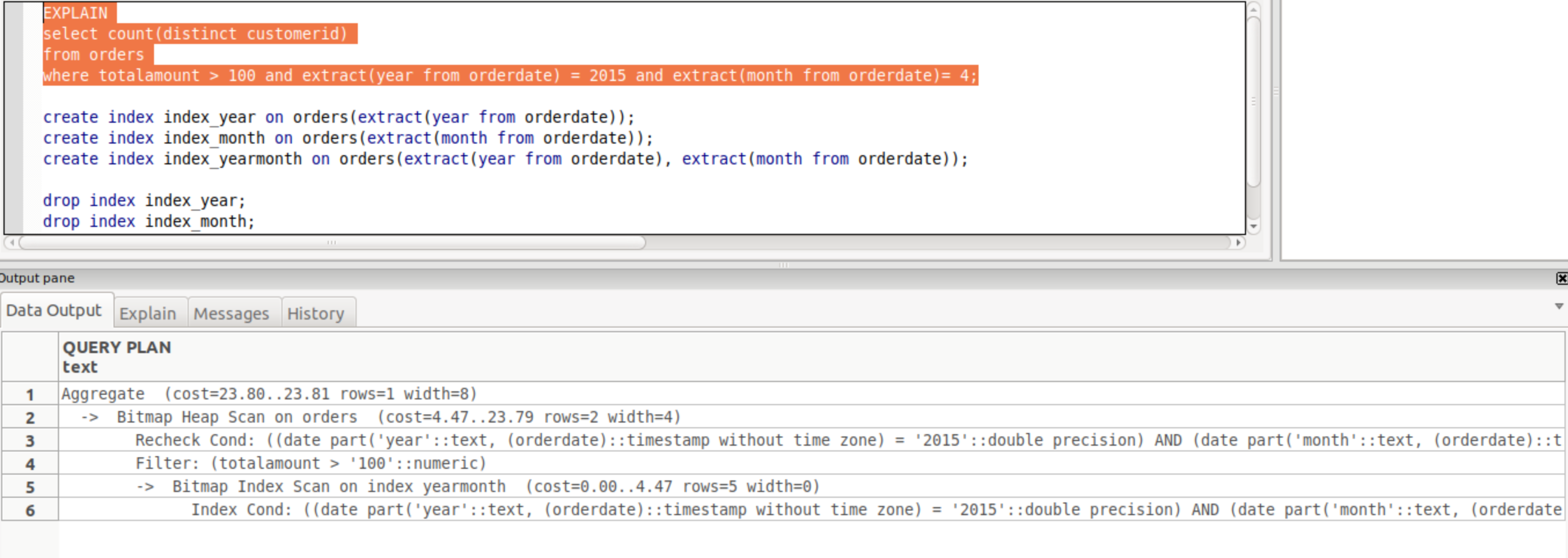
Al ejecutar de nuevo la query vimos una notable mejora en el rendimiento, pues el coste se reducía de 5627 a 1509. El cambio se observa en el cambio de ‘seq scan’ que se realiza en orders por un ‘bitmap heap scan’. Este se divide en dos pasos: la búsqueda de las filas que cumplan la condición en el índice creado y luego ‘traerlas’ del disco para mostrarlo. Traer las filas de una en una es más costoso que leerlas de manera secuencial, el motivo de que esto sea menos costoso es porque gracias al índice no tendremos que leer toda la tabla, pues el subconjunto de filas que tenemos que buscar es notablemente menor que el número total (aproximadamente un 12%).

Viendo esto decidimos probar a añadir otro índice en este caso para la extracción del mes, es decir:

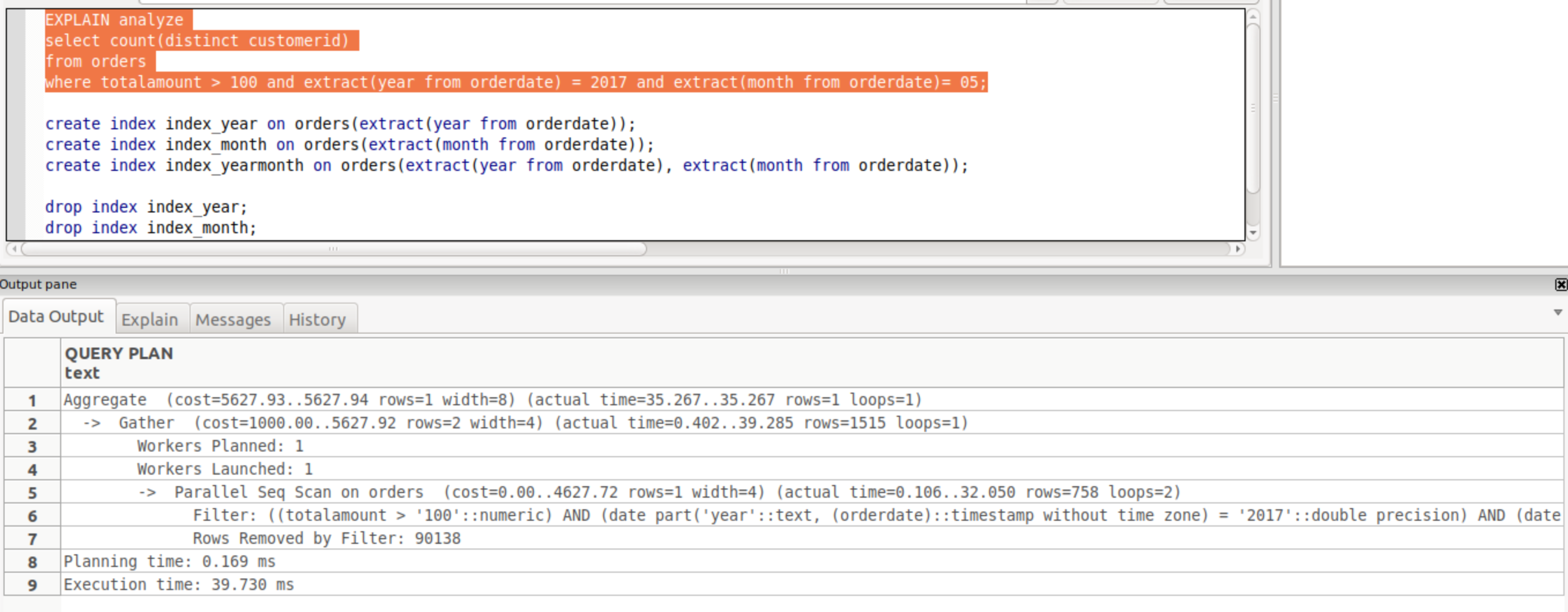
*create index index\_month* *on orders(extract(year from orderdate));*

