**MEMORIA PRÁCTICA 4**

Práctica realizada por: Victoria Pelayo y Sofía Sánchez

**OPTIMIZACIÓN**

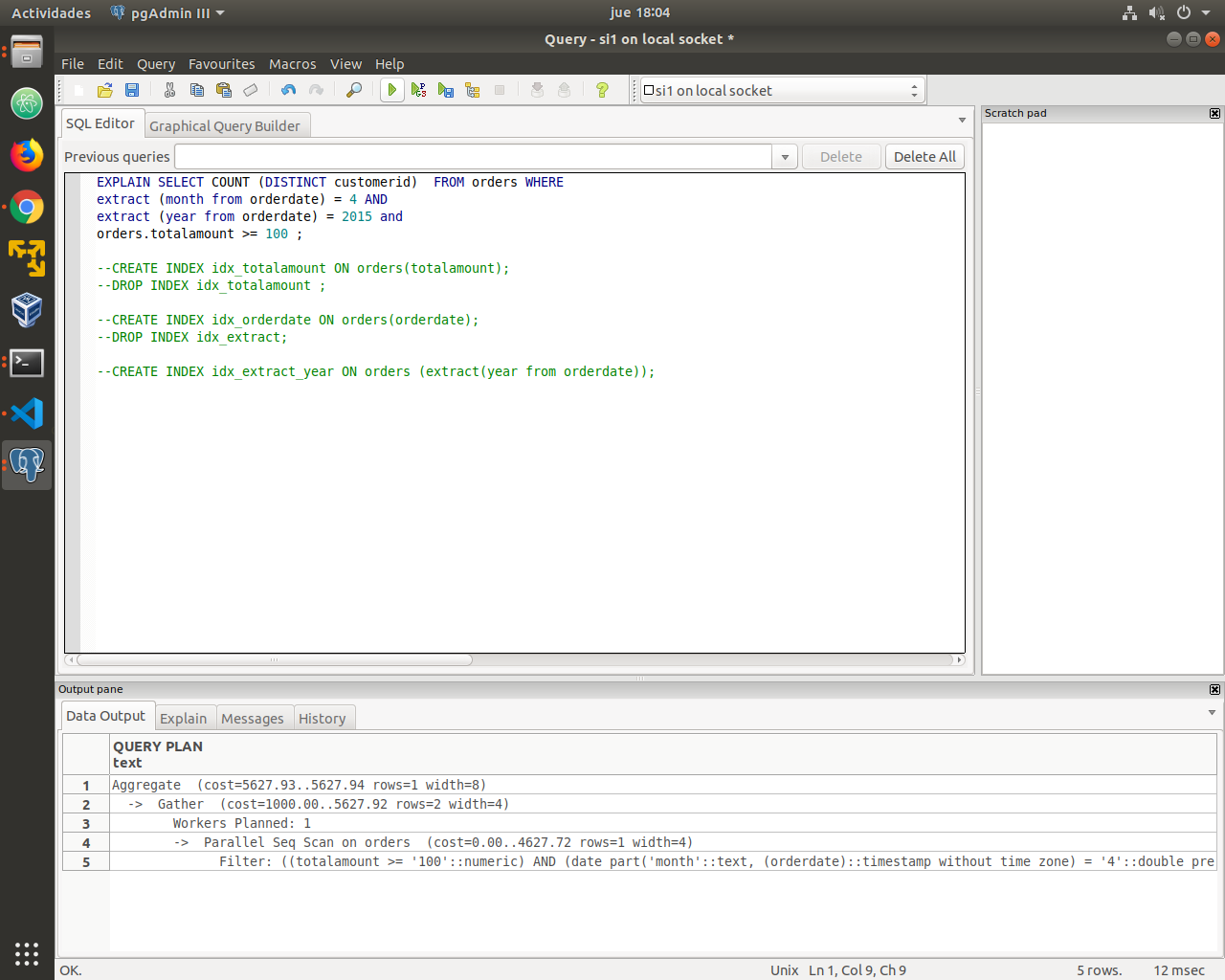
**APARTADO A:**

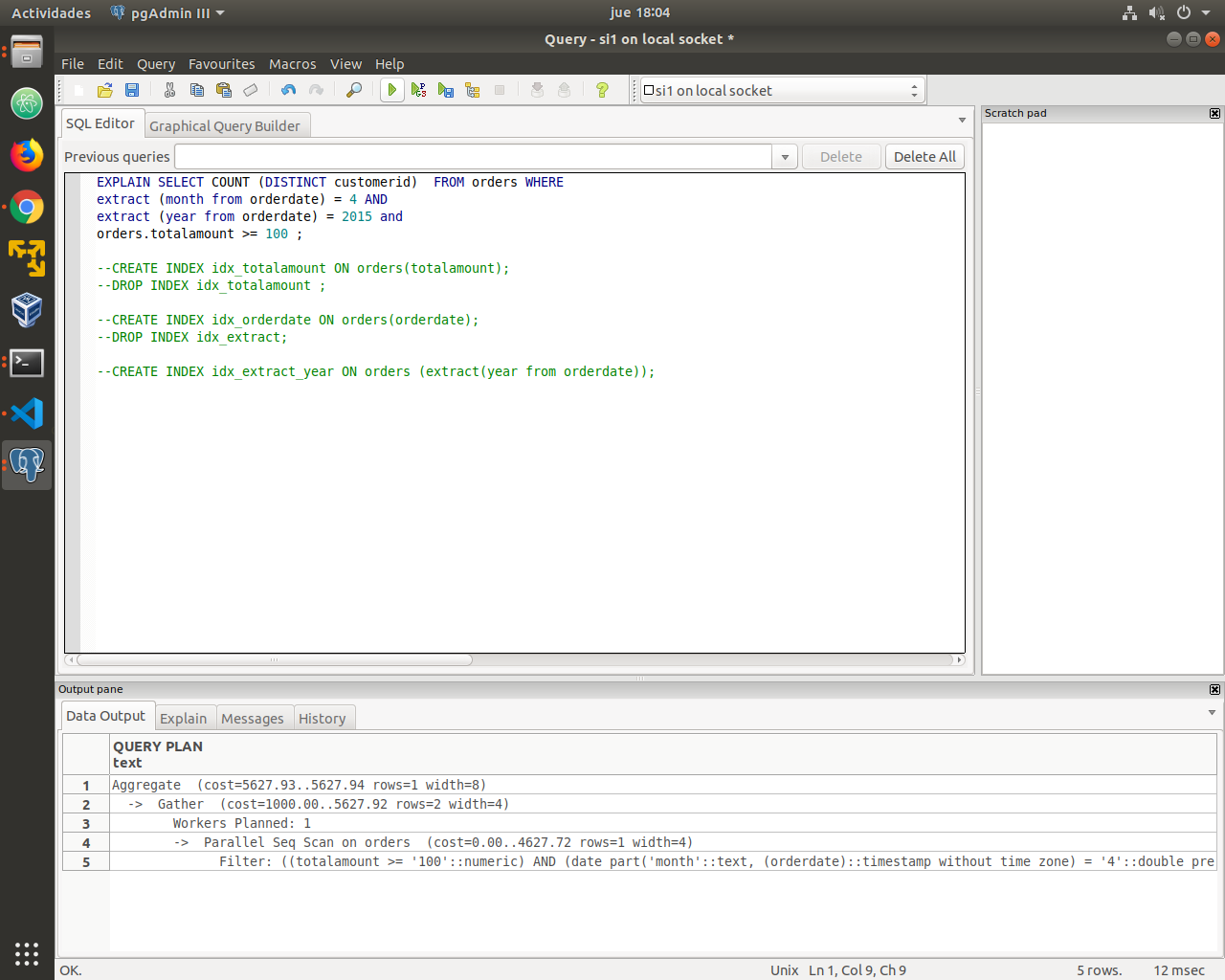
En este apartado tratamos de estudiar el impacto de un índice. Para ello crearemos una consulta en **clientesDistintos.sql**, la cual muestra el número de clientes distintos que tienen pedidos en un mes y año dados, en este caso Abril de 2015, con importe superior a un umbral dado, en este caso 100.

Dicha consulta es la siguiente:

SELECT COUNT ( DISTINCT customerid) FROM orders WHERE extract (mont from orderdate) = 4 AND extract (year from ordedate) = 2015 AND orders.totalamount >= 100;

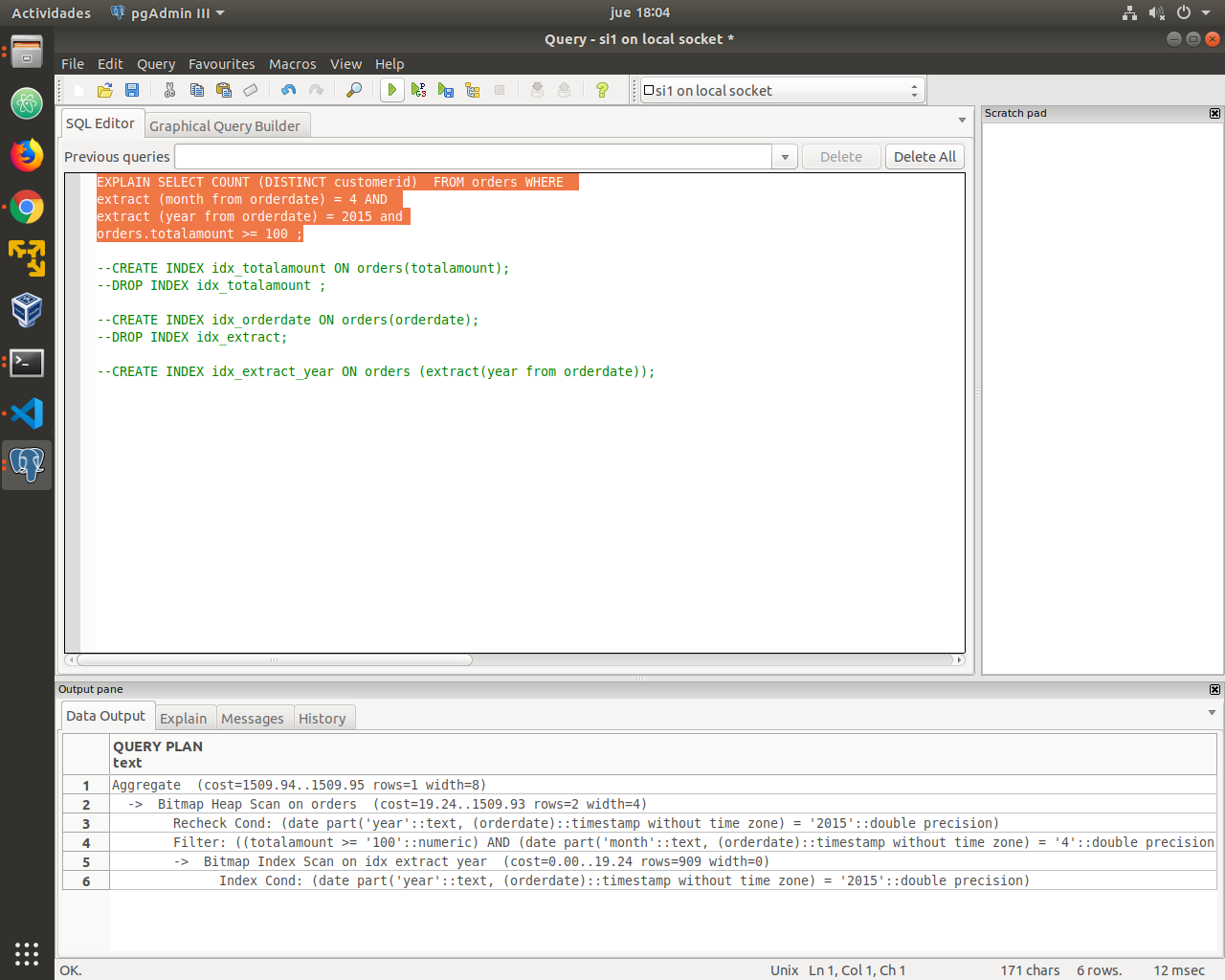
Primero mostramos el resultado de la consulta (con EXPLAIN) sin ningún índice:





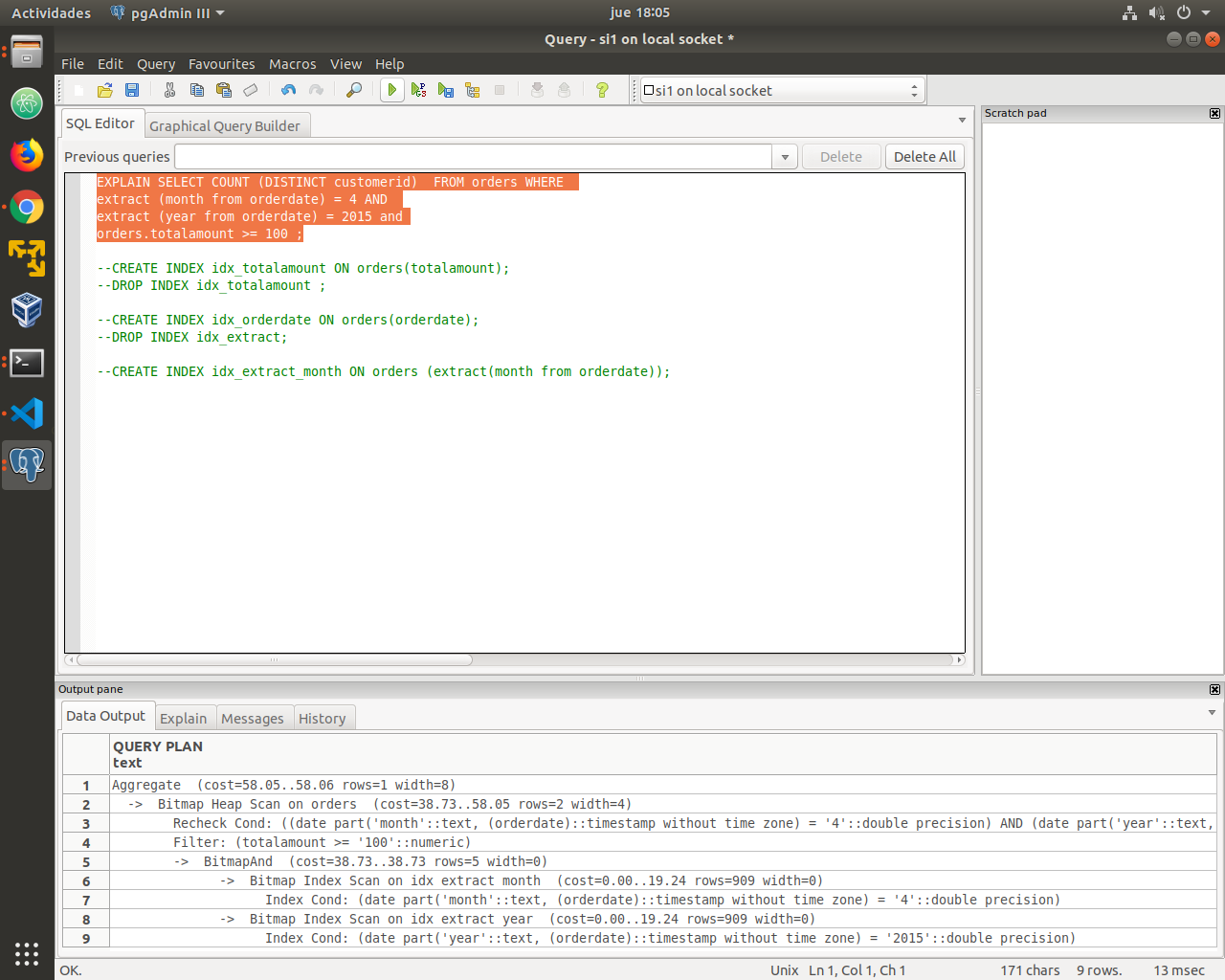
Vemos que se realiza un seq scan sobre la tabla orders que es lo más costoso ya que se realiza un escaneo secuencial que comienza en la fila 1 y continua hasta que se cumpla la consulta (que puede no ser toda la tabla), por eso hemos pensado en crear un índice en esa tabla.

En primer lugar, hemos decidido crear un índice en la operación que nos extrae el año, es decir en extract (year from orderdate).



Vemos que se ha sustituido el *seq scan* por un *index scan* y que el coste ha pasado de 5627 a 1505, una mejora notable. Esto se debe a que *Bitmap Index Scan* ha encontrado un pequeño subconjunto de filas para recuperar, en este caso las del índice creado, y va a buscar solo en esas filas.

Ahora probamos con índices en extract (year from orderdate) y extract (month from orderdate)



Con ambos índices vemos que de nuevo mejora la consulta. Pues ha pasado el coste de 1505 a 50 y de 5627 a 50 si comparamos con la consulta sin ningún índice. Esto se debe a lo que hemos explicado anteriormente pero aplicado a estos dos índices.

