**Titulación: Grado en Ingeniería Informática e Ingeniería en Sistemas de Información**

**Curso: 2020-2021. Convocatoria Ordinaria de Junio**

**Asignatura: Bases de Datos Avanzadas – Laboratorio**

## **Practica 2: Carga Masiva de Datos, Procesamiento y Optimización de Consultas**

**ALUMNO 1:**

**Nombre y Apellidos:** Guillermo Díaz García

**DNI:** 09072643V

**ALUMNO 2:**

**Nombre y Apellidos:** Borja Ordóñez Bello

**DNI:** 06021787L

**Fecha:**

**Profesor Responsable: Iván González Diego**

Mediante la entrega de este fichero los alumnos aseguran que cumplen con la normativa de autoría de trabajos de la Universidad de Alcalá, y declaran éste como un trabajo original y propio.

En caso de ser detectada copia, se calificará la asignatura como Suspenso – Cero.

## **Plazos**

Tarea en Laboratorio: Semana 22 de Marzo, Semana 6 de Abril, Semana 12 de Abril, semana 19 de Abril y semana 26 de Abril.

Entrega de práctica: Semana 4 de Mayo (Martes). Aula Virtual

Documento a entregar: Este mismo fichero con las respuestas a las cuestiones planteadas y el programa que genera los datos de carga de la base de datos. No se pide el script de carga de los datos en bruto de la base de datos. Se entregará en un ZIP comprimido llamado: **DNI'sdelosAlumnos\_PECL2.zip**

**AMBOS ALUMNOS DEBEN ENTREGAR EL FICHERO EN LA PLATAFORMA.**

## **Introducción**

El contenido de esta práctica versa sobre la monitorización de la base de datos, manipulación de datos, técnicas para una correcta gestión de los mismos, así como tareas de mantenimiento relacionadas con el acceso y gestión de los datos. También se trata el tema de procesamiento y optimización de consultas realizadas por PostgreSQL (13.x). Se analizará PostgreSQL en el proceso de carga masiva y optimización de consultas.

En general, la monitorización de la base de datos es de vital importancia para la correcta implantación de una base de datos, y se suele utilizar en distintos entornos:

* Depuración de aplicaciones: Cuando se desarrollan aplicaciones empresariales no se suele acceder a la base de datos a bajo nivel, sino que se utilizan librerías de alto nivel y mapeadores ORM (Hibernate, Spring Data, MyBatis…) que se encargan de crear y ejecutar consultas para que el programador pueda realizar su trabajo más rápido. El problema en estos entornos está en que se pierde el control de qué están haciendo las librerías en la base de datos, cuántas consultas ejecutan, y con qué parámetros, por lo que la monitorización en estos entornos es vital para saber qué consultas se están realizando y poder optimizar la base de datos y los programas en función de los resultados obtenidos.
* Entornos de prueba y test de rendimiento: Cuando una base de datos ha sido diseñada y se le cargan datos de prueba, una de las primeras tareas a realizar es probar que todos los datos que almacenan son consistentes y que las estructuras de datos dan un rendimiento adecuado a la carga esperada. Para ello se desarrollan programas que simulen la ejecución de aquellas consultas que se consideren de interés para evaluar el tiempo que le lleva a la base de datos devolver los resultados, de cara a buscar optimizaciones, tanto en la estructura de la base de datos como en las propias consultas a realizar.
* Monitorización pasiva/activa en producción: Una vez la base de datos ha superado las pruebas y entra en producción, el principal trabajo del administrador de base de datos es mantener la monitorización pasiva de la base de datos. Mediante esta monitorización el administrador verifica que los parámetros de operación de la base de datos se mantienen dentro de lo esperado (pasivo), y en caso de que algún parámetro salga de estos parámetros ejecuta acciones correctoras (reactivo). Así mismo, el administrador puede evaluar nuevas maneras de acceso para mejorar aquellos procesos y tiempos de ejecución que, pese a estar dentro de los parámetros, muestren una desviación tal que puedan suponer un problema en el futuro (activo).

Para la realización de esta práctica será necesario generar una muestra de datos de cierta índole en cuanto a su volumen de datos. Para ello se generarán, dependiendo del modelo de datos suministrado, para una base de datos denominada **NETFLIX**. Básicamente, la base de datos guarda información sobre las cuentas de los usuarios que se suscriben a la plataforma NETFLIX para poder acceder a los contenidos que ofrece, en este caso, series y películas. Hay una serie de suscripciones definidas en el sistema y el usuario en todo momento esta suscrito a una de ellas. Cada mes que el usuario está dado de alta en la plataforma, se produce un pago de la cantidad por la que está suscrito. Esa suscripción del usuario le permite acceder solamente a los contenidos de películas y series que proporciona esa suscripción. Por último, cada uno de los contenidos de la plataforma tiene asociado una serie de géneros de tal manera que es posible clasificar los contenidos por géneros.

Los datos referidos al año 2020 que hay que generar deben de ser los siguientes:

- Hay 2.000.000 de usuarios dados de alta, donde cada uno tiene una cuenta.

- Existe un total de 100 tipos de suscripción diferente.

- Por cada usuario se almacenan los pagos realizados mensualmente durante el año 2020. Por lo tanto, hay que generar un total de 12 pagos por cada usuario.

- Existen un total de 20.000.000 de contenidos. El decidir si un contenido es película o serie se debe de hacer de manera aleatoria por medio de la función random.

- Cada contenido se asocia aleatoriamente a un tipo de suscripción.

- Existen 20 géneros de contenidos diferentes, y el número de géneros que tiene asociado un contenido es un valor aleatorio que va entre 1 y 6.

- El campo duración de la tabla películas debe tener valores aleatorios entre 80 y 120 minutos.

- El campo capítulos de la tabla series debe tener valores aleatorios entre 10 y 20.

- El campo temporadas de la tabla series debe tener valores aleatorios entre 1 y 15.

- El campo valoración de los contenidos debe tener valores aleatorios entre 1 y 10.

- El campo fecha de la tabla pago debe tener los días generados de forma aleatoria para cada mes de pago de la suscripción de cada usuario.

## **Actividades y Cuestiones**

Cuestión 1: ¿Tiene el servidor postgres un recolector de estadísticas sobre el contenido de las tablas de datos? Si es así, ¿Qué tipos de estadísticas se recolectan y donde se guardan?

El servidor PostgreSQL tiene un recolector de estadísticas sobre el contenido de las tablas de datos que tiene un parámetro llamado ‘track\_counts’ que recopila estadísticas sobre los accesos a la tabla y al índice, se pueden acceder a las estadísticas a través de las vistas ‘pg\_stat’ y ‘pg\_statio’.

El recopilador de estadísticas transmite la información recopilada a otros procesos de PostgreSQL a través de archivos temporales. Estos archivos se almacenan en el directorio nombrado por el parámetro ‘stats\_temp\_directory’, ‘pg\_stat\_tmp’ de forma predeterminada.

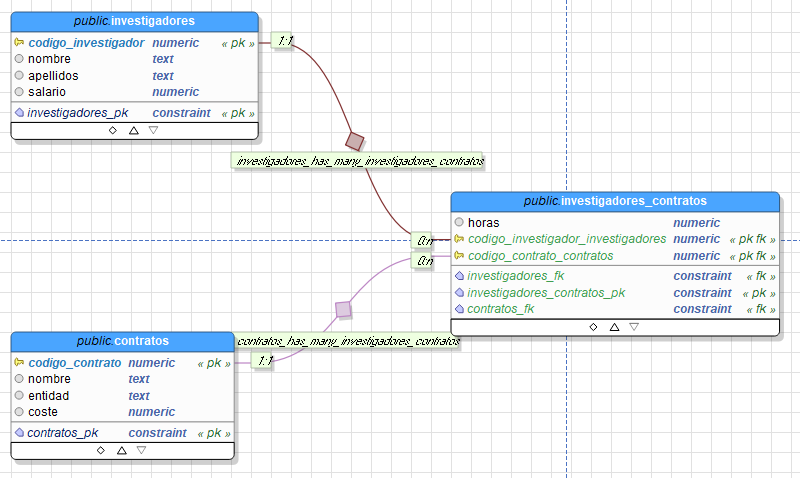
Cuestión 2: Crear una nueva base de datos llamada **investigar** y que tenga las siguientes tablas con los siguientes campos y características:

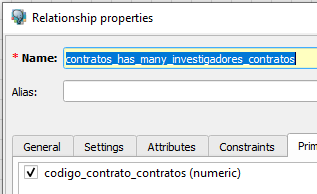
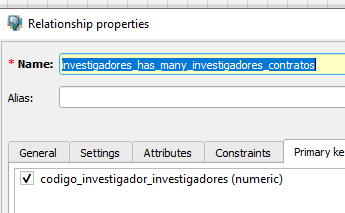
* investigadores(codigo\_investigador tipo numeric PRIMARY KEY, nombre tipo text, apellidos tipo text, salario tipo numeric)
* contratos(codigo\_contrato tipo numeric PRIMARY KEY, nombre tipo text, entidad tipo text, coste tipo numeric)
* investigadores\_contratos(codigo\_investigador tipo numeric que sea FOREIGN KEY del campo codigo\_investigador de la tabla investigadores con restricciones de tipo RESTRICT en sus operaciones, codigo\_contrato tipo numeric que sea FOREIGN KEY del campo codigo\_contrato de la tabla contratos con restricciones de tipo RESTRICT en sus operaciones, horas de tipo numeric. La PRIMARY KEY debe ser compuesta de codigo\_investigador y codigo\_contrato.

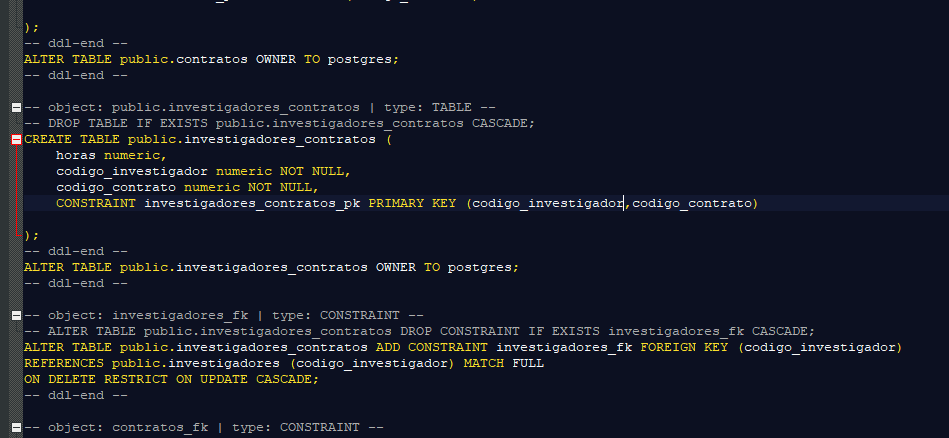
Se pide:

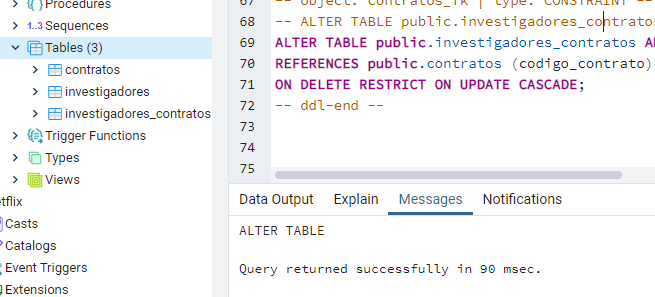
* Indicar el proceso seguido para generar esta base de datos.
* Cargar la información del fichero datos\_investigadores.csv, datos\_contratos.csv y datos\_investigadores\_contratos.csv en dichas tablas de tal manera que sea lo más eficiente posible.
* Indicar los tiempos de carga.

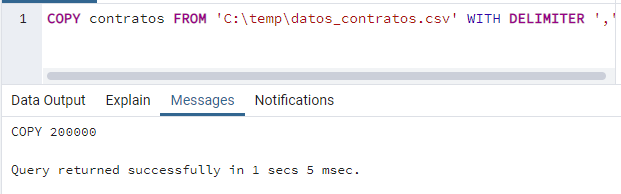
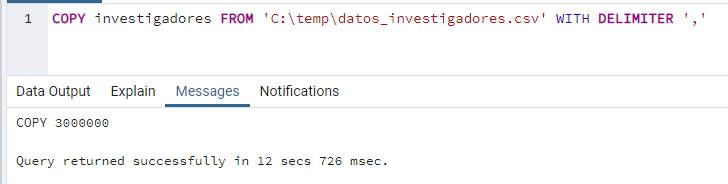
**Proceso para generar base de datos:**  
Accedemos a pgModeler, a continuación le damos un nombre a la base de datos, para ello en el panel del lado izquierdo seleccionamos la opción ‘Properties’ y nos aparecerá una nueva ventana en la que le indicaremos el nombre que es ‘investigar’.

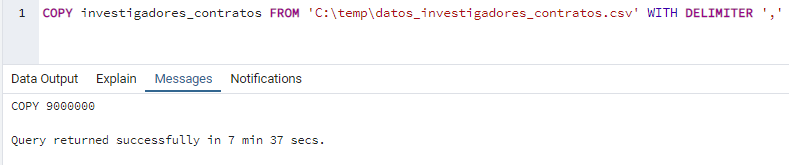
Después creamos el esquema en pgModeler, como aprendimos en el primer cuatrimestre:  




Una vez finalizado los pasos previos, exportamos el resultado a sql y nos aseguramos de que las columnas tengan el nombre correcto:  


Y por último ejecutamos el .sql en pgAdmin:  


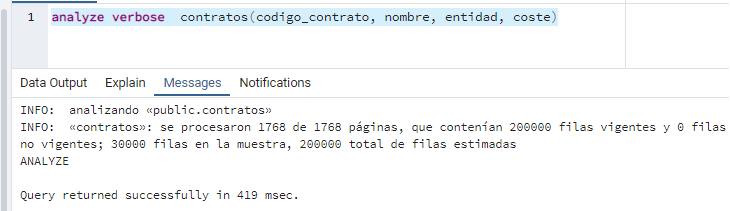
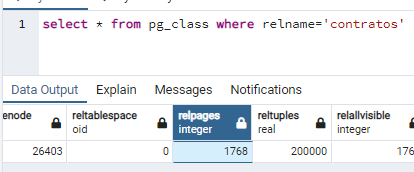
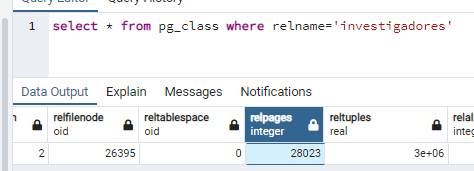
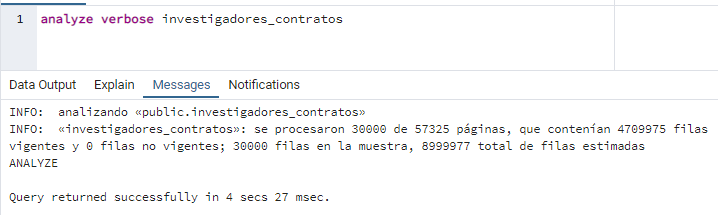
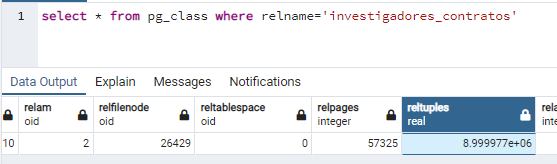
**Proceso de importado de datos:**  
Utilizamos COPY para que sea más eficiente que varias sentencias de INSERT, de la siguiente manera:  
  




Como se puede ver, las copias han tardado 1 segundo, 12 segundos y 7 minutos 37 segundos respectivamente.