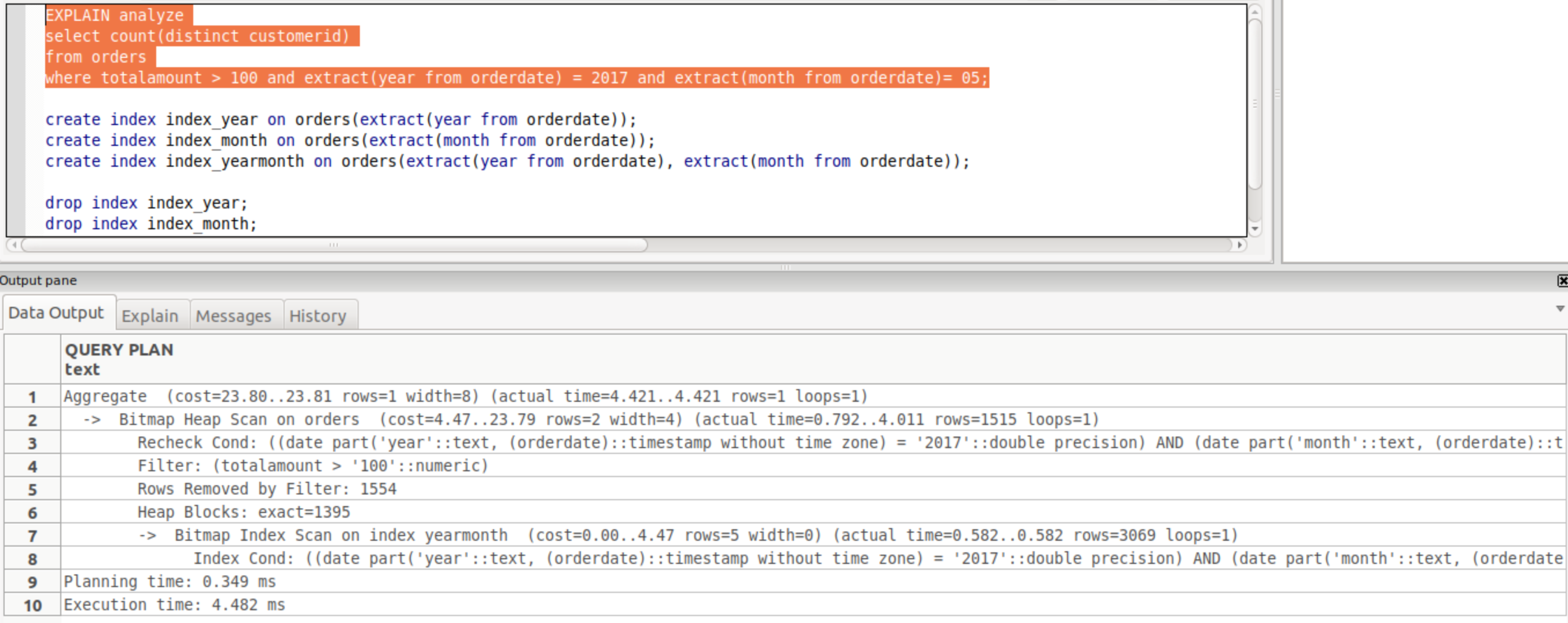
Observamos que el rendimiento de la consulta mejora de nuevo, pasando de un coste de 1509 a un coste total de 58. El razonamiento es el mismo que el anterior.

Viendo esto se nos ocurrió crear un índice más específico uniendo las dos situaciones anteriores en lugar de tenerlos por separados, es decir, un índice para la extracción de la fecha completa. De este modo nos lo imaginamos como un índice de un libro donde cada capítulo es una fecha, y en ellos están los pedidos que se realizaran ese día. Es claro que el número de filas que tendremos que ‘traer’ será menor, por lo que debería mejorar el rendimiento.

Así obtuvimos nuestro mejor resultado de rendimiento, pasando de 5627 a 23.80:

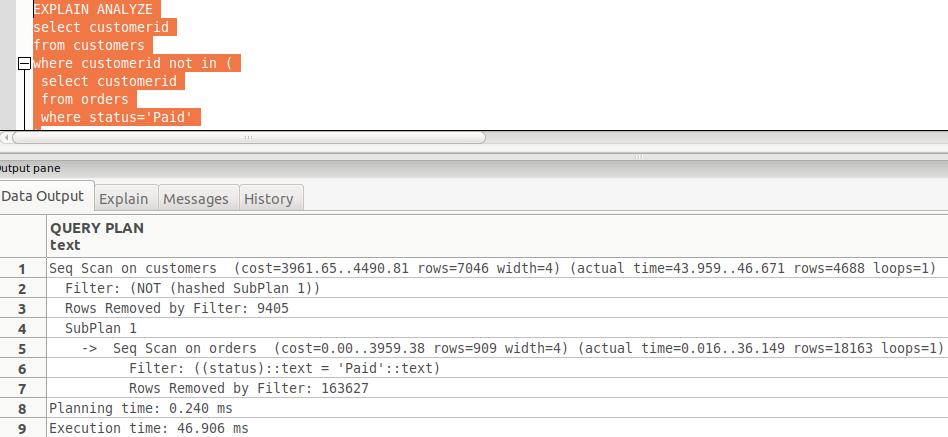
Haciendo uso de **EXPLAIN ANALYZE** podemos comprobar también una mejora en el tiempo de ejecución (a costa de tardar un poco más en tiempo de planificación pues ‘bitmap heap scan’ consta de dos pasos en su planificación) entre ejecutar la consulta sin índices, y con el último índice discutido (el de fecha completa):