Finalmente, vemos que este es el mejor resultado que hemos obtenido. El coste pasa de 5627 a 50, es decir, obtenemos 5627/50 = 112, lo cual es una mejora bastante destacable.

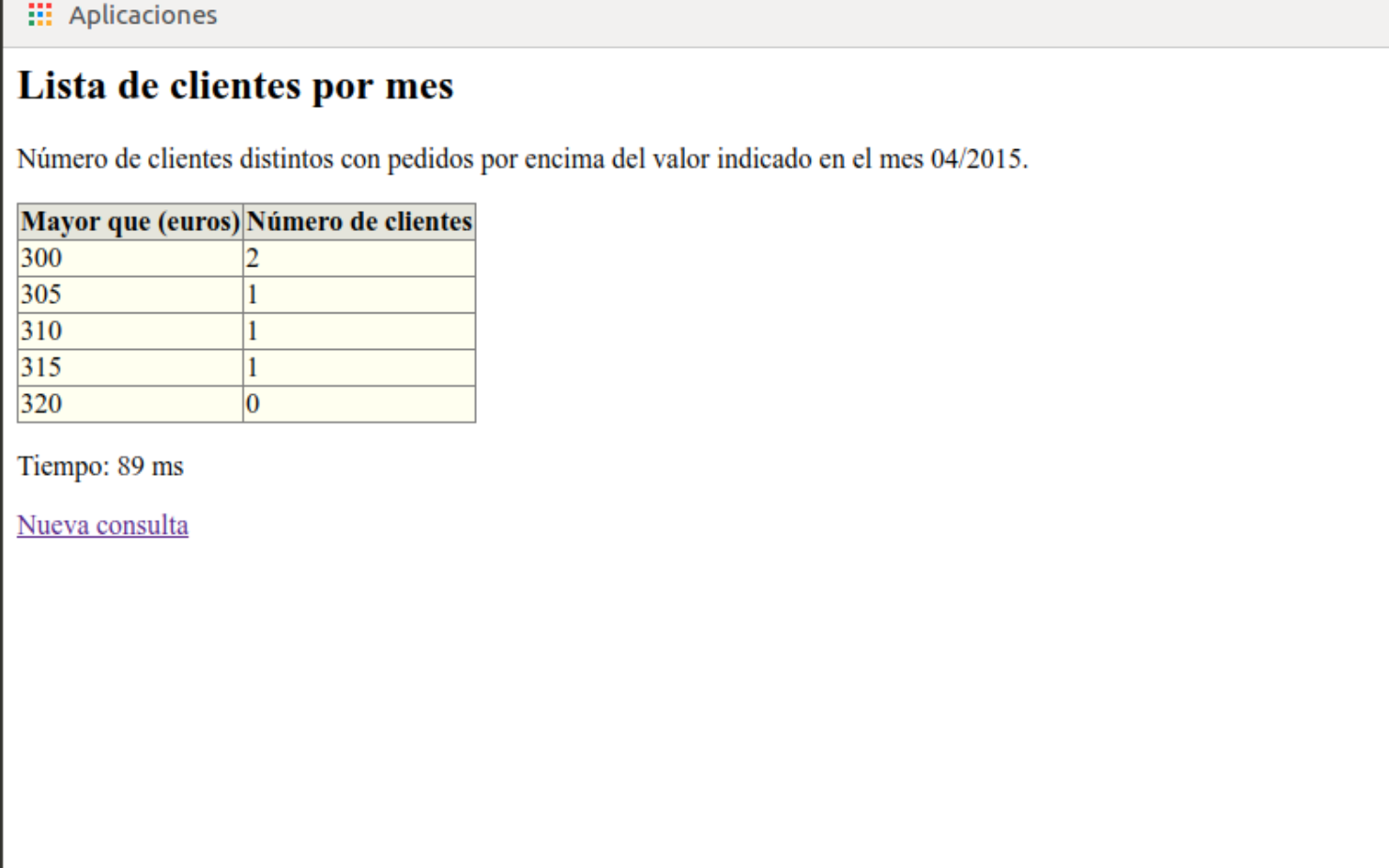
**APARTADO B:**

En este apartado tratamos de estudiar el impacto de preparar sentencias SQL. Para ello creamos una página, **listaClientesMes** que usa la consulta del apartado anterior para poder mostrar el número de clientes distintos.

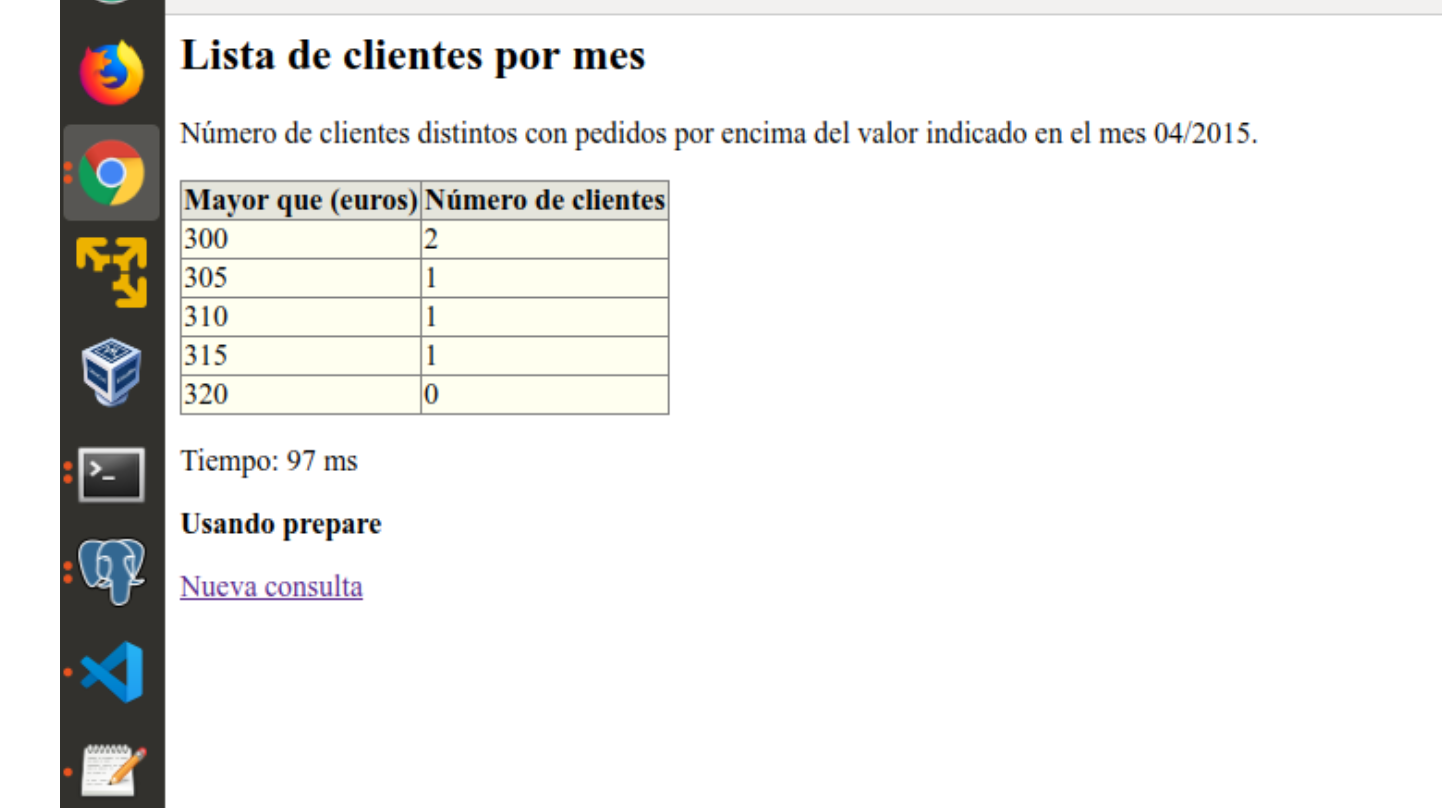
Para ello hemos modificado el fichero *database.py* y hemos diferenciado dos casos: uno cuando tenemos que preparar la consulta y otro en la que no tenemos que prepararla. En el primer caso, la consulta será PREPARE getListaCliMes (int, int, int) as (consulta del apartado anterior) y se realizará fuera del bucle puesto que tenemos que preparla. Dentro del bucle que nos indica el ejercicio solo tendremos que realizar dos acciones: si tenemos la consulta la ejecutaremos mediante EXECUTE o si no tenemos la consulta preparada, realizaremos la del apartado anterior. Además, debemos de usar DEALLOCATE para poder usarla varias veces seguidas.

A continuación, mostramos los resultados obtenidos.

* Sin prepare y sin ningún índice:

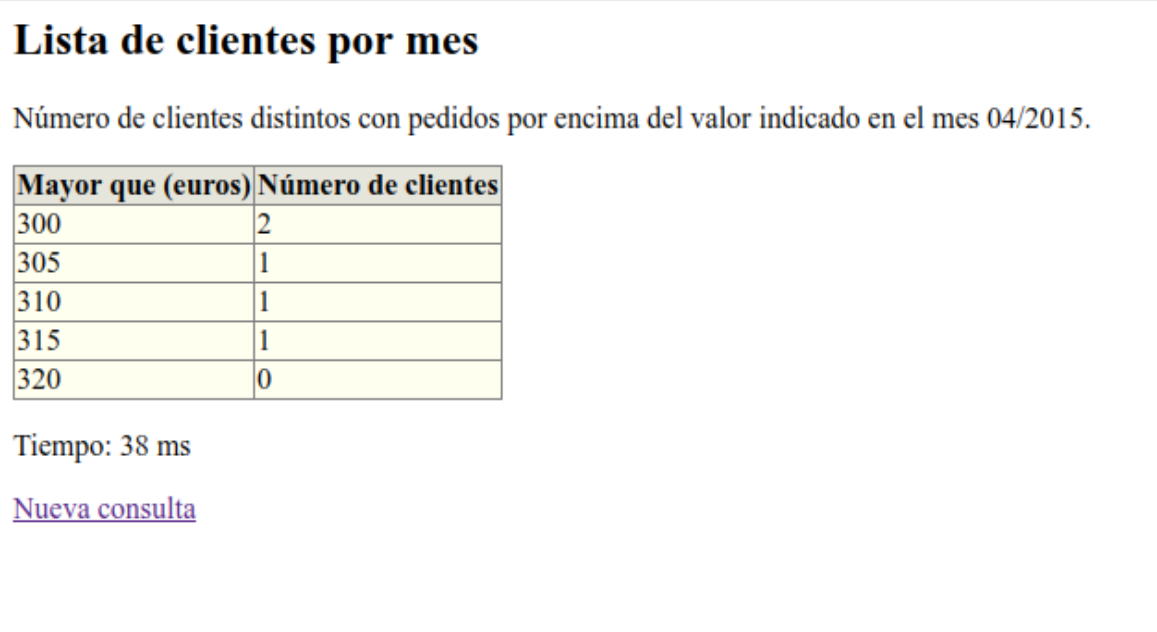


* Con prepare y sin ningún índice:

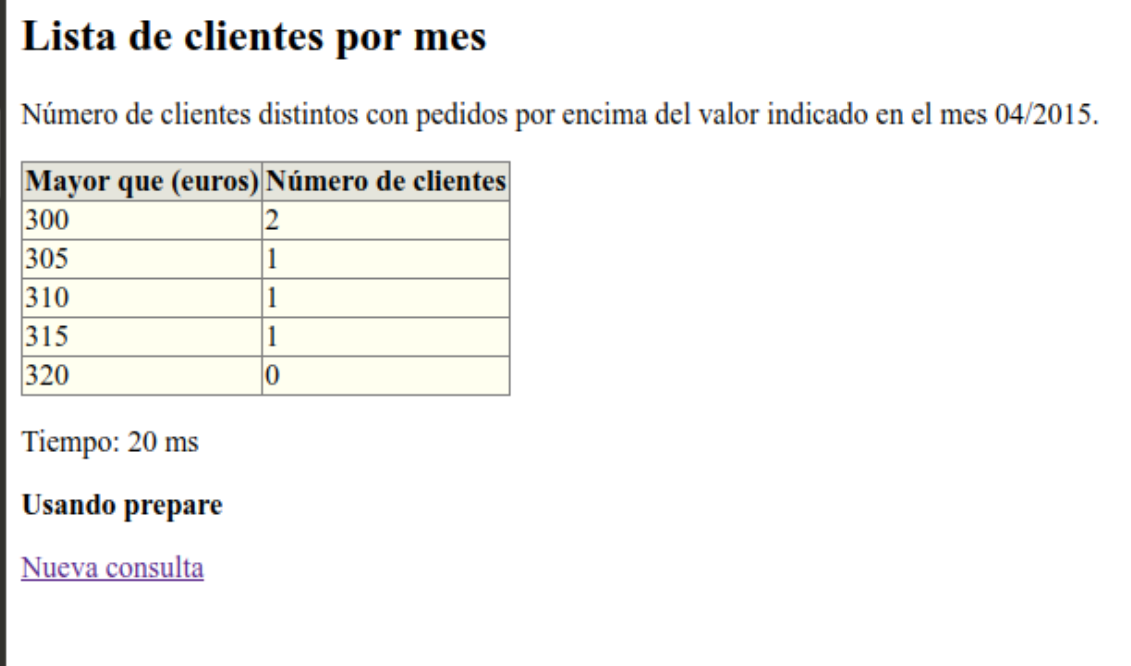


Observamos que con el prepare no hay mucha diferencia. Hemos dejado la captura donde la diferencia era la más “significativa” (son solo 6ms) pero al ejecutar las consultas varias veces observamos que obtuvimos tiempos muy similares entre ambas.

* Sin prepare y con index:



* Con prepare y con index:



Observamos que aquí si hay una mejora notable (hay una diferencia de 18 ms). Esto se debe a que las sentencias preparadas en sql reducen el tiempo de análisis ya que la preparación de la consulta se realiza solo una vez aunque luego se ejecute las veces que sea necesarias.

Cuando tratamos consultas con gran cantidad de datos si notamos mejoras notables con prepare, por ejemplo, con una consulta con mínimo 0 e incremento 1. Sin embargo, en ese tipo de consultas el index no proporciona mucha diferencia de tiempos.

PONER EJEMPLO DE ESTO

Cabe destacar que en el caso de generar estadísticas no hemos encontrado diferencias notables tras ejecutar la consulta varias veces. Tampoco hemos encontrado ningún caso en el que con PREPARE empeore el rendimiento al ejecutar la consulta varias veces.

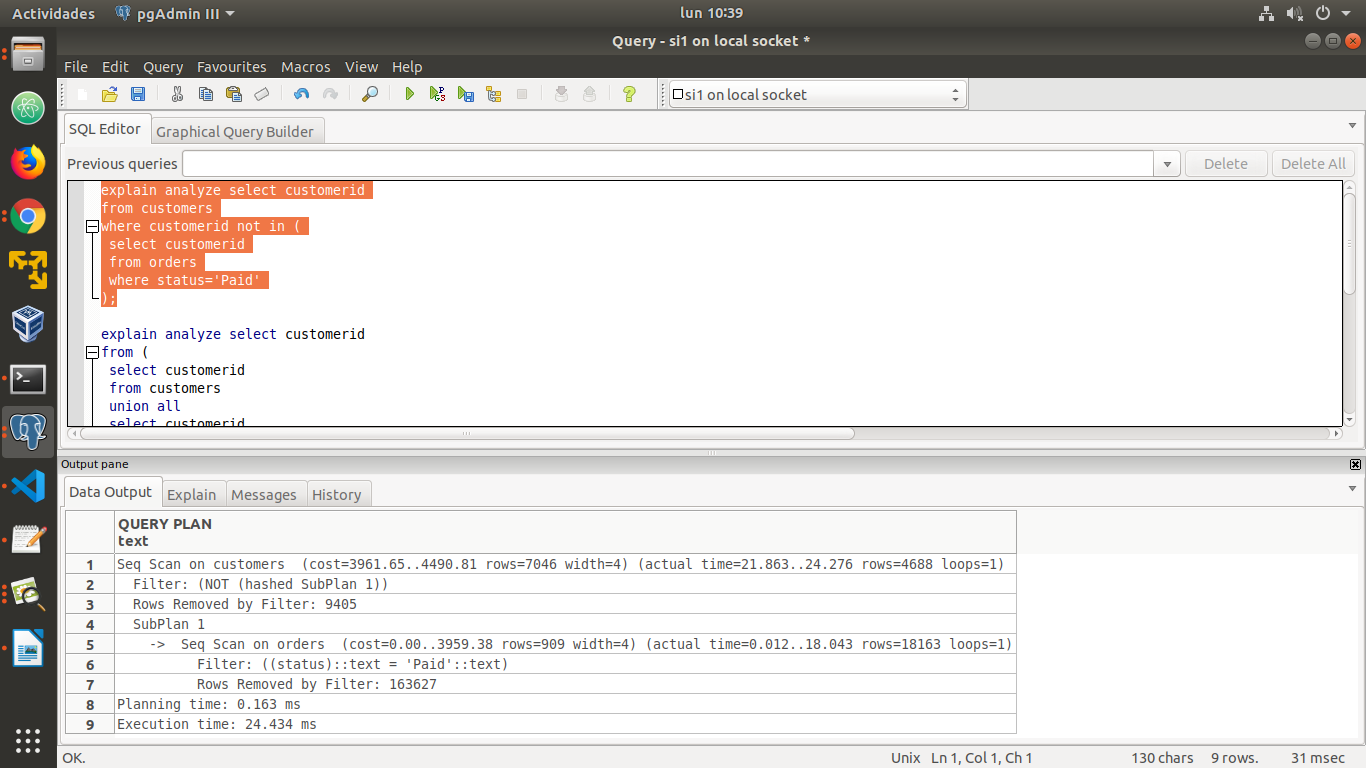
**APARTADO C**

En este apartado vamos a estudiar el impacto de cambiar la forma de realizar una consulta. Para ello vamos a estudiar los planes de ejecución de las consultas alternativas mostradas en el apéndice 1 y los compararemos.

