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

**Curso: 2019-2020. 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:** JAVIER GARCÍA JIMÉNEZ

**DNI:** 09099503J

**ALUMNO 2:**

**Nombre y Apellidos:** ISABEL MARTÍNEZ GÓMEZ

**DNI:** 06027983M

**Fecha:** 26/04/2020

**Profesor Responsable:** JOSÉ CARLOS HOLGADO

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 2 de Marzo, Semana 9 de Marzo, Semana 16 de Marzo, semana 23 de Marzo y semana 30 de Marzo.

Entrega de práctica: Semana 26 de Abril (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 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 (12.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 **TIENDA**. Básicamente, la base de datos guarda información sobre las tiendas que tiene una empresa en funcionamiento en ciertas provincias. La empresa tiene una serie de trabajadores a su cargo y cada trabajador pertenece a una tienda. Los clientes van a las tiendas a realizar compras de los productos que necesitan y son atendidos por un trabajador, el cuál emite un tickect en una fecha determinada con los productos que ha comprado el cliente, reflejando el importe total de la compra. Cada tienda tiene registrada los productos que pueden suministrar.

Los datos referidos al año 2019 que hay que generar deben de ser los siguientes:

* Hay 200.000 tiendas repartidas aleatoriamente entre todas las provincias españolas.
* Hay 1.000.000 productos cuyo precio está comprendido entre 50 y 1.000 euros y que se debe de generar de manera aleatoria.
* Cada una de las empresas tiene de media en su tienda 100 productos que se deben de asignar de manera aleatoria de entre todos los que hay; y además el stock debe de estar comprendido entre 10 y 200 unidades, que debe de ser generado de manera aleatoria también.
* Hay 1.000.000 trabajadores. Los trabajadores se deben de asignar de manera aleatoria a una tienda y el salario debe de estar comprendido entre los 1.000 y 5.000 euros. Se debe de generar también de manera aleatoria.
* Hay 5.000.000 de tickets generados con un importe que varía entre los 100 y 10.000 euros. La fecha corresponde a cualquier día y mes del año 2019. Tanto el importe como la fecha se tiene que generar de manera aleatoria. El trabajador que genera cada ticket debe de ser elegido aleatoriamente también.
* Cada ticket contiene entre 1 y 10 productos que se deben de asignar de manera aleatoria. La cantidad de cada producto del ticket debe de ser una asignación aleatoria que varíe entre 1 y 10 también.

**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 de postgreSQL tiene un [recolector de estadísticas](http://www.postgresql.org/docs/devel/static/monitoring-stats.html) que almacena la información en la tabla *pg\_statistic*. La tabla *pg\_statistic* almacena datos estadísticos sobre el contenido de la base de datos. Las entradas son creadas por ANALYZE y posteriormente utilizadas por el planificador de consultas.

Todos los datos estadísticos que recoge *pg\_statistic* son aproximados, incluso suponiendo que están actualizados. La tabla *pg\_statisic* también almacena datos estadísticos sobre los valores de las expresiones de índice.

La tabla *pg\_statistic* no debe ser legible por el público, ya que incluso la información estadística sobre un contenido de la tabla puede considerarse sensible como los valores mínimos y máximos de salario. Por tanto, *pg\_stats* es una vista públicamente legible en *pg\_statistic* que solo expone información sobre esas tablas que el usuario puede leer.

Las estadísticas que recolecta la tabla *pg\_statistic* son:

* *starelid*: identificador de la tabla o índice que describe la columna a la que pertenece
* *staattnum*: el número de la columna descrita
* *stainherit*: si es true, las estadísticas incluyen columnas de herencia secundarias, no sólo los valores de la relación especificada.
* *stanullfrac*: la fracción de las entradas de la columna que son nulas.
* *stawidth*: el ancho promedio almacenado, en bytes, sin considerar las entradas nulas.
* *stadistinct*: el número de valores distintos de datos no nulos en la columna. Un valor mayor que 0 es el número real de valores distintos.
* *stakindN*: un código numérico que indica el tipo de estadísticas almacenadas en la enésima “espacio” de la fila *pg\_statistic*.
* *staopN*: un operador solía derivar las estadísticas almacenadas en la enésima “espacio”.
* stacollN: la colación solía derivar las estadísticas almacenadas en la enésima “espacio”
* *stanumbersN*: estadísticas numéricas del tipo apropiado para la enésima “espacio”, o nula si el tipo de espacio no involucra valores numéricos.
* *stavaluesN*: valores de datos de columna del tipo apropiado para la enésima “espacio” o nula si el tipo de espacio no almacena ningún valor de datos.

Cuestión 2: Modifique el log de errores para que queden guardadas todas las operaciones que se realizan sobre cualquier base de datos. Indique los pasos realizados.

Como ya sabemos, la ruta del directorio de datos por defecto se encuentra en “C:/Program Files/PostgreSQL/12/data” por tanto sabemos que en esa ruta se encontrará el archivo de configuración de postgresql donde podremos modificar el log de errores.





Por defecto, el logging collector ya está a “on” y descomentado.



Para activar el log de errores de postgres de tal forma que queden guardadas todas las operaciones que se realizan sobre cualquier base de datos editaremos el archivo postgresql.conf.

Para ello descomentamos y modificamos la línea *log\_statement* que anteriormente estaba comentada y con valor none.



Si queremos sacar una consulta al log, tendrá que ver con la duración de la misma por lo que tendremos que modificar también la línea *log\_min\_duration\_statement* del apartado “When to log” que anteriormente estaba comentada y con valor -1.



Cuestión 3: Crear una nueva base de datos llamada **empresa** y que tenga las siguientes tablas con los siguientes campos y características:

* empleados(numero\_empleado tipo numeric PRIMARY KEY, nombre tipo text, apellidos tipo text, salario tipo numeric)
* proyectos(numero\_proyecto tipo numeric PRIMARY KEY, nombre tipo text, localización tipo text, coste tipo numeric)
* trabaja\_proyectos(numero\_empleado tipo numeric que sea FOREIGN KEY del campo numero\_empleado de la tabla empleados con restricciones de tipo RESTRICT en sus operaciones, numero\_proyecto tipo numeric que sea FOREIGN KEY del campo numero\_proyecto de la tabla proyectos con restricciones de tipo RESTRICT en sus operaciones, horas de tipo numeric. La PRIMARY KEY debe ser compuesta de numero\_empleado y numero\_proyecto.

Se pide:

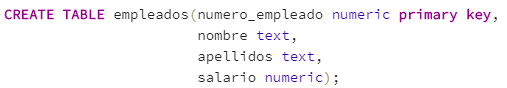
* Indicar el proceso seguido para generar esta base de datos.
* Cargar la información del fichero datos\_empleados.csv, datos\_proyectos.csv y datos\_trabaja\_proyectos.csv en dichas tablas de tal manera que sea lo más eficiente posible.
* Indicar los tiempos de carga.

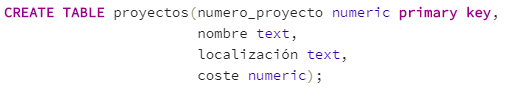
Para generar la base de datos según se ha indicado en el enunciado, hacemos lo siguiente:

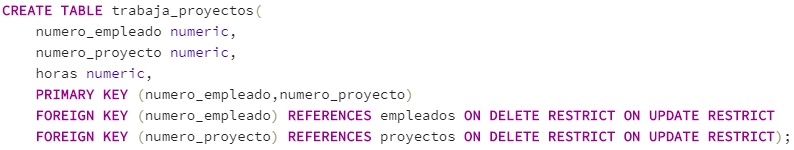
-         En primer lugar, creamos la base de datos:



-         Para continuar, creamos las diferentes tablas:



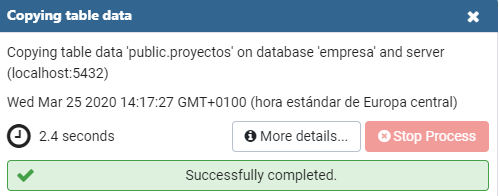


**

* Para cargar los datos en las tablas, hemos usado el comando COPY ya que está optimizado para cargar grandes cargas de datos.
* Por último, los tiempos de carga de los ficheros csv son los siguientes:
* Tabla empleados

**

* Tabla proyectos

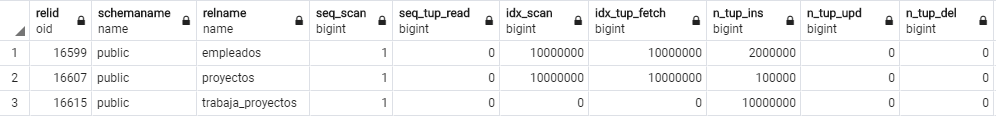
**

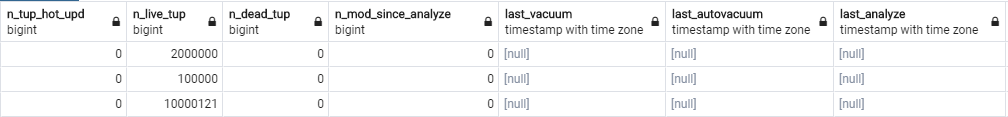
* Tabla trabaja\_proyectos

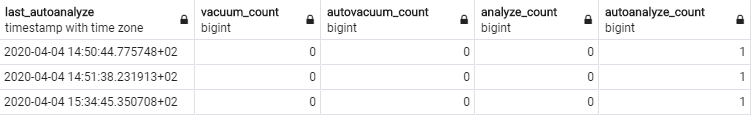
**

Cuestión 4: Mostrar las estadísticas obtenidas en este momento para cada tabla. ¿Qué se almacena? ¿Son correctas? Si no son correctas, ¿cómo se pueden actualizar?