**Sin índices:**

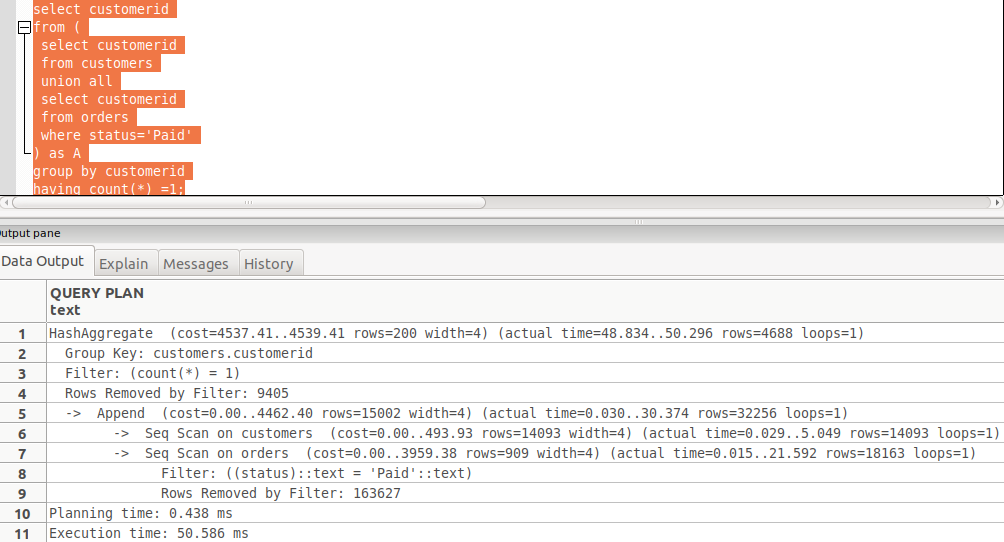
****

**Con índice:**

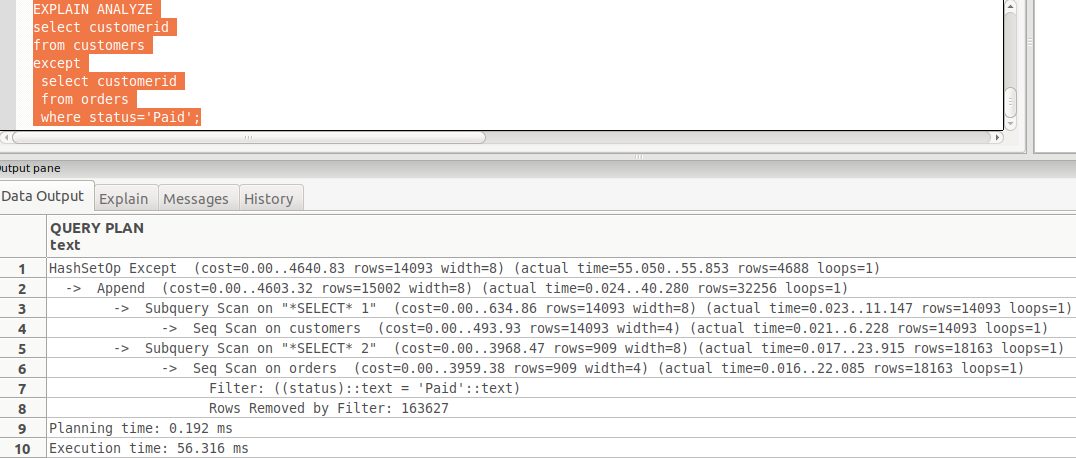
F) Vamos a estudiar la forma de las consultas del Anexo 1.



En la primera consulta obeservamos que se realizan dos *seq scans*, en primer lugar uno de la tabla *orders* y posteriormente uno de la tabla *costumers* con la restricción de *status=“paid“*.



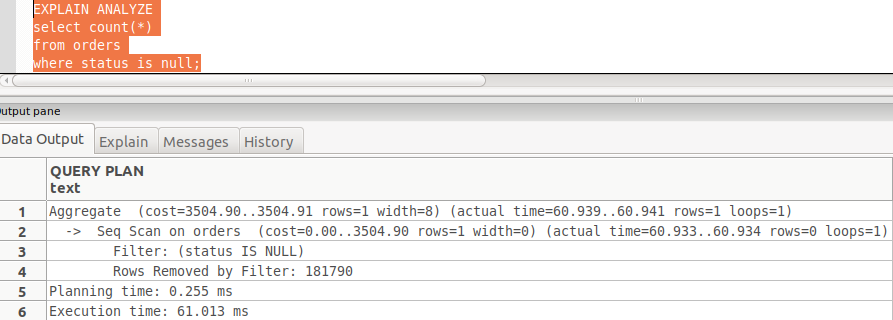
En la segunda consulta se realiza una operación de *HashAggregate* y posteriormente se realizan en paralelo dos *seq* *scan*, donde se recorren secuencialmente las tablas de *orders* y *costumers*. Por este motivo esta query se beneficia de la ejecución en paralelo.



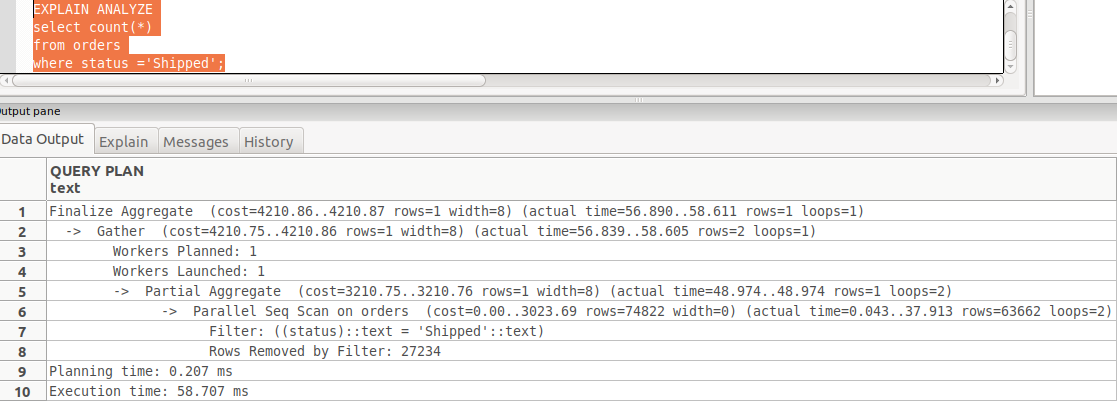
En la tercera consulta se realiza una operacion de *HashSetOp* cuyo coste de ejecución a diferencia de las dos anteriores es (cost=0.00..4640.83 rows=14093 width=8) por lo que mostrará resultados nada más comenzar la ejecución. Además, esta consulta realiza un *append*, que tiene dos *subquerys* que, al igual que la anterior consulta, se benefician de la ejecución en paralelo.

G) Estudiamos el coste de ejecución de las dos consultas del anexo 2 con la sentencia EXPLAIN. Para este primer analisis no usamos ningún índice y obtenemos los siguientes resultados:

* **Consulta 1**

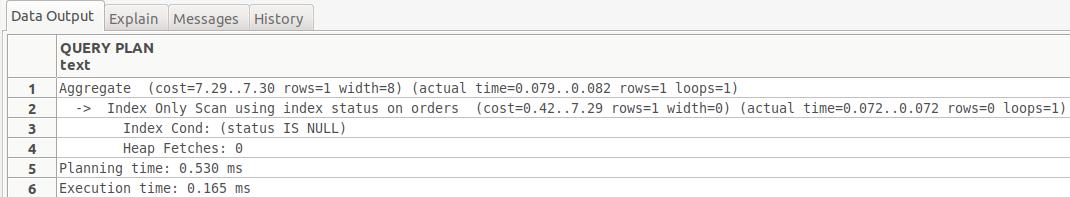
****

* **Consulta 2**

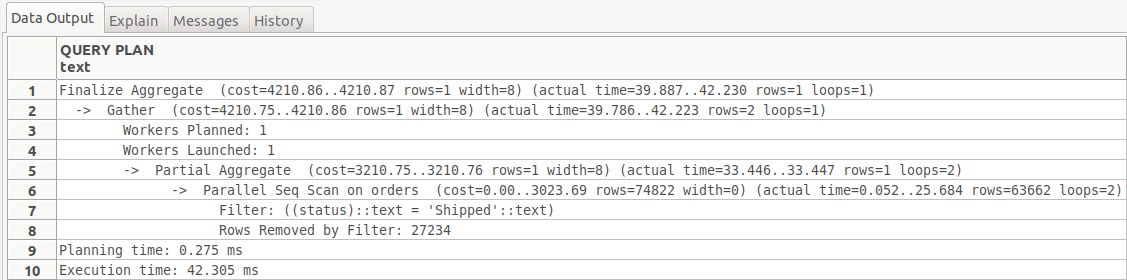
****

A continuación creamos un índice para la tabla orders y la columna status y volvemos a analizar el coste de ejecución de ambas consultas. Observamos que el coste de ejecución mejora considerablemente. Tambien observamos que el tiempo de planificación se incrementa levemente. En cambio, el tiempo de ejecución se reduce enormemente gracias a la introducción del índice. La razón es exactamente la misma que hemos explicado anteriormente en el apartado E.