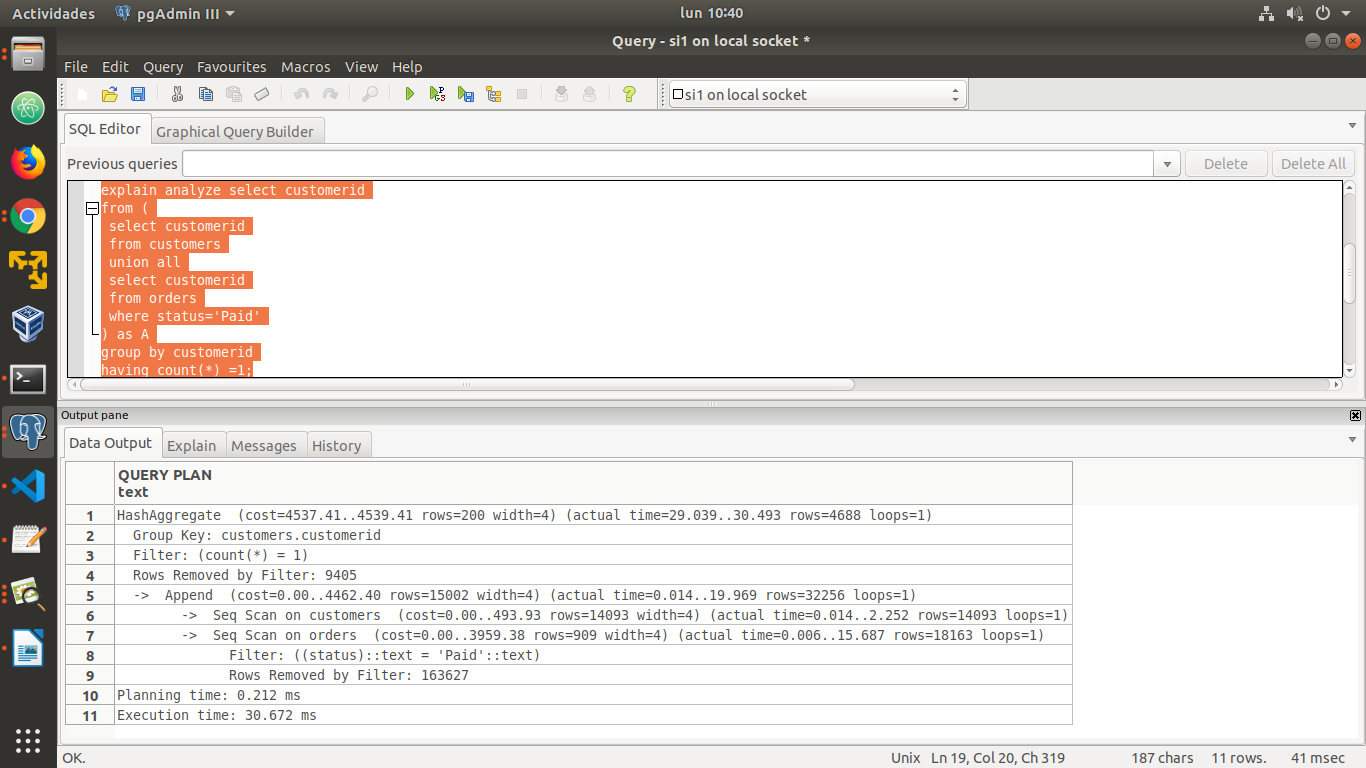
Para analizar los planes de ejecución de cada consulta y poder ver cual es la más eficiente y cuál devuelve algún resultado nada más comenzar su ejecución, haremos uso de EXPLAIN ANALYZE. También veremos que consulta se puede beneficiar de la ejecución en paralelo.

Los resultados (planes de ejecución y costes) obtenidos tras ejecutar las consultas han sido los siguientes:

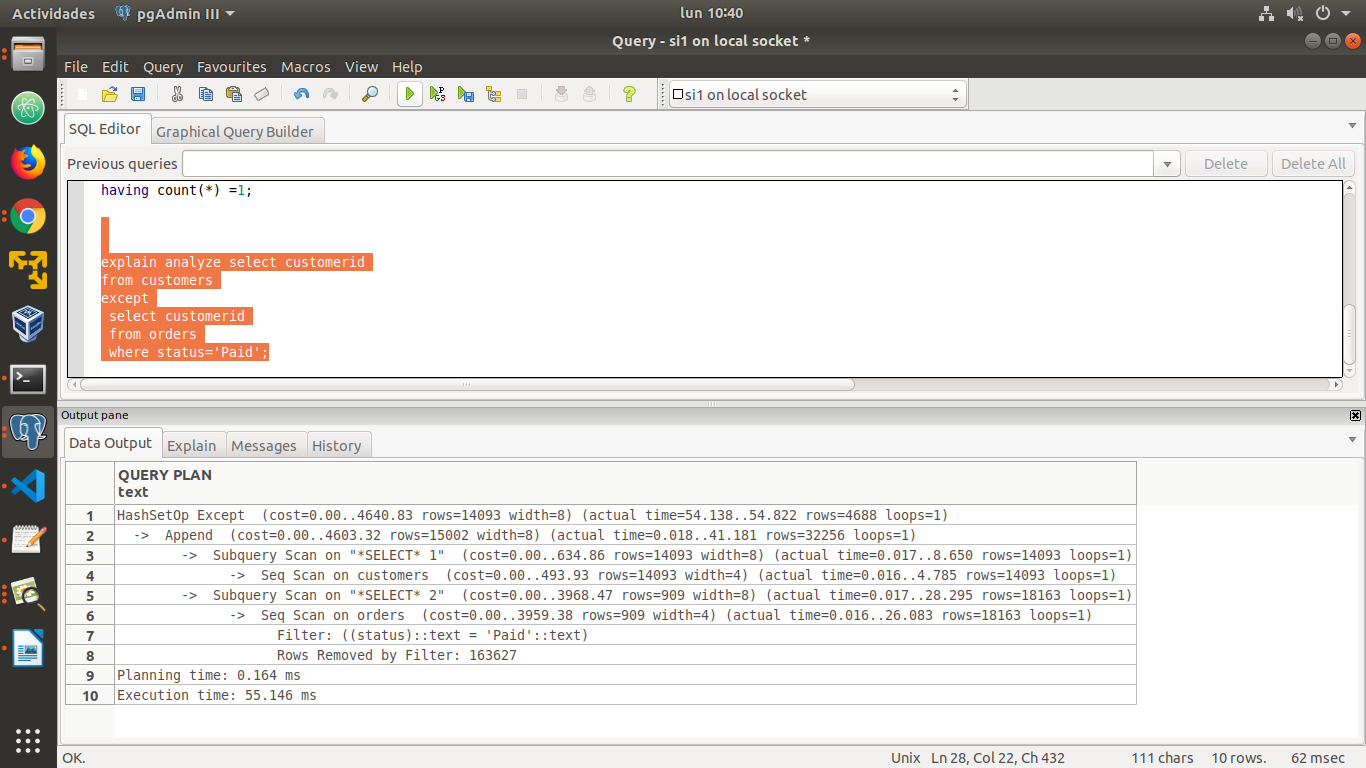
* Consulta 1



* Consulta 2



* Consulta 3



Como podemos observar, de las tres consultas la única que devuelve resultados nada mas comenzar su ejecución es la ultima pues como podemos apreciar en la captura vemos que el coste es 0 *(cost=0.00..4640.93)*. Mientras que los otros costes de ejecución son de 3961 y 4537 para la primera y la segunda consulta, respectivamente.

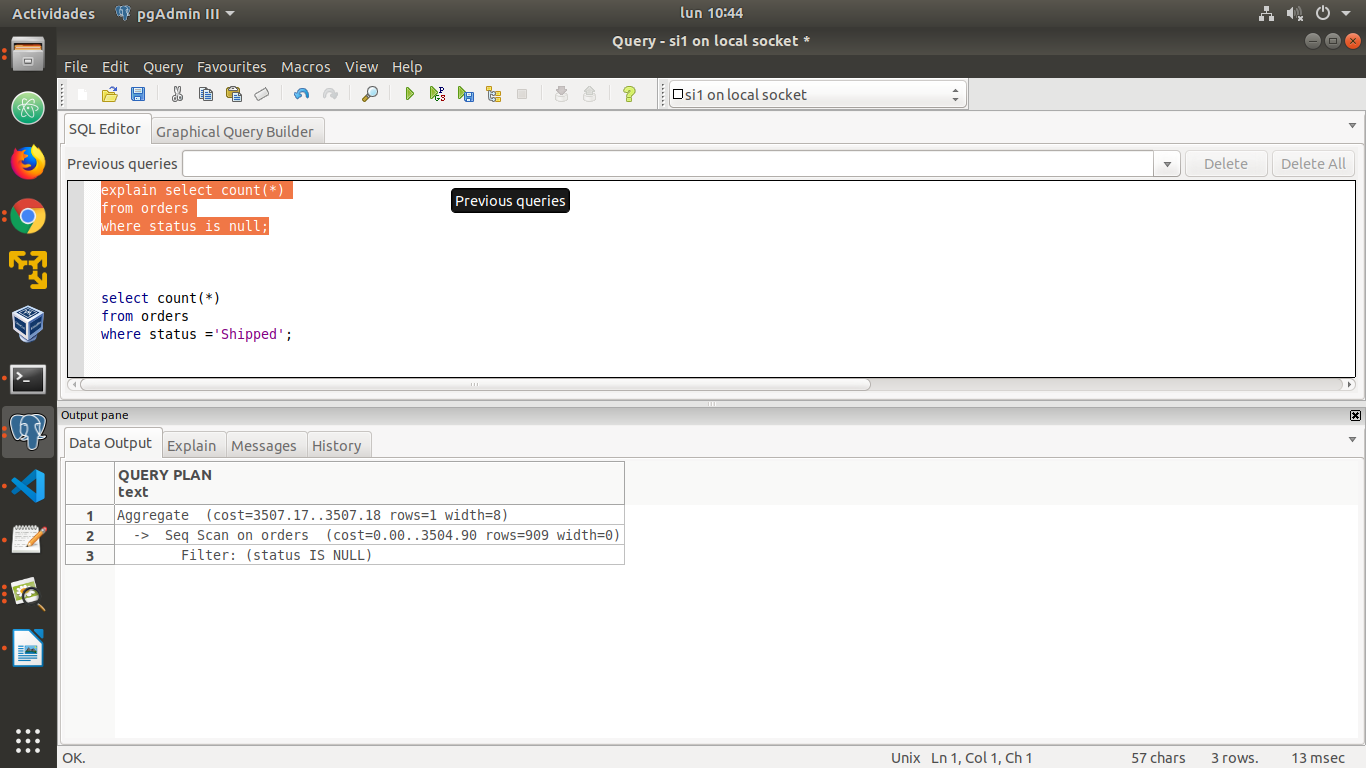
Analizando los planes de ejecución de las tres consultas, apreciamos que para la segunda se realizan dos Seq Scan paralelamente, lo que indica que se beneficia de la ejecución en paralelo. Algo parecido ocurre con la tercera consulta ya que se realizan dos subquerys paralelamente, por lo que también se beneficia de la ejecución en paralelo. No ocurre lo mismo en la primera consulta ya que solo nos encontramos con un Seq Scan para la consulta.

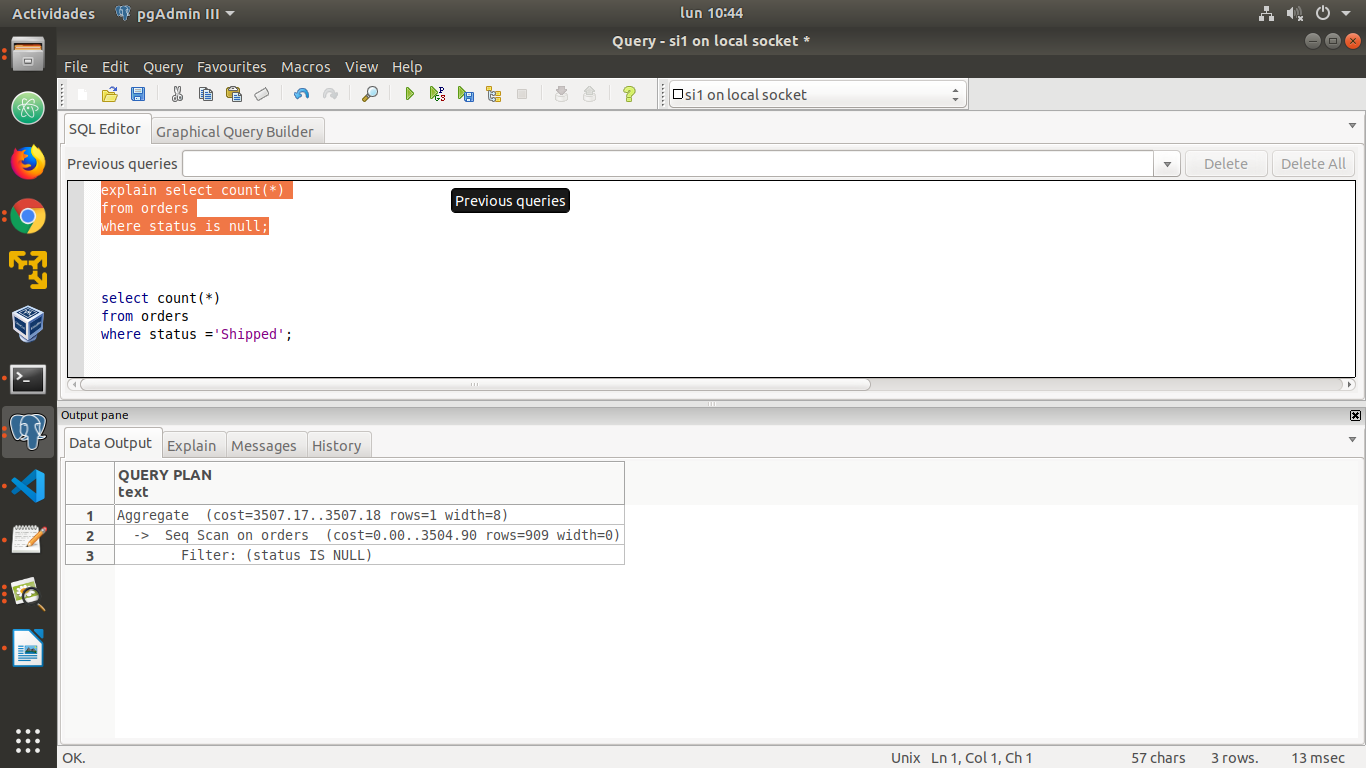
**APARTADO D:**

En este apartado estudiaremos el impacto de la generación de estadísticas. Para ello partiremos de nuevo de la base de datos que nos han suministrado.