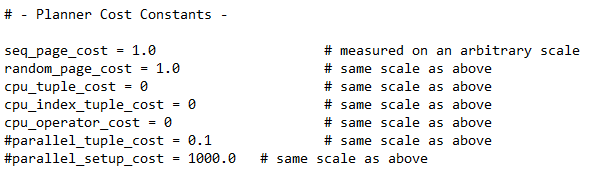
Con la sentencia select \* from pg\_stat\_user\_tables obtenemos una serie de datos estadísticos acerca de las tablas de usuario existentes en la base de datos. Entre esta serie de datos se almacenan lo siguiente:

* *relid:* es el OID de la tabla (identificador)
* *schemaname*: es el nombre del esquema en el que se encuentra la tabla
* *relname*: muestra el nombre de la tabla
* *seq\_scan:* el número de escaneos secuenciales iniciados en esa tabla
* *seq\_tup\_read:* muestra el número de tuplas vivas traídas por dicho escaneo secuencial
* *idx\_scan*: el número de escaneos de índice realizados sobre la tabla
* *idx\_tup\_fetch*: el número de tuplas vivas traídas tras la realización de los escaneos sobre los índices.
* *n\_tup\_ins*: el número de filas insertadas
* *n\_tup\_upd*: el número de filas actualizadas
* *n\_tup\_del:* el número de filas borradas
* *n\_tup\_hot\_upd*: el número de filas HOT actualizadas
* *n\_live\_tup:* número estimado de filas vivas
* *n\_dead\_tup:* número estimado de filas muertas
* *last\_vacuum*: la última vez en la que se le realizó un vacuum manual a la tabla
* *last\_autovacuum*: la última vez en la que se le realizó un vacuum automático a la tabla
* *last\_analyze:* la última vez en la que esta tabla fue analizada por el autovacuum daemon
* *vacuum\_count*: el número de veces que se ha realizado un vacuum manual a esta tabla
* *autovacuum\_count*: el número de veces que se ha realizado un vacuum automático a esta tabla
* *analyze\_count*: el número de veces que se ha analizado manualmente a esta tabla
* *autoanalyze\_count*: número de veces que esta tabla ha sido analizada por el autovacuum.

Todos los datos estadísticos recogidos por postgresql y descritos anteriormente son correctos ya que de momento no hemos manipulado datos, simplemente hemos insertado los datos de los archivos csv. En caso de que algún dato estadístico fuese incorrecto, habría que aplicar un VACUUM sobre la tabla para actualizar las estadísticas y reasignar el espacio.

Cuestión 5: Configurar PostgreSQL de tal manera que el coste mostrado por el comando EXPLAIN tenga en cuenta solamente las lecturas/escrituras de los bloques en el disco de valor 1.0 por cada bloque, independientemente del tipo de acceso a los bloques. Indicar el proceso seguido y la configuración final.

Los costos se miden en unidades arbitrarias determinadas por los parámetros de costos del planificador. Para configurar esta opción tendremos que modificar de nuevo el archivo postgresql.conf de forma que accederemos a la sección del planificador de costes y modificaremos algunos parámetros para que quede de la siguiente manera:



Descomentamos *seq\_page\_cost* y *random\_page\_cost* y le damos valor 1.0.

*random\_page\_cost* establece la estimación del planificador del costo de una página de disco no recuperada secuencialmente. Este valor predeterminado es 4.0. La reducción de este valor en relación con *seq\_page\_cost* hará que el sistema prefiera escaneos de índice. Aumentarlo hará que los escaneos de índice parezcan relativamente más caros. Cambiarlo ambos valores servirá para cambiar la importancia de los costos de E/S de disco en relación con los costos de CPU, que se describen por los siguientes parámetros. Es decir, al cambiarlo a 1.0, significa que por cada bloque/página que Postgres lea, contará 1 bloque como lo estamos viendo en teoría.

El acceso aleatorio al almacenamiento en disco suele ser mucho más costoso que cuatro veces el acceso secuencia. Sin embargo, se usa un valor predeterminado más bajo (4.0) porque la mayoría de los accesos aleatorios al disco, como las lecturas indexadas, se supone que están en caché.

En consecuencia, si nuestros datos están principalmente en caché como cuando tenemos una base de datos más pequeña que el total de memoria del servidor, disminuir el *random\_page\_cost* puede ser apropiado.

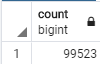
El almacenamiento que tiene una lectura aleatoria baja podrían modelarse mejor con un valor más bajo para *random\_page\_cost.*

Por otra parte el comando *seq\_page\_coste* establece la estimación del planificador del costo de una búsqueda de página de disco que es parte de una búsqueda secuencial. El valor predeterminado es 1.0.

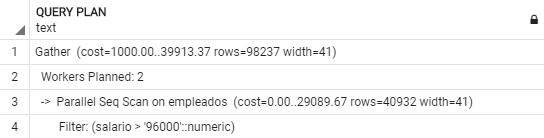
Por último, hemos modificado los costes de la CPU a 0 ya que en teoría, despreciamos estos costes.

Cuestión 6: Aplicar el comando EXPLAIN a una consulta que obtenga la información de los empleados con salario de más de 96000 euros. ¿Son correctos los resultados del comando EXPLAIN? ¿Por qué? Comparar con lo que se obtendría con lo visto en teoría.









PostgreSQL consta de una característica conocida como consulta paralela. La consulta paralela puede aprovechar múltiples CPU para responder consultas más rápido. La consulta paralela es a menudo muy significativa, llegando algunas consultas a ejecutarse más del doble de rápido cuando se usa la consulta paralela y otras llegando a ejecutarse cuatro veces más rápido o incluso más.

En este caso, el optimizador de Postgres determina que la consulta paralela es la estrategia de ejecución más rápida para la consulta y crea un plan de consulta que incluye un nodo “Gather”.

Usando el comando EXPLAIN podemos ver que la cantidad de “workers” elegidos por el planificador es de 2. Cuando se alcanza al nodo “Gather” durante la ejecución de la consulta, el proceso que implementa la sesión del usuario solicitará una cantidad de procesos workers en segundo plano igual a la cantidad de workers elegidos por el planificador. La cantidad de workers elegidos por el planificador está limitado a lo más por max\_parallel\_workers\_per\_gather, en nuestro caso 2. Todos los procesos workers en segundo plano de una consulta paralela dada ejecutarán una porción paralela del plan.

Se realiza una búsqueda secuencial ya que al no existir ningún índice y al no estar ordenado por ningún campo, la forma posible de realizar una selección es aplicando una búsqueda secuencial.

Lo que se obtiene al aplicar el explain es lo siguiente:

* Coste de inicio estimado: Es el tiempo empleado antes de que pueda comenzar la fase de salida. En este caso el coste de inicio estimado es 1000,00.
* Coste estimado total: Se asume que el nodo del plan se ejecuta hasta su finalización. En nuestro caso 39913,37.
* Número estimado de filas de salida por el nodo del plan. De nuevo, se asume que el nodo se ejecuta hasta su finalización. En nuestro caso 98237 filas de salida.
* Ancho promedio estimado de filas de salida por este nodo del plan en bytes. En nuestro caso 41 bytes.

En resumen, Postgres hace una búsqueda secuencial en paralelo con 2 workers y estima que se recuperarán 98.237 tuplas de la consulta realizada. Por tanto, esta aproximación es buena y se acerca mucho a la realidad que como hemos visto antes eran 99.523, por lo que se podría decir que los resultados son correctos.

**CÁLCULO TEÓRICO DEL COSTE**

Para calcular el coste teórico de la siguiente consulta consideramos lo siguiente:

Al no haber ningún índice y no estar el archivo ordenado por ningún campo, solo podremos aplicar una búsqueda secuencial.

Los campos de la tabla empleados son los siguientes:

* numero\_empleado de tipo numeric
  + LNUMERO\_EMPLEADO = 6 bytes
* nombre de tipo text
  + LNOMBRE = 13 bytes
* apellidos de tipo text
  + LAPELLIDOS = 16 bytes
* salario de tipo text
  + LSALARIO = 6 bytes

LR = Σ LCAMPO = 6 + 13 + 16 + 6 = 41 bytes/registro

El factor de bloque viene dado por el tamaño de un bloque entre la longitud del registro. El tamaño de un bloque en PostgreSQL por defecto es de 8.192 bytes. Por tanto:

fR = B/LR = 8.192/41 = 154.566 = 199 registros/bloque

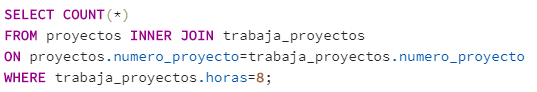
Por último, sacaremos el número de bloques en total que necesitamos si tenemos 2.000.000 de registros:

bR = nR/fR = 2.000.000/199 = **10.051 bloques en total**

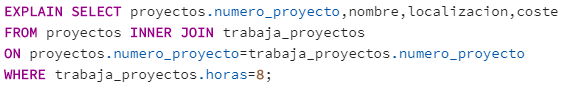
Como podemos ver, el coste calculado teóricamente es mucho menor al que estima postgres, esto es ya que postgres calcula de una forma distinta los costes y además, nosotros no estamos considerando por ejemplo las longitudes de control de las cabeceras.

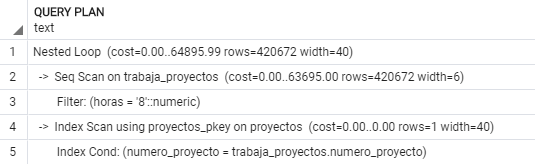
La estimación del explain es correcta ya que se acerca bastante a los valores reales aplicando reglas heurísticas. Como hemos podido observar, la estimación de tuplas se acercaba al valor real aunque no es exacto.

Cuestión 7: Aplicar el comando EXPLAIN a una consulta que obtenga la información de los proyectos en los cuales el empleado trabaja 8 horas. ¿Son correctos los resultados del comando EXPLAIN? ¿Por qué? Comparar con lo que se obtendría con lo visto en teoría.