* **Consulta 1**

****

* **Consulta 2**

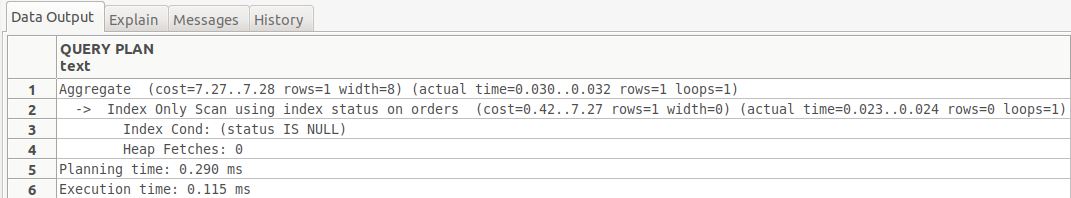
****

A continuación ejecutamos la sentencia ANALYZE sobre la tabla orders y volvemos a analizar los costes, el tiempo y el plan de ejecución de ambas consultas.

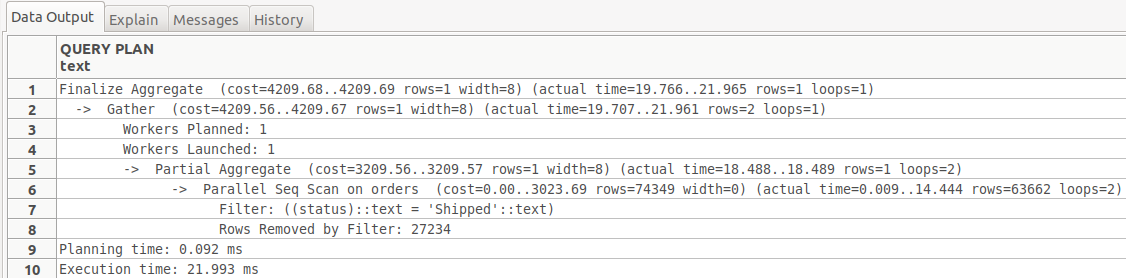
En el caso de la consulta 1, observamos que la planificación de la consulta cambia. Se sustituye el *seq scan* de la tabla orders por una búsqueda por índice. Por este motivo el tiempo de ejecución, el tiempo y el coste mejora considerablemente.

Sin embargo, en la consulta 2 la mejora de tiempo no es tan notable, aunque sigue siendo considerable. Observamos que la planificación no cambia, y la diferencia del coste es insignificante.

* **Consulta 1**

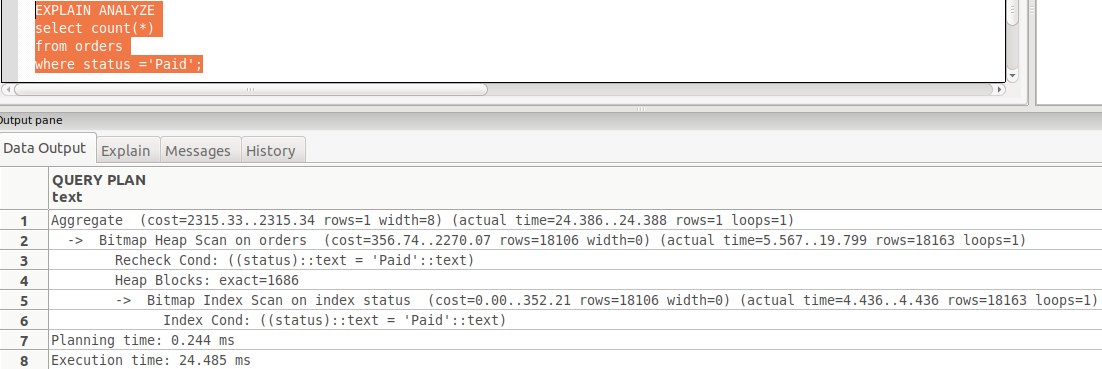
****

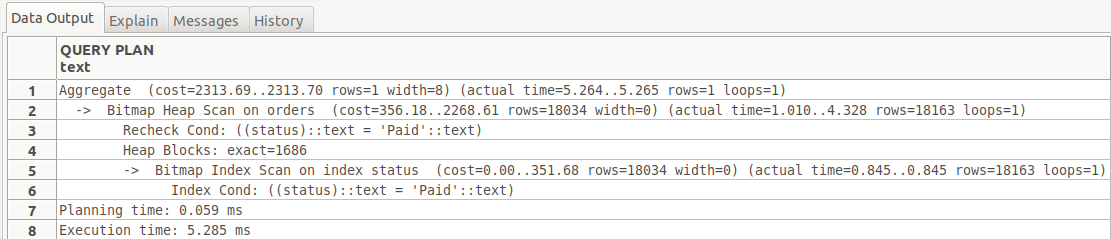
* **Consulta 2**

****

A continuación analizamos las otras dos consultas del anexo 2.

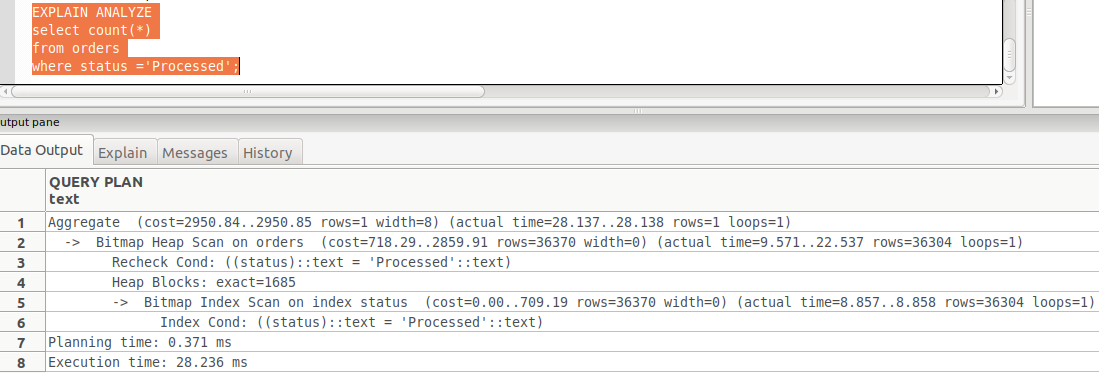
* **Consulta 3**

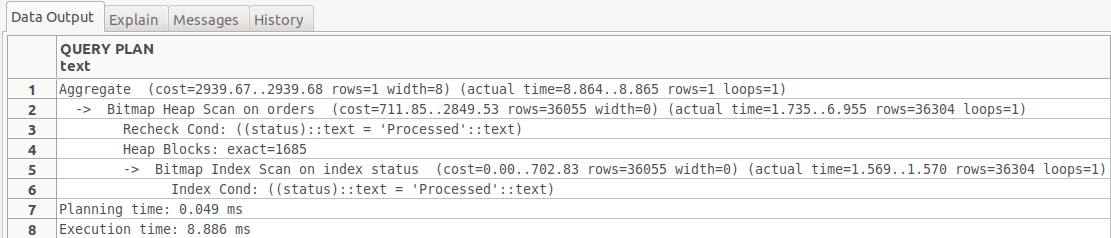
****

****

Ejecutamos la consulta 3 con y sin la ejecución de la sentencia ANALYZE. Observamos que con la sentencia, la planificación permanece idéntica, el coste de ejecución se mantiene y disminuye considerablemente el tiempo de ejecución al igual que en las otras sentencias.