Cuestión 3: Mostrar las estadísticas obtenidas en este momento para cada tabla. ¿Qué se almacena? ¿Son correctas? Si no son correctas, ¿cómo se pueden actualizar?

Seleccionando una tabla mientras miramos la pestaña de estadísticas podemos ver los datos de cada una de ellas:

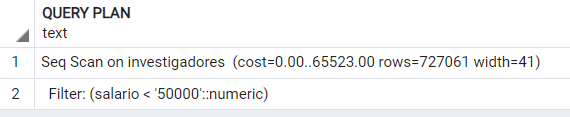
* Contratos:  
  Utilizando el siguiente comando:  
  ANALYZE VERBOSE contratos(codigo\_contrato, nombre, entidad, coste)  
  podemos ver los siguientes datos:  
    
  De aquí sacamos que se utilizan 1.768 bloques para almacenar las 200.000 tuplas activas, que no hay tuplas “no vigentes” (borradas pero sin purgar), y que se han utilizado 30.000 tuplas como muestra para interpolar los datos.  
  Si comparamos este resultado con otras formas de obtener esos datos, vemos que son correctos. Por ejemplo, a través de pg\_class podemos ver que en efecto se utilizan 1.768 bloques, y que hay 200.000 tuplas activas:  
  
* Investigadores:  
  Utilizamos la misma metodología:  
    
  Podemos ver que se utilizan 28.023 bloques, que almacenan 3.000.000 tuplas.  
  De nuevo comparamos con pg\_class y vemos que los datos son los mismos:  
  
* Investigadores\_contratos:  
    
  Como se puede ver, en este caso tenemos que sólo se han procesado 30.000 de los 57.325 bloques que ocupa la tabla. En los bloques analizados se han encontrado 4.709.975 tuplas, por lo que se interpola que el total son 8.999.977 tuplas en total.  
  De nuevo pg\_class confirma estos datos:  
  

En caso de que no fuesen correctas las estadísticas seleccionaremos una tabla, base de datos, índice, etc. y hacemos click derecho sobre este y en las opciones que nos aparece seleccionamos ‘Refresh’ y actualizaría las estadísticas.

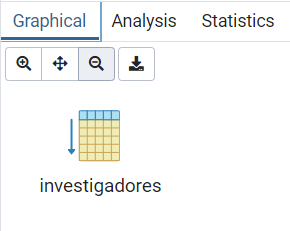
Cuestión 4: Aplicar el comando EXPLAIN a una consulta que obtenga la información de los investigadores con salario de menos de 50.000 euros. ¿Son correctos los resultados del comando EXPLAIN? ¿Por qué? Comparar con lo que se obtendría con el procedimiento visto en teoría.

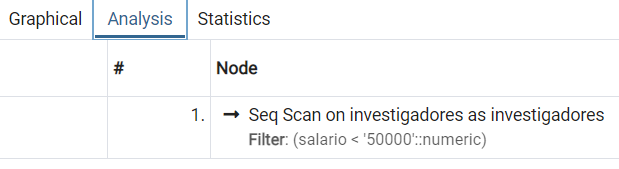
El comando empleado para la consulta sería: EXPLAIN SELECT \* FROM investigadores WHERE salario<50000;.

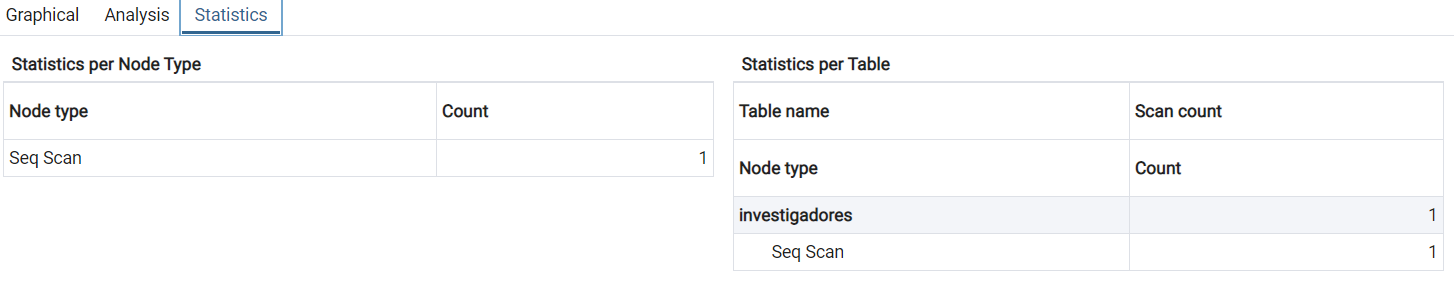
El resultado nos dice que le ha costado 65.523, ha impreso 727.061 filas y la salida tiene un tamaño de 41 bytes.



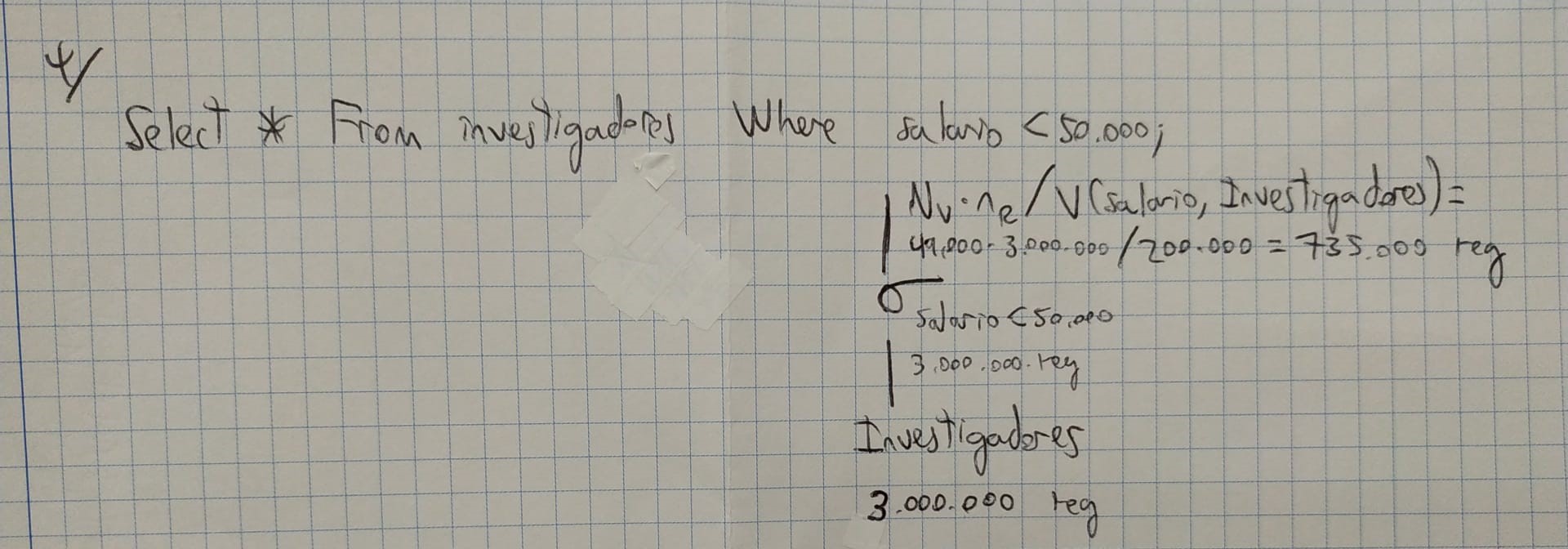
Si pulsamos F7 con el comando: SELECT \* FROM investigadores WHERE salario<50000; veremos el resultado gráficamente y nos dice que ha sido un escaneo secuencial.







Lo calculamos teóricamente:



El resultado nos devuelve que hemos obtenido 735.000 tuplas. Si lo comparamos con la consulta realizada anteriormente vemos que este número se aproxima al que nos da pgAdmin (727.061).

Si ahora calculamos los costes veremos:



El resultado nos devuelve que hemos obtenido 15.076 bloques. Si lo comparamos con la consulta realizada anteriormente vemos que este número no se aproxima al que nos da pgAdmin (65.523).

Comparando PostgreSQL con lo realizado teóricamente vemos que hemos seguido el mismo procedimiento, un escaneo secuencial con una selección.

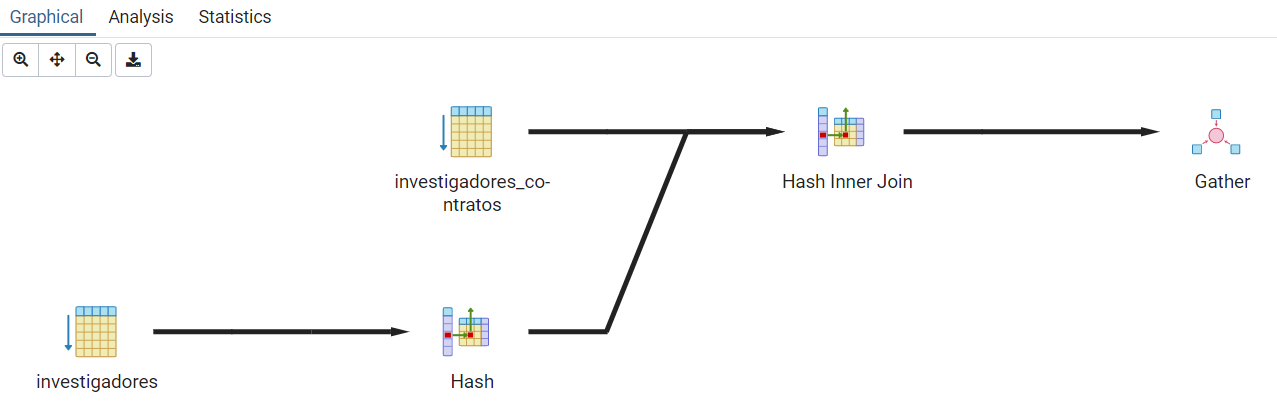
Cuestión 5: Aplicar el comando EXPLAIN a una consulta que obtenga el nombre de los investigadores en los cuales están asignados a contratos por 8 horas o tienen un salario de 30.000 euros. ¿Son correctos los resultados del comando EXPLAIN? ¿Por qué? Comparar con lo que se obtendría con el procedimiento visto en teoría.

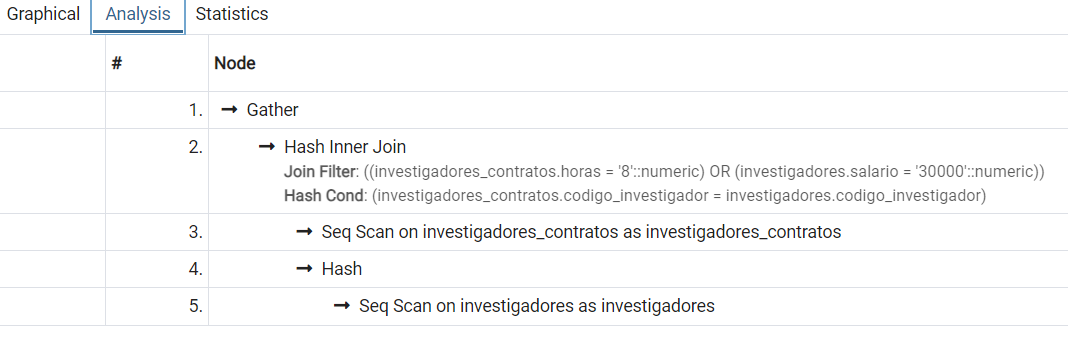
El comando empleado para la consulta sería: EXPLAIN SELECT nombre FROM investigadores NATURAL JOIN investigadores\_contratos WHERE horas=8 OR salario=30000;.

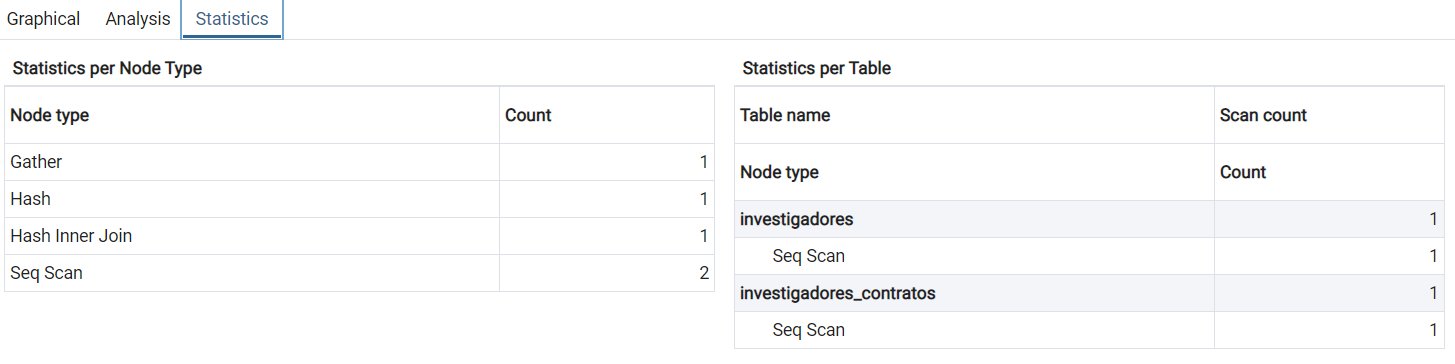
El resultado nos dice que le ha costado 253.843,03, ha impreso 383.144 filas y la salida tiene un tamaño de 13 bytes.