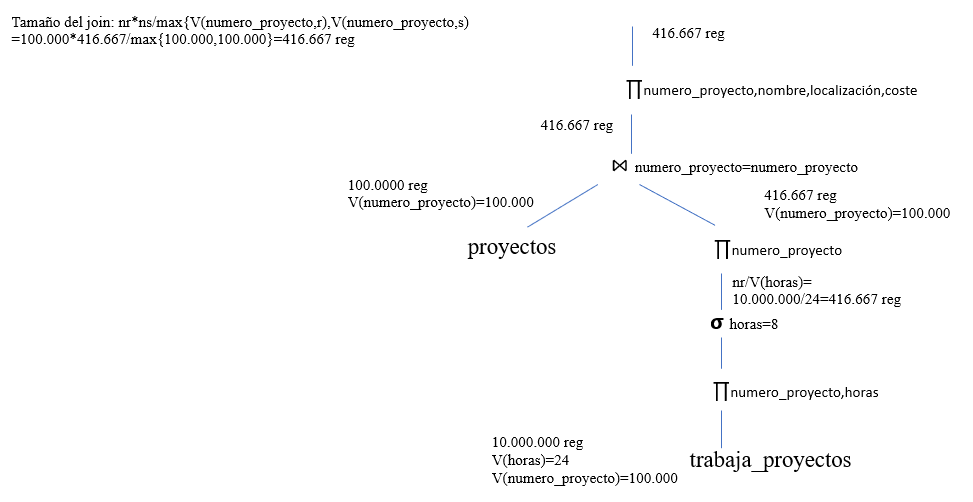


De nuevo vemos que se realiza una búsqueda secuencial sobre la tabla *trabaja\_proyectos* ya que Postgres no encuentra una manera mejor de recorrer la tabla que hacer una búsqueda secuencial. Por otra parte, se hace una búsqueda mediante el índice *proyectos\_pkey* en la tabla proyectos.

El coste estimado total de la tabla sería 64.896 bloques y estima que el número total de filas que cumplen la query son 420.672. Como hemos visto el número real de tuplas que cumplen la condición son 416.698, por lo que Postgres se acerca considerablemente al resultado real.

**CÁLCULO TEÓRICO DEL COSTE**

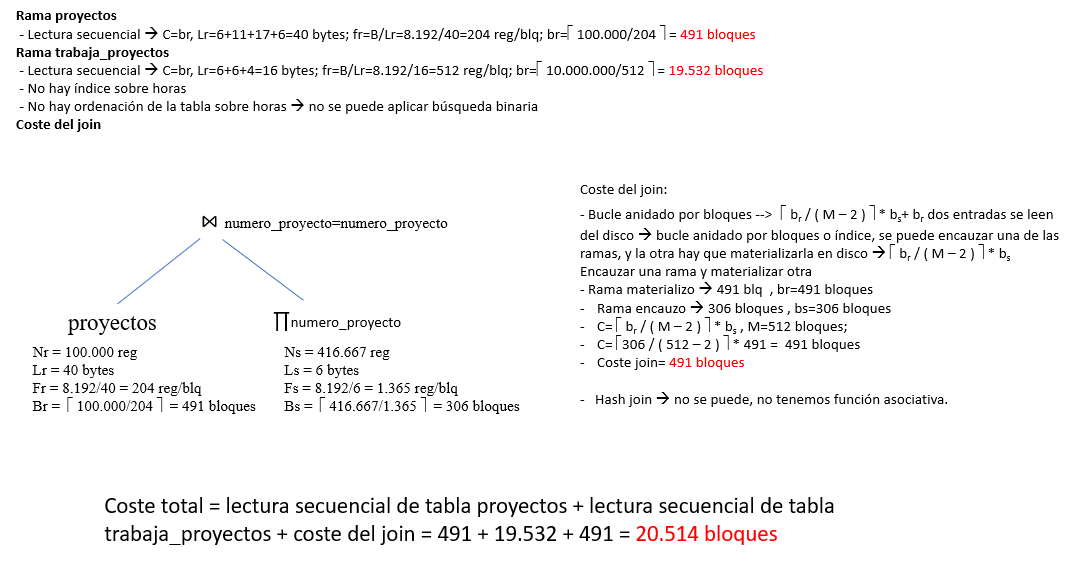
Para calcular el coste teórico de la siguiente consulta consideramos lo siguiente:



Para calcular el coste teórico hemos considerado lo siguiente:

* El join está sacando directamente los datos de la tabla proyectos, es decir, no hay ninguna selección antes del join, por lo que tira directamente de la tabla realizando una búsqueda secuencial sobre ella que como hemos calculado son 9.804 bloques.
* La rama *trabaja\_proyectos* llega al join con 100.000 registros. Previamente se ha realizado una selección sobre el campo horas, como no tenemos índice ni la tabla ordenada sobre el campo horas, sólo se podrá aplicar una búsqueda secuencial, que hemos calculado y es 19.532 bloques.
* Para calcular el coste del join hemos realizado un bucle anidado por bloques.
  + Para calcular el tamaño de la memoria hemos accedido a los archivos de postgres de configuración donde hemos podido observar que el tamaño de la memoria es de 4MB. Como un bloque en postgres es de 8KB y trabajamos con la memoria en bloques, dividimos 4MB/8KB = 512 bloques, por lo que el tamaño de M que usa postgres es de 512 bloques.
  + Como el join tira directamente de la tabla proyectos, significa que la tabla proyectos ya está materializada por lo que sólo nos queda encauzar la rama de la tabla *trabaja\_proyectos.*
  + Finalmente nos sale un coste de 491 bloques para realizar el join
* Si sumamos todos los costes, teóricamente sacamos un coste de **20.514 bloques** para realizar esta consulta.

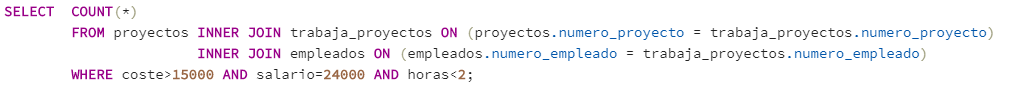
Adjuntamos una captura de cómo hemos realizado el coste del ejercicio detalladamente:



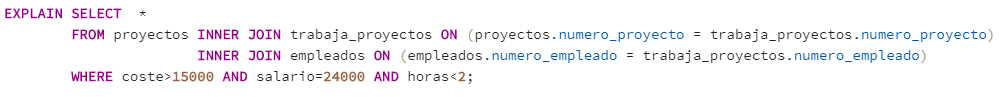
Como vemos, el coste calculado teóricamente es mucho menor que el que estima Postgres. Esto pasa ya que postgres calcula de forma distinta los costes, además nosotros no consideramos la longitud de control en las cabeceras de los bloques, por lo que las tablas en postgres ocupan más de lo que nosotros consideramos.

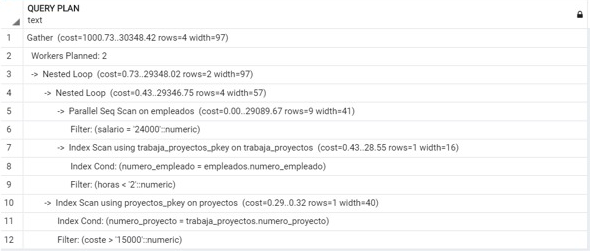
Por otra parte, los resultados del explain no son los reales, sino que son valores estimado para la consulta ejecutada. El explain saca un coste estimado de la consulta basándose en una heurística que se aproxima considerablemente al coste real.

Cuestión 8: Aplicar el comando EXPLAIN a una consulta que obtenga la información de los proyectos que tienen un coste mayor de 15000, y tienen empleados de salario de 24000 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.









En este caso, el optimizador de Postgres crea un plan de consulta que incluye un nodo “Gather”. La cantidad de “workers” elegidos por el planificador es de 2. Por otra parte, Postgres realiza la estimación haciendo lo siguiente:

Realiza un “Nested Loop” en el que incluye:

* Realiza una búsqueda sobre el índice *proyectos\_pkey* en la tabla proyectos filtrando aquellas tuplas que tengan coste mayor que 15.000
* Realiza un “Nested Loop” en el que incluye:
  + Una búsqueda paralela secuencial sobre empleados filtrando aquellas tuplas que tengan salario igual a 24.000
  + Una búsqueda sobre el índice *trabaja\_proyectos\_pkey* en la tabla *trabaja\_proyectos* filtrando aquellas que cumplen horas menor que 2.