* **Consulta 4**

****

****

Ejecutamos la consulta 4 igual que con la 3. La planificación se mantiene idéntica una vez más, y el coste prácticamente invariable. El tiempo de ejecución en cambio vuelve a bajar considerablemente gracias a la sentencia ANALYZE.

**TRANSACCIONES**

APARTADO H

Para el apartado H usando el esqueleto suministrado, hemos completado la función correspondiente a *borraCliente* de *database*.*py* para cumplir todos los requisitos de las transacciones de este apartado. Además cambiamos *borraCliente*.*html* para que el form sea de tipo POST.

Empezamos realizando la sentencia *BEGIN* para comenzar la transacción y la dividimos en tres etapas:

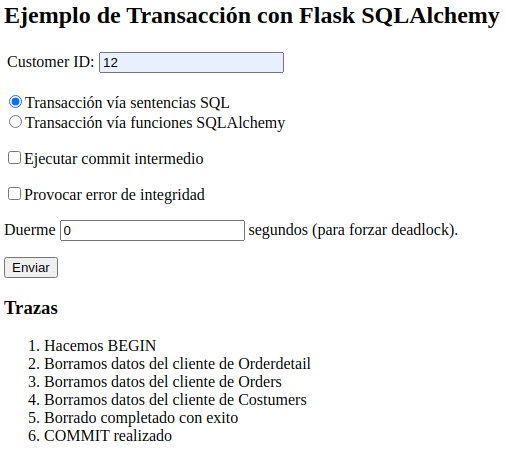
En primer lugar se borran los datos de la tabla *orderdetail* de ese cliente (query1). A continuación, se borran los datos de la tabla *orders* para ese cliente (query2). Por último borramos al cliente de la tabla *customers* (query3).

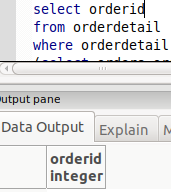
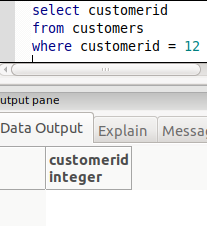
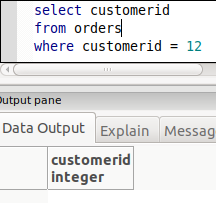
Si no ha habido ningún error, se realiza la sentencia *COMMIT* para que se agreguen los nuevos cambios a nuestra base de datos. Si hay algun error, se realiza la sentencia *ROLLBACK* para dejar los datos como estaban antes del *BEGIN* y evitar inconsistencias.

Para el caso que realicemos un commit intermedio, lo que hacemos es dividir la transacción en dos sub-transacciones. Por lo que después de ejecutar query1 realizamos un *COMMIT* para guardar los cambios en la base de datos, y un BEGIN para continuar con el resto de la transacción.

A continuación vamos a mostrar los resultados obtenidos para cada una de las opciones que podiamos escoger para la transacción:

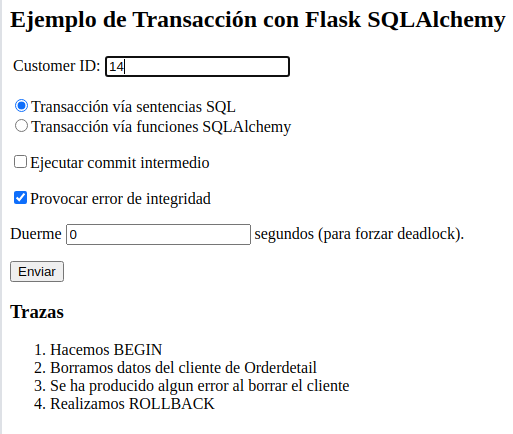
* **Transacción sin errores**





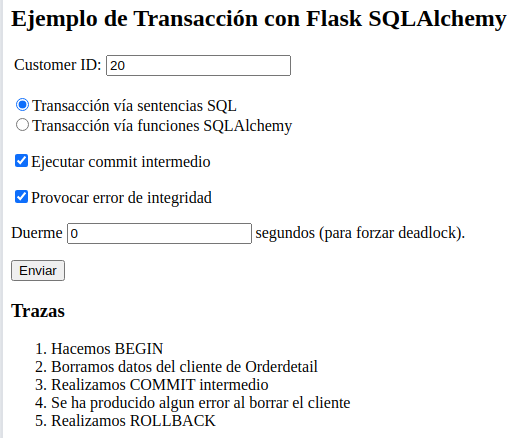
Observamos que se han borrado los datos de las tablas correspondientes.

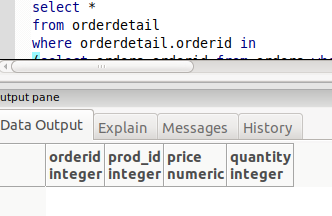
* **Transacción con errores**



Obervamos que al hacer el *ROLLBACK*, la base de datos se mantiene idéntica a como estaba antes del *BEGIN*, no se efectúa ninguno de los cambios.

* **Transacción con errores y COMMIT intermedio**

****

Al realizar un commit intermedio, los cambios efectuados en la tabla orderdetail se aplicaran a nuestra base de datos, mientras que los de las tablas orders y customers, al producirse un error, se realiza un *ROLLBACK* y no se efectúan. Realizamos un BEGIN después de hacer *COMMIT* debido a que si hay un error, como en este caso, al hacer *ROLLBACK*, volverá al estado antes del *BEGIN*, manteniendo así los cambios que se hayan realizado en ese *COMMIT*.

Observamos que el borrado sobre realizados en la tabla orderdetail si que se ha efectudao, a diferencia de los otros dos.

APARTADO I

En este apartado estudiamos los bloqueos y deadlocks que se pueden producir.

Para ello partimos de nuevo de la base de datos que se nos proporciona limpia.

A continuación, se nos pedía implementar en un script la creación de una nueva columna promo en *customers*.

Esto lo llevamos a cabo con la siguiente sentencia:

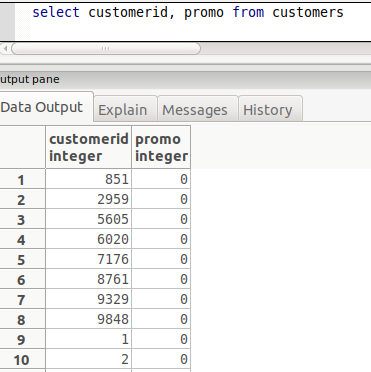
*alter table customers add column promo integer default 0;*

También se nos pedía crear un *trigger* sobre esta tabla *customers* que hiciese un descuento en los artículos de un carrito cuando se alterase la columna

recién creada promo de un cliente, del porcentaje indicado por esta. Añadimos también un sleep al comienzo de nuestro trigger, con el fin de crear el deadlock que se nos pide más adelante.

Todo esto se lleva a cabo en el archivo llamado **updPromo.sql.**

Mostramos a continuación capturas del funcionamiento correcto tanto de la creación de la columna promo, como del trigger.