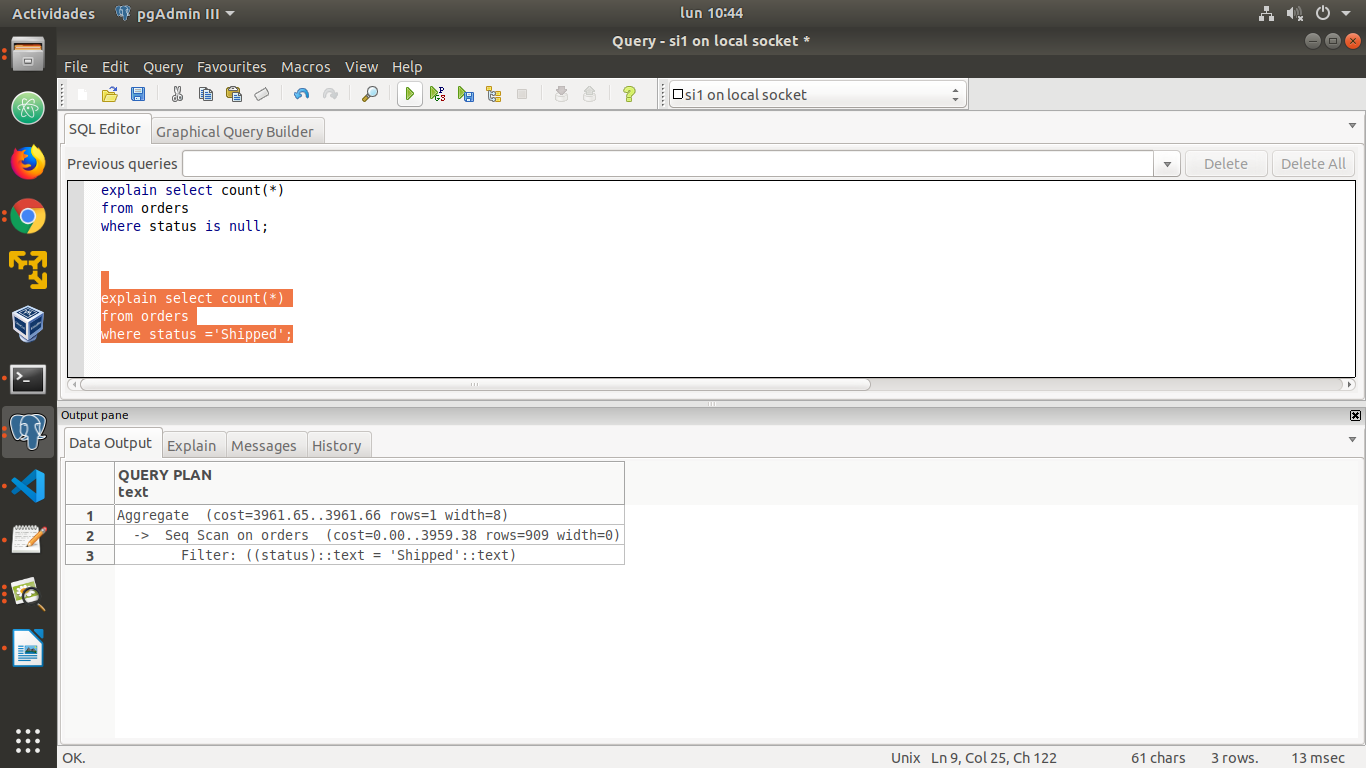
En primer lugar, mediante la senetencia EXPLAIN obtenemos los planes de ejecución de las dos consultas proporcionadas en el Apéndice 2 **sin ningún índice**.

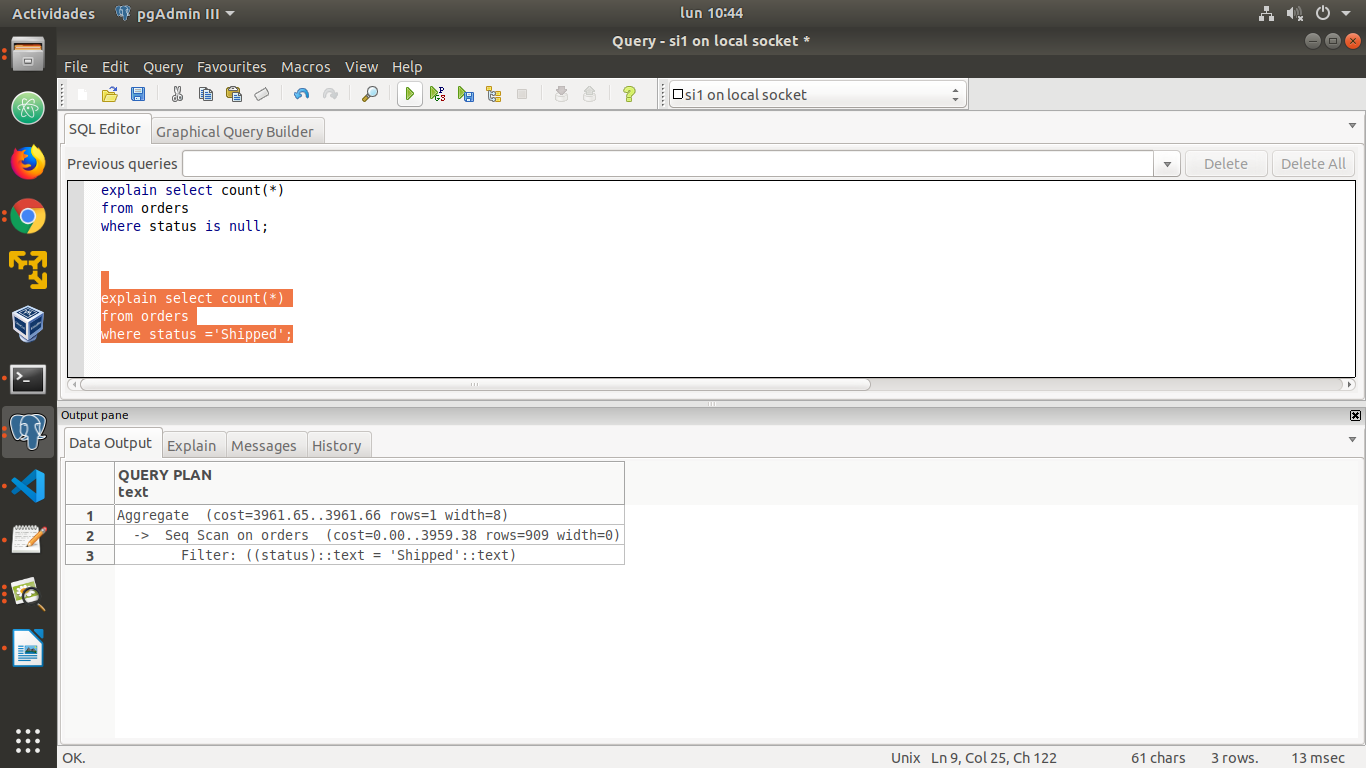
* Consulta 1





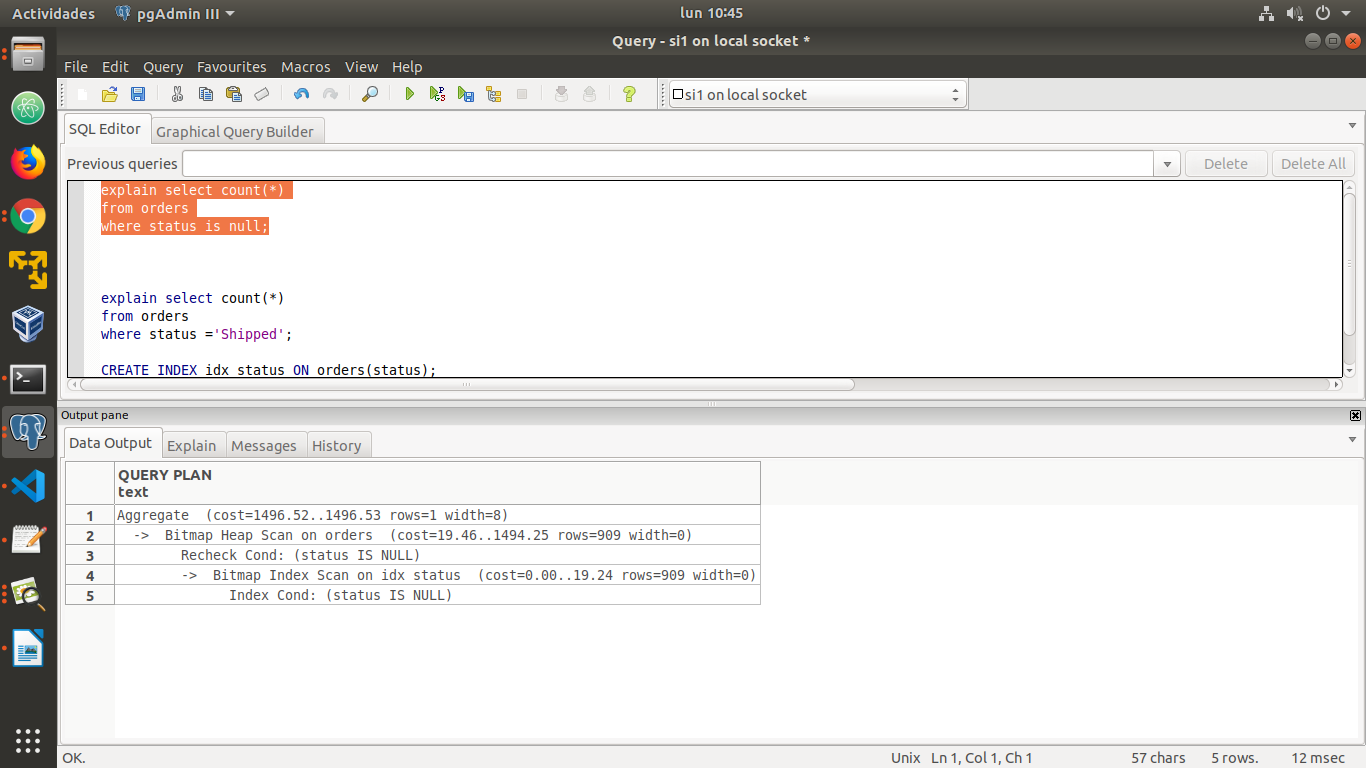
* Consulta 2



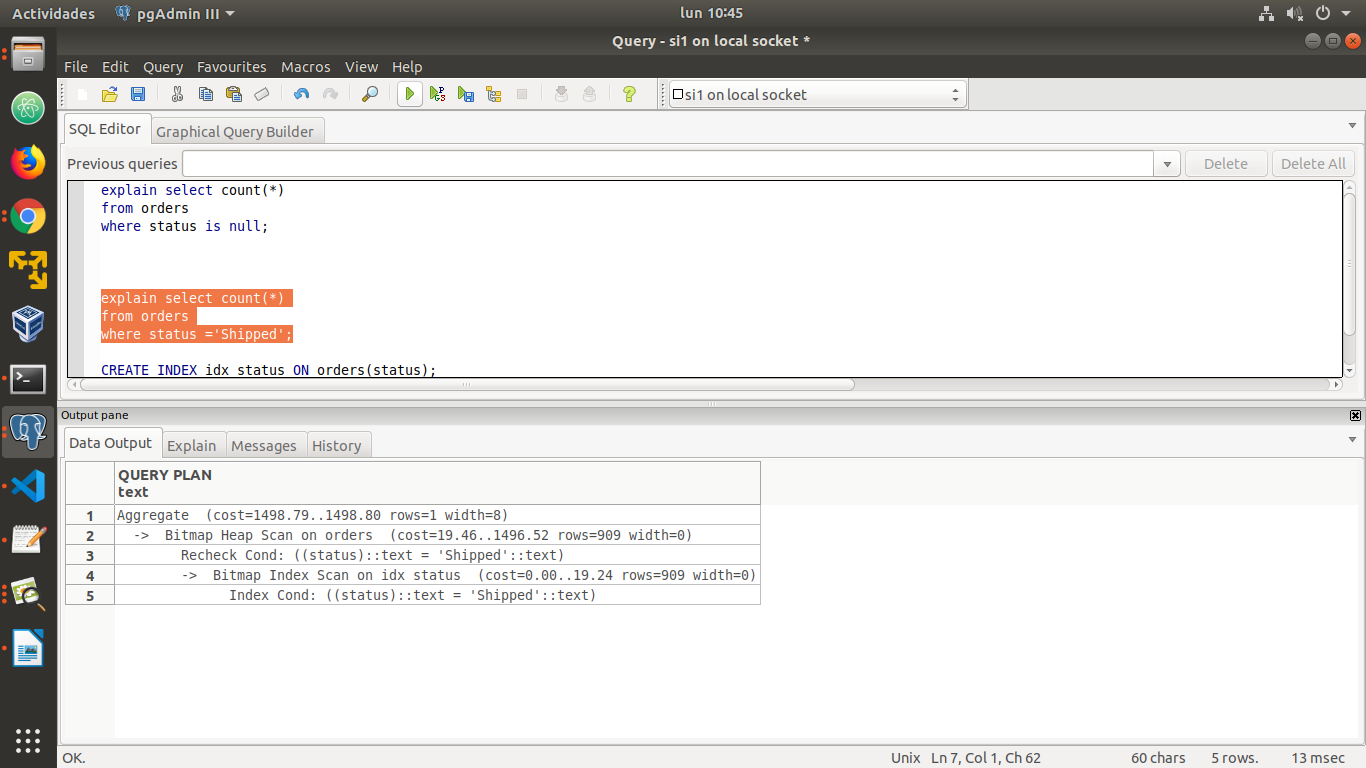


Ahora **creamos un índice** en la tabla *orders* para la columna *status* y volvemos a obtener los planes de ejecución de ambas consultas:

* Consulta 1:



* Consulta 2:



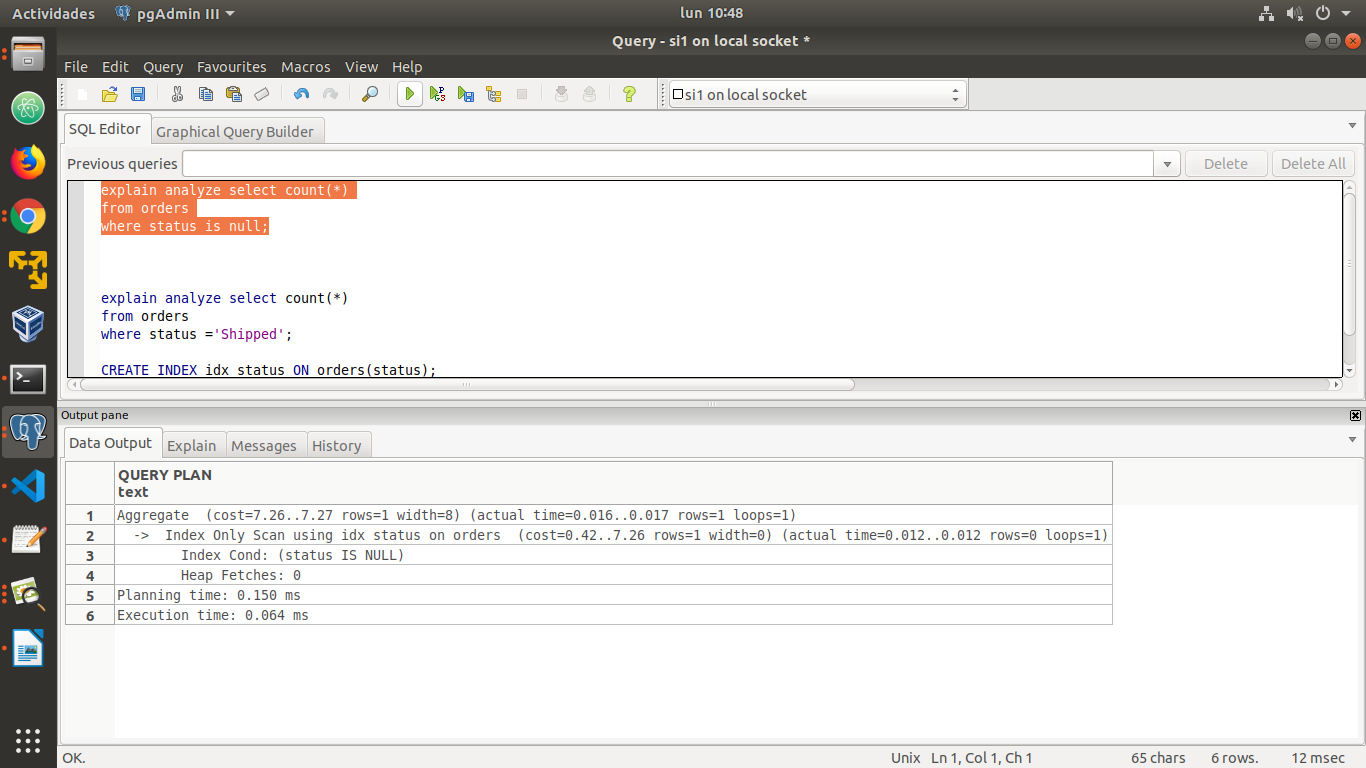
Viendo los resultados obtenidos podemos decir apreciar que los planes de ejecucion para las consultas con indice y sin indice son diferente. A su vez también apreciamos variaciones en los tiempos de ejecucion de estas. Esto se debe a lo que ya hemos explicado en apartados anteriores, es decir, por las diferencias entre el escaneo secuencial y el escaneo mediante un indice.