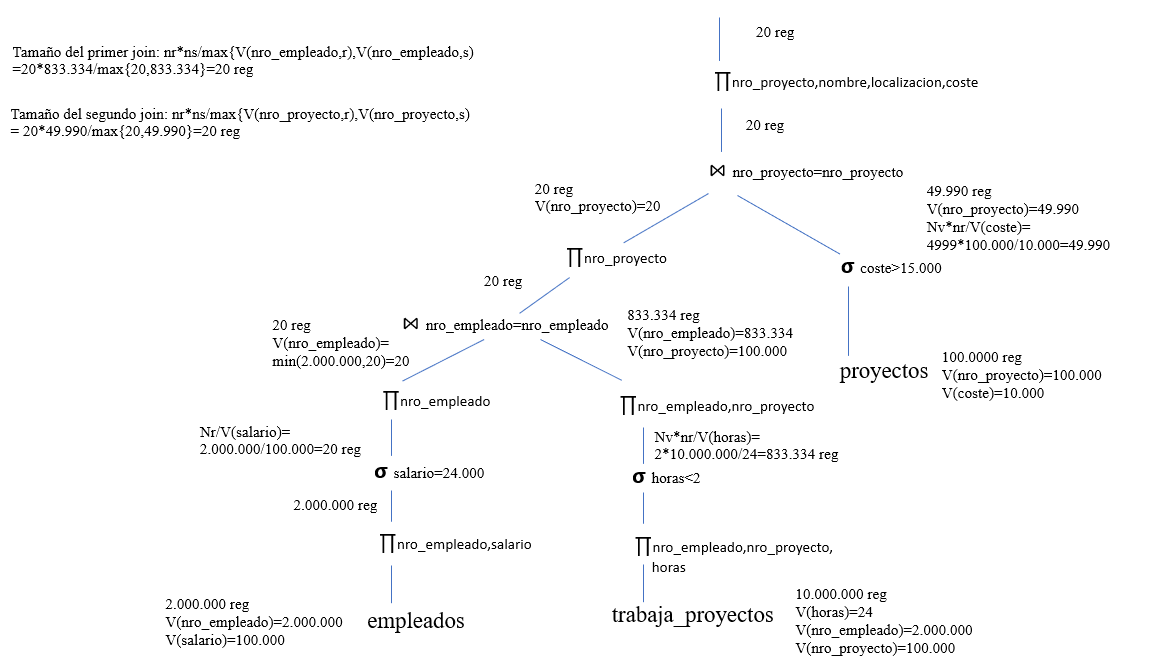
Finalmente, lo que se obtiene al aplicar el EXPLAIN se resumen en el nodo Gather:

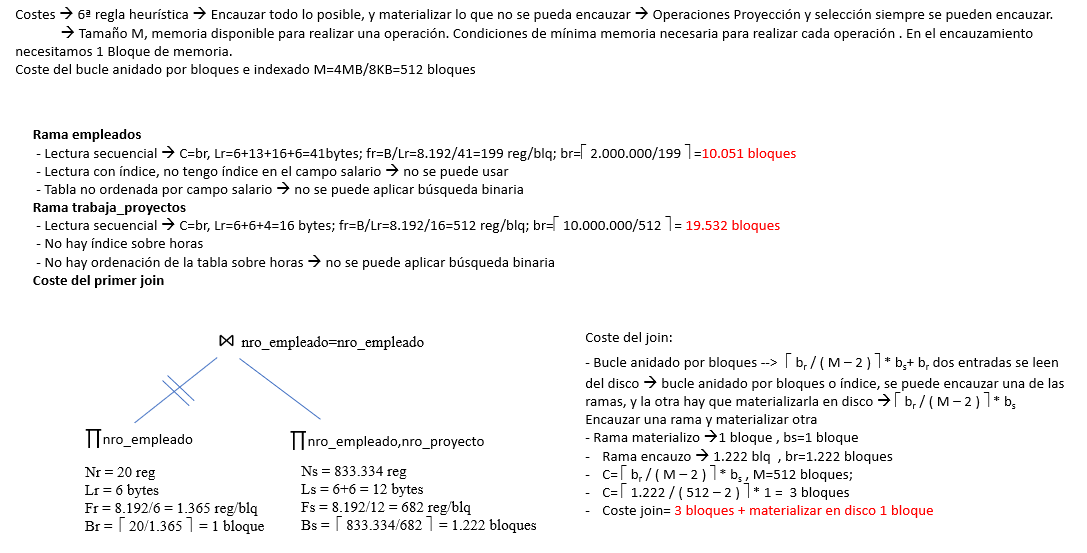
* Coste de inicio estimado: 1000,73
* Coste estimado total: 30348,32
* Número estimado de filas de salida por el nodo del plan: 4 filas
* Ancho promedio estimado de filas de salida por este nodo del plan en bytes. En nuestro caso 97 bytes.

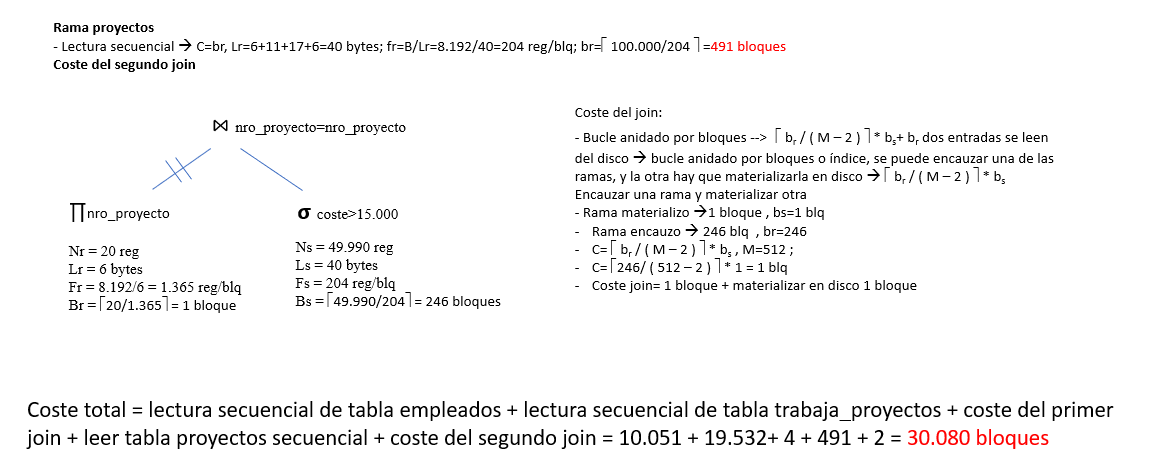
En resumen, Postgres hace una búsqueda secuencial en paralelo con 2 workers y estima que se recuperarán 98.237 tuplas de la consulta realizada. Por tanto, esta aproximación es buena y se acerca mucho a la realidad que como hemos visto antes eran 99.523, por lo que se podría decir que los resultados son correctos.

**CÁLCULO TEÓRICO DEL COSTE**

Para calcular el coste teórico de la siguiente consulta consideramos lo siguiente:







Como se puede observar el coste teórico, nos ha salido que es 30.080 bloques. Este coste vemos que se parece mucho a lo que ha salido como resultado de la estimación del explain.

Como podemos observar, en este caso los resultados del explain también se aproximan al resultado real, aunque estima que saldrán 4 filas y realmente salen 2. Como hemos dicho anteriormente, el explain solo saca una estimación usando heurísticas y es por ello que podría coincidir con el valor real o no, lo que sí sabemos es que los valores serán aproximados a los reales.

Cuestión 9: Realizar la carga masiva de los datos mencionados en la introducción con la integridad referencial deshabilitada (tomar tiempos) utilizando uno de los mecanismos que proporciona postgreSQL. Realizarlo sobre la base de datos suministrada TIENDA. Posteriormente, realizar la carga de los datos con la integridad referencial habilitada (tomar tiempos) utilizando el método propuesto. Especificar el orden de carga de las tablas y explicar el porqué de dicho orden. Comparar los tiempos en ambas situaciones 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?

El orden de aparición de los tiempos de carga en la siguiente tabla especifica el orden en que han sido introducidos los datos.

|  |  |  |
| --- | --- | --- |
| **Tabla** | **Tiempo sin integridad** | **Tiempo con integridad** |
| Tienda | 1.2 segundos | 1.21 segundos |
| Productos | 14.7 segundos | 13.77 segundos |
| Trabajador | 11.8 segundos | 21.15 segundos |
| Ticket | 31.82 segundos | 82.82 segundos |
| Tienda\_Productos | 144.26 segundos | 713.63 segundos |
| Ticket\_Productos | 188.33 segundos | 961.2 segundos |

En este ejercicio se ha realizado la carga masiva de datos en la base de datos, primero eliminando la integridad referencial de los datos y, posteriormente, con dicha integridad activada, para comprobar y analizar diferencias.

Lo primero que debemos observar es el orden de carga de los datos dentro de la base de datos. Esto no es algo aleatorio, ya que necesitamos tener un orden para poder introducir los datos.

Básicamente el orden está tal y como se ve en la tabla superior porque debemos introducir primero los datos de las tablas tienda, productos y trabajadores ya que estas no dependen de ningún atributo de otras tablas. Sin embargo, si observamos *ticket, tienda\_productos* y *ticket\_productos,* nos damos cuenta de que algunos de sus atributos son claves de otras tablas, por tanto, es necesario introducir los datos en ese orden porque si no, sería imposible introducir los datos que hemos introducido en último lugar.

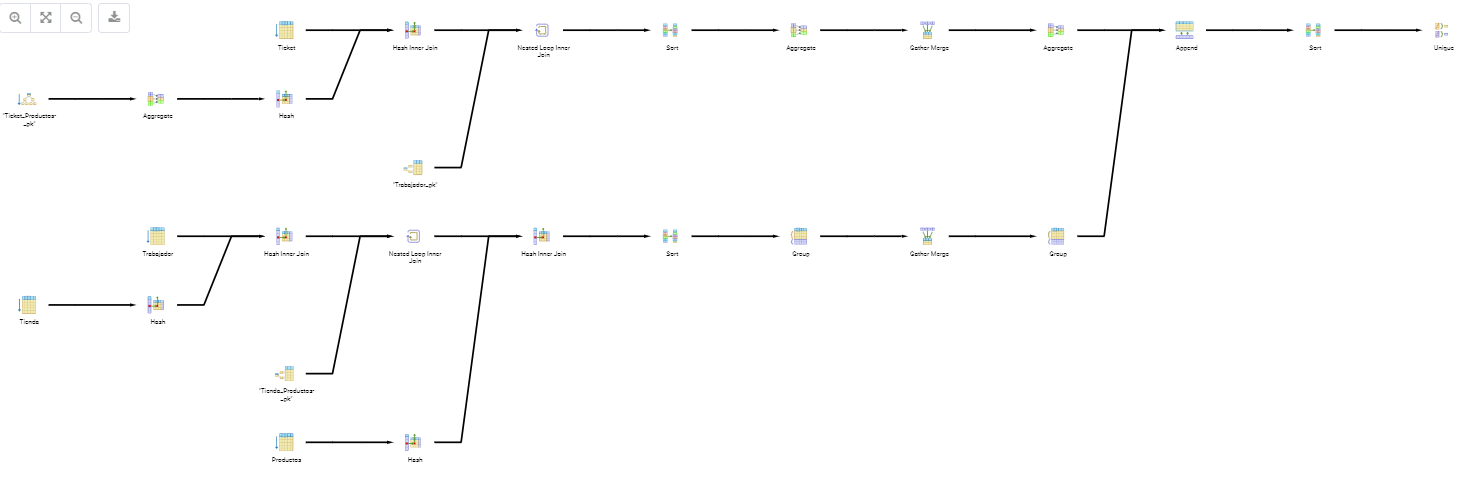
A partir de este momento en adelante, se deben de realizar las siguientes cuestiones con la base de datos que tiene la integridad referencial activada.

