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 Martín Gómez_____

DNI: 47231977M_____

ALUMNO 2:

Nombre y Apellidos: Alberto González Martínez_____

DNI: 09072311F_____

Fecha: 15-04-2020____

Profesor Responsable: Oscar Gutiérrez_____

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 14 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?

Sí, guarda estadísticas tanto de los datos como de la actividad.

La herramienta pg_statistic almacena estadísticas sobre información y contenidos sobre las tablas y los índices de la base de datos. También guarda información sobre la herencia de las tablas. La diferencia entre pg_statistic y pg_stats es que la primera está restringida a los superusuarios, mientras que el resto de los usuarios pueden acceder a la segunda.

Además, en postgres existe un subsistema llamado colector de estadísticas que recoge información sobre la actividad del servidor (tablas, índices...). El sistema, por defecto, recolecta comandos y accesos a tablas e índices.

Postgres diferencia las estadísticas de datos y las de actividad. Las primeras se guardan en una tabla asociada a cada base de datos y las segundas se guardan en pg_stat y temporalmente en pg_stat_temp.

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.

Hay que modificar el archivo postgresql.conf y cambiar lo siguiente:

```
#log_statement = 'all'
```

```
#log_min_duration_statement = 0 # -
#log_checkpoints = on
#log_connections = on
#log_disconnections = on
#log_duration = on
#log_error_verbosity = default
#log_hostname = on
#log_line_prefix = '%m [%p] '
```

De esta manera en el log guardará todas las operaciones que realizaremos en la base de datos.

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.
-

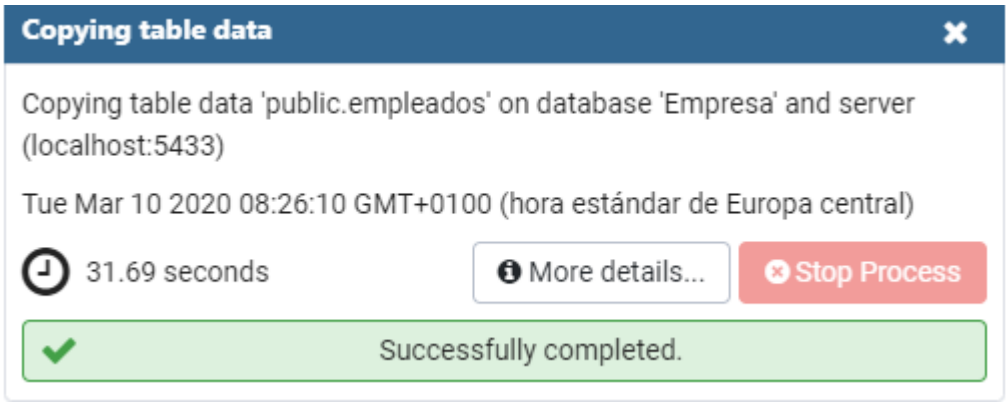
Se pide:

- 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.

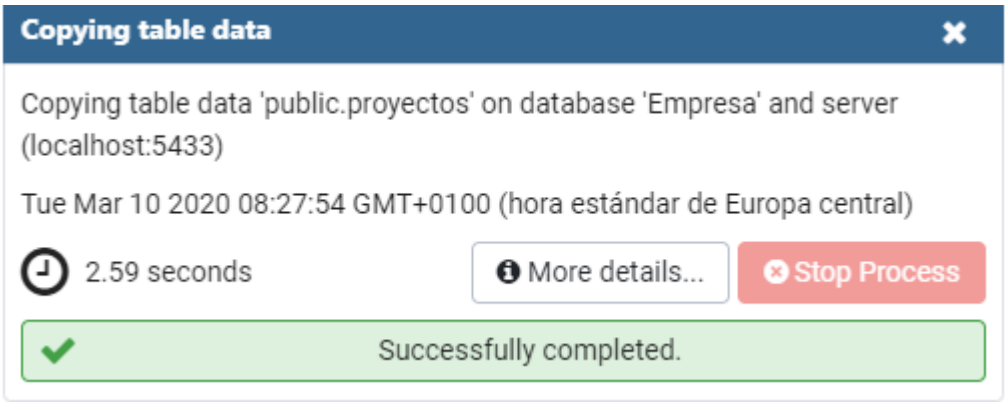
Primero creamos las tablas empleados y proyectos con sus campos correspondientes. Posteriormente, para relacionar las dos tablas mediante la tabla “trabaja_proyectos” lo realizamos de la siguiente manera:

```
CREATE TABLE public.trabaja_proyectos
(    numero_empleado_fk numeric NOT NULL,
    numero_proyecto_fk numeric NOT NULL,
    horas numeric,
    CONSTRAINT "trabaja_proyectos_pk" PRIMARY KEY (numero_empleado_fk,numero_proyecto_fk)
);

ALTER TABLE public.trabaja_proyectos ADD CONSTRAINT "empleados_fk" FOREIGN KEY (numero_empleado_fk)
REFERENCES public.empleados (numero_empleado) MATCH FULL
ON DELETE RESTRICT ON UPDATE CASCADE;

ALTER TABLE public.trabaja_proyectos ADD CONSTRAINT "proyectos_fk" FOREIGN KEY (numero_proyecto_fk)
REFERENCES public.proyectos (numero_proyecto) MATCH FULL
ON DELETE RESTRICT ON UPDATE CASCADE;
```