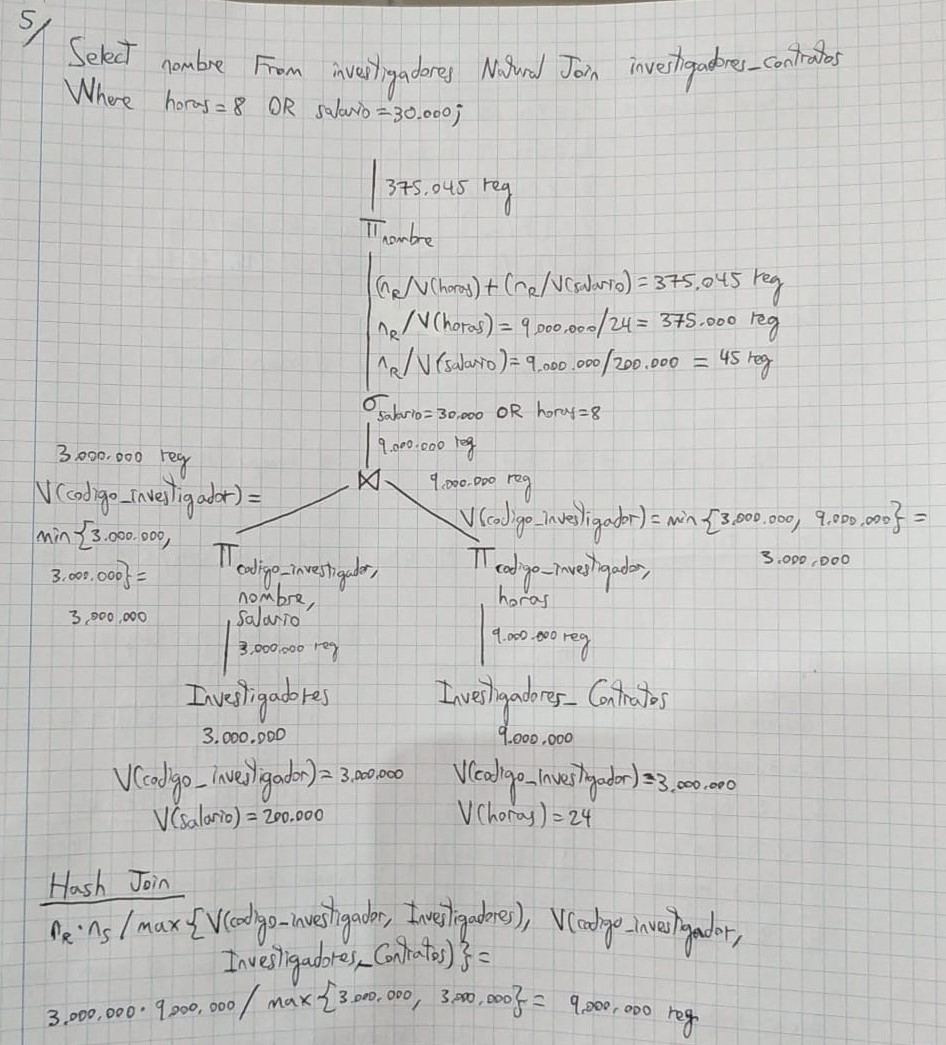
Si pulsamos F7 con el comando: SELECT nombre FROM investigadores NATURAL JOIN investigadores\_contratos WHERE horas=8 OR salario=30000; veremos el resultado gráficamente y nos dice que ha sido dos escaneos secuenciales con un hash sobre uno de ellos, y con este hash y el otro escaneo secuencial se ha realizado un hash inner join y sobre este último se ha aplicado un recolector de estadísticas.





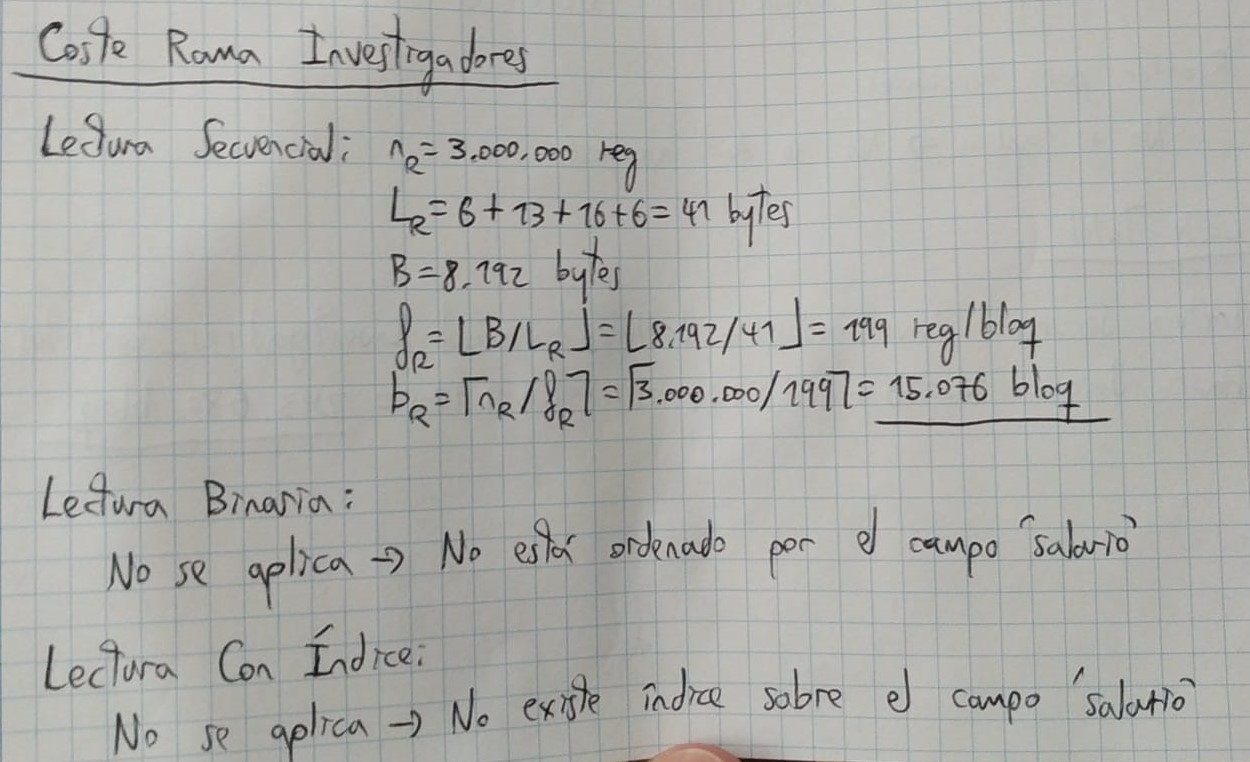


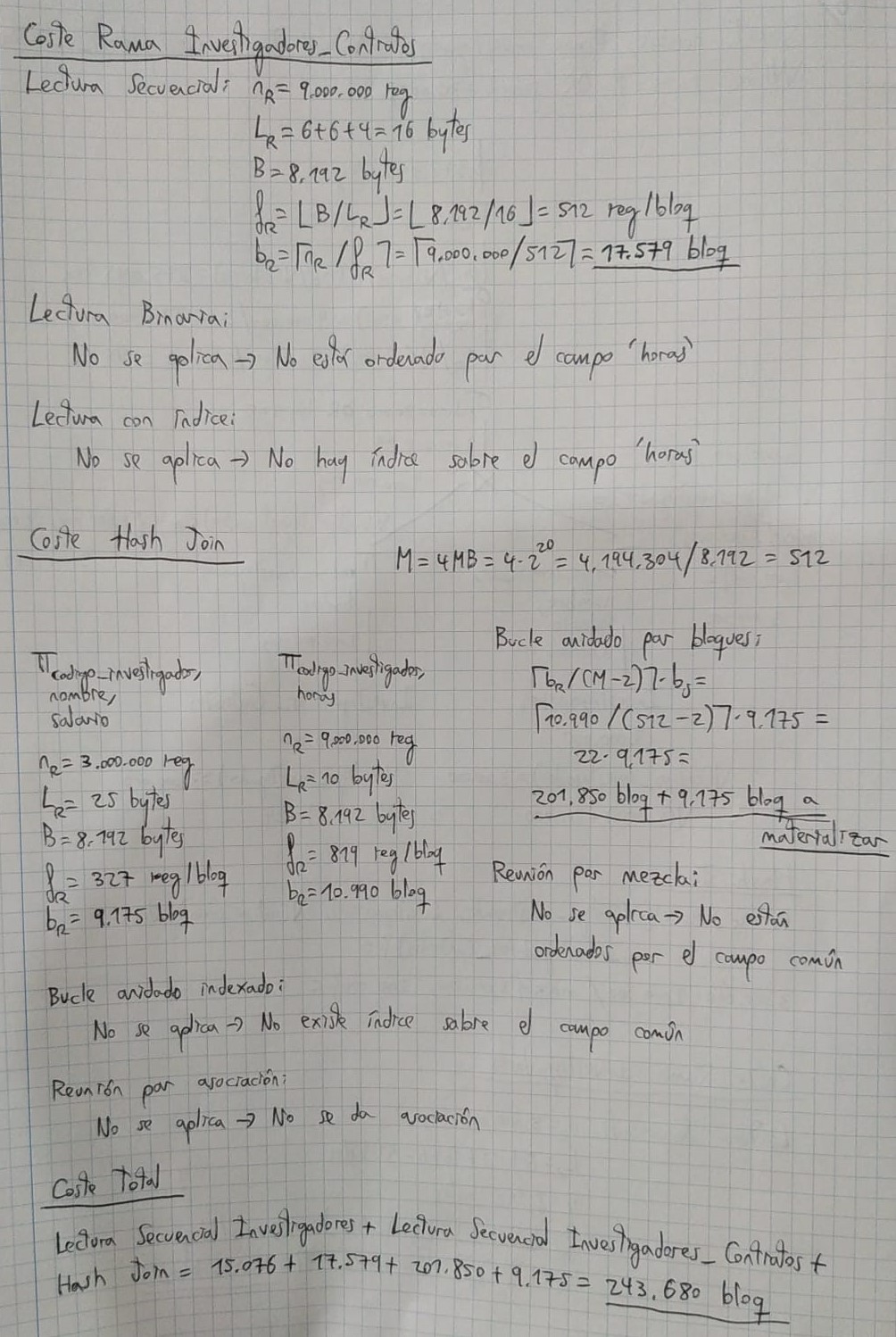
Lo calculamos teóricamente:



El resultado nos devuelve que hemos obtenido 375.045 tuplas. Si lo comparamos con la consulta realizada anteriormente vemos que este número se aproxima al que nos da pgAdmin (383.144).

Si ahora calculamos los costes veremos:





El resultado nos devuelve que hemos obtenido 243.680 bloques. Si lo comparamos con la consulta realizada anteriormente vemos que este número se aproxima al que nos da pgAdmin (253.843,03).

Comparando PostgreSQL con lo realizado teóricamente vemos que hemos seguido el mismo procedimiento, un escaneo secuencial en cada tabla unidos por un Hash Join con una selección posterior.

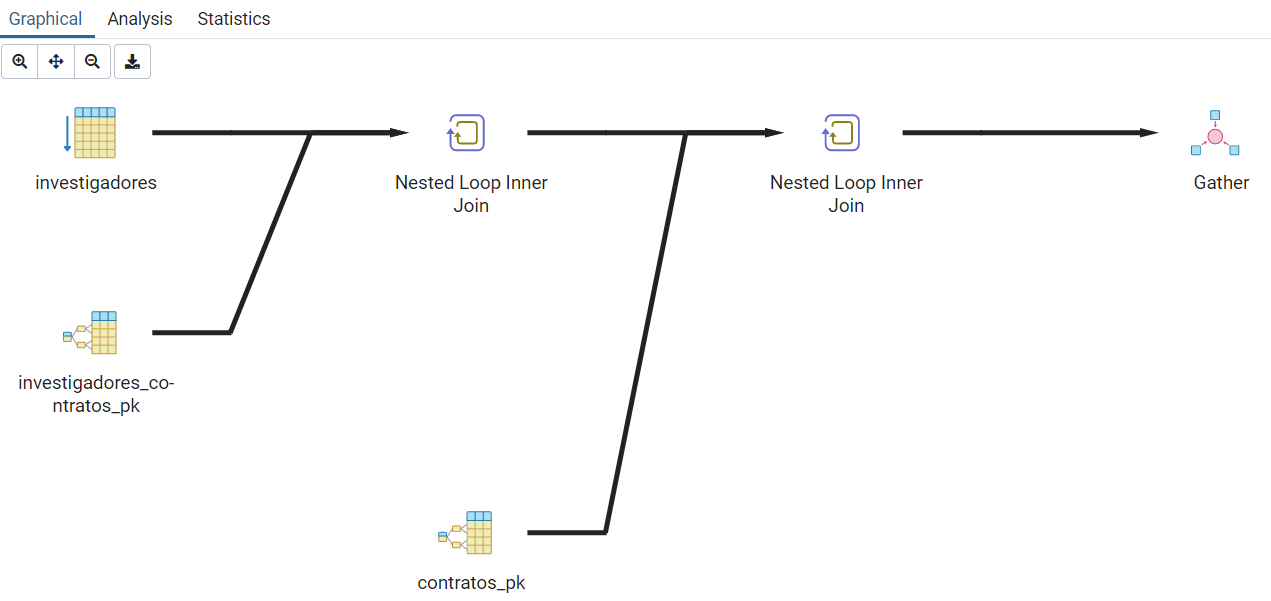
Cuestión 6: Aplicar el comando EXPLAIN a una consulta que obtenga la información del nombre de los contratos y entidades que tienen un coste mayor de 15.000, y tienen empleados con un salario de 24.000 euros y trabajan menos de 2 horas en ellos. ¿Son correctos los resultados del comando EXPLAIN? ¿Por qué? Comparar con lo que se obtendría con lo visto en teoría.