Cuestión 10: Realizar una consulta SQL que muestre “el nombre y DNI de los trabajadores que hayan vendido algún ticket en los cuatro últimos meses del año con más de cuatro productos en los que al menos alguno de ellos tenga un precio de más de 500 euros, junto con los trabajadores que ganan entre 3000 y 5000 euros de salario en la Comunidad de Madrid en las cuales hay por lo menos un producto con un stock de menos de 100 unidades y que tiene un precio de más de 400 euros.”

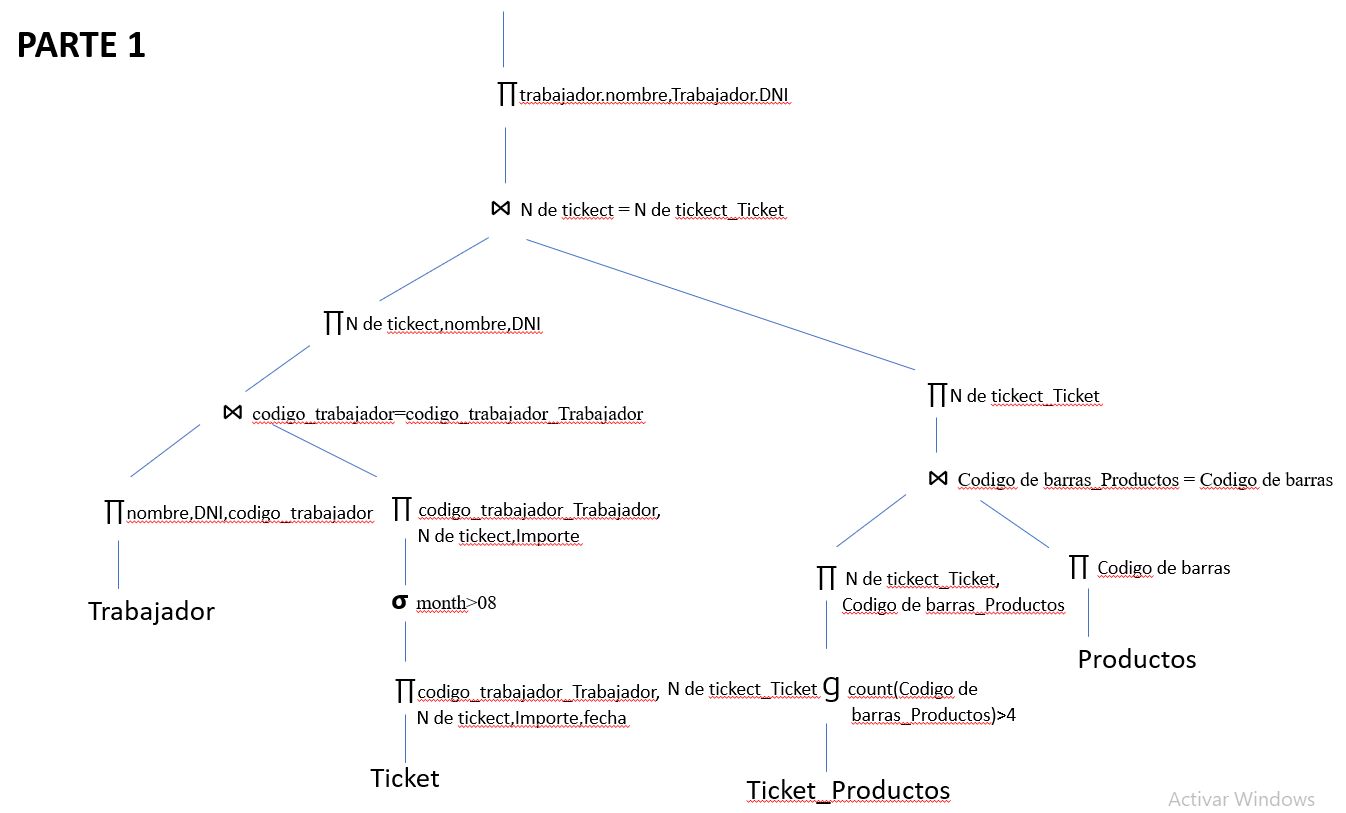
Obtener el plan de ejecución con el resultado del comando EXPLAIN en forma de árbol de álgebra relacional. Explicar la información obtenida en el plan de ejecución de postgreSQL. Comparar el árbol obtenido por nosotros al traducir la consulta original al álgebra relacional y el que obtiene postgreSQL.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

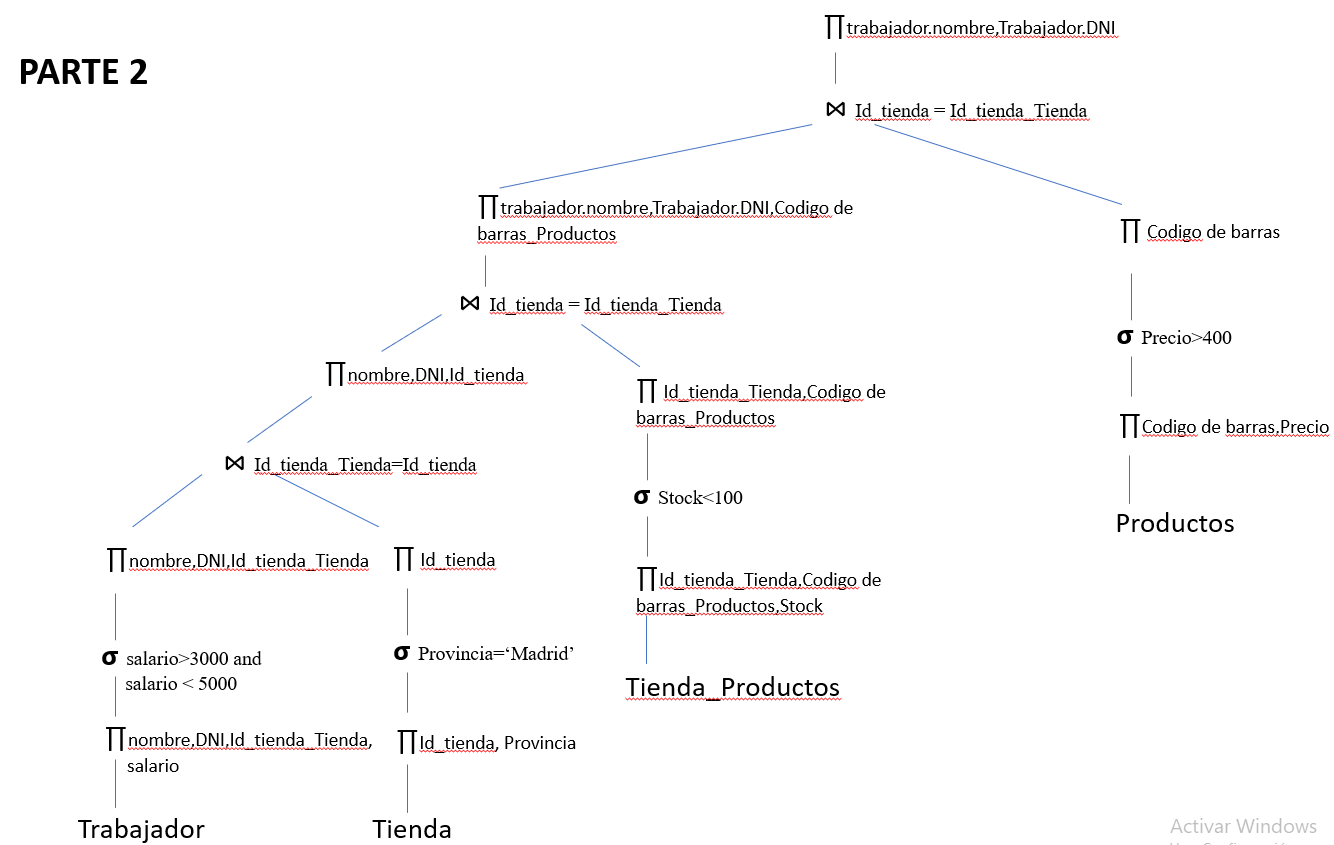
Para realizar la obtención de datos propuesta en el enunciado, hemos realizado la siguiente consulta:

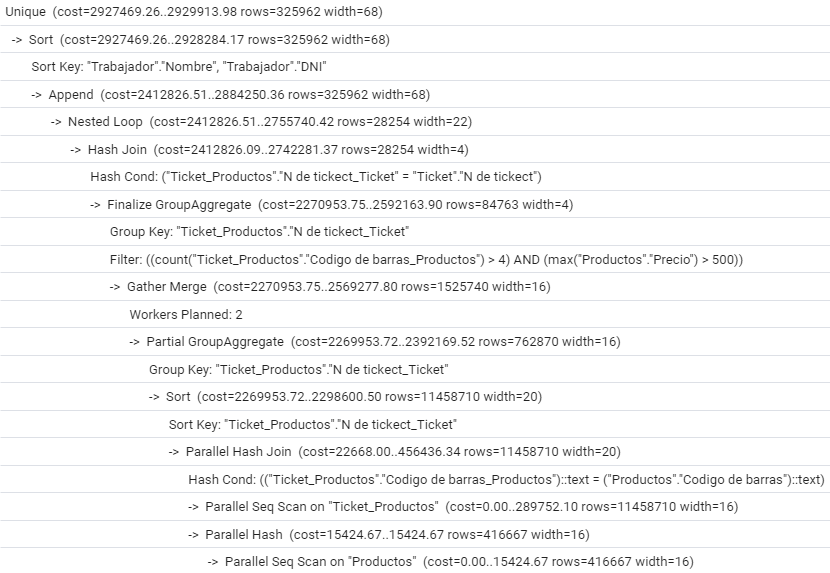


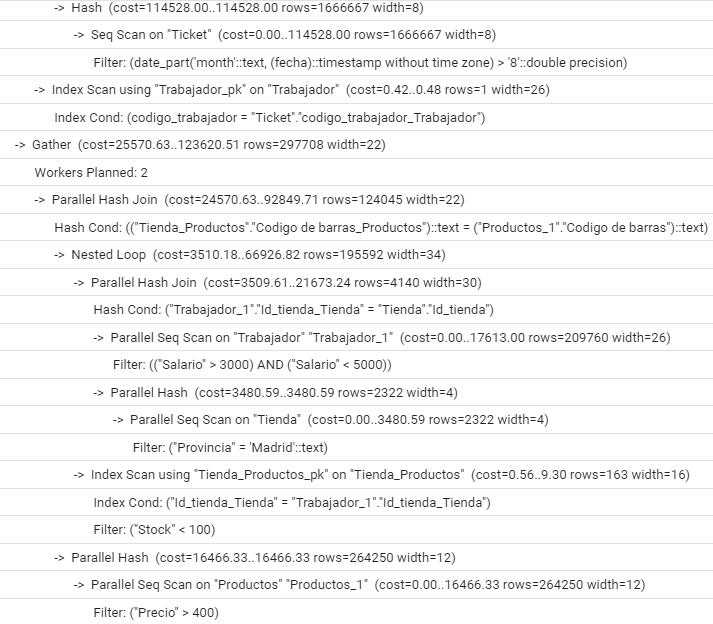
~~~~

En las siguientes dos imágenes adjuntamos el árbol de consulta relacional obtenido según lo visto en teoría, lo hemos realizado en dos partes, cada una correspondiente con las partes de la consulta, la primera y la segunda parte respectivamente.