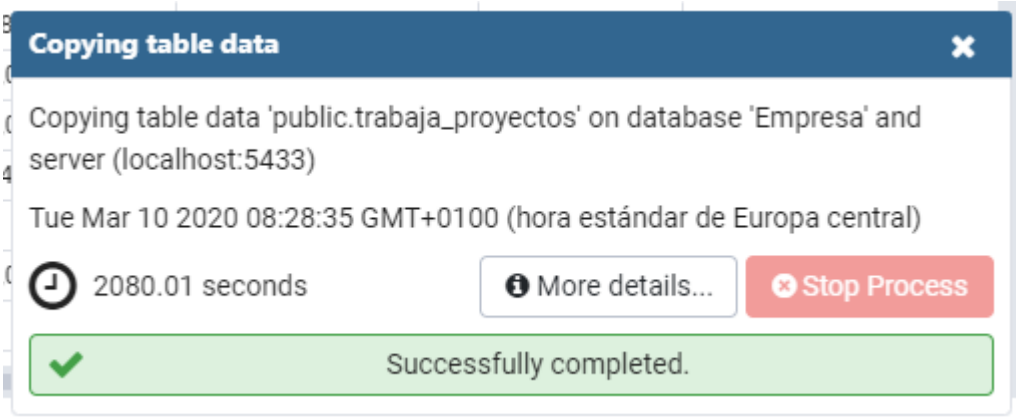
Primero se crea la tabla con sus campos correspondientes y con los campos que componen la PK. Después, a través de Alter table, relacionamos los campos numero_empleado_fk y numero_proyecto_fk como Foreign Key de las tablas proyectos y empleados.



Tiempo necesitado para los datos de la tabla empleados



Tiempo necesitado para los datos de la tabla proyectos



Tiempo necesitado para los datos de la 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?

Se almacenan los valores más comunes de los campos, la correlación, la frecuencia, el número de nombres distintos y los límites del histograma.

```
30  select * from pg_stats where tablename='proyectos' or tablename='empleados' or tablename='trabaja_proyectos';
31
32
```

Data Output | Explain | Messages | Notifications

	schemaname name	tablename name	attname name	inherited boolean	null_frac real	avg_width integer	n_distinct real	most_common_vals anyarray	most_common_freqs real[]	histogram_bounds anyarray	correlation real
1	public	trabaja_proyect...	horas	false	0	4	24	{6,11,19,18,8,4,0,1,9,13,16,...	0333334,0.039633334,0.0393}	[null]	0.034875594
2	public	empleados	numero_em...	false	0	6	-1	[null]	[null]	{12,18280,38420,55121,...	1
3	public	empleados	nombre	false	0	13	-1	[null]	[null]	{nombre1000005,nombr...	-0.3961672
4	public	empleados	apellidos	false	0	16	-1	[null]	[null]	{apellidos1000005,apelli...	-0.3961672
5	public	empleados	salario	false	0	6	97623	[null]	[null]	{1002.000,2034.000,300...	0.0123782065
6	public	proyectos	numero_pro...	false	0	6	-1	[null]	[null]	{3,1004,1953,2978,3949,...	1
7	public	proyectos	nombre	false	0	11	-1	[null]	[null]	{nombre10002,nombre1...	0.8265681
8	public	proyectos	localizacion	false	0	17	-1	[null]	[null]	{localizacion10002,locali...	0.8265681
9	public	proyectos	coste	false	0	6	9820	{12430.00,10185.00,1165...	,0.0003,0.0003,0.0003,0.0003}	{10000.00,10104.00,102...	0.0031728598
10	public	trabaja_proyect...	numero_em...	false	0	6	-0.17225634	[null]	[null]	{160,20235,40067,58805...	0.0061144633
11	public	trabaja_proyect...	numero_pro...	false	0	6	98041	[null]	[null]	{1,1040,2023,3034,3976,...	0.003655018

Para comprobar si son correctos, tenemos que ejecutar diferentes consultas. Por ejemplo, para saber los valores diferentes del campo horas de la tabla trabaja_proyectos, ejecutamos la consulta “select count(distinct horas) from "trabaja_proyectos";” y nos aparece el mismo valor que en la columna n_distinct (24).