El comando empleado para la consulta sería: EXPLAIN SELECT contratos.nombre, entidad FROM contratos INNER JOIN investigadores\_contratos ON contratos.codigo\_contrato = investigadores\_contratos.codigo\_contrato INNER JOIN investigadores ON investigadores\_contratos.codigo\_investigador = investigadores.codigo\_investigador WHERE coste>15000 AND salario=24000 AND horas<2;.

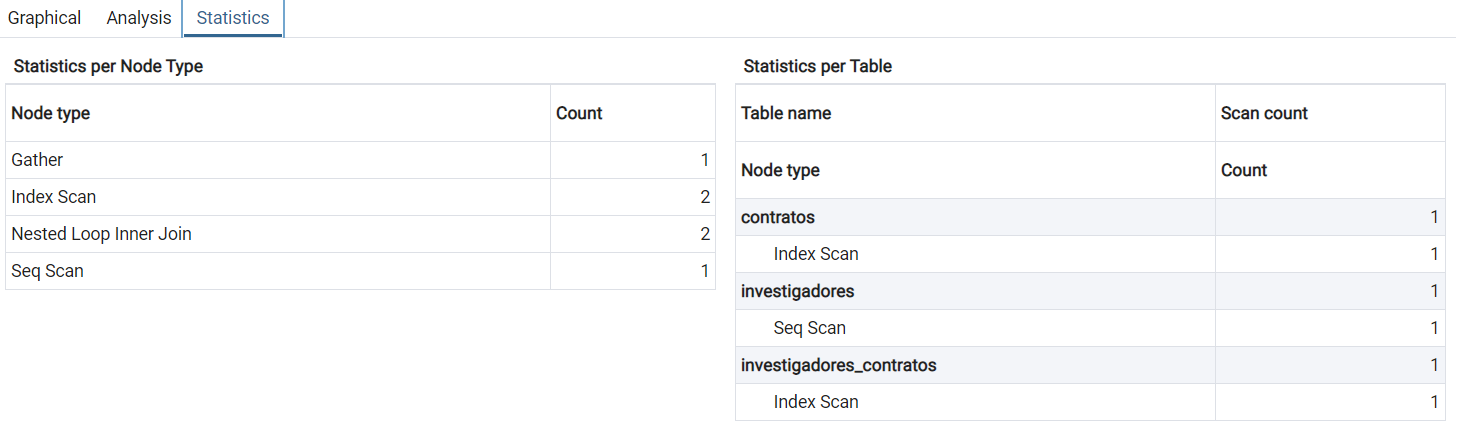
El resultado nos dice que le ha costado 44.792,88, ha impreso 2 filas y la salida tiene un tamaño de 25 bytes.



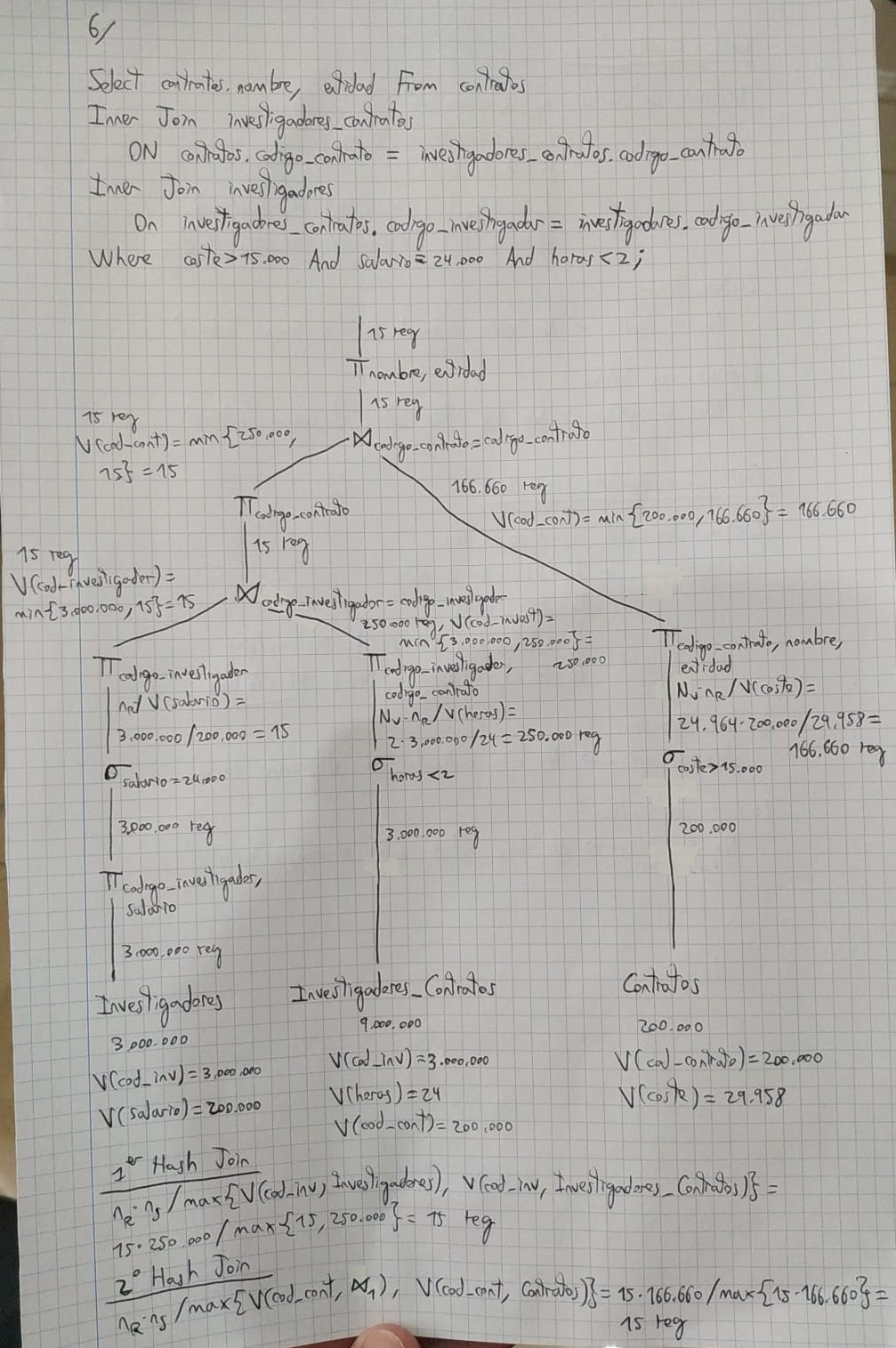
Si pulsamos F7 con el comando: EXPLAIN SELECT contratos.nombre, entidad FROM contratos INNER JOIN investigadores\_contratos ON contratos.codigo\_contrato = investigadores\_contratos.codigo\_contrato INNER JOIN investigadores ON investigadores\_contratos.codigo\_investigador = investigadores.codigo\_investigador WHERE coste>15000 AND salario=24000 AND horas<2; veremos el resultado gráficamente y nos dice que ha sido un escaneo secuencial con un escaneo por índice, sobre estos dos se ha realizado un bucle inner join anidado y este junto con un segundo escaneo por índice nos ha realizado un segundo bucle inner join anidado y sobre este último nos ha realizado un recolector de estadísticas.





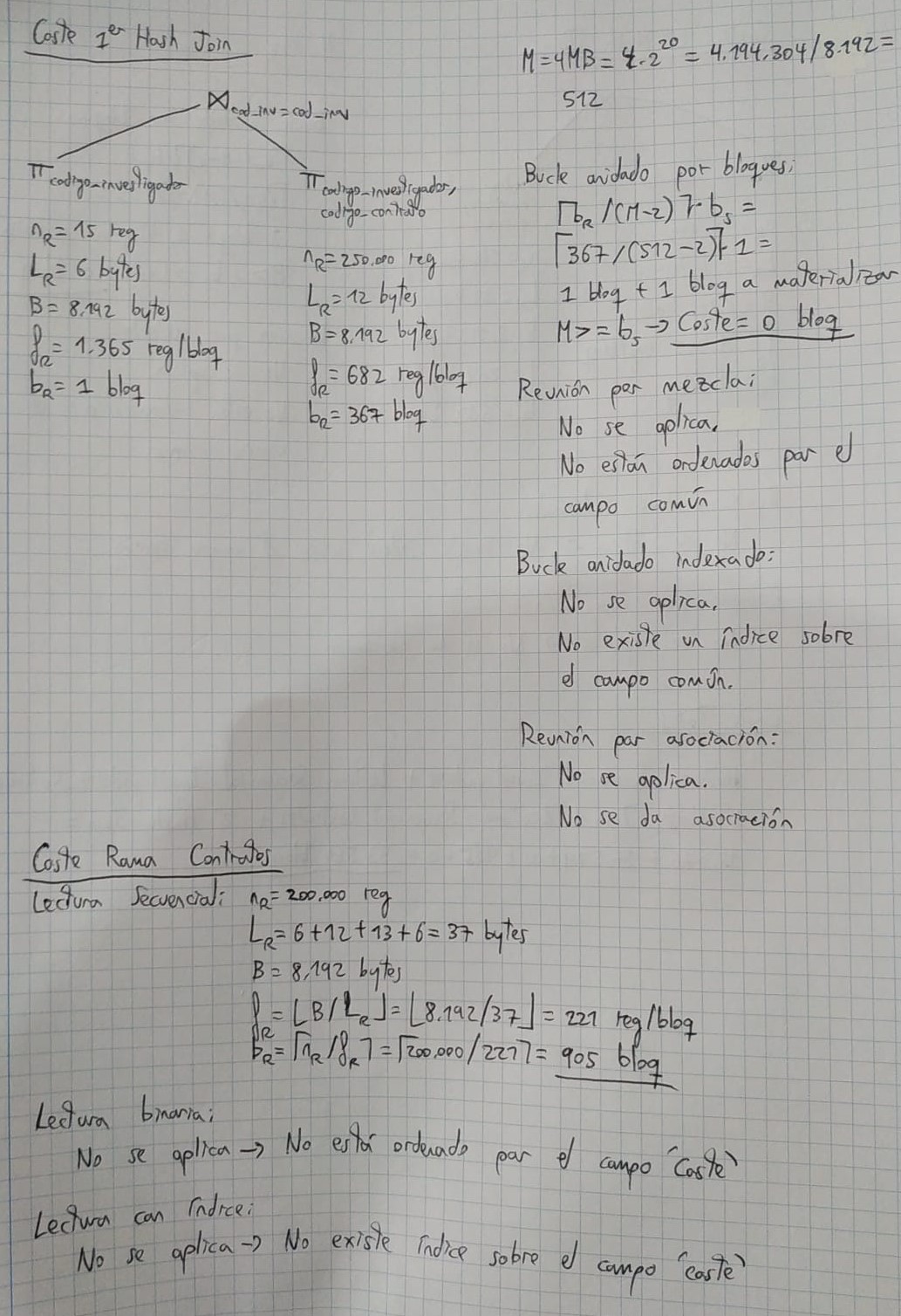
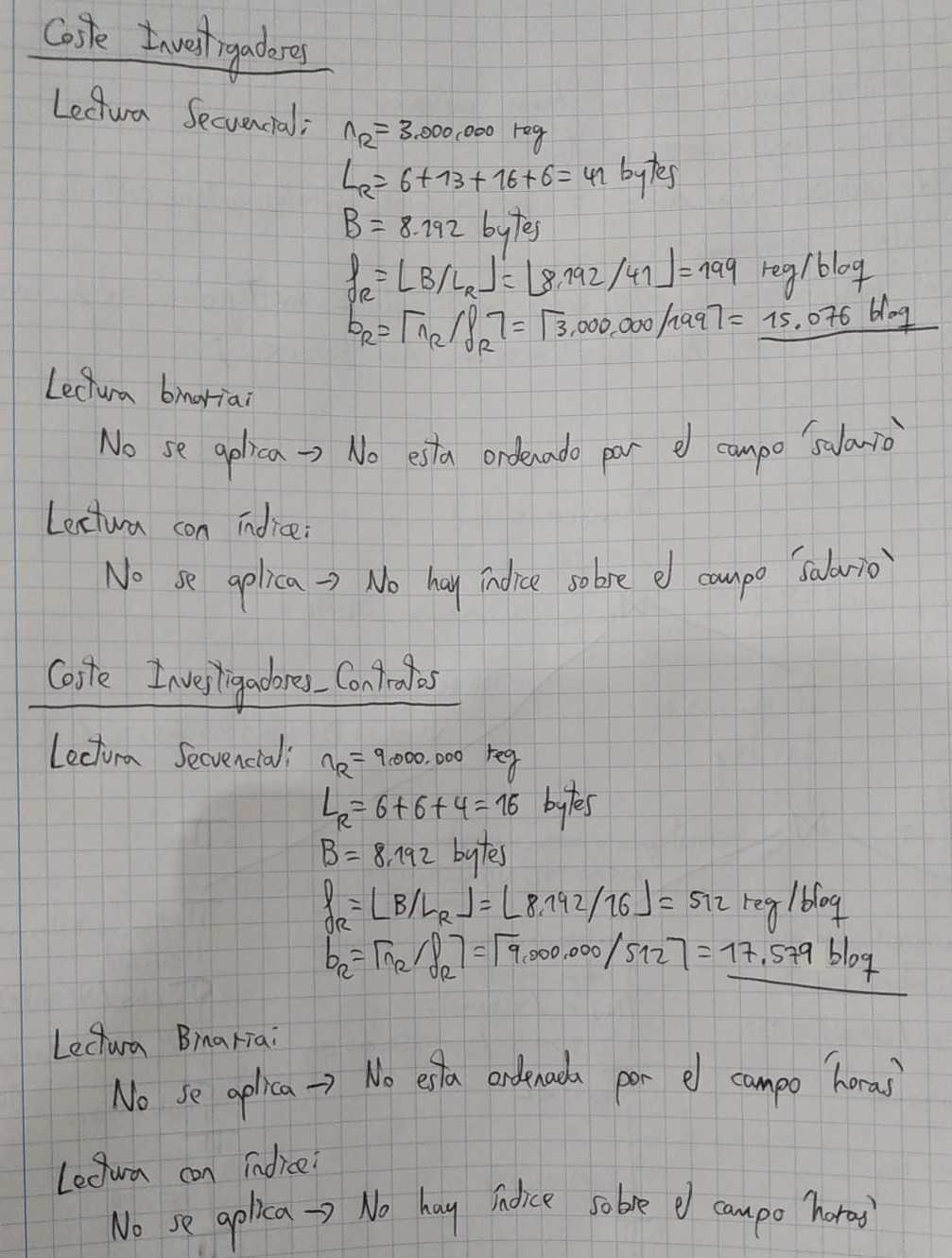