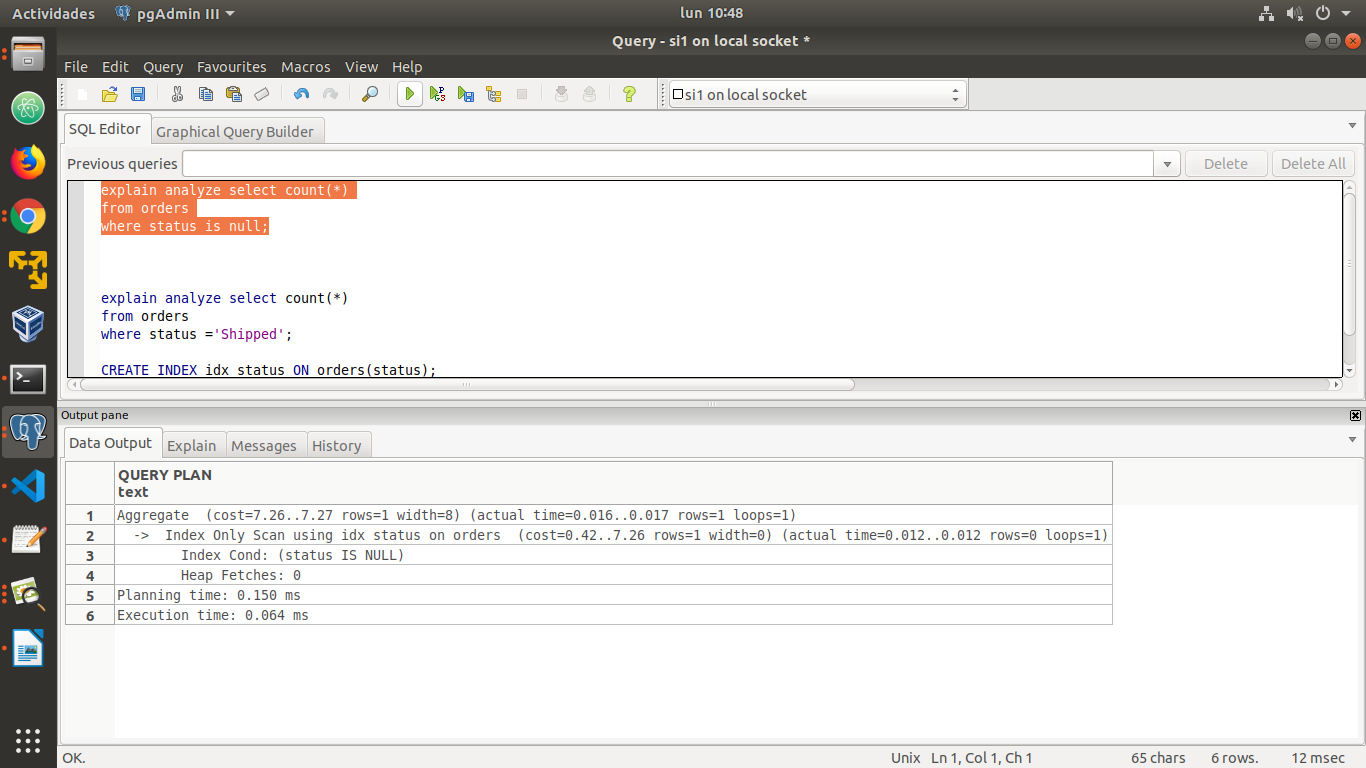
Estos resultados se encuentran dentro de lo esperado , pues ya lo habiamos visto en los apartados anteriores (Optimización).

Ahora ejecutaremos la sentencia ANALYZE para generar las estadísticas sobre la tabla *orders*. Para ello realizamos en PgAdmin la consulta ANALYZE orders;

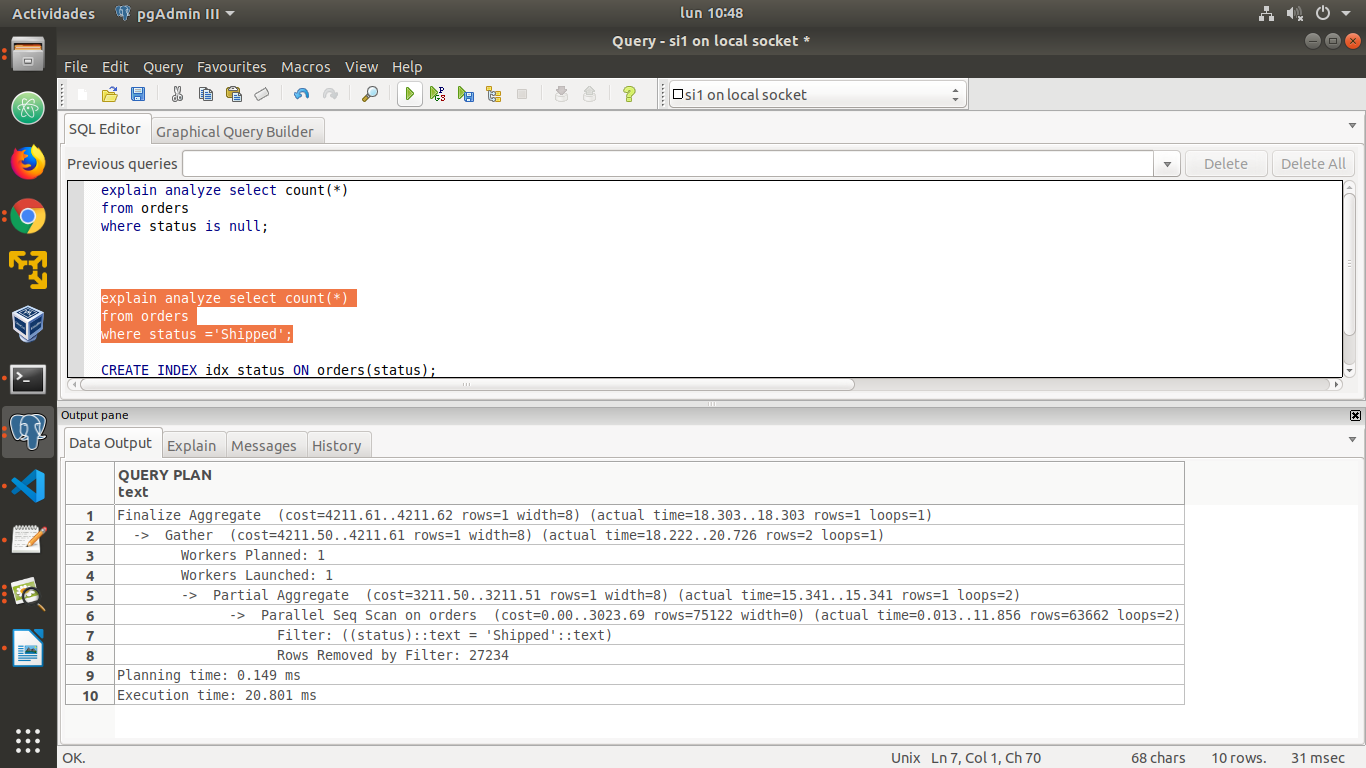
Los resultados de ambas consultas son los siguientes:

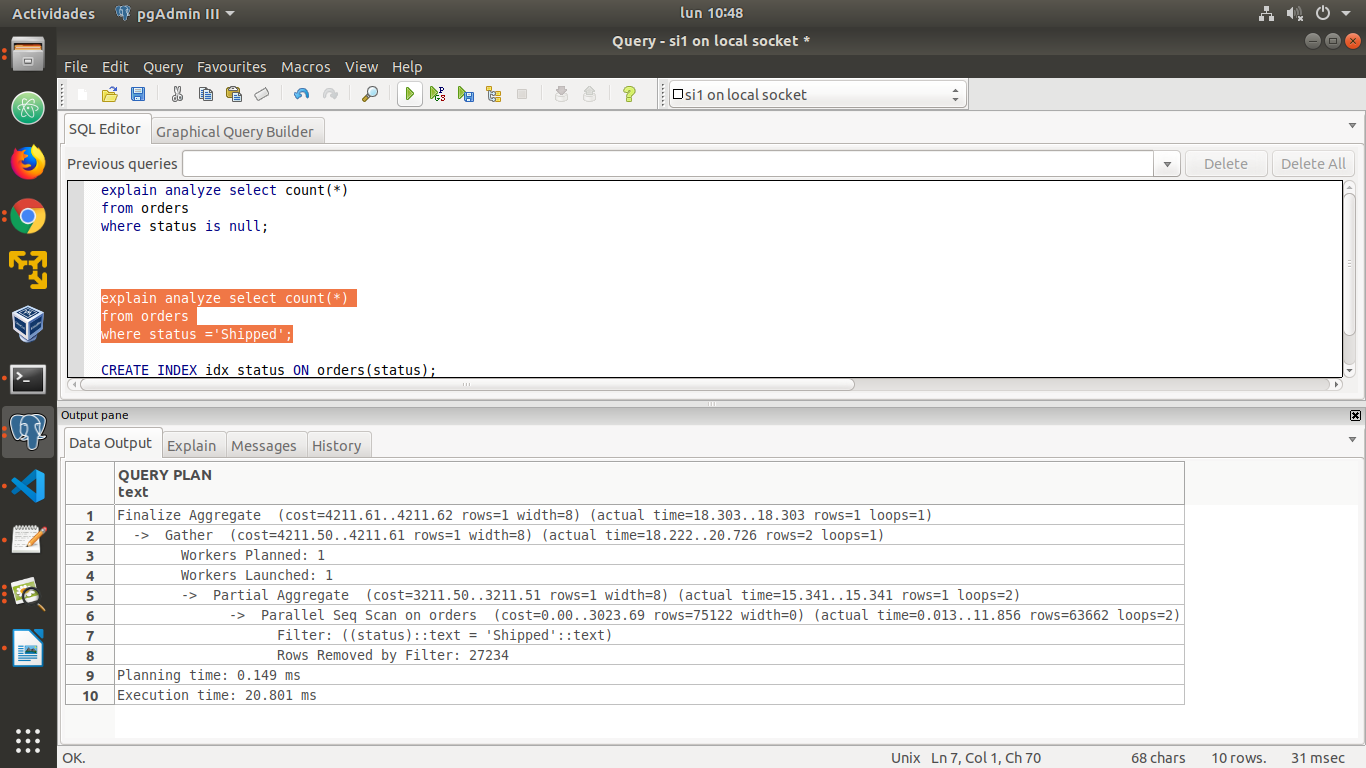
* Consulta 1:





* Consulta 2:

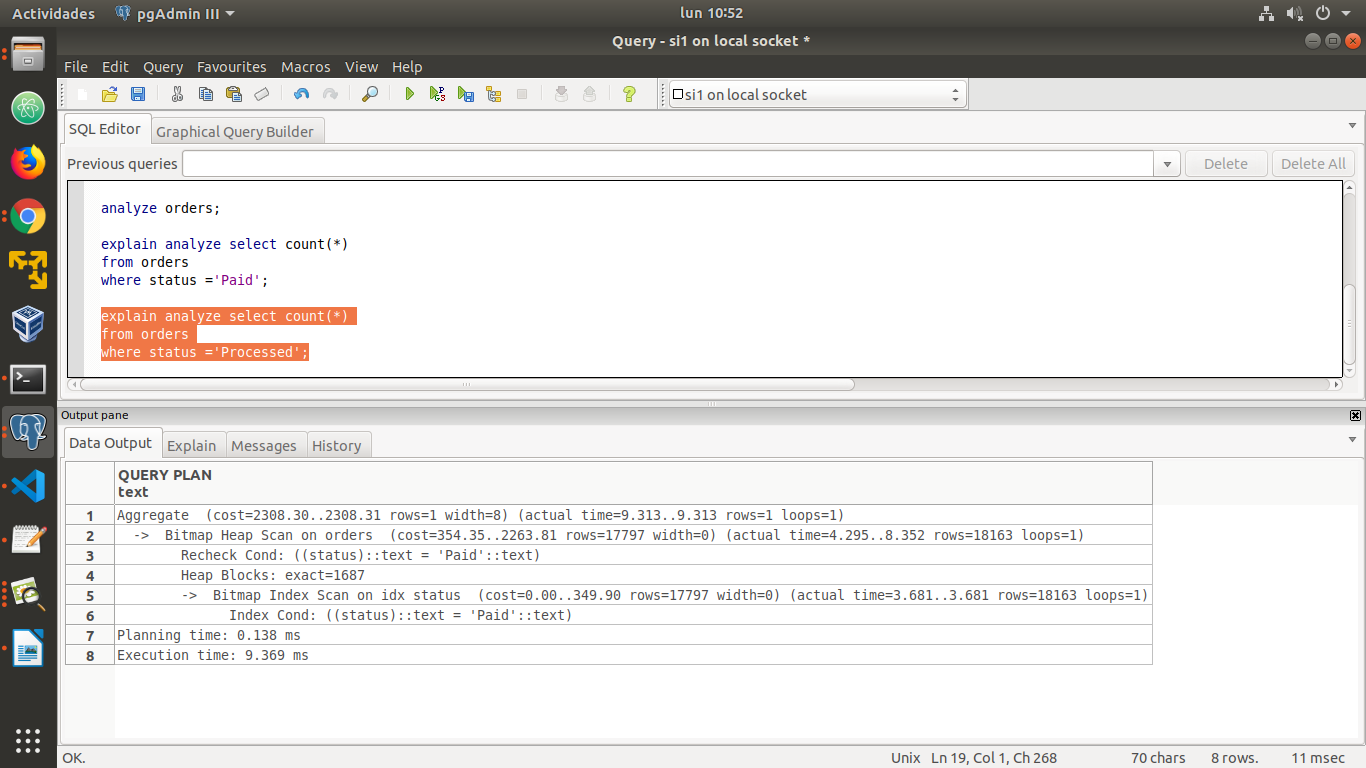


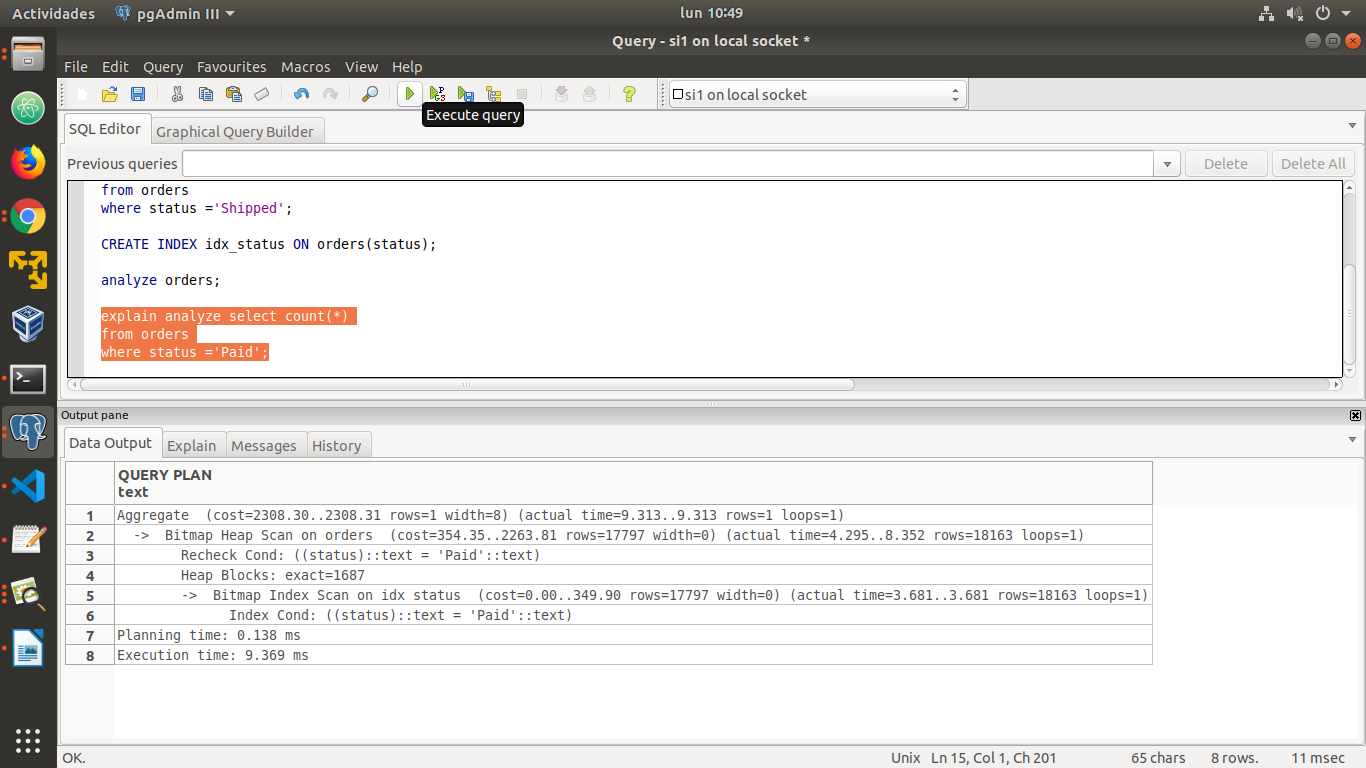


Podemos apreciar que los planes de ejecucion han cambiado y que la primera consulta que se ha visto afectada de manera que ha quedado mejorada; pues su tiempo de ejecucion ahora es inferior. Esto se debe a ¿????????