~~~~

~~~~





En las capturas adjuntadas anteriormente se puede observar tanto la consulta realizada para extraer la información solicitada como la ruta seguida por el planificador de la base de datos al ejecutarla, tanto en forma de árbol como con el comando explain.

Como se puede observar el coste de esta consulta es bastante alto. Esto se debe a que se requiere la extracción de una gran cantidad de datos, todos ellos relacionados entre sí y con varias condiciones que afectan a más de una tabla.

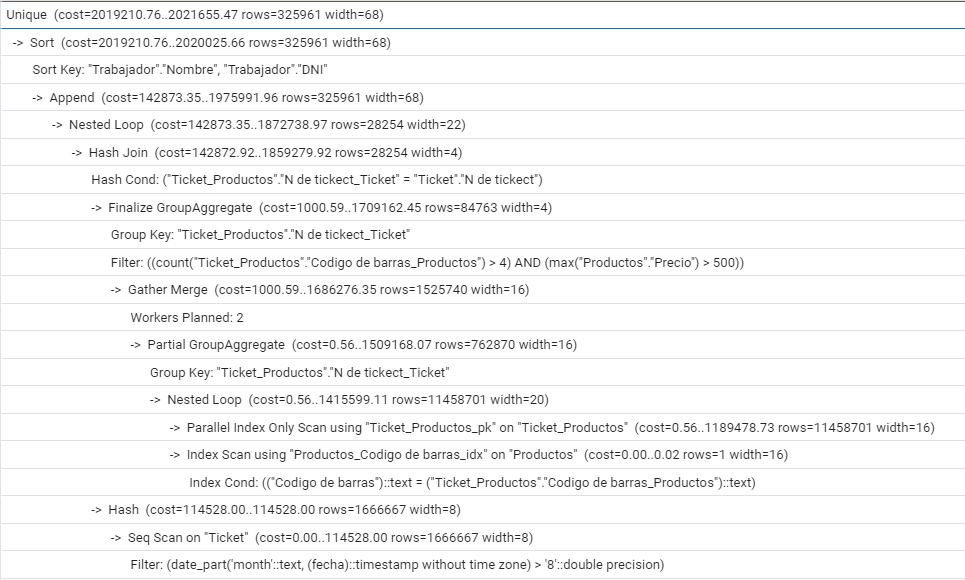
La consulta por tanto está dividida en dos partes que se juntan gracias a la expresión union.

Para la primera parte el planificador realiza un inner join entre la tabla Ticket\_Productos y la tabla Ticket, teniendo en cuenta la selección con las características correspondientes (la fecha > 08 en el caso del Ticket y los Tickets con más de 4 productos en el caso de la tabla Ticket\_Productos) mediante los índices que se crea postgres automáticamente a través de la clave primaria de ambas tablas. Como no existe ningún otro índice para las tablas pues postgres considera que usar dicho índice creado automáticamente es el más adecuado a la hora de llevar a cabo la consulta.

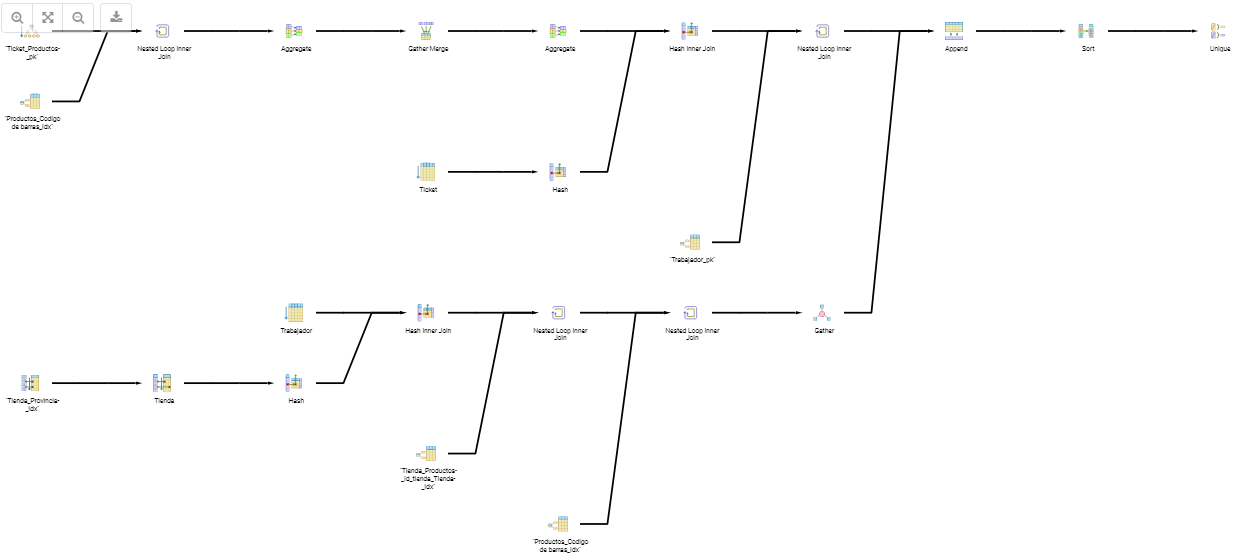
Para el resto de la consulta podemos observar que postgres realiza la misma operación que acabamos de comentar para realizar los inner join entre las distintas tablas.

Dado que no existen índices creados por nosotros, el coste resultante de realizar esta consulta es muy elevado, dado el número grande de tuplas que existen en cada tabla.

Cuestión 11: 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 10 y explicar los resultados. Obtener el plan de ejecución con el resultado del comando EXPLAIN en forma de árbol de algebra relacional. Comentar los resultados obtenidos y comparar con la cuestión anterior.







Al observar y ejecutar la consulta de la pregunta 10, cómo se ha comentado anteriormente el coste es bastante elevado, ya que al elevado número de datos que existen en la tabla, se añaden unas cuantas condiciones y joins que hacen que la consulta se complique. Sin embargo, existen maneras para realizar la misma de una forma más eficiente.

La manera que hemos elegido ha sido la introducción de índices en las tablas de la base de datos.

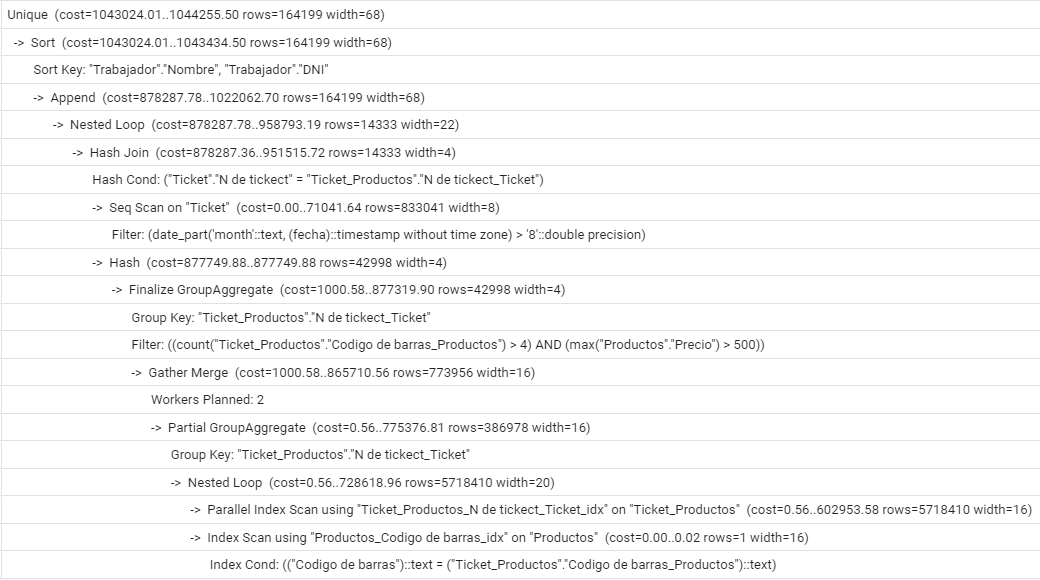
Para realizar la optimización hemos introducido una variedad de índices usando tanto hash como árboles btree, dichos índices en los campos que se utilizan en la consulta.

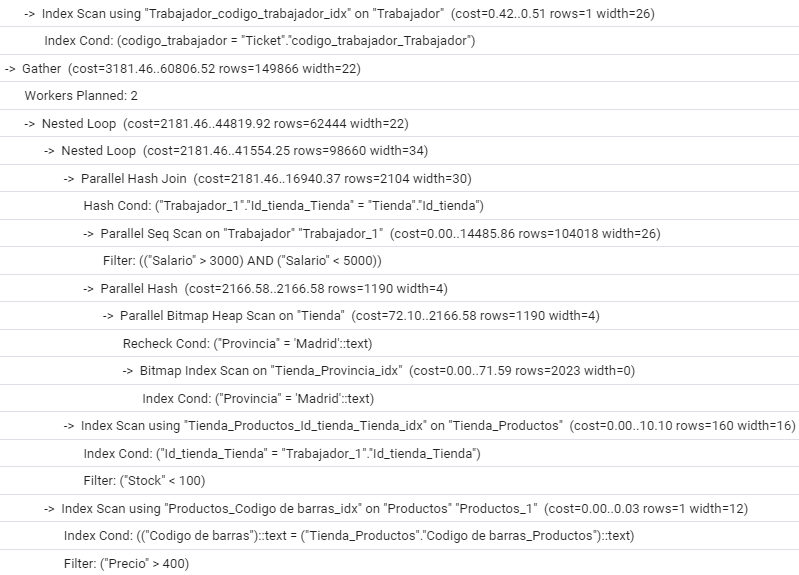
Como se puede observar si hacemos una comparación entre las informaciones del comando explain en la pregunta 10 y en la 11, podemos observar un cambio significativo en el coste de la consulta.

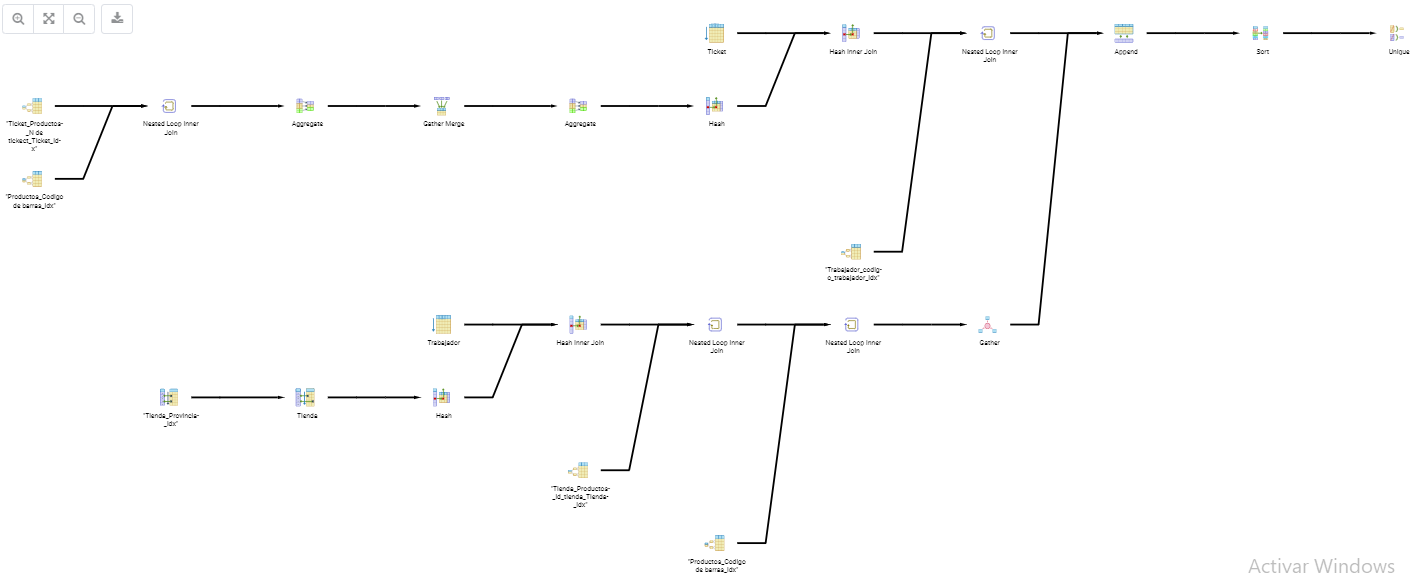
Una vez implementados los índices podemos ver que el coste se reduce en más de 900.000 tuplas.

Por tanto, podemos observar cómo gracias a los índices, nuestra consulta ha sido optimizada con éxito.

Cuestión 12: Usando PostgreSQL, borre el 50% de las tiendas almacenadas 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 algebra relacional. Comparar con los resultados anteriores.







En esta cuestión referente al borrado, se han realizado numerosas operaciones para conseguir borrar todos los datos pedidos en el enunciado, de manera eficiente.

Esta cuestión requiere de diversas operaciones previas al borrado, ya que, como hemos comentado anteriormente, las tablas de nuestra base de datos están muy pobladas con millones de datos.

Además de contener muchísimos datos, las tablas poseen integridad referencial, esto es que hay tablas relacionadas con otras, por lo que, si se borra una tupla de una tabla, se debe de borrar también dicha tupla de las tablas que sea necesario.

Por tanto, realizar un borrado masivo de datos “a lo bruto”, que sería directamente borrar sin tener en cuenta nada, nos llevaría una cantidad inmensa de tiempo, incluso hablaríamos de varios días realizando borrado.

Para evitar este colapso de nuestro ordenador, se ha seguido un proceso con diversas fases:

1. **Selección de un orden a través del cual borrar los datos de las tablas:**

En este apartado hemos elegido un orden para ir borrando los datos tabla a tabla.

Pero ¿por qué es necesario un orden? Bien, como hemos comentado anteriormente, las tablas están relacionadas entre sí mediante la integridad referencial, por tanto, si borramos tuplas de una tabla “padre”, deberíamos de borrar también las mismas de la tabla “hija”. Por tanto, igual que a la hora de insertar los datos hemos tenido en cuenta un orden, al borrarlos hay que seguir el mismo orden de manera invertida.

Por tanto, se han borrado los datos de las tablas de la siguiente manera:

Primero los datos de Ticket\_Productos y Tienda\_Productos, tras esto los datos de tickets, luego Trabajador y por último la tabla Tienda.

Gracias a este orden conseguiremos evitar tener que borrar datos de dos tablas a la vez.

1. **Creación de tablas temporales en las cuáles guardar los datos que se van a borrar.**

En este apartado lo que se pretende hacer es un conjunto de tablas temporales cuya intención es guardar temporalmente datos que se van a borrar.

Para ello tendremos tres tablas temporales:

* temp\_Tienda: esta será la primera tabla que crearemos. Su único componente es una columna llamada id\_tienda que guardará 100.000 ids de tienda generados aleatoriamente.
* temp\_Trabajadores: una vez tenemos la tabla anterior, seleccionamos los códigos de los trabajadores que trabajan en las tiendas que hemos guardado en la tabla temp\_Tienda y los guardamos en la tabla temp\_Trabajadores. Con esto tendríamos una tabla temporal que guarda los códigos de todos los trabajadores que vamos a borrar.
* temp\_Tickets: una vez tenemos los trabajadores, es necesario borrar los tickets generados por estos trabajadores. Por tanto, en esta tabla guardaremos todos los números de ticket que han producido los trabajadores que se encuentran en temp\_Trabajadores.

1. **Creación de índices en las tablas en las que vamos a borrar datos y en las tablas temporales.**

En este punto hemos creado un índice en cada una de las tablas sobre las que vamos a borrar datos. Este índice estará basado en los campos guardados en las tablas temporales. Esto consigue que los borrados sean más eficientes, ya que la búsqueda de datos sería más rápida.

1. **Deshabilitación temporal de las foreign key de las tablas que lo necesiten**.

Si deshabilitamos mientras estamos borrando datos (algo que no afecta a la integridad de la base de datos) las claves foráneas de las distintas tablas, conseguiremos que nuestro borrado sea más liviano. Por ello, a través de los siguientes comandos, deshabilitamos algunas claves foráneas temporalmente.

* *alter table “Trabajadores” drop constraint “Tienda\_fk”*
* *alter table “Ticket” drop constraint “Trabajador\_fk”*

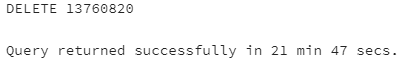
Sin embargo, tenemos que tener en cuenta esto cuando terminamos de realizar el borrado. Con esto nos referimos a que cuando terminemos de borrar, hay que volver a activar las claves foráneas que hemos deshabilitado anteriormente. Esto lo haremos mediante lo siguiente:

* *alter table “Trabajador” add constraint “Tienda\_fk” foreign key (“Id\_tienda\_Tienda”) references table “Tienda”(“Id\_tienda”) match full on delete cascade on update cascade*
* *alter table “Ticket” add constraint “Trabajador\_fk” foreign key (“codigo\_trabajador\_Trabajador”) references “Trabajador” (“codigo\_trabajador”) match full on delete cascade on update cascade*

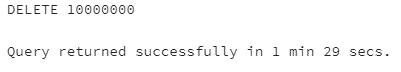
Por tanto, aquí tendríamos todo el proceso a realizar para borrar los datos pedidos en el ejercicio 12.

Para borrar de las tablas hemos utilizado la siguiente consulta:

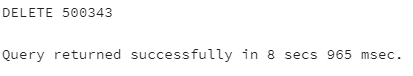
* Tabla Tickets\_Productos:
  + *delete from “Ticket\_Productos” where “N de tickect\_Ticket” in (select “n\_ticket” from temp\_Tickets)*

**

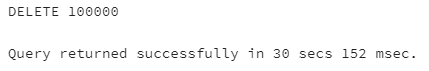