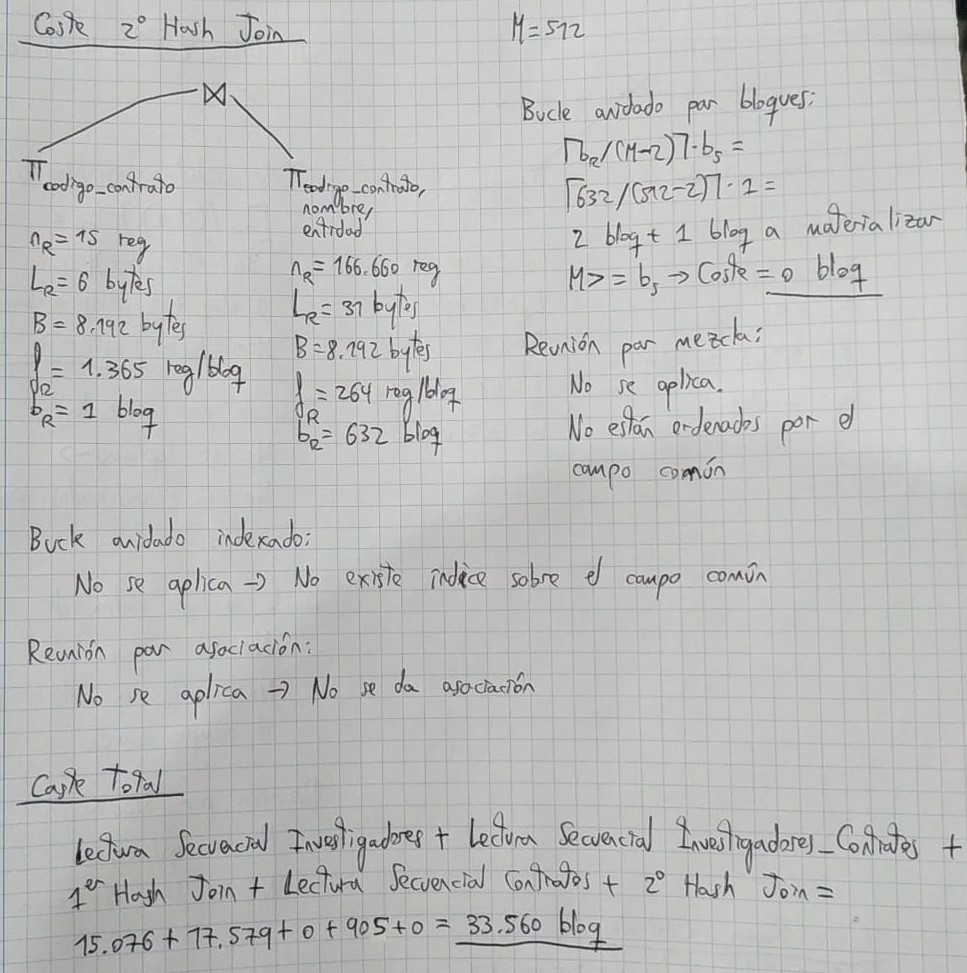
Lo calculamos teóricamente:



El resultado nos devuelve que hemos obtenido 15 tuplas. Si lo comparamos con la consulta realizada anteriormente vemos que este número se aproxima al que nos da pgAdmin (2).

Si ahora calculamos los costes veremos:



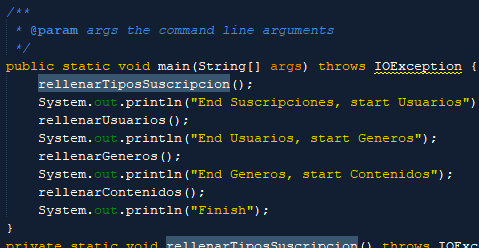
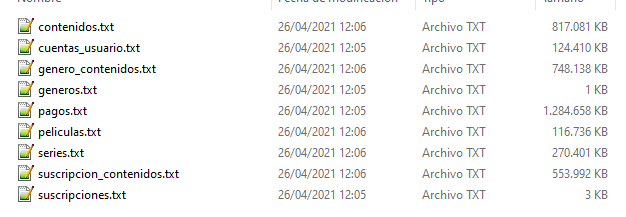


El resultado nos devuelve que hemos obtenido 33.560 bloques. Si lo comparamos con la consulta realizada anteriormente vemos que este número se aproxima al que nos da pgAdmin (44.792,88).

Comparando PostgreSQL con lo realizado teóricamente vemos que hemos seguido casi el mismo procedimiento, nosotros hemos realizado un escaneo secuencial, en vez de un escaneo por índice, en todas las tablas con sus respectivas selecciones posteriores y después al igual que en PostgreSQL hemos realizado un Hash Join de ‘investigadores’ con ‘investigadores\_contratos’ y con el resultado de este hemos ejecutado otro Hash Join con la tabla ‘contratos’.

Cuestión 7: Generar los datos solicitados al comienzo de la práctica para la base de datos **NETFLIX**.

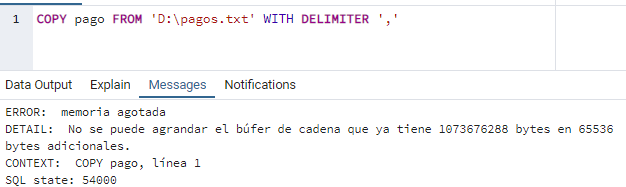
Utilizando el código de generación de datos que adjuntamos hemos generado los nueve ficheros .txt necesarios para rellenar los datos:

Cuestión 8: Realizar la carga masiva de los datos generados en la cuestión 7 en la base de datos **NETFLIX**. Indicar el proceso seguido y el orden de carga de las tablas, explicando el porqué de dicho orden. Comparar los tiempos en las tablas implicadas y explicar a qué es debida la diferencia. ¿Existe diferencia entre los tiempos que ha obtenido y los que aparecen en el LOG de operaciones de postgreSQL? ¿Por qué?

Siguiendo el orden de rellenado natural (es decir, según vayamos necesitando dependencias por las relaciones):

|  |  |
| --- | --- |
| **Tabla** | **Tiempo (seg)** |
| suscripcion | 0,096 s (96 ms) |
| cuenta\_usuario | 23,809 s (23 s 809 ms) |
| pago | primer txt: 591 s (9 min 51 s)  segundo txt: 371 s (6 min 11 s) |
| genero | 0,038 s (38 ms) |
| contenidos | 123 s (2 min 3 s) |
| genero\_contenidos | 1.701 s (28 min 21 s) |
| series | 347 s (5 min 47 s) |
| peliculas | 210 s (3 min 30 s) |
| suscripcion\_contenidos | 782 s (13 min 2 s) |

Nota: hemos tenido que dividir el archivo de pagos en dos para poder rellenar los pagos, ya que eran demasiados datos para el tamaño de buffer inicial:  


A partir de este momento en adelante, se deben de realizar las siguientes cuestiones con la base de datos que tiene la integridad referencial activada.

Cuestión 9: Realizar una consulta SQL que muestre “el nombre y el email de los usuarios que pueden acceder a contenidos con una valoración de más de 5 y tengan películas de una duración entre 90 y 100 minutos o series con menos de 6 temporadas o más de 15 capítulos; y además que esos usuarios puedan ver contenidos que tienen más de 3 géneros asociados habiendo realizado por lo menos algún pago en los 7 primeros días del mes de mayo.

Obtener el plan de ejecución con el resultado del comando EXPLAIN en forma de árbol de álgebra relacional. Explicar la información obtenida en el plan de ejecución de postgreSQL. Comparar el árbol obtenido por nosotros al traducir la consulta original al álgebra relacional y el que obtiene postgreSQL. Comentar las posibles diferencias entre ambos árboles.

Para la consulta introducimos el comando:

EXPLAIN SELECT DISTINCT nombre, e\_mail FROM cuenta\_usuario

INNER JOIN suscripcion ON "cuenta\_usuario"."suscripcion\_ID\_suscripcion" = "suscripcion"."suscripcion\_ID"

INNER JOIN suscripcion\_contenidos ON "suscripcion\_contenidos"."suscripcion\_ID\_suscripcion"="suscripcion"."suscripcion\_ID"

INNER JOIN contenidos ON "contenidos"."producto\_ID" = "suscripcion\_contenidos"."producto\_ID\_contenidos"

INNER JOIN peliculas ON "peliculas"."producto\_ID\_contenidos" = "contenidos"."producto\_ID"

INNER JOIN series ON "series"."producto\_ID\_contenidos" = "contenidos"."producto\_ID"

INNER JOIN

(SELECT "genero\_contenidos"."producto\_ID\_contenidos" FROM genero\_contenidos

GROUP BY "genero\_contenidos"."producto\_ID\_contenidos"

HAVING count("genero\_contenidos"."genero\_ID\_genero")>3) AS temp

ON "temp"."producto\_ID\_contenidos" = "contenidos"."producto\_ID"

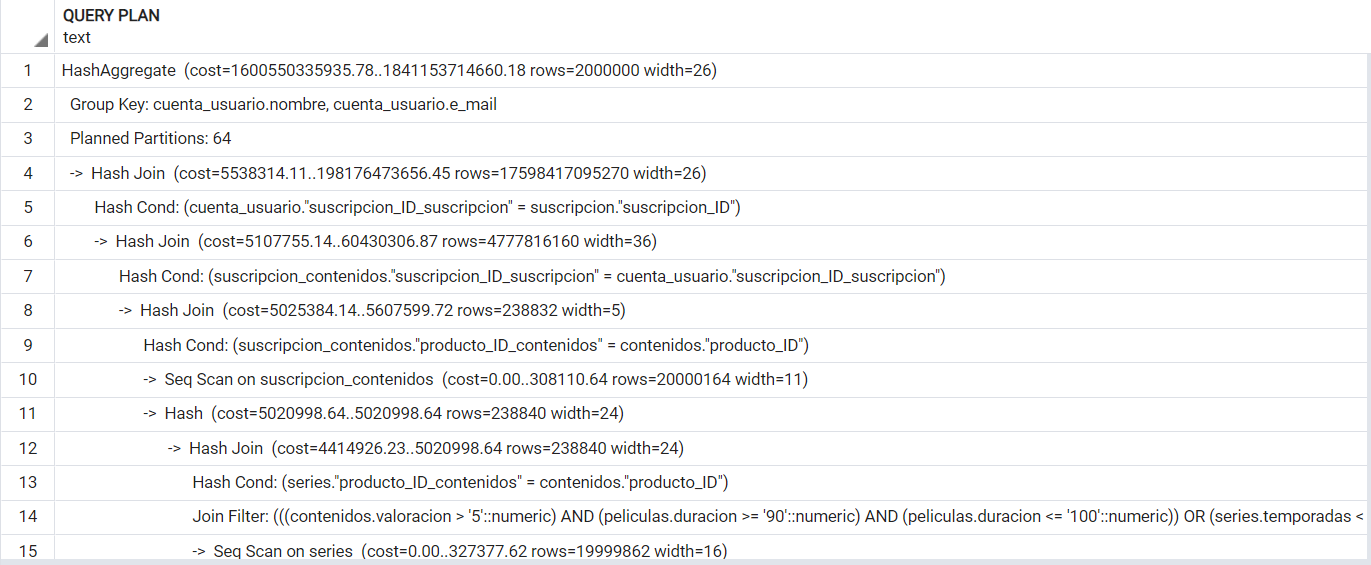
INNER JOIN pago ON "pago"."suscripcion\_ID\_suscripcion" = "suscripcion"."suscripcion\_ID"

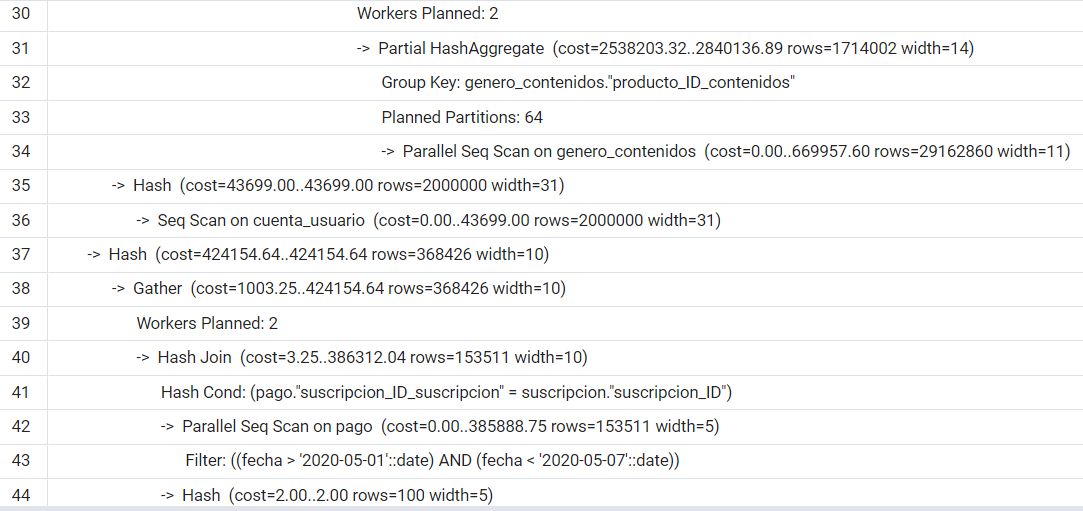
WHERE ((valoracion>5 AND (duracion>=90 AND duracion<=100)) OR temporadas<6 OR capitulos>15)

AND ("pago"."fecha" > '2020/05/01' AND "pago"."fecha" < '2020/05/07')

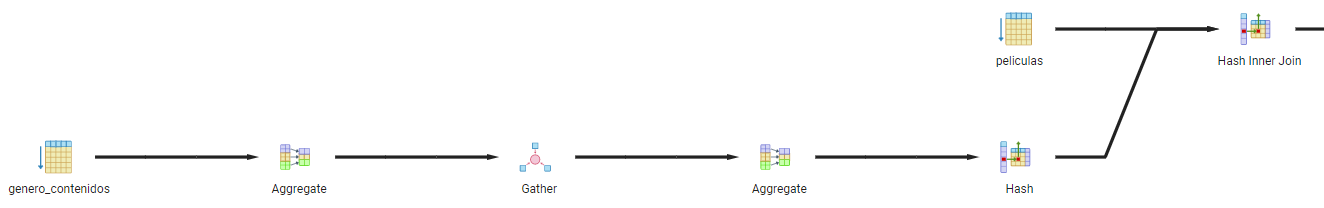
Esta consulta se queda ejecutando todo el tiempo, no finaliza.