En el caso de la segunda consulta vemos que tambien ha mejorado el tiempo de ejecucion pero no tan notablemente como en el caso anterior.

También se nos pedía realizar estas otras dos consultas una vez realizadas las estadísticas para compararlas con ellas.





FALTA : Comparar con la planificación de las otras dos consultas proporcionadas y comentar los resultados. 🡪 de estas no se que decir

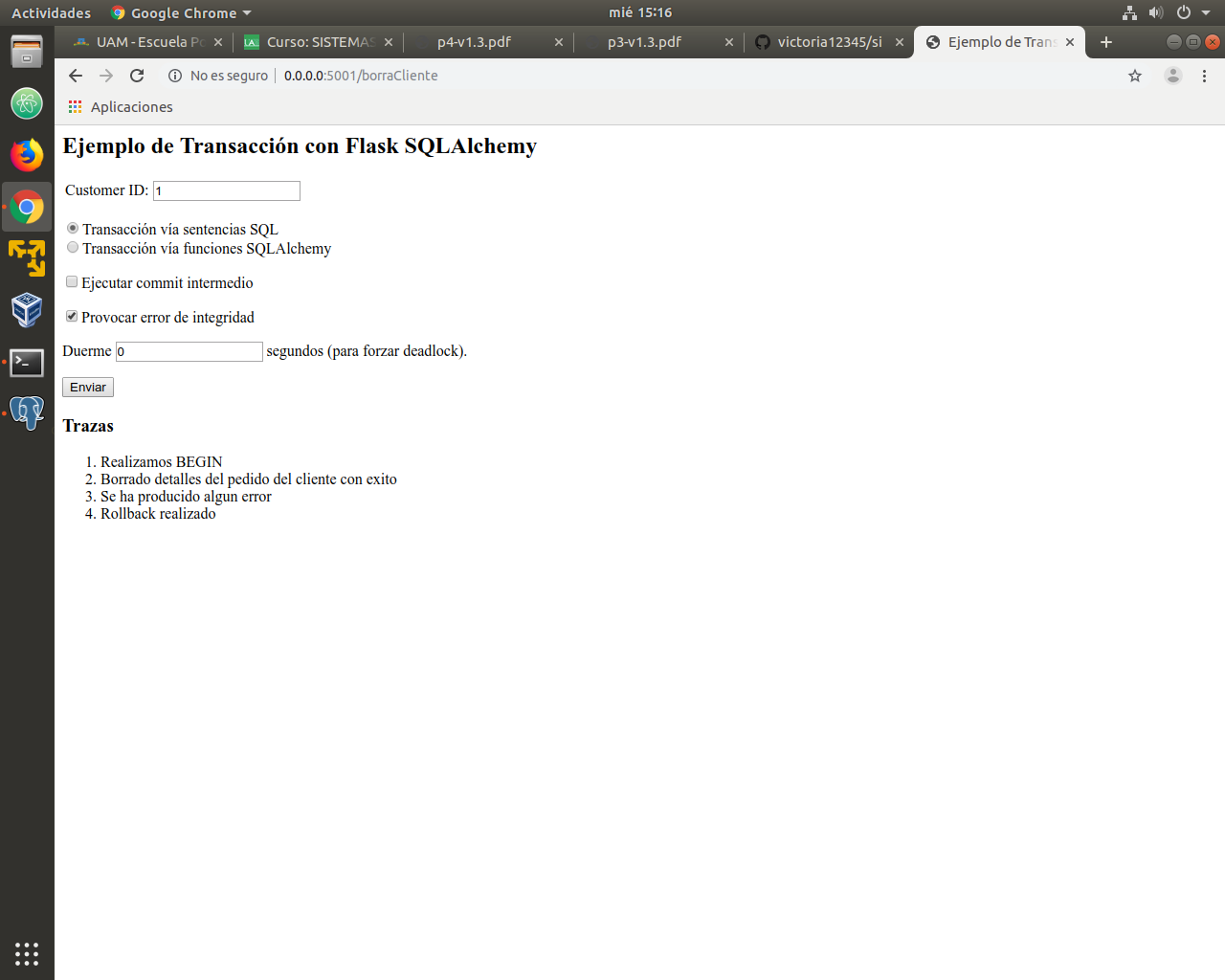
**TRANSACCIONES Y DEADLOCKS**

**APARTADO E**

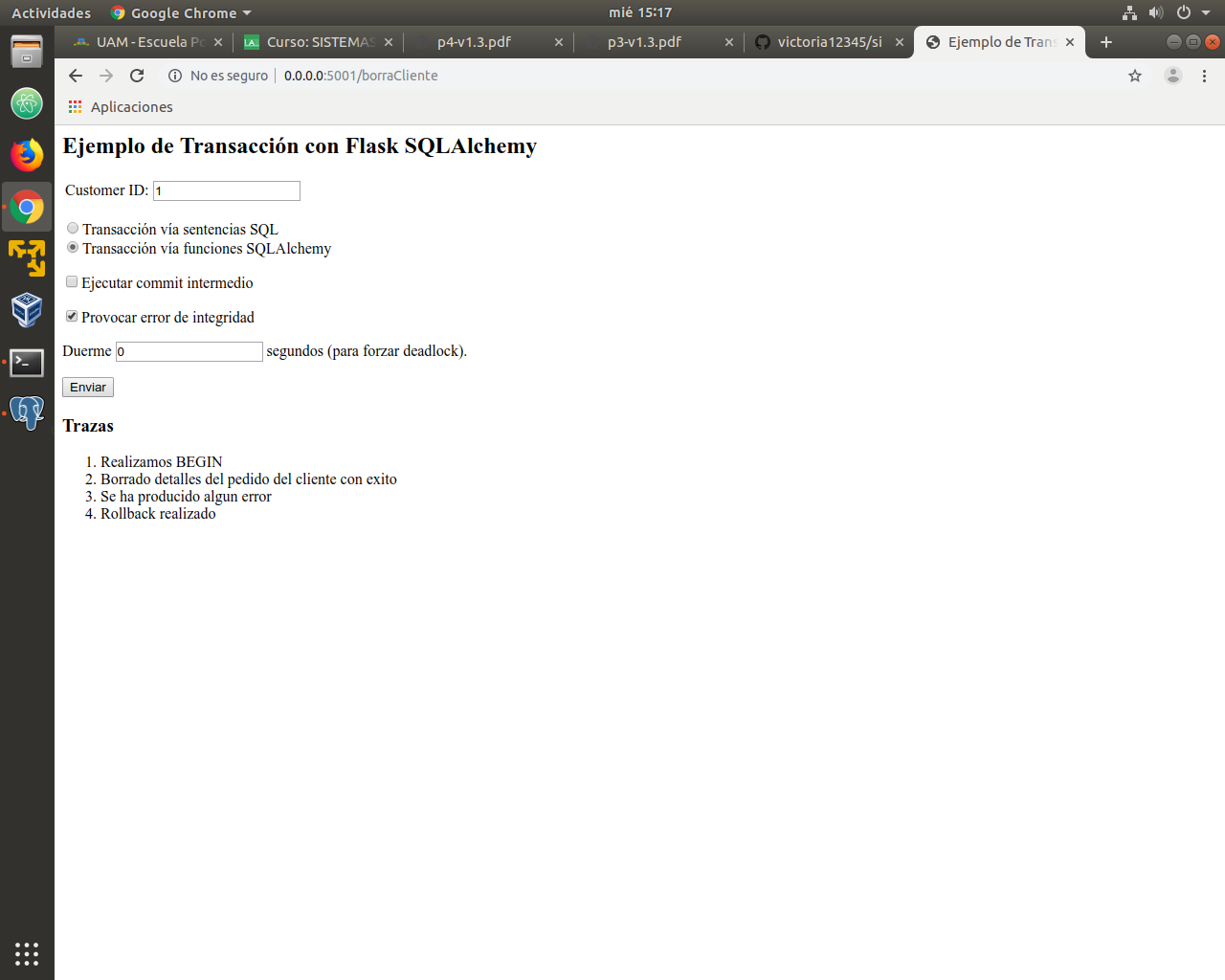
En este apartado estudiaremos las transacciones. Para ello realizaremos la pagina **borraCliente** que ejecuta una transacción que borra un cliente y toda su información (carrito, historial y pedidos con su detalle; es decir, se verán afectadas las tablas *orders, orderdetail* y *customers*)

En la página resultante se muestra la traza de los cambios parciales que vamos realizando y como estos cambios se deshacen al realizar un rollback. Para ello mostramos capturas de pantalla de las pruebas realizadas:

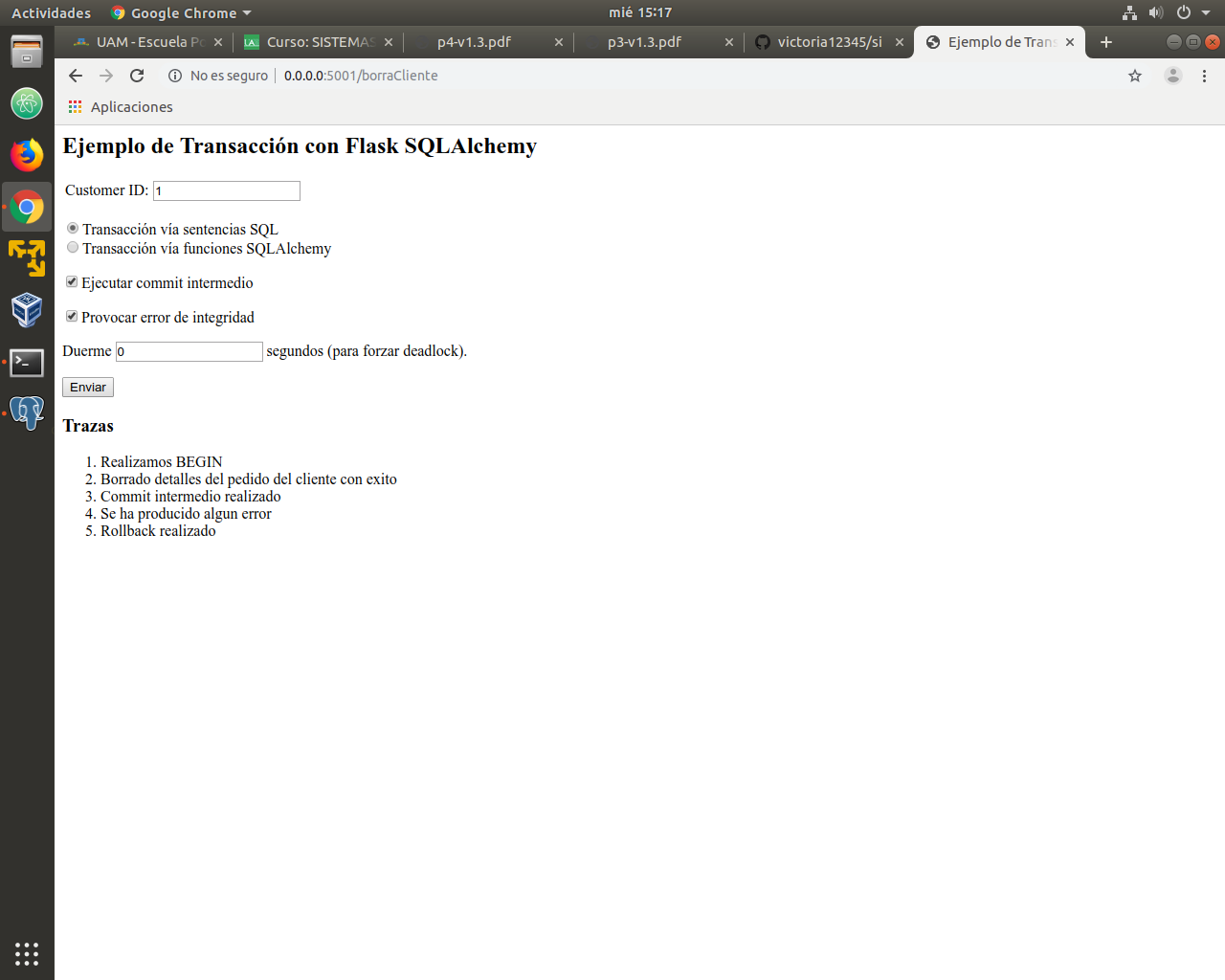
Primero, vemos la traza que realizamos para una transaccion via SQL provocando un error de integridad y sin realizar ningun commit intermedio.



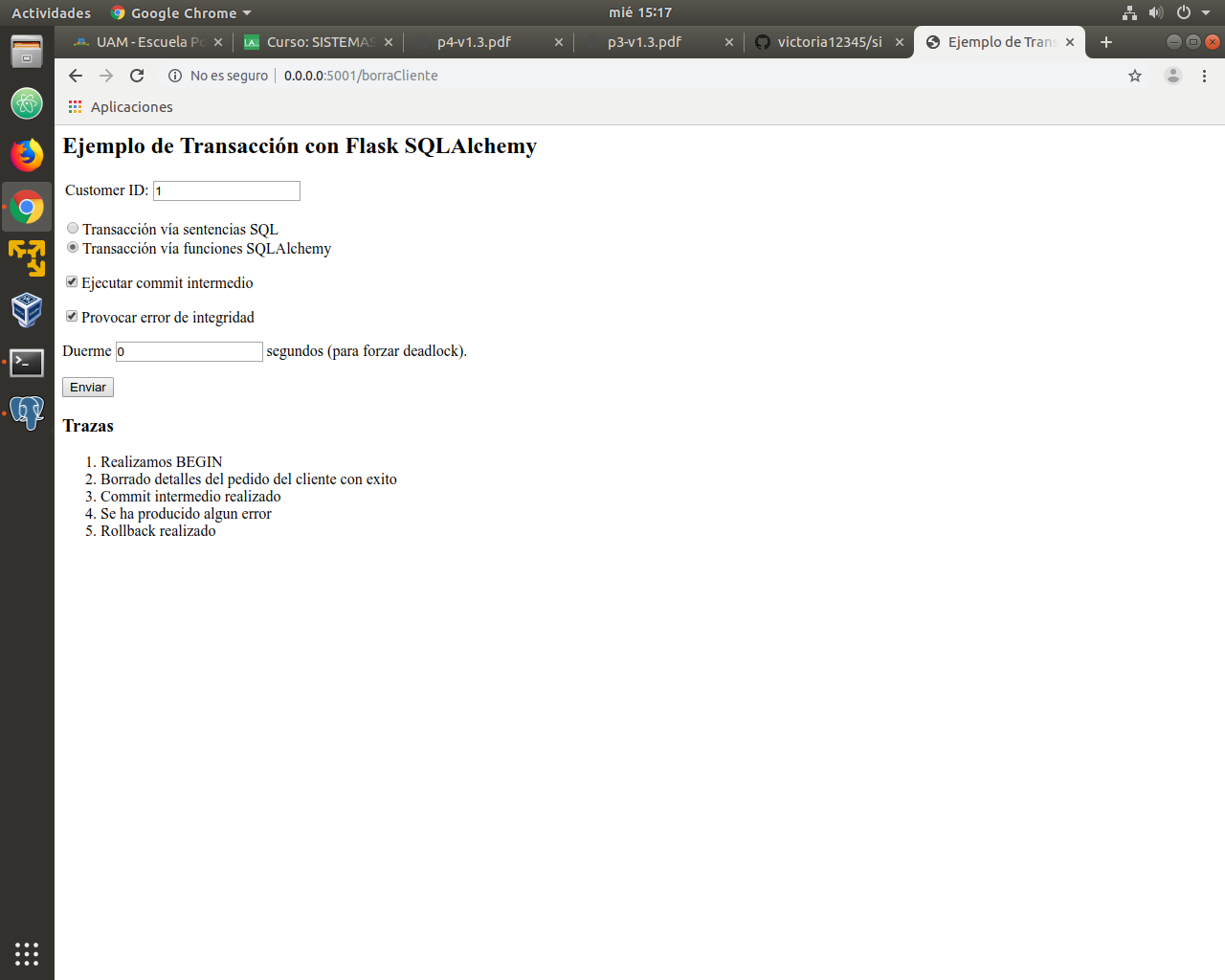
Realizamos esta misma operación pero en este caso para una transaccion via SQLAlchemy y obtenemos la misma traza.