Para poder comprobar el funcionamiento de nuestro *trigger* necesitábamos un customer con el carrito a *null*, por lo que buscamos en nuestra base de datos mediante la consulta:

*select \* from orders where status is NULL; (1)*

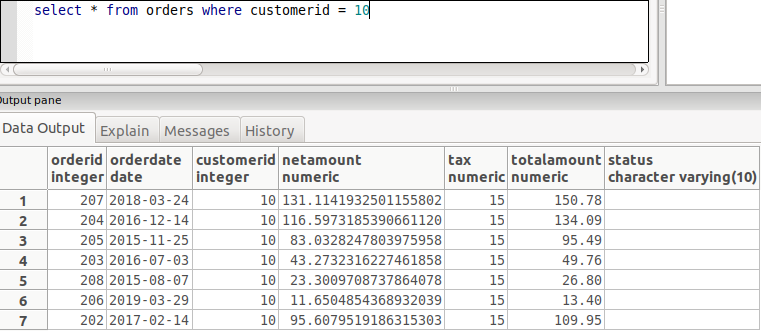
Observamos entonces que no había ningún caso en la base de datos con el status a *null*, por lo que procedimos a crearlo. Pusimos la cesta de algunos *customerid* a *null* para poder trabajar con ellos, mediante la sentencia:

*update customers set status = null where customerid = x (2)*

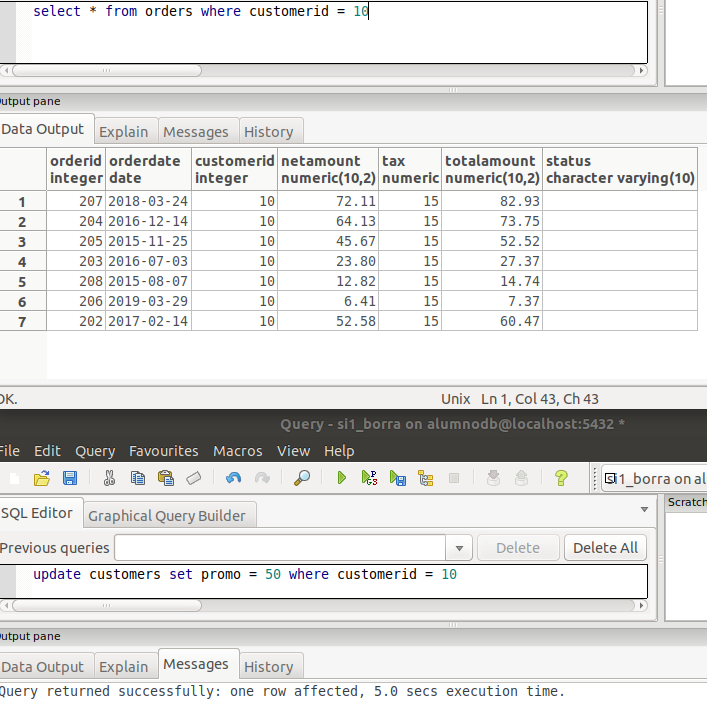
con x ciertos *customerids* presentes en la base de datos.

Tras obtener el resultado cambiamos la columna promo al 50% para este cliente mediante la sentencia:

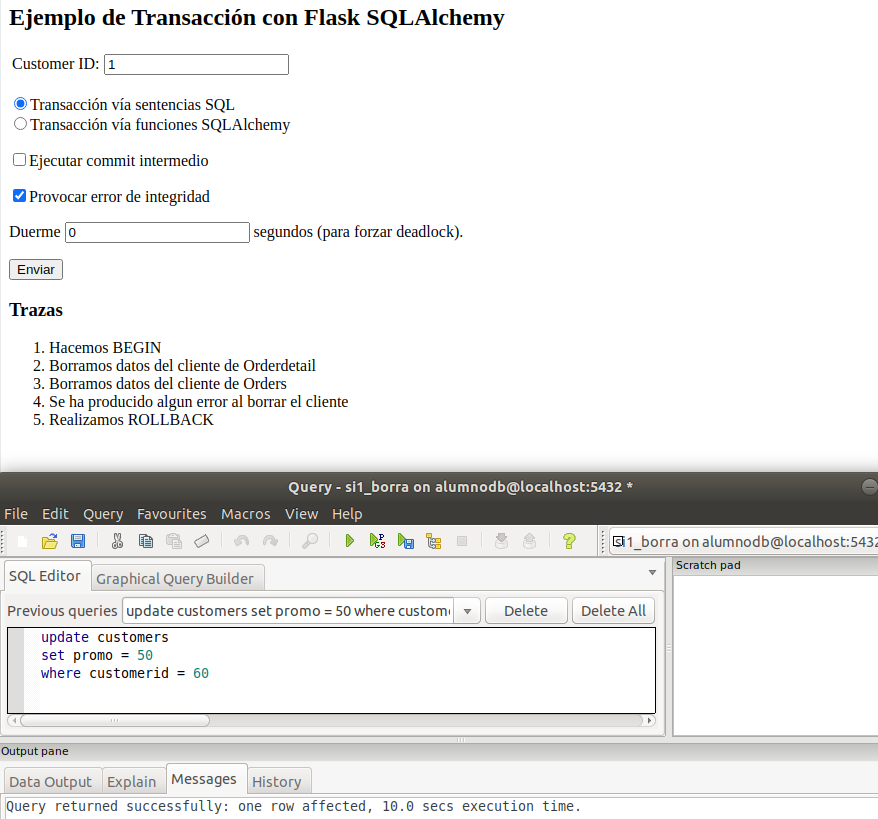
*update customers set promo = 50 where customerid = x (3)*



Volvemos a ejecutar la sentencia (1) para comprobar que, efectivamente, netamount y total amount se han actualizado, y en netamount tenemos la mitad del precio anterior, pues el descuento se ha aplicado.