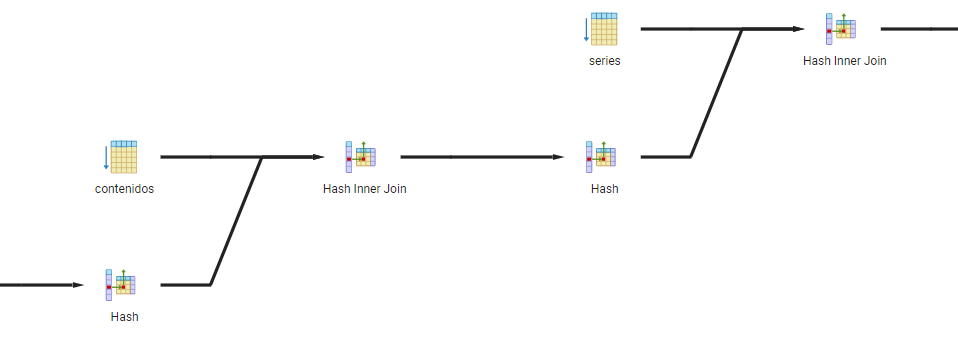
Añadimos el EXPLAIN al principio de la consulta anterior, lo ejecutamos y obtenemos lo siguiente:

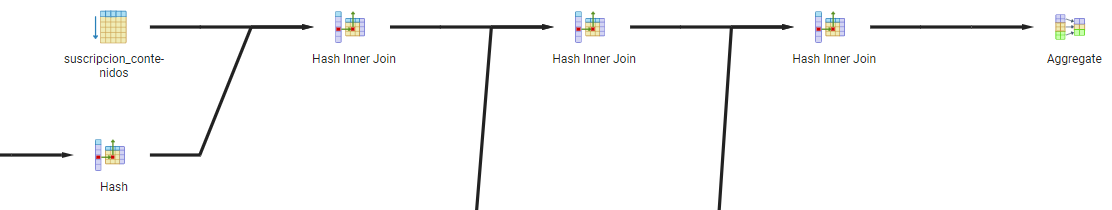




Si ahora ejecutamos F7 sobre la consulta anterior, sin el EXPLAIN, veremos una representación gráfica de la resolución de la consulta:

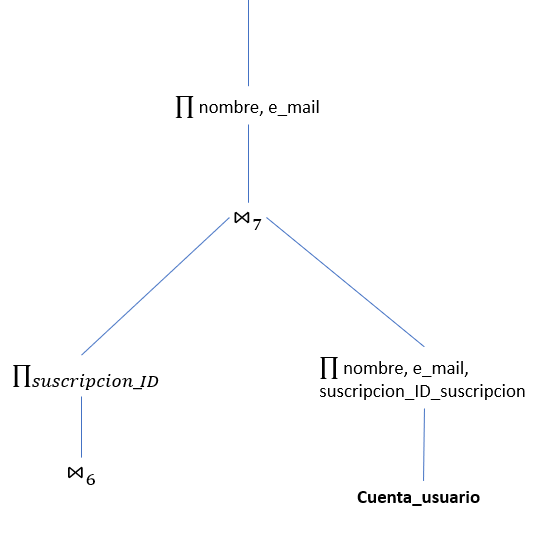


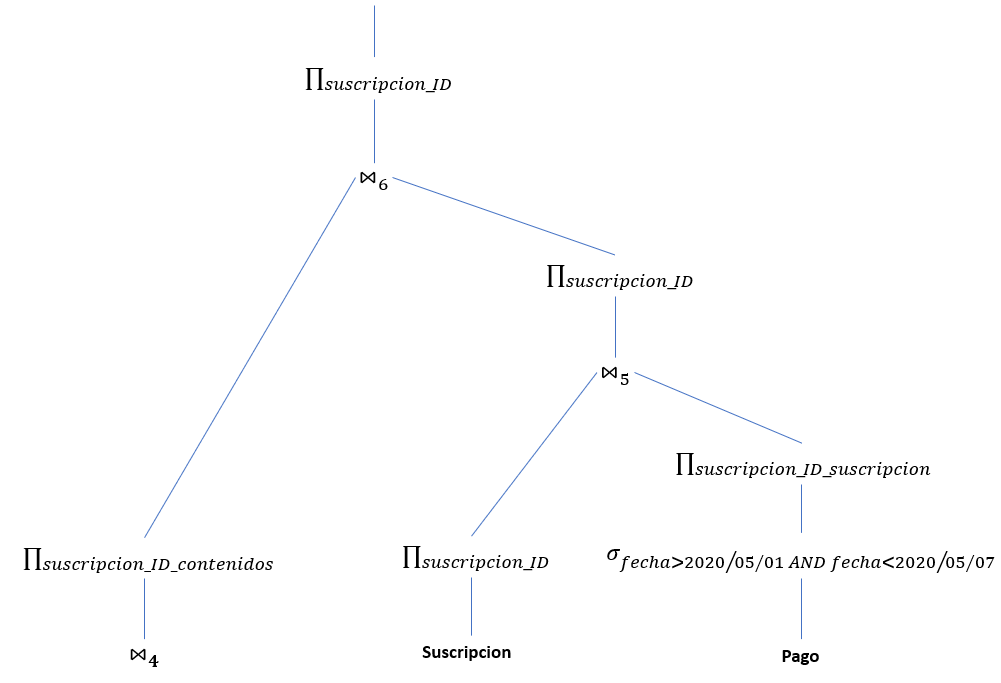


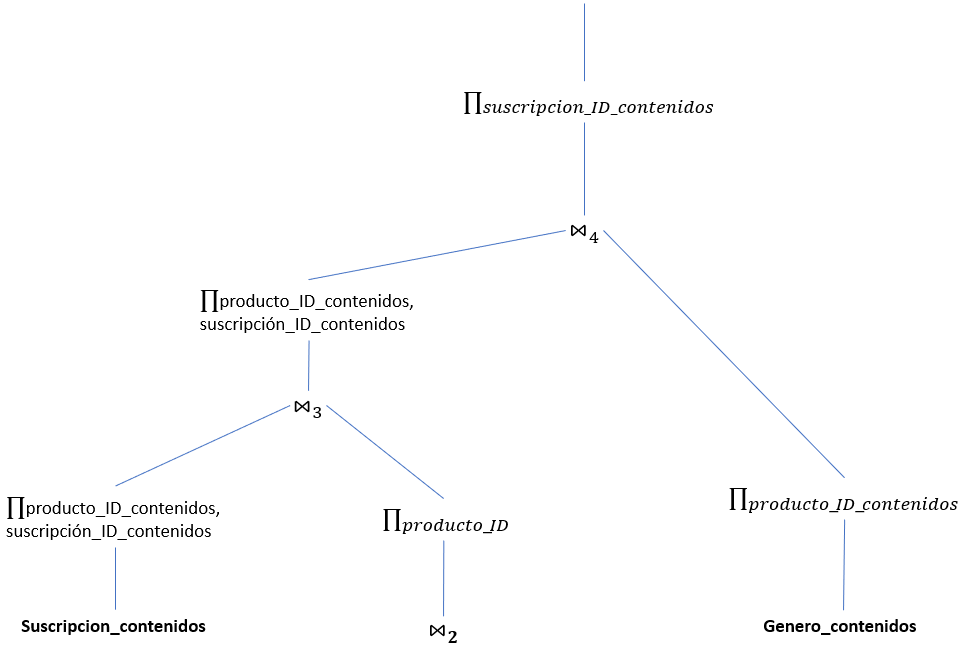
 

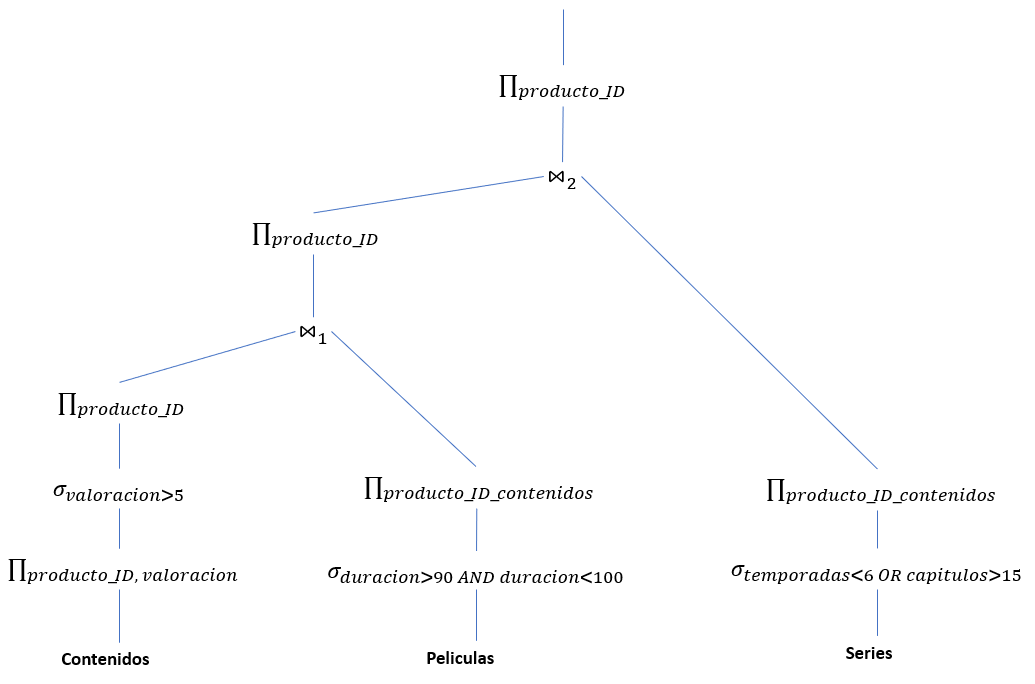
De las imágenes previas que hemos obtenido podemos decir que ha ido uniendo las tablas una por una por hash inner join, salvo la unión de cuenta\_usuario, pago y suscripción que las hace en paralelo. Inicialmente hace una agregación sobre la tabla genero\_contenidos, seguido de una recolección de datos y una segunda agregación. Paralelamente del resultado hash inner join entre pago y suscripción hace una recolección. Y finalmente al resultado final hace otra agregación.

Si ahora creamos el árbol teóricamente veremos lo siguiente:









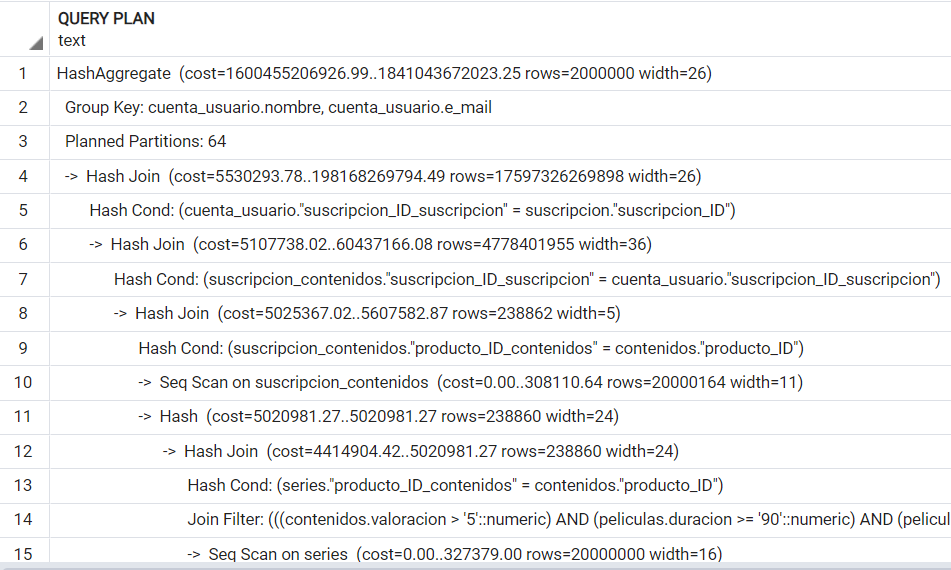
Si comparamos el árbol obtenido por nosotros con el árbol que obtiene PostgeSQL podemos ver que PostgreSQL comienza uniendo ‘genero\_contenidos’ con ‘peliculas’, mientras que nosotros unimos ‘peliculas’ con ‘contenidos’, y continua uniendolo con ‘contenidos’ y ‘series’, mientras que nosotros lo unimos con ‘series’, ‘suscripcion\_contenidos’ y ‘genero\_contenidos’; a esto se le añade que paralelamente PostgreSQL une ‘pago’ y ‘suscripcion’ y del resultado de este junto con la tabla ‘cuenta\_usuario’ lo une a lo que ya ha realizado y nosotros teóricamente hacemos igual.