Para comprobar los distintos de salario de la tabla empleados hacemos la consulta “select count(distinct salario) from "empleados";” y nos da distinto (100000) a lo de la tabla (97624), como se puede ver en la captura**********

Para comprobar los distintos costes de la tabla proyectos ejecutamos la consulta select count(distinct coste) from "proyectos"; y nos da distinto (10000) a lo de la tabla (9820) como se puede ver en la captura.

Podemos ejecutar estas consultas anteriores para ver si son correctos el campo n_distinct para cada tupla. Para ver los límites de los diferentes campos de cada tabla usaremos los comandos min y max y lo comprobaremos.

Para actualizarlos, la instrucción UPDATE STATISTICS actualiza todas las estadísticas en la tabla y las vistas indexadas.

**falta ver si son correctos

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.

En el archivo postgresql.conf modificamos las constantes de coste poniendo a 1.0 las de bloque (sq_page_cost y random_page_cost) y el resto las ponemos a 0.0

Se configura poniendo los costes de acceso a bloque a 1 y el resto a 0.

```
# - Planner Cost Constants -

#seq_page_cost = 1.0                    # measured on an arbitrary scale
#random_page_cost = 1.0         # same scale as above
#cpu_tuple_cost = 0.0                   # same scale as above
#cpu_index_tuple_cost = 0.0             # same scale as above
#cpu_operator_cost = 0.0                # same scale as above
#parallel_tuple_cost = 0.0              # same scale as above
#parallel_setup_cost = 0.0      # same scale as above

#jit_above_cost = 0             # perform JIT compilation if available
                                        # and query more expensive than this;
                                        # -1 disables
#jit_inline_above_cost = 0      # inline small functions if query is
                                        # more expensive than this; -1 disables
#jit_optimize_above_cost = 0    # use expensive JIT optimizations if
                                        # query is more expensive than this;
                                        # -1 disables
```

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.

Si hacemos la consulta:

```
14
15  select count(*) from "empleados" where salario>96000
16
17
```

Data Output | Explain | Messages | Notifications

	count bigint
1	99523

El coste nos sale 99523 registros estimados. Hacemos la siguiente consulta:

```
15  explain (format json) select * from "empleados" where salario>96000
16
17
```

Data Output | Explain | Messages | Notifications

Graphical | Analysis | Statistics

#	Node	Rows Plan
1.	→ Gather (cost=1000..40067.97 rows=99783 width=41)	99783
2.	→ Seq Scan on empleados as empleados (cost=0..29089.67 rows=41576 width=41) Filter: (salario > '96000'::numeric)	41576

Vemos que ahora lee 99783 registros (reales), vemos que no coinciden, pero se acercan bastante por lo que se puede considerar una buena estimación. Los resultados no son 100% correctos ya que la estimación no es perfecta.

Procedimiento teoría:

Nr=2000000

Nv=100999-96000=4999 al estar salario comprendido entre 1000 y 100999

V(A,R)=100999-1000=99999 se le suma 1 por lo tanto 100000

T(σsalario>9600(empleados))=4999*2000000/100000=**99980 registros**

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.

```
17  explain (format json)select "proyectos".* from "proyectos" inner join "trabaja_proyectos" on
18  "trabaja_proyectos".numero_proyecto_fk="proyectos".numero_proyecto where horas=8;
19
```

Data Output | Explain | Messages | Notifications

Graphical | Analysis | Statistics

#	Node	Rows Plan
1.	→ Gather (cost=4957..166075.55 rows=427005 width=40)	427005
2.	→ Hash Inner Join (cost=3957..122375.05 rows=177919 width=40) Hash Cond: (trabaja_proyectos.numero_proyecto_fk = proyectos.numero_proyecto)	177919
3.	→ Seq Scan on trabaja_proyectos as trabaja_proyectos (cost=0..115778.99 rows=177919 width=6) Filter: (horas = '8'::numeric)	177919
4.	→ Hash (cost=1925..1925 rows=100000 width=40)	100000
5.	→ Seq Scan on proyectos as proyectos (cost=0..1925 rows=100000 width=40)	100000

El coste nos sale 427005 registros estimados. Hacemos la siguiente consulta:

```
17  select count("proyectos".*) from "proyectos" inner join "trabaja_proyectos" on
18   "trabaja_proyectos".numero_proyecto_fk="proyectos".numero_proyecto where horas=8;
19
```

Data Output | Explain | Messages | Notifications

	count bigint
1	416698

Ahora lee 416698 registros (reales) y vemos que no son los mismos que los estimados, pero se acercan bastante, por lo que la estimación es buena. Los resultados no son 100% correctos ya que la estimación no es perfecta.