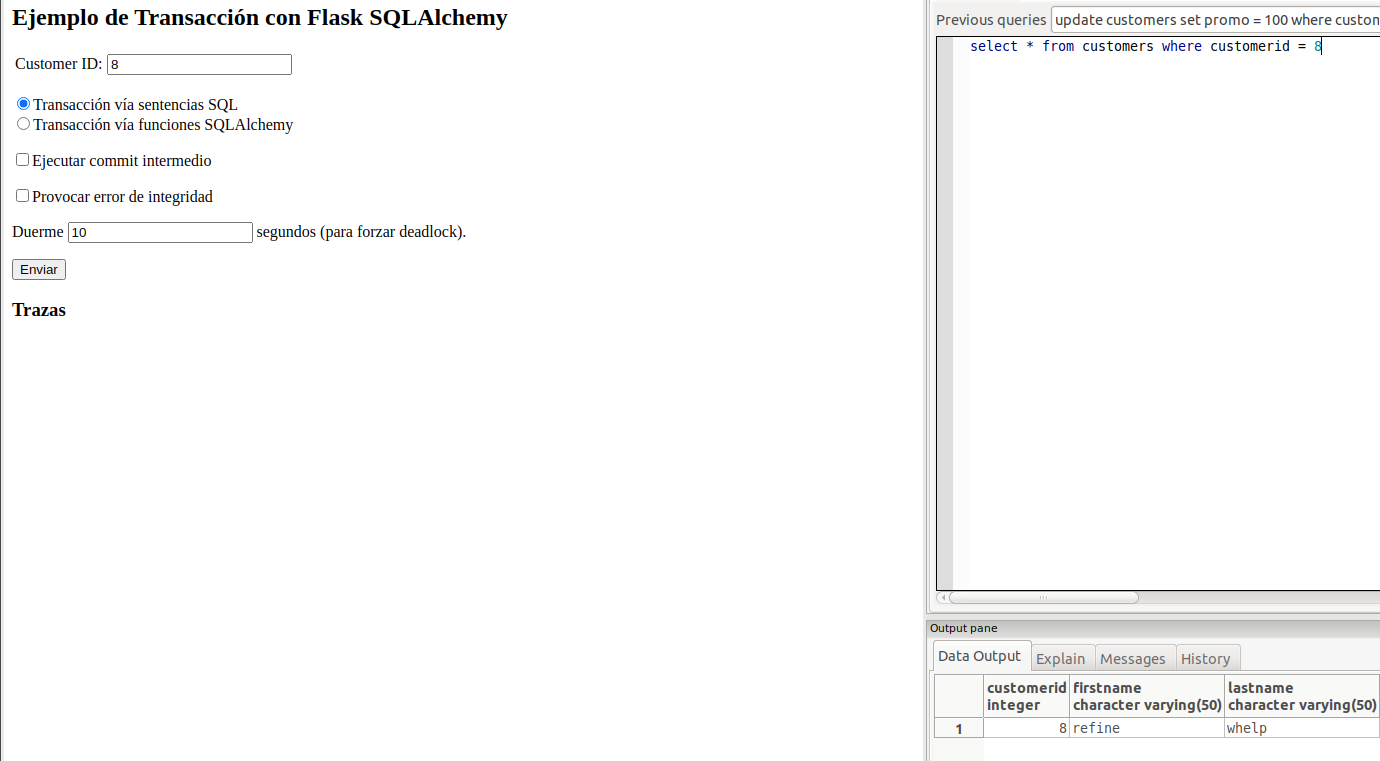
Para el apartado G, tomamos diez segundos de sleep para borraCliente y cinco segundos para el trigger y obervamos que el trigger espera a que finalice la transacción, que acabará con error (*ROLLBACK*) y ya se actualizará la tabla orders de forma correcta. Explicaremos los bloqueos más adelante.



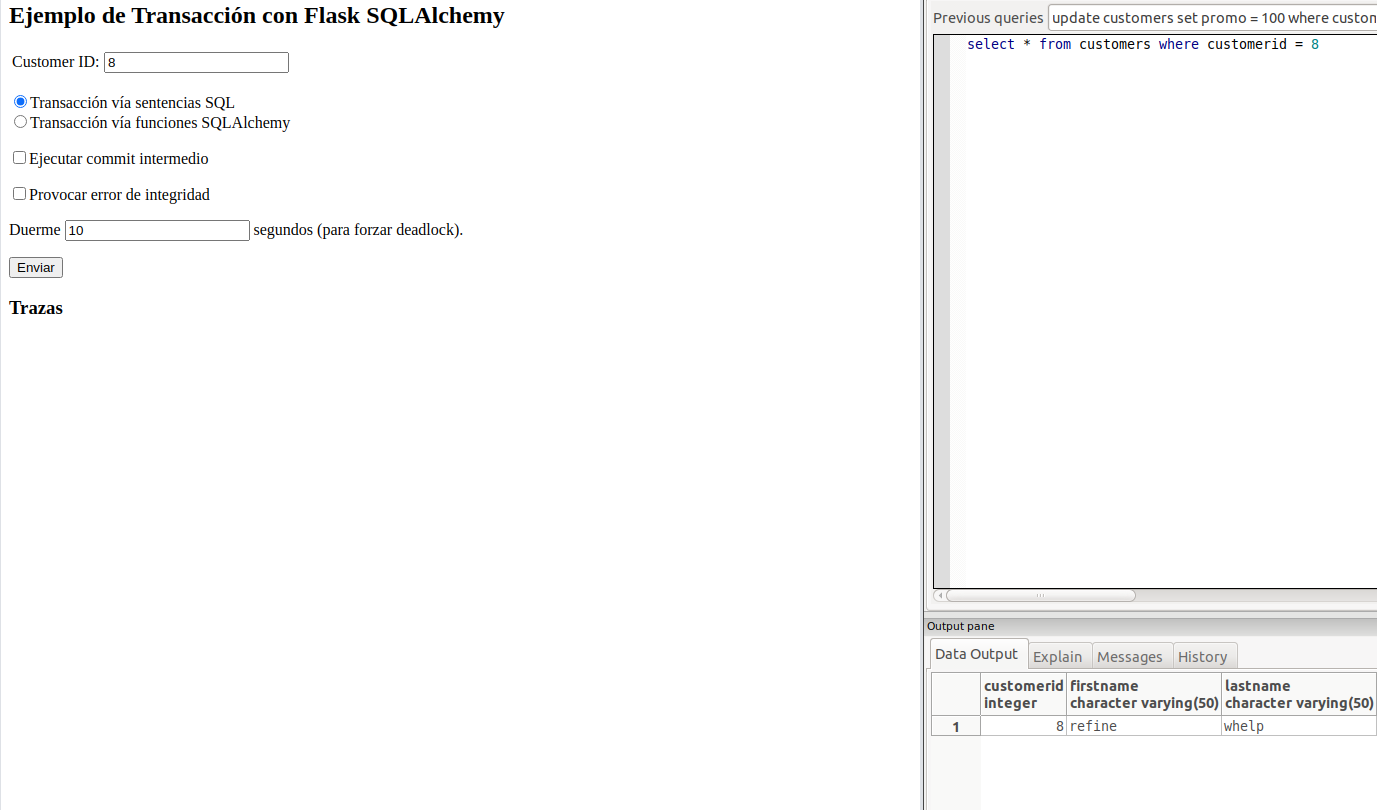
Por otro lado, se nos pide comprobar que durante el *sleep* en la página de borrado (que nosotros insertamos dentro del *database.py*, en la función de borrar cliente delCustomer, antes de borrar los datos del cliente de *customers* cuando no hay fallo) o el contenido en el trigger, los datos alterados por la página o por el *trigger* no son visibles.

Esto se debe en el caso de *delCostumer()* a que los cambios efectuados no se producen hastaque se realiza la sentencia *COMMIT* ( que es después del *sleep*). En el caso del *trigger*, como realizamos el *sleep* antes de modifica la tabla orders, no se mostrarán las modificaciones durante el *sleep*.