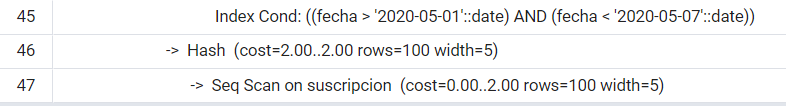
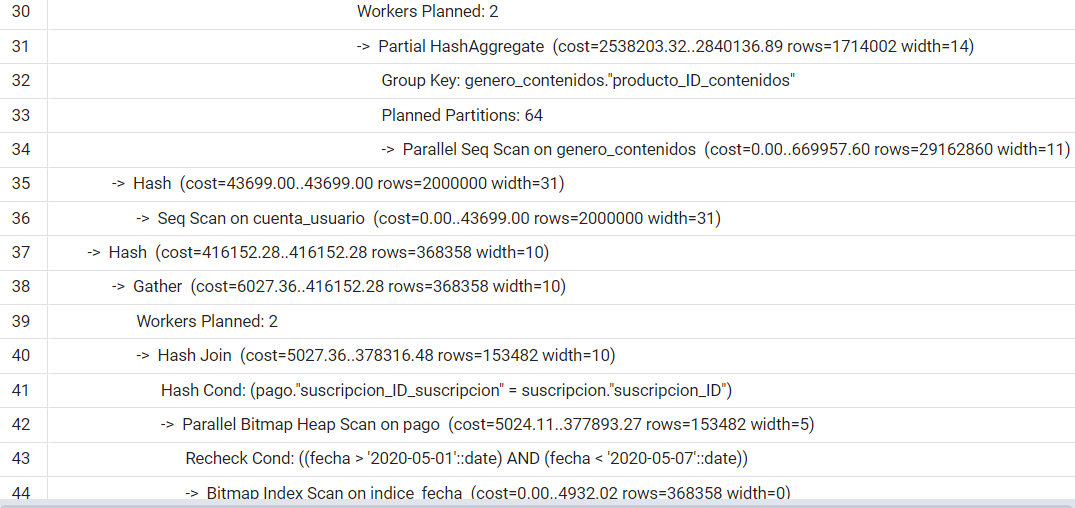
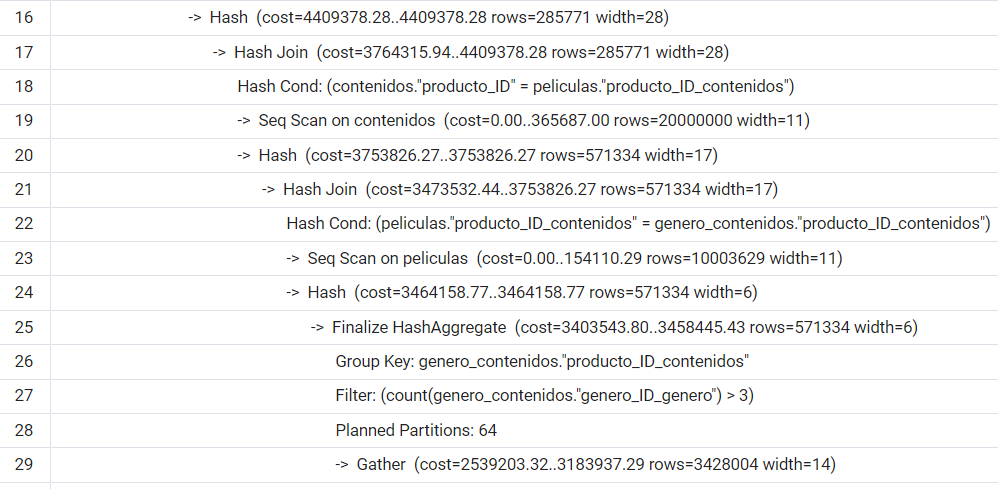
Cuestión 10: Usando PostgreSQL, y a raíz de los resultados de la cuestión anterior, ¿qué modificaciones realizaría para mejorar el rendimiento de la misma y por qué? Obtener la información pedida de la cuestión 9 y explicar los resultados. Obtener el plan de ejecución con el resultado del comando EXPLAIN en forma de árbol de álgebra relacional. Comentar los resultados obtenidos y comparar con la cuestión anterior.

Las modificaciones que realizamos para mejorar el rendimiento de la cuestión anterior son crear índices para cada una de las condiciones de WHERE, para ello escribimos las consultas:

* CREATE INDEX indice\_valoracion ON contenidos (valoracion);
* CREATE INDEX indice\_duracion ON peliculas (duracion);
* CREATE INDEX indice\_temporadas ON series (temporadas);
* CREATE INDEX indice\_capitulos ON series (capitulos);
* CREATE INDEX indice\_fecha ON pago (fecha);

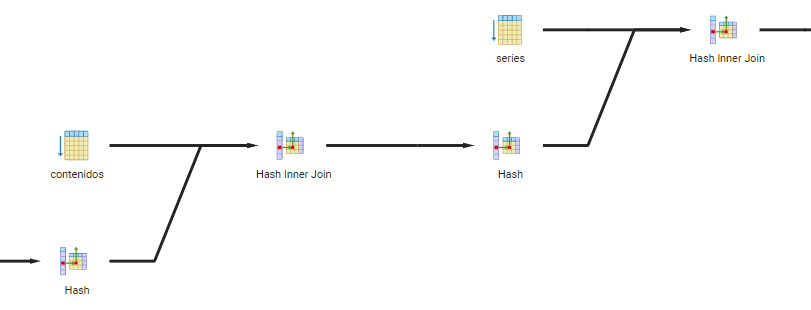
Si ahora ejecutamos nuevamente la consulta de la cuestión 9 con EXPLAIN veremos:

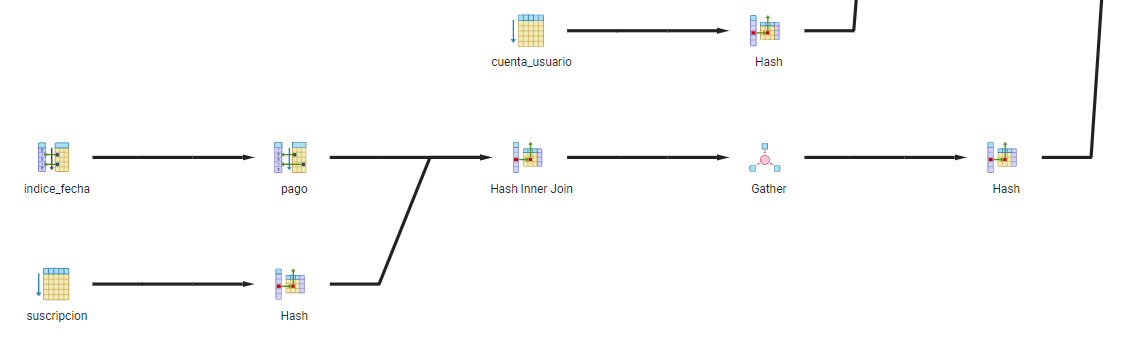
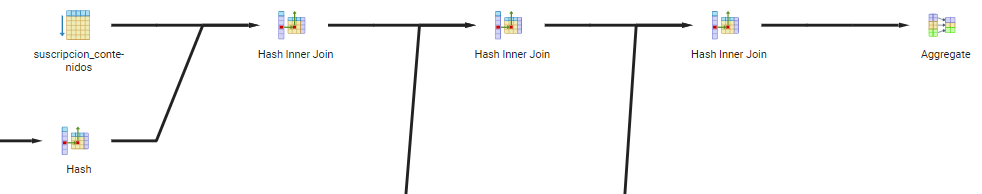




Si ahora ejecutamos F7 sobre la consulta, sin el EXPLAIN, veremos una representación gráfica de la resolución de la consulta:





Si comparamos los resultados obtenidos de la cuestión anterior con los nuevos resultados obtenidos en esta cuestión veremos que el coste ha disminuido en 110.042.636,9 unidades. También se puede observar que el plan de ejecución es el mismo con la única diferencia de que en vez de hacer un escaneo secuencial sobre la tabla ‘pago’ se ha realizado una búsqueda por índice sobre el campo ‘fecha’.

Cuestión 11: Usando PostgreSQL, borre el 50% de los contenidos almacenados de manera aleatoria y todos sus datos relacionados ¿Cuál ha sido el proceso seguido? ¿Y el tiempo empleado en el borrado? Ejecute la consulta de nuevo. Obtener el plan de ejecución con el resultado del comando EXPLAIN en forma de árbol de álgebra relacional. Comparar con los resultados anteriores.

Para ejecutar el borrado, creamos una tabla temporal, llamada temp1, que borraremos al final. En esa tabla almacenamos 10.000.000 de elementos de contenidos elegidos aleatoriamente mediante la siguiente manera:

CREATE TABLE temp1 AS (SELECT "producto\_ID" FROM contenidos ORDER BY random() LIMIT 10000000);

Después creamos, mediante transacciones únicas para asegurarnos de que no se comen la memoria unos de otros, borrados selectivos utilizando esa tabla intermedia. Empezamos por las tablas relacionadas con contenidos, y cuando éstas hayan tenido ese borrado borramos en contenidos. La razón para hacerlo así es que de esta forma podemos evitar que el borrado en cascada de la base de datos se coma la memoria del sistema al intentar cargar 5 tablas completas al mismo tiempo.

De esta forma, el código completo es el siguiente:

BEGIN;

CREATE TABLE temp1 AS (SELECT "producto\_ID" FROM contenidos ORDER BY random() LIMIT 10000000);

COMMIT;

END;

BEGIN;

DELETE FROM series WHERE "producto\_ID\_contenidos" IN (SELECT "producto\_ID" FROM temp1);

COMMIT;

END;

BEGIN;

DELETE FROM peliculas WHERE "producto\_ID\_contenidos" IN (SELECT "producto\_ID" FROM temp1);

COMMIT;

END;

BEGIN;

DELETE FROM suscripcion\_contenidos WHERE "producto\_ID\_contenidos" IN (SELECT "producto\_ID" FROM temp1);

COMMIT;

END;

BEGIN;

DELETE FROM genero\_contenidos WHERE "producto\_ID\_contenidos" IN (SELECT "producto\_ID" FROM temp1);

COMMIT;

END;

BEGIN;

DELETE FROM contenidos WHERE "producto\_ID" IN (SELECT "producto\_ID" FROM temp1);

COMMIT;

END;

BEGIN;

DROP TABLE IF EXISTS temp1;

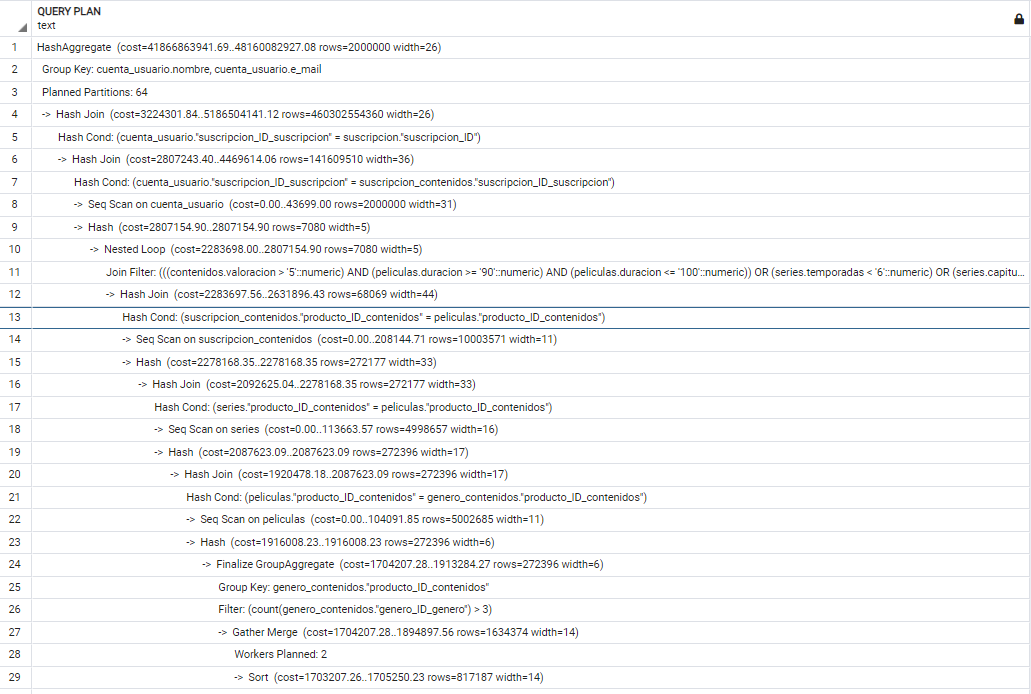
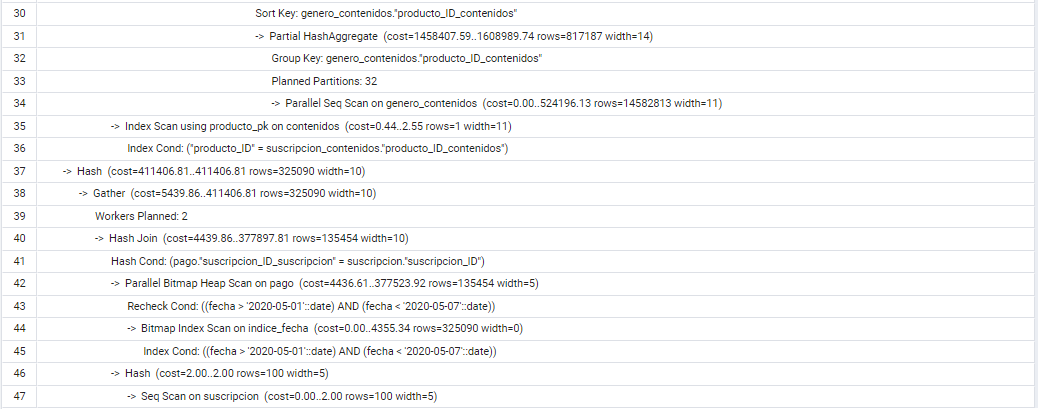
COMMIT;

END;

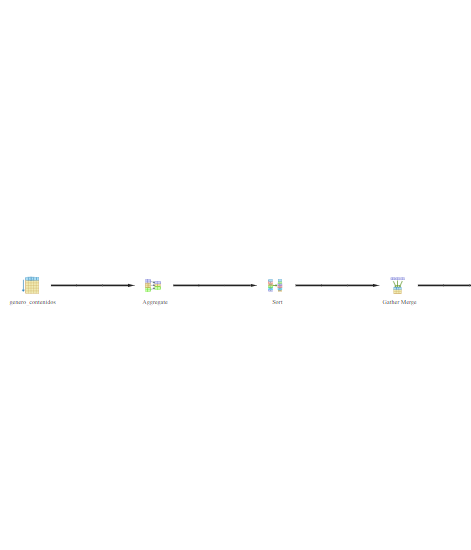
Dado que todas esas consultas llevan su tiempo, las ejecutamos una por una para asegurarnos de que todo funciona como debe y no se ha quedado pillado. De esta forma, los tiempos utilizados para cada transacción son los siguientes:

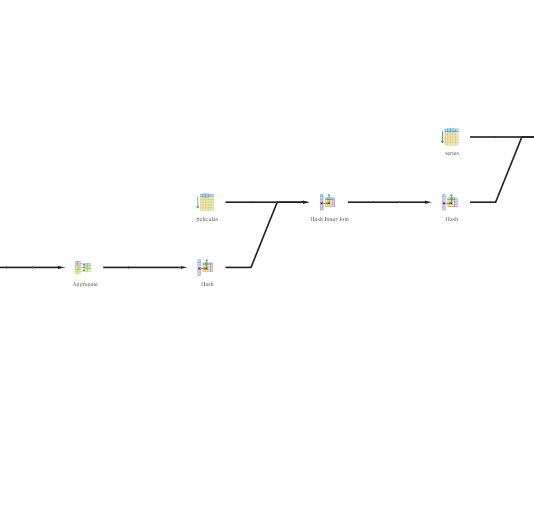
* Creación de la tabla intermedia: 38,9 segundos
* Borrado de series: 4 minutos 34 segundos
* Borrado de peliculas: 2 minutos 46 segundos
* Borrado de suscripcion\_contenidos: 25 minutos 41 segundos
* Borrado de genero\_contenidos: 1 hora 10 minutos
* Borrado de contenidos: Hemos tenido que cortarlo a mitad de ejecución, debido a que estuvo 16 horas y no terminó. Hicimos pruebas con menos tuplas y nos tardó lo siguiente:
  + 10 tuplas: 2 minutos.
  + 1.000 tuplas: 4 horas 9 minutos
* Por tanto, hemos extrapolado que tardaría 1.729 DÍAS en borrar 10.000.000 de tuplas.