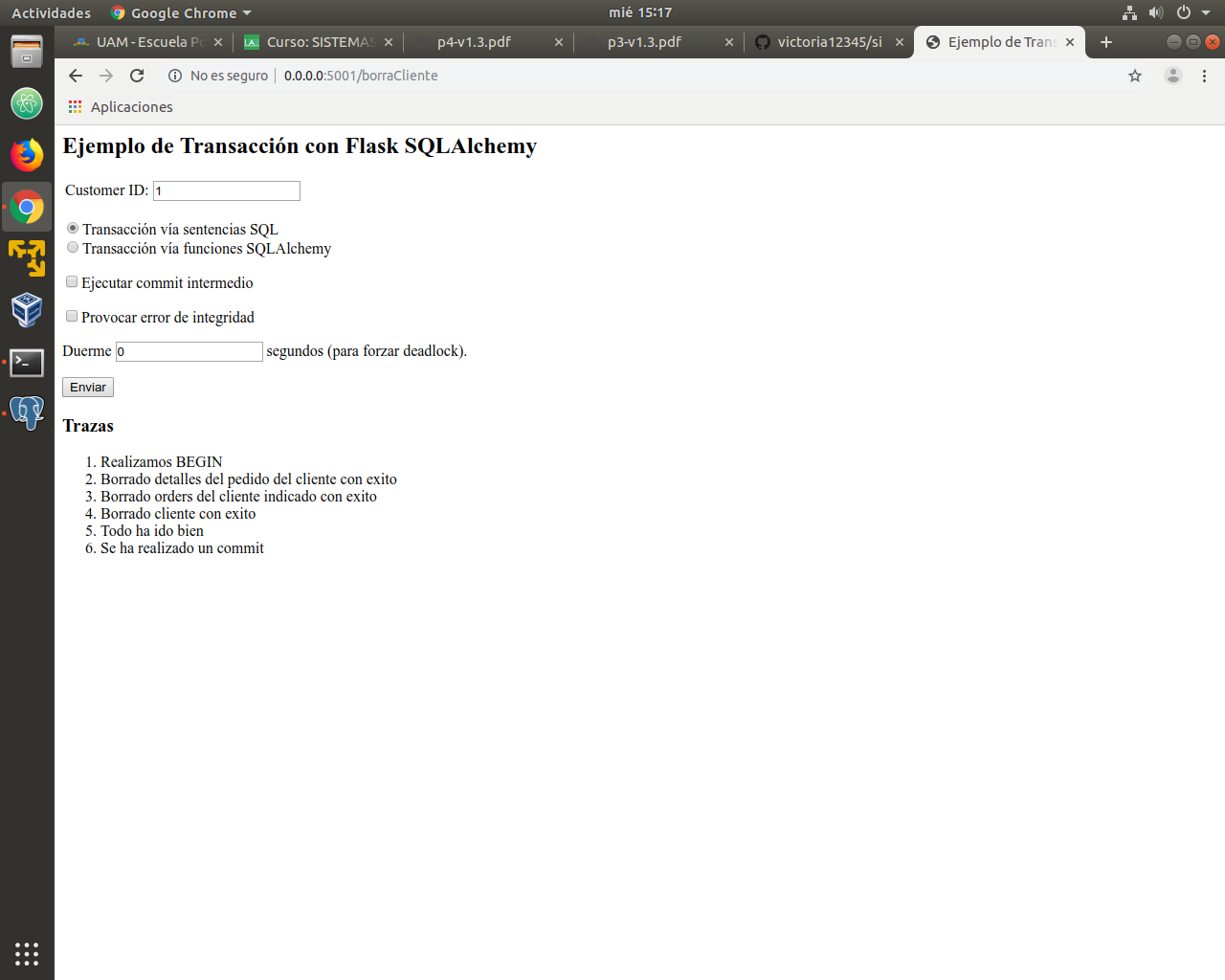
Ahora vemos la traza que realizamos para una transaccion via SQL provocando un error de integridad y realizando un commit intermedio.



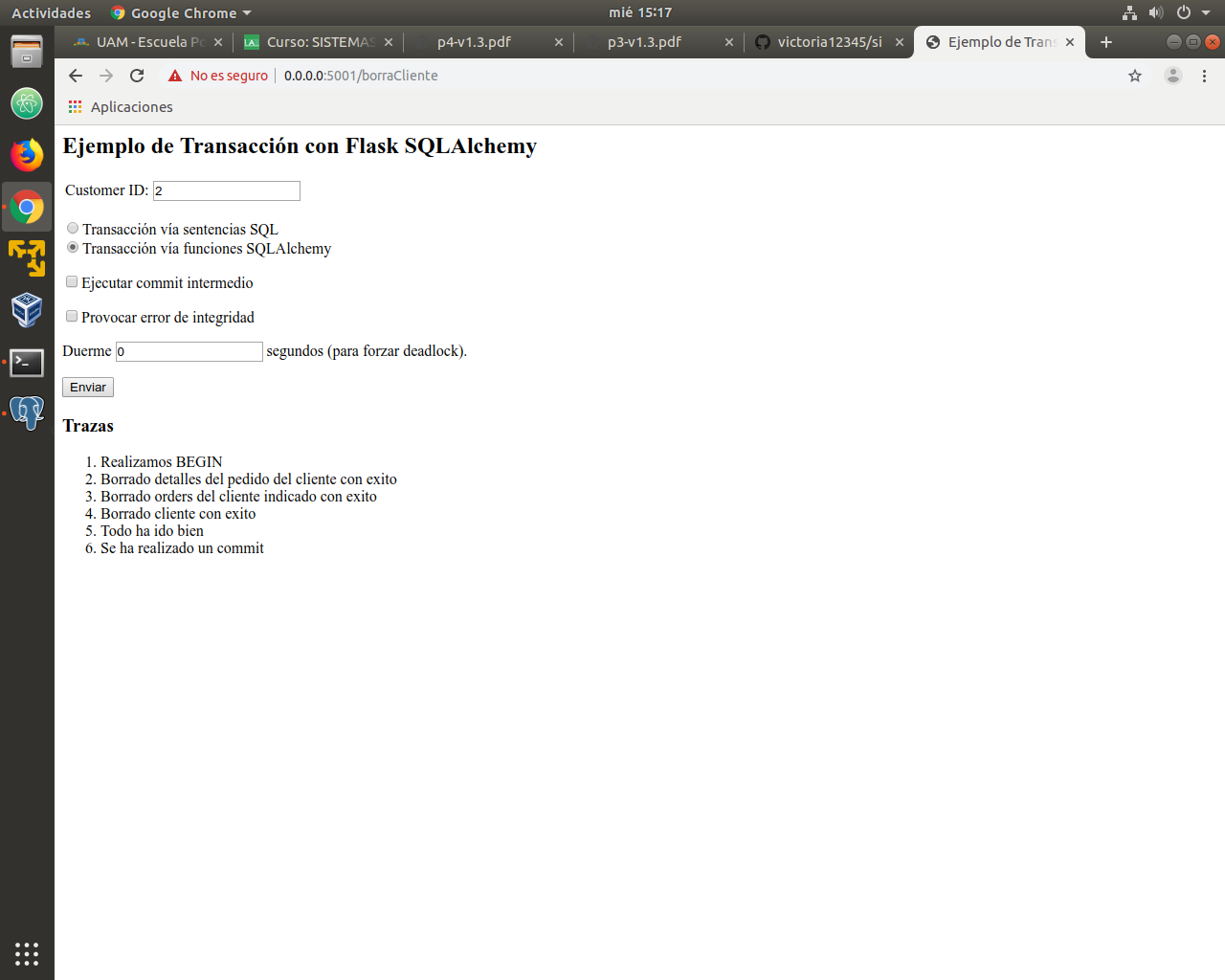
Repetimos la misma operación pero en este caso para una transaccion via SQLAlchemy por lo que obtenemos la misma traza.



En segundo lugar, borraremos correctamente un cliente y veremos la traza obtenida para una transacción via sentencias SQL y sin commit intermedio (con commit intermedio es la misma traza pues como todo se realiza correctamente no se indica ningún commit)



Realizamos la misma operación pero en este caso con una transaccion via SQLAlchemy y para el customer con id 2. Vemos que la traza obtenida es igual que la anterior.



**APARTADO F**

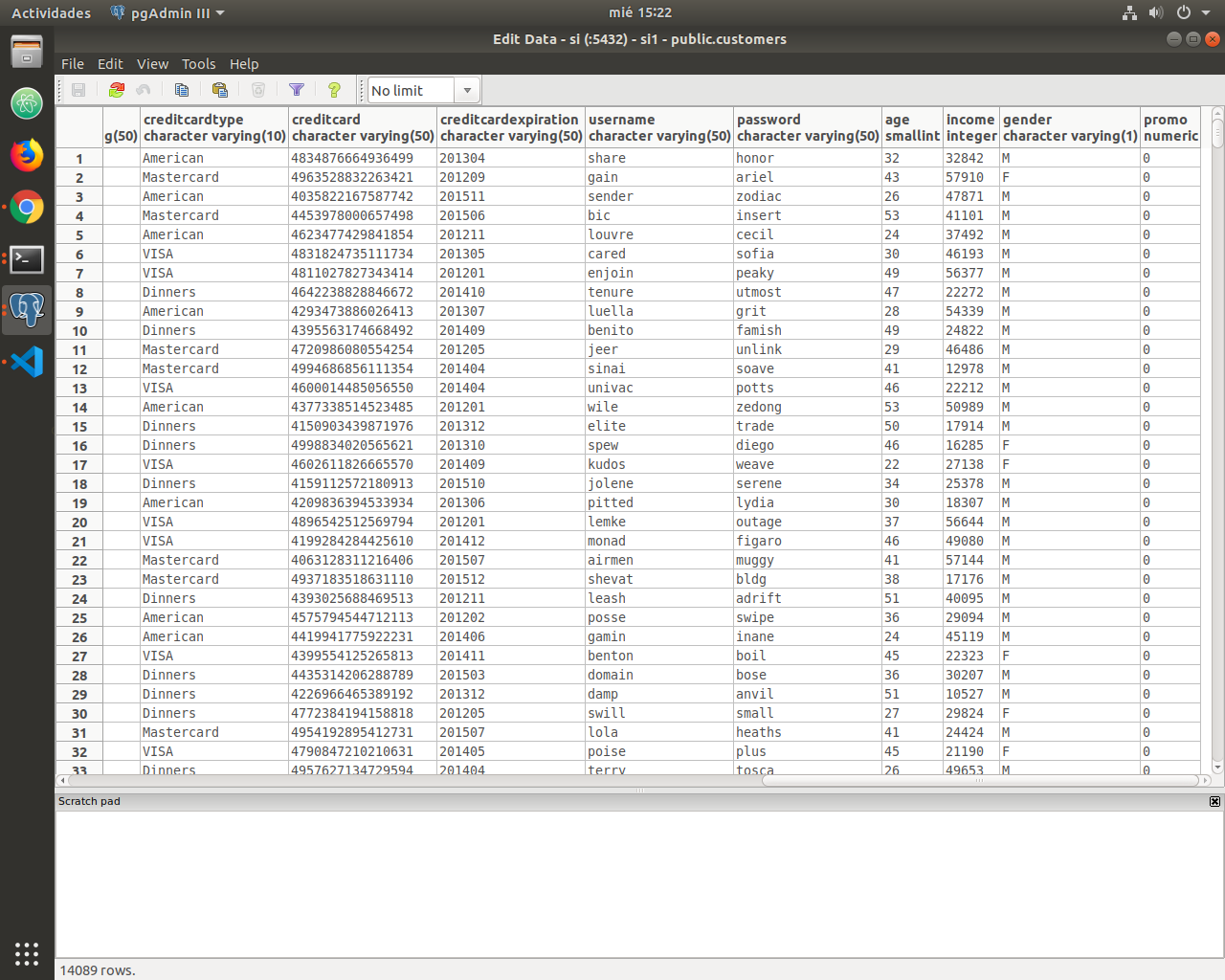
En este apartado estudiaremos los bloqueos y los deadlocks. Para ello tendremos que partir de nuevo de la base de datos limpia y crearemos el script **updPromo.sql** que crea una nueva columna promo en la tabla customers. En nuestro caso hemos puesto que por defecto el descuento sea de 0 mediante la siguiente consulta:

ALTER TABLE customers ADD promo numeric DEFAULT 0;

A continuación creamos un trigger sobre la tabla customers de forma que cuando se altera la columna *promo* de un cliente se le hace un descuento en los artículos de su cesta o carrito del porcentaje indicado en dicha columna sobre el precio de la tabla *products*. Además, en dicho trigger hemos realizado el sleep en --------poner donde---- y hemos insertado el sleep en los momentos correctos en la página **borraCliente**

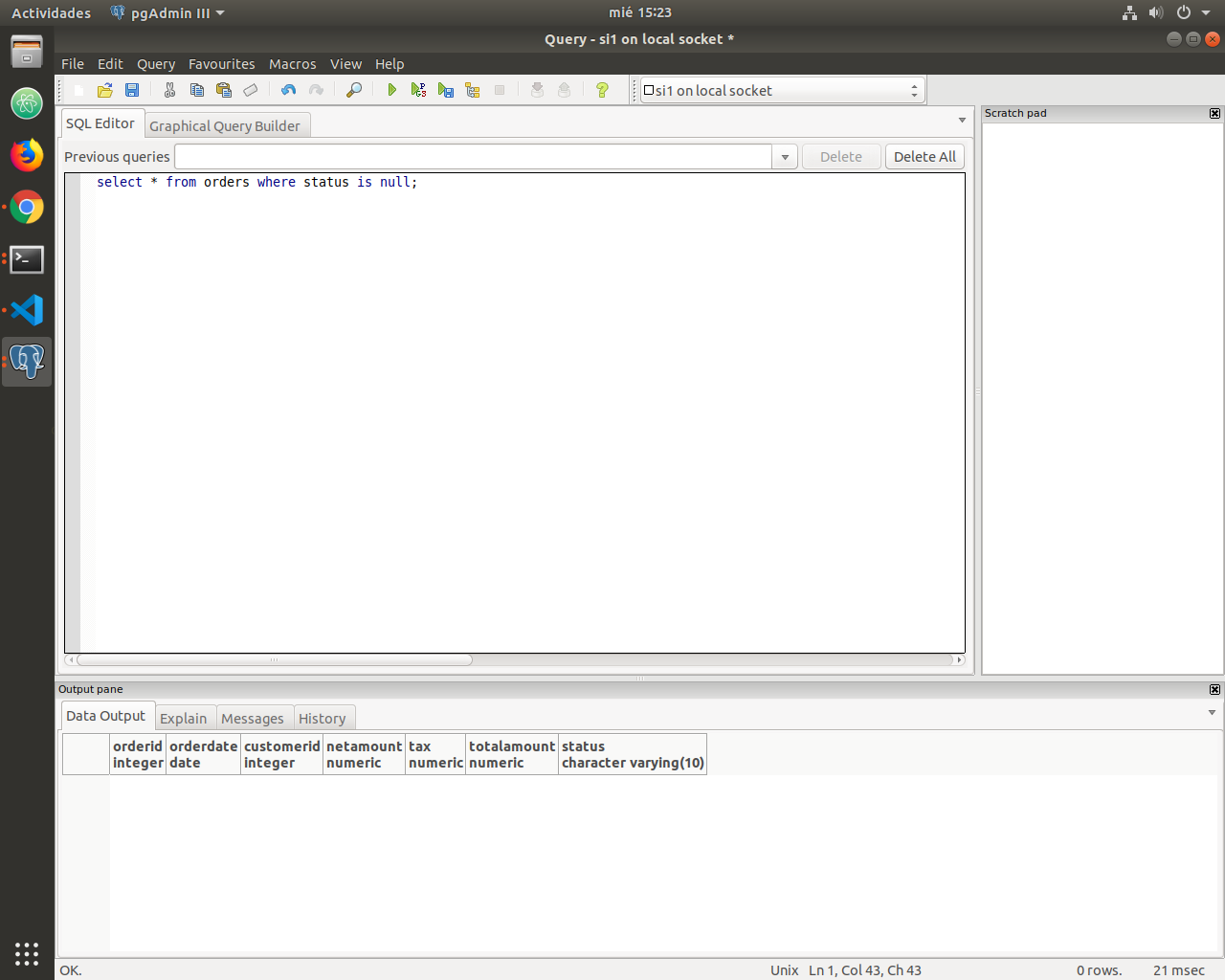
Para comprobar el correcto funcionamiento de este trigger mostramos algunas capturas de pantallas donde aparecen las comprobaciones en PgAdmin.

Primero comprobamos que la columna promo se ha añadido correctamente a nuestra tabla.