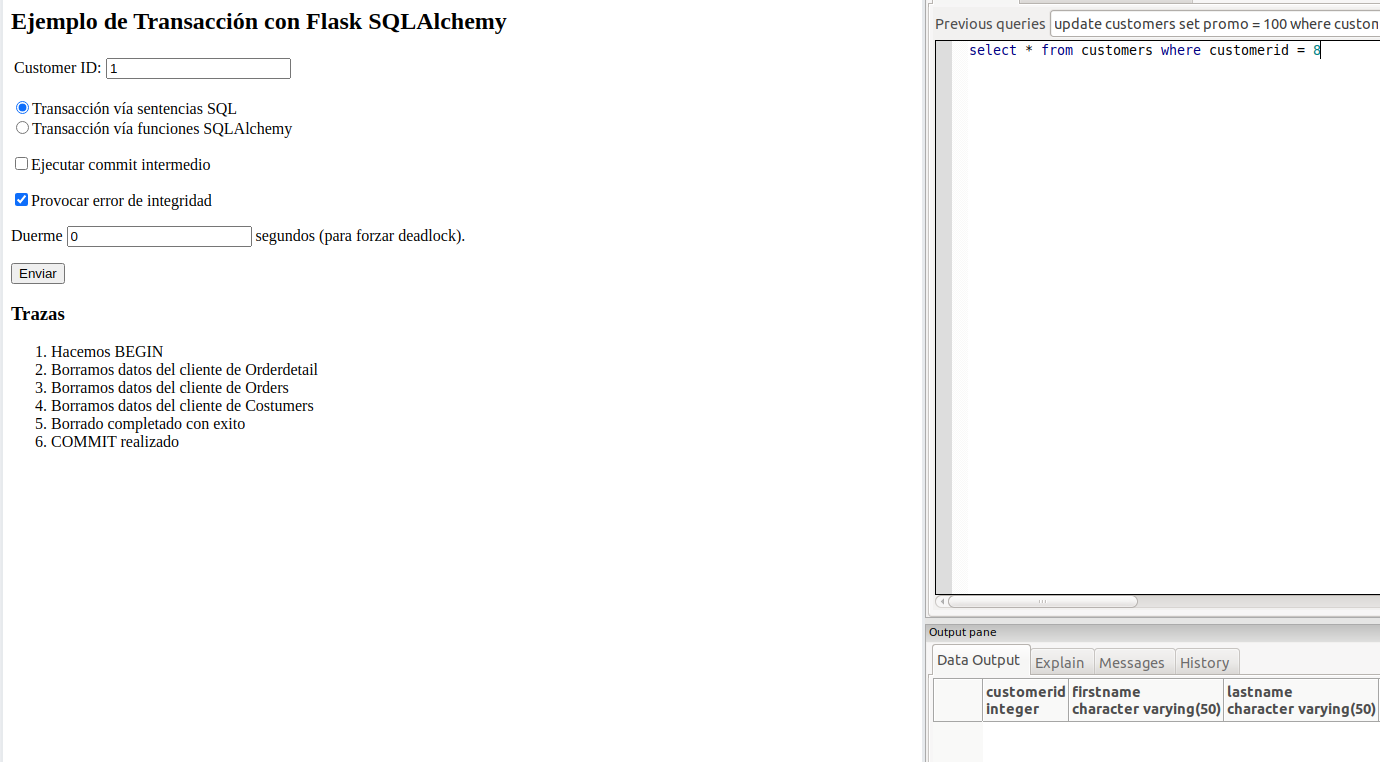
* **Caso *delCostumer()*** 
  1. Antes de iniciar la transacción:

****

* 1. Mientras se realiza la transacción:

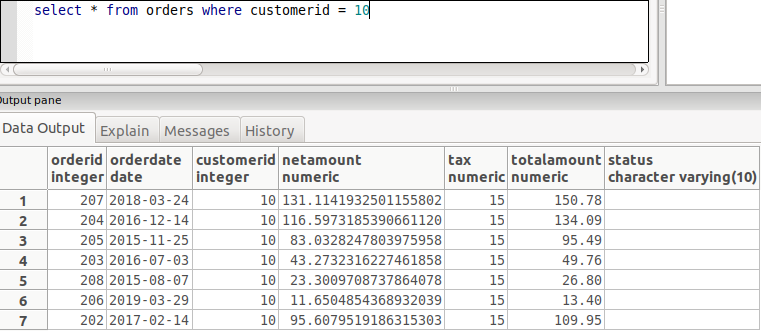


* 1. Al finalizar la transacción

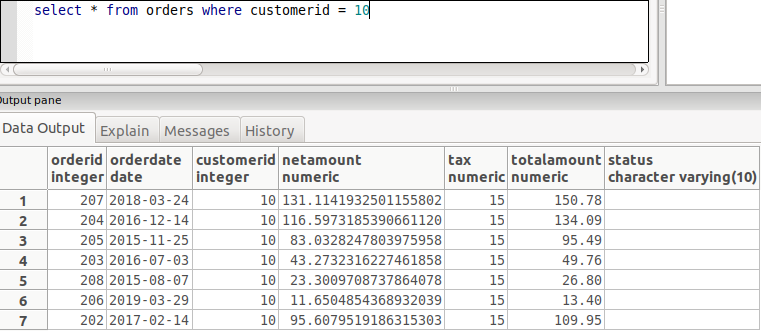
****

* **Caso trigger**

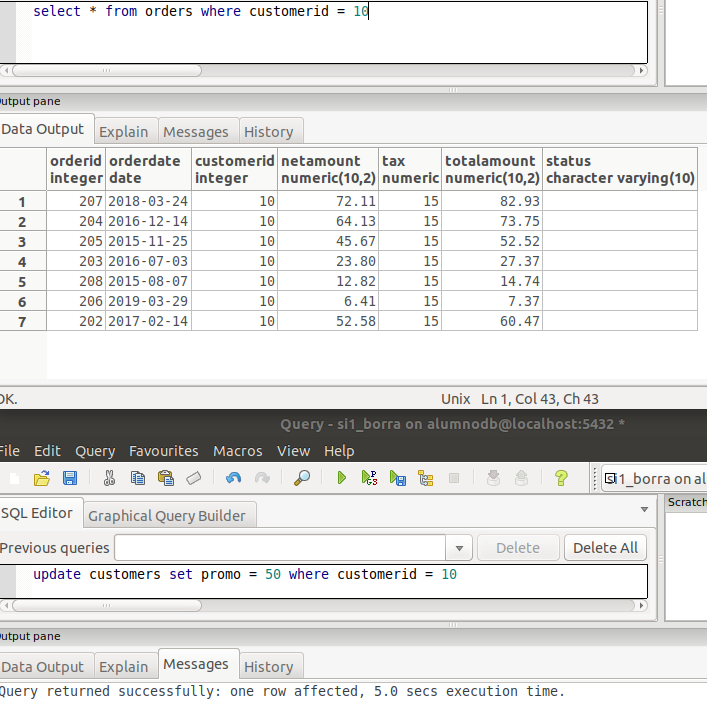
1. Antes de actualizar promo:



1. Nada mas actualizar promo



1. Pasado el tiempo del *sleep* del trigger



Se producen dos bloqueos. El trigger se activa cuando se produce un cambio en la tabla customers y se bloquean las filas que son alteradas de dicha tabla. Por otro lado, la transacción al borrar los datos de la tabla orderdetail y de la tabla orders, se produce un bloqueo en las filas de dichas tablas.

Además, al acabar los *sleep* de ambas partes el trigger quiere acceder a una fila bloqueada por la transacción de *orders* mientras que la transacción quiere acceder a una fila de customers bloqueada por el trigger. Por esta razón, se produce un interbloqueo entre ambas partes.

A continuación, ajustamos los tiempos de ambos sleeps (ambos a cinco segundos) y obervamos que se produce un *deadlock* y PostgreSQL automáticamente finaliza el trigger para resolverlo.