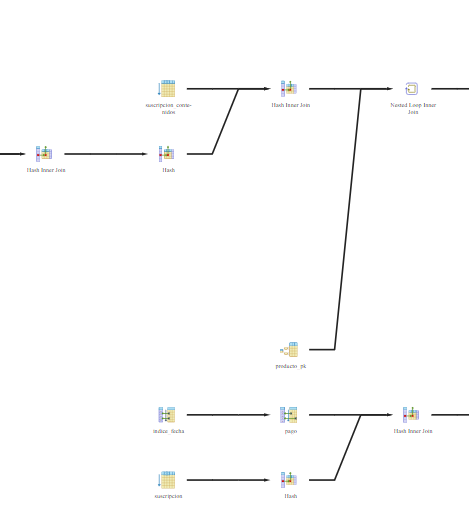
Si ahora ejecutamos la consulta con el EXPLAIN, de la cuestión 9, obtenemos lo siguiente:

Si ahora ejecutamos F7 sobre la consulta, sin el EXPLAIN de la cuestión 9, veremos una representación gráfica de la resolución de la consulta:









Si comparamos los resultados obtenidos anteriormente con los nuevos resultados obtenidos en esta cuestión veremos:

La gran similitud es que ha tenido que hacer un escaneo secuencial de casi todas las tablas afectadas (genero\_contenidos, series, suscripcion\_contenidos, peliculas y cuenta\_usuario) a excepcion de la tabla ‘pago’ que ha utilizado el índice basado en las fechas gracias a que no hemos borrado nada en ella, y por tanto su índice sigue siendo válido.

Por otra parte, el coste es considerablemente inferior al del resultado del ejercicio 9, debido a que han desaparecido tantos datos, además de contar con el índice de la tabla pago. También tiene menos coste que en el resultado del ejercicio 10, aunque esa diferencia es menor.

Cuestión 12: ¿Qué técnicas de optimización de la BD propondría para mejorar los resultados de dicho plan sin modificar el código de la consulta? ¿Por qué?

Las técnicas de optimización de la base de datos para mejorar los resultados sin modificar el código serían:

* VACUUM porque ayuda a recuperar o reutilizar el espacio en disco ocupado por filas actualizadas o eliminadas, actualiza las estadísticas de datos utilizadas por el planificador de consultas de PostgreSQL, actualiza el mapa de visibilidad lo que acelera los [escaneos de solo índice](https://www.postgresql.org/docs/current/indexes-index-only-scans.html) y para protegerse contra la pérdida de datos muy antiguos debido al envolvente de ID de transacción o al envolvente de ID de multixact.
* ANALYZE para poder actualizar las estadísticas de datos utilizadas por el planificador de consultas de PostgreSQL. Es importante tener estadísticas razonablemente precisas, de lo contrario, las malas elecciones de planes podrían degradar el rendimiento de la base de datos.
* REINDEX para reconstruir un índice utilizando los datos almacenados en la tabla del índice reemplazando la copia anterior del índice.

Cuestión 13: Usando PostgreSQL, lleve a cabo las operaciones propuestas en la cuestión anterior y ejecute el plan de ejecución de la misma consulta. Obtener el plan de ejecución con el resultado del comando EXPLAIN en forma de árbol de álgebra relacional. Compare los resultados del plan de ejecución con los de los apartados anteriores. Coméntelos.

Para VACUUM escribimos la consulta:  
VACUUM FULL VERBOSE ANALYZE contenidos, cuenta\_usuario, genero, genero\_contenidos, pago, peliculas, series, suscripcion, suscripcion\_contenidos;  
Tiempo de ejecución: 7 minutos 41 segundos.

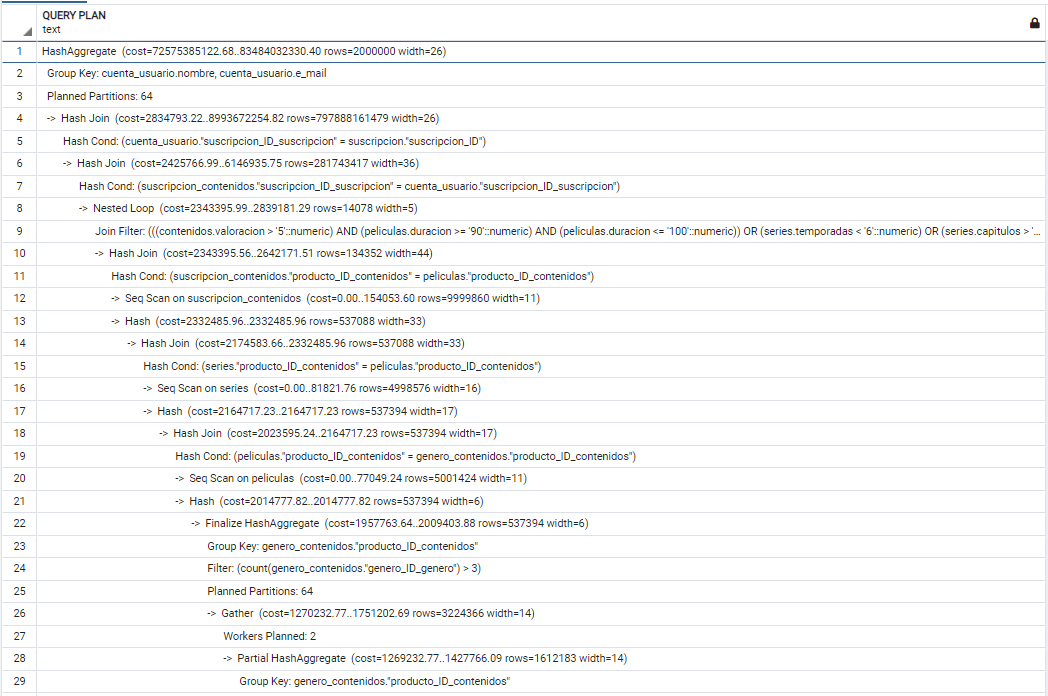
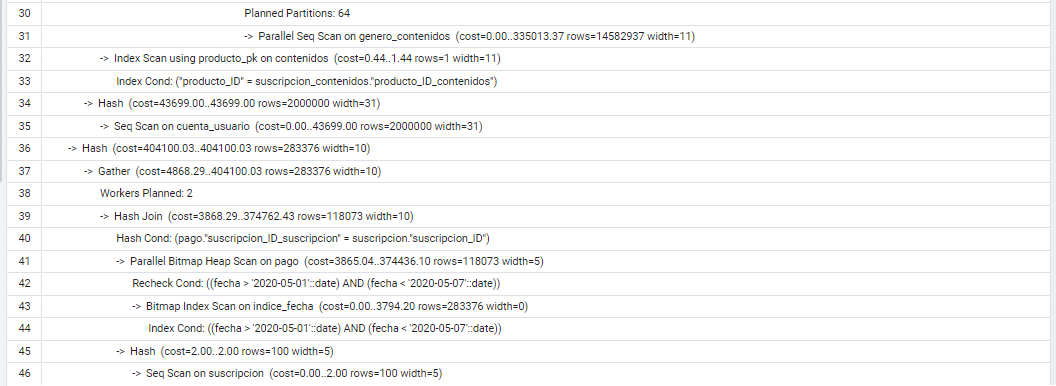
Para ANALYZE escribimos la consulta:  
ANALYZE VERBOSE contenidos, cuenta\_usuario, genero, genero\_contenidos, pago, peliculas, series, suscripcion, suscripcion\_contenidos;  
Tiempo de ejecución: 38 segundos.

Para REINDEX escribimos las consultas:

* REINDEX INDEX indice\_valoracion;
* REINDEX INDEX indice\_duracion;
* REINDEX INDEX indice\_temporadas;
* REINDEX INDEX indice\_capitulos;
* REINDEX INDEX indice\_fecha;

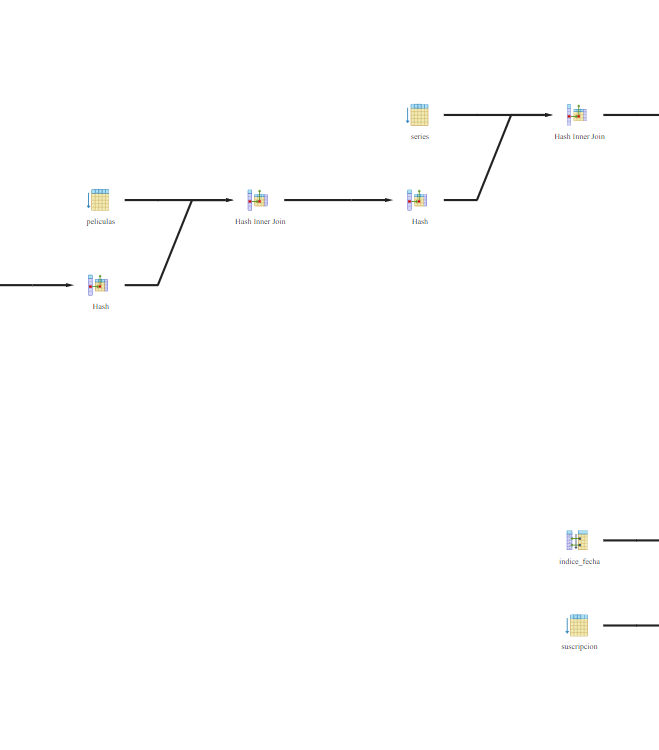
Tiempo de ejecución: 1 minuto 47 segundos.

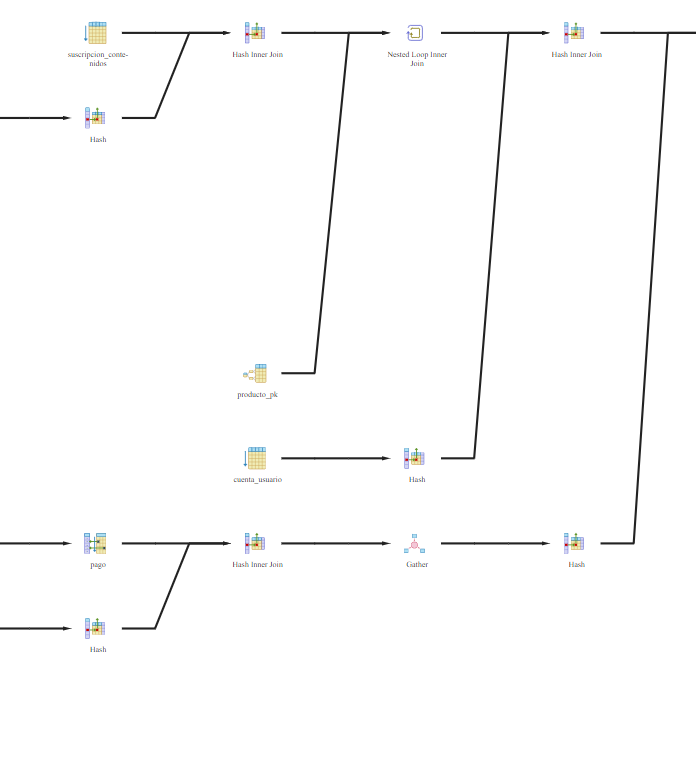
Una vez aplicadas las técnicas de optimización si ahora ejecutamos la consulta con el EXPLAIN, de la cuestión 9, obtenemos lo siguiente:

Si ahora ejecutamos F7 sobre la consulta, sin el EXPLAIN de la cuestión 9, veremos una representación gráfica de la resolución de la consulta:









Si comparamos los resultados obtenidos anteriormente con los nuevos resultados obtenidos en esta cuestión veremos que a pesar de tener los índices, sigue utilizando el mismo árbol de ejecución, y por tanto no presenta ninguna mejora.

Cuestión 14: A partir de lo visto y recopilado en toda la práctica. Describir y comentar cómo es el proceso de procesamiento y optimización que realiza PostgreSQL en las consultas del usuario.

Para el proceso de procesamiento:

* PostgreSQL crea varios planes de ejecución y el que es el más barato lo ejecuta (el más rápido).
* Una vez que sabe que plan va a ejecutar crea un árbol de planes completo para la ejecución.
* PostgreSQL escanea cada relación individual utilizada en la consulta.
* PostgreSQL siempre crea un plan de escaneo secuencial para cada tabla y con alguna/s puede crear otro plan con un escaneo por índice.
* Los escaneos por índices también se generan para índices que tienen un orden de clasificación que puede coincidir con la cláusula ORDER BY o un orden de consultas que puede ser útil para la combinación de combinaciones.
* Para unir dos o más relaciones se tienen que encontrar todos los planes factibles para escanear relaciones.
* Cuando la consulta involucra más de dos relaciones, el resultado final debe construirse mediante un árbol de pasos de combinación, cada uno con dos entradas. El planificador examina diferentes secuencias de combinaciones posibles para encontrar la más barata.

Para optimizar:

* En PostgreSQL hay que realizar tareas de mantenimiento con VACUUM, ANALYZE y REINDEX para optimizar la ejecución de las consultas.
* Han de crearse índices para optimizar las ejecuciones.
* Se puede modificar el fichero ‘postgresql.conf’ para añadir características a la base de datos, modificar dichas características o suprimirlas. Por ejemplo, se puede modificar el atributo ‘shared\_buffer’ del apartado ‘Memory’.

## **Bibliografía**

PostgreSQL (13.x)

- Capítulo 14: Performance Tips.

- Capítulo 19: Server Configuration.

- Capítulo 15: Parallel Query.

- Capítulo 24: Routine Database Maintenance Tasks.

- Capítulo 50: Overview of PostgreSQL Internals.

- Capítulo 70: How the Planner Uses Statistics.