* Tabla Tienda\_Productos:
  + *delete from “Tienda\_Productos” where “Id\_tienda\_Tienda” in ( select “id\_tienda” from temp\_Tienda)*

**

* Tabla Ticket:
  + *delete from “Ticket” where “N de tickect” in (select “n\_ticket” from temp\_Tickets*
  + Se eliminó en algo más de 1 minuto, se nos olvidó hacer la captura.
* Tabla Trabajador:
  + *delete from “Trabajador” where “Id\_tienda\_Tienda” in (select “id\_tienda” from temp\_Tienda)*

**

* Tabla Tienda:
  + *delete from “Tienda” where “Id\_tienda” in (select “id\_tienda” from temp\_Tienda)*

**

Con esto habríamos terminado el borrado de datos requerido.

Cuestión 13: ¿Qué técnicas de mantenimiento de la BD propondría para mejorar los resultados de dicho plan sin modificar el código de la consulta? ¿Por qué?

Para mejorar los resultados del plan de ejecución podemos usar las siguientes técnicas de mantenimiento:

**VACUUM**

Las bases de datos de PostgreSQL requieren mantenimiento periódico conocido como vacuuming.

En Postgres una actualización o eliminación de una tupla no elimina inmediatamente la tupla. Eventualmente, las filas eliminadas ya no son de interés para ninguna transacción y están ocupando espacio que puede ser reutilizado por nuevas filas, para evitar un crecimiento ilimitado del disco.

El vacuum, por tanto, es el proceso en el cual se eliminan definitivamente tuplas marcadas para borrar y hay una reorganización de datos a nivel físico.

También existe la opción del autovacuum cuya funcionalidad es ir realizando de manera paulatina la mantención de nuestra base de datos.

Si eliminamos las tuplas, reducimos el tamaño de la tabla y por tanto, el coste de buscar en la tabla sería menor.

**AUTOVACUUM DAEMON**

PostgreSQL tiene una característica opcional pero altamente recomendada llamada autovacuum, cuyo propósito es automatizar la ejecución de los comandos VACUUM y ANALYZE. Cuando está habilitado, las comprobaciones de vacío automático para tablas que han tenido una gran cantidad de tuplas insertadas, actualizadas o eliminadas.

El Autovacuum Daemon en realidad consiste en múltiples procesos. Hay un proceso daemon persistente, llamado iniciador de autovacuum, que se encarga de iniciar los procesos de trabajo autovacuum para todas las bases de datos. El iniciador distribuirá el trabajo a lo largo del tiempo, intentando iniciar un worker dentro de cada base de datos cada *autovacuum\_naptime* segundos.

Por último, *log\_autovacuum\_min\_duration* se puede configurar para monitorear la actividad de los workers de autovacuum.

**REINDEX**

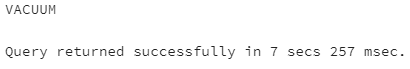
En algunas situaciones, vale la pena reconstruir índices periódicamente con el comando REINDEX o una serie de paso de reconstrucción individuales. Para los índices de árbol B, un índice recién construido es ligeramente más rápido de acceder que uno que ha sido actualizado muchas veces por lo que reindexar periódicamente mejorará la velocidad de acceso.

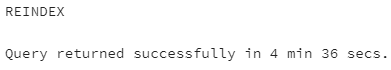
Si eliminamos las tuplas, entonces debemos reconstruir el índice ya que no estará actualizado.

**LOG FILE MAINTENANCE**

Postgres almacena la salida del log del servidor de la base de datos. Sin embargo, la salida del log tiende a ser voluminosa por lo que no se desea tenerla guardada indefinidamente. Para ello, necesitaremos rotar los archivos del log de forma que los nuevos se guarden y los viejos sean eliminados después de un periodo razonable de tiempo.

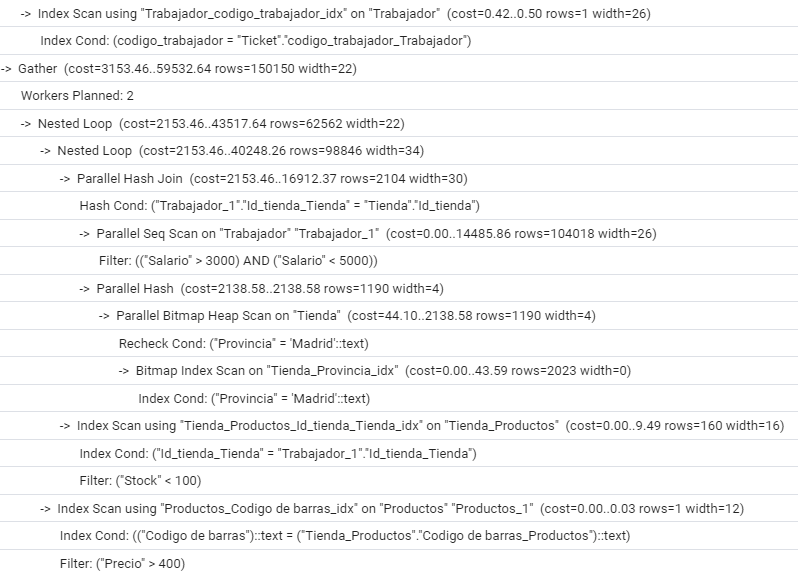
Cuestión 14: Usando PostgreSQL, lleve a cabo las operaciones propuestas en la cuestión anterior y ejecute el plan de ejecución de la misma consulta. Obtener el plan de ejecución con el resultado del comando EXPLAIN en forma de árbol de algebra relacional. Compare los resultados del plan de ejecución con los de los apartados anteriores. Coméntelos.





Tras realizar las operaciones mencionadas en el ejercicio 13 como se puede observar en las dos imágenes anteriores, este es el resultado del explain en la consulta:





Como se puede observar, el coste de la consulta tras realizar el borrado y tras realizar las operaciones de optimización, se reduce aún más.

Esto se debe a todos los cambios que producen en la base de datos las operaciones realizadas (los cuales se han explicado en la cuestión 13).

Cuestión 15: Usando PostgreSQL, analice el LOG de operaciones de la base de datos y muestre información de cuáles han sido las consultas más utilizadas en su práctica, el número de consultas, el tiempo medio de ejecución, y cualquier otro dato que considere importante.

La siguiente información la podemos encontrar en el directorio de archivos de log que está disponible en “C:\Program Files\PostgreSQL\12\data\log”

Si accedemos a dicha carpeta encontramos varios archivos en los que su nombre sigue el siguiente formato *postgresql-2020-01-28\_154644.*

Podemos ver todas las estadísticas que almacena el LOG de operaciones en *pg\_stat\_statements.*

Las consultas más utilizadas en la práctica son selecciones de elementos de tablas que cumplen una condición, consultas para ver estadísticas o consultas para contar el número de elementos que hay en una tabla que cumplan una cierta condición.

El número de consultas que se han realizado lo podemos ver con la siguiente consulta. 



Por lo que, desde que activamos el log de postgres, se han registrado 312 consultas.

Por otra parte, el tiempo medio de ejecución de estas 312 consultas es de 32130,597 milisegundos, es decir, 32,13 segundos. Esto es debido a que hay muchas consultas que han tardado mucho tiempo en ejecutarse, como por ejemplo, cuando cargamos los archivos de *trabaja\_proyectos*, *proyectos* y *empleados*. Sin embargo, la mayoría de consultas han tardado menos de 1 o 2 minutos.





Cuestión 16: 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.

**PROCESO DE PROCESAMIENTO**

Para obtener el resultado de una consulta, ésta deberá de pasar por cinco etapas:

1. Se debe establecer una conexión desde un programa de aplicación al servidor PostgresSQL. El programa de aplicación transmite una consulta al servidor y espera a que el servidor le devuelva los resultados.
2. El analizador sintáctico (parser) comprueba que la consulta introducida por el programa de aplicación es correcta sintácticamente y crea un árbol de consulta.
3. El sistema “rewriter” utiliza el árbol de consulta creado en la etapa anterior por el analizador sintáctico y busca cualquier regla (almacenada en los catálogos del sistema) para aplicarla al árbol de consulta.
4. El planificador/optimizador coge el árbol de consulta reescrito y crea un plan de consulta óptimo para la consulta. Esto lo hace creando todas las rutas posibles que conducen al mismo resultado y después estima el costo de la ejecución de cada ruta escogiendo finalmente la ruta más barata.
5. El ejecutor recorre recursivamente el árbol del plan y recupera las filas de la forma representada por el plan.

**PROCESO DE OPTIMIZACIÓN**

Como decíamos anteriormente, una de las etapas por las que pasa una consulta es por el optimizador.

La tarea del optimizador de Postgres es crear un plan de ejecución óptimo. Una consulta SQL dada y por tanto, un árbol de consulta puede ejecutarse en una gran variedad de formas diferentes, las cuales darán todas el mismo resultado. Por tanto, el optimizador examinará cada uno de estos posibles planes de ejecución, seleccionando de todos ellos aquel que se espera ejecutar el más rápido.

El procedimiento de búsqueda del optimizador funciona con estructuras de datos llamadas rutas o planes que son representaciones reducidas de planes que solo contienen la cantidad de información que el optimizador necesita para determinar decisiones.

Una vez determinado el plan más barato, se construye el árbol de plan completo para pasárselo al ejecutor. Esto representará el plan de ejecución deseado con suficiente detalle para que el ejecutor lo pueda ejecutar.

**Bibliografía**

PostgreSQL (12.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.