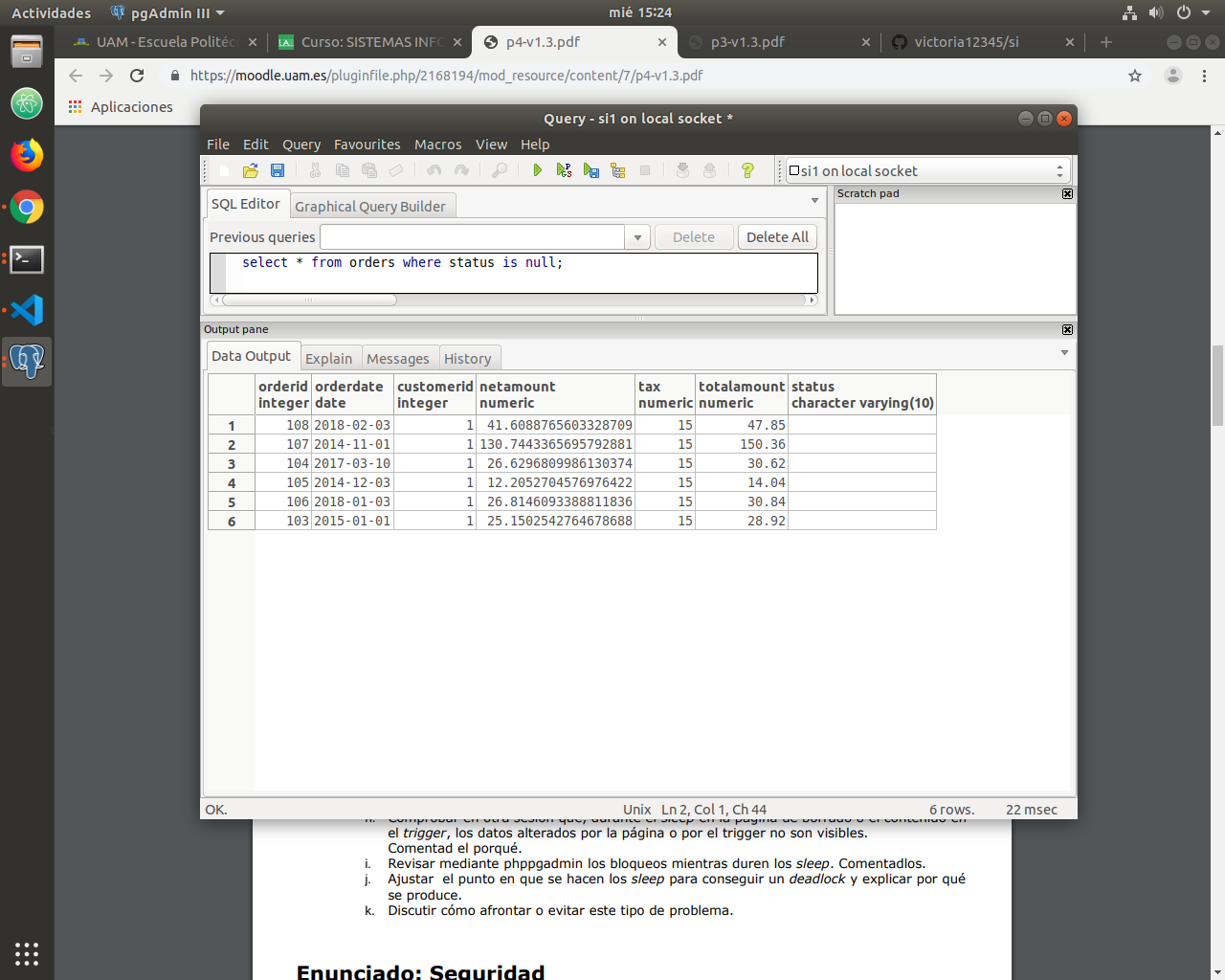


Posteriormente instalaremos el trigger que hemos implementado y realizamos las siguientes consultas:

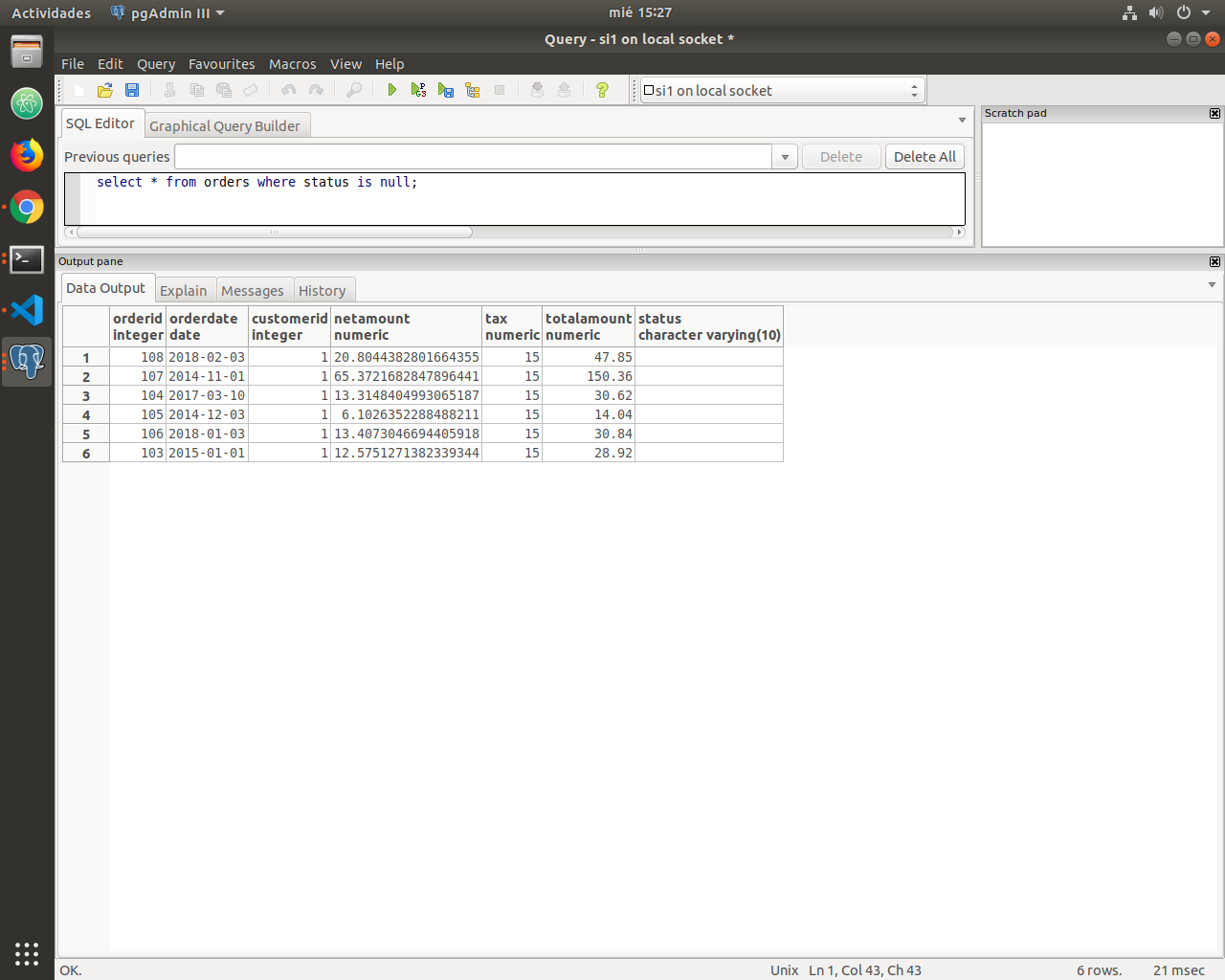
* Comprobamos que no tenemos ninguna cesta a null para poder poner nosotros una, tal y como se dice en el enunciado



* Ponemos la cesta del customerid a null para poder trabjar con este. Para ello realizamos la consulta: UPDATE customers SET status = null WHERE customerid = 1; Obtenemos los siguientes resultados



* Ahora ponemos la promo de este cliente al 50% mediante la consulta: UPDATE customers SET promo = 50 WHERE customerid = 1; Y volvemos a realizar la operación anterior para ver si se ha actualizado netamount.



Vemos que es correcto puesto que tenemos la mitad en netamount para cada compra del carrito.

**SEGURIDAD**

**APARTADO G**

En este apartado el objetivo que tenemos es acceder a nuestro sitio web protegido con usuario y contraseña sin disponer ni de usuario ni de contraseña.

Lo primero que haremos será ver en *routes.py* que realiza la función de validación **xLoginInjection** y nos damos cuenta que hace uso de la función *getCustomer* implementada en *database.py*.

Sabiendo esto, analizaremos esta ultima función, pues es la que contiene la consulta para validar el login. Como podemos ver se realiza una consulta de la siguiente manera:

query="select \* from customers where username='" + username + "' and password='" + password + "'"

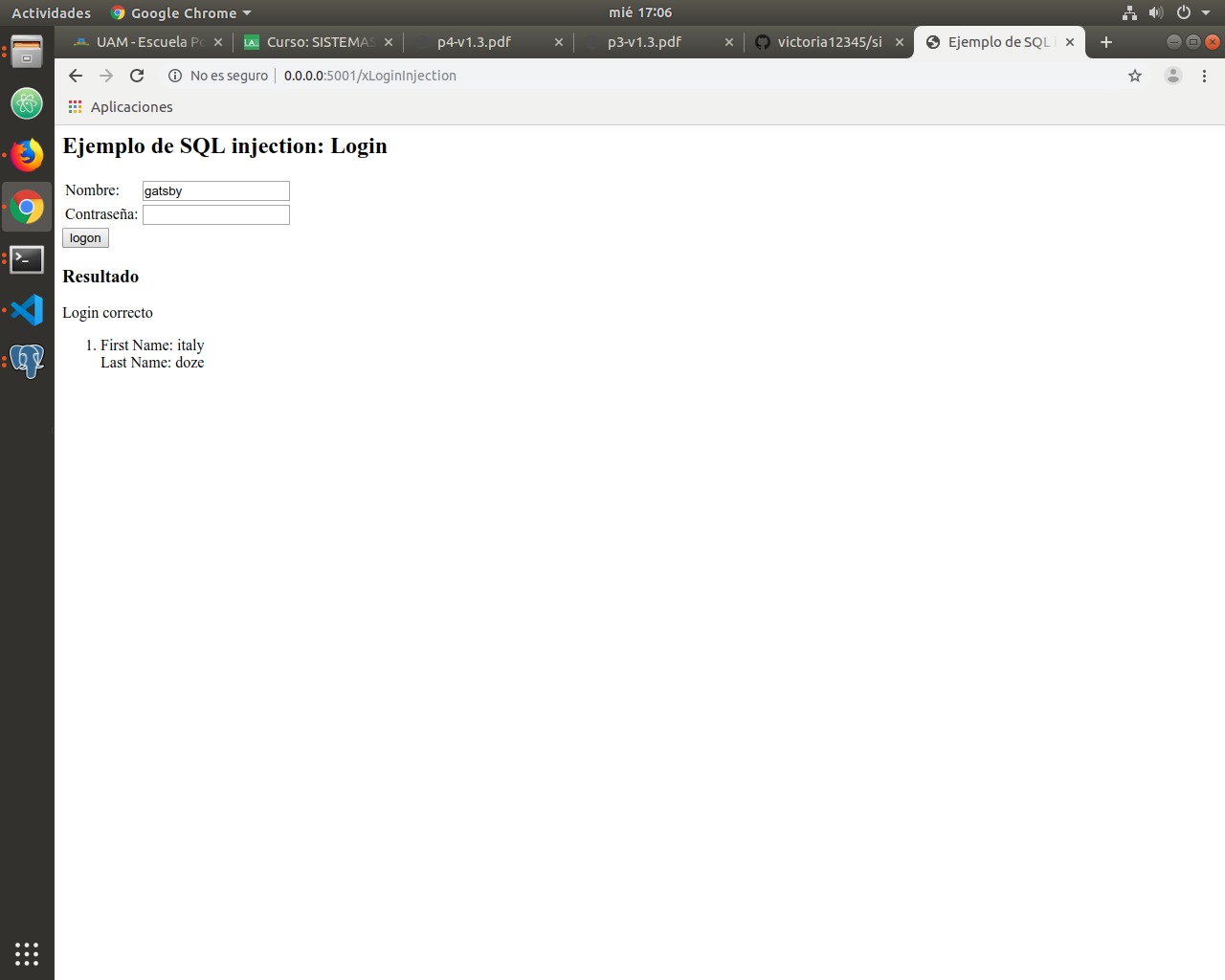
Por lo tanto lo que tenemos que conseguir es que la consulta se pare y se realice una de esta manera:

query="select \* from customers where username='" + username + "'"

Para ello lo único que tenemos que hacer es introducir en nuestra aplicación el nombre gatsby’;-- que hace que se compone de un operador mal intencionado (en este caso es ’; ) y de la manera de comentar en sql que hara que no se realice la segunda parte de la consulta de login.

Si nos fijamos en *getCustomer* esta función devolverá el nombre y el apellido del usuario introducido.

Se muestra a continuación un ejemplo del resultado:

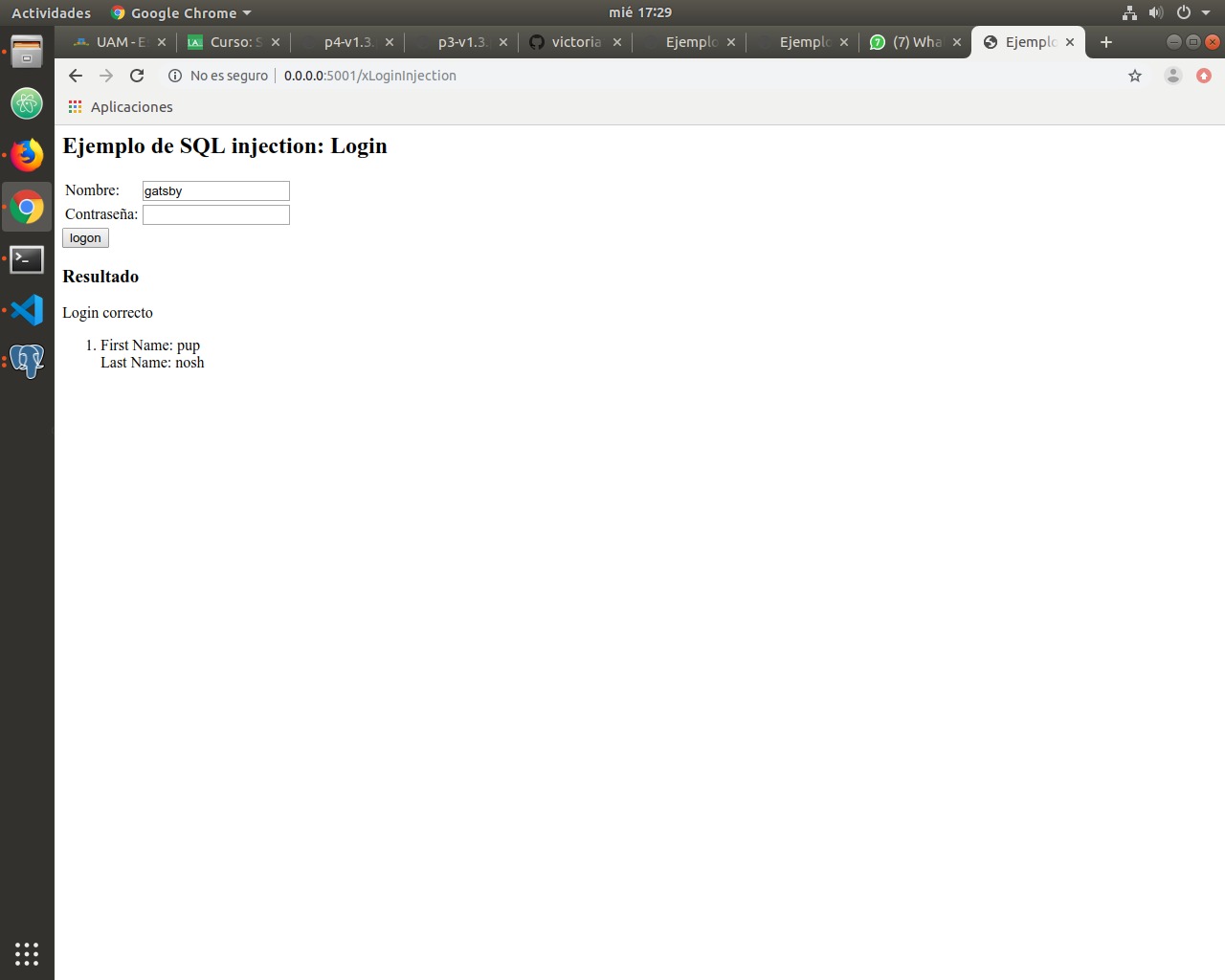


Ahora tendremos que realizarla misma operación que en el ejercicio anterior pero sin conocer el nombre de usuario. Para ello queremos una consulta que sea “siempre cierta” con lo cual bastaría con tener una consulta que fuera del tipo:

query="select \* from customers where username='g' or 1=1;

Puesto que la condición 1=1 siempre es cierta, en esta consulta siempre tendremos resultado y por lo tanto se nos permitirá realizar el login. Con lo cual, solo tenemos que introducir en nuestra aplicación el nombre g’ or 1=1;--.

Se muestra a continuación un ejemplo del resultado:



Para realizar este apartado hemos buscado en internet como evitar estas inyecciones SQL y entre las multiples respuestas que hemos encontrado cabe destacar una, pues son las que nosotras hemos visto a lo largo de la asignatura y de esta práctica.

Una de las medidas es usar consultas preparadas ya que los valores de los parámetros son transmitidos después usando un protocolo diferente. Si la `plantilla de la sentencia original no es derivada de un input externo, los ataques SQL injection no pueden ocurrir.

(También aparecía información sobre cómo hacerlo para diferentes leguajes como PHP)