Procedimiento teoría:

$\sigma_{horas=8}(proyectos \bowtie_{numero\_proyecto=numero\_proyectos\_fk} trabaja\_proyectos)$

nr(trabaja_proyectos)=10000000

V(A,R)=24

T(σhoras=8)=10000000/24=416667 registros

T(proyectos)=100000

Min{41667,100000}=41667

T(⋈)=416667*100000/max{416667,100000}=**416667registros**

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.

```
19
20 explain (format json) select "proyectos".* from "proyectos" inner join "trabaja_proyectos" on
21  "trabaja_proyectos".numero_proyecto_fk="proyectos".numero_proyecto inner join "empleados" on
22 "trabaja_proyectos".numero_empleado_fk="empleados".numero_empleado
23 where salario=24000 and horas<2 and coste>15000;
24
25
26
```

Data Output | Explain | Messages | Notifications

Graphical | Analysis | Statistics

#	Node	Rows Plan
1.	→ Gather (cost=1000.73..30319.54 rows=4 width=40)	4
2.	→ Nested Loop Inner Join (cost=0.73..29319.14 rows=2 width=40)	2
3.	→ Nested Loop Inner Join (cost=0.43..29318.18 rows=3 width=6)	3
4.	→ Seq Scan on empleados as empleados (cost=0..29089.67 rows=8 width=6) Filter: (salario = '24000'::numeric)	8
5.	→ Index Scan using trabaja_proyectos_pk on trabaja_proyectos as trabaja_proyectos (cost=0.43..28.55 rows=1 width=12) Filter: (horas < '2'::numeric) Index Cond: (numero_empleado_fk = empleados.numero_empleado)	1
6.	→ Index Scan using proyectos_pkey on proyectos as proyectos (cost=0.29..0.32 rows=1 width=40)	1

El coste estimado son 4 registros. Se realiza la siguiente consulta:

```
20 select count("proyectos".*) from "proyectos" inner join "trabaja_proyectos" on
21  "trabaja_proyectos".numero_proyecto_fk="proyectos".numero_proyecto inner join "empleados" on
22 "trabaja_proyectos".numero_empleado_fk="empleados".numero_empleado
23 where salario=24000 and horas<2 and coste>15000;
24
25
26
```

Data Output | Explain | Messages | Notifications

	count bigint
1	2

Vemos que el valor real del coste, son 2 registros. Sigue sin ser el correcto, pero se sigue aproximando bastante al valor real. Los resultados no son 100% correctos ya que la estimación no es perfecta.

Procedimiento teoría:

$\sigma_{horas<2 \cap salario=24000 \cap coste>15000}(Empleados \underset{numero\_proyecto\_fk=numero\_proyecto}{\bowtie} trabaja\_proyectos \underset{numer\_empleado=numero\_empleado\_fk}{\bowtie} proyectos)$

$T(\sigma_{horas<2})=2*1000000/24=833334$

$T(\sigma_{salario=24000})=200000/100000=200$

$Min\{833334,1986530\}=833334$

$Min\{200,2000000\}=200$

$T(\bowtie 1)=200*833334/max\{833334,200\}=200$ registros

$T(\sigma_{coste>15000})=100000*5000/20000=25000$

$Min\{200,100000\}=200$

$Min\{25000,100000\}=25000$

$T(\bowtie 2)=25000*200/\max\{25000,200\}=$**200 registros**

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?

Para empezar, hemos realizado un programa en Python para que se generen los datos aleatorios pedidos con las restricciones dadas en la base de datos tienda.

Después, pasamos a la inserción de estos datos generados. En la primera carga, deshabilitamos todos los triggers ejecutando la línea "alter table "tabla" disable trigger all" para todas las tablas, y así, deshabilitamos la integridad referencial.

Después, realizamos la carga de datos con la integridad referencial activada, por lo que ejecutamos la línea "alter table "tabla" enable trigger all" para todas las tablas.

Antes de hacer la primera carga de datos ejecutamos lo siguiente:

```
1  ALTER TABLE "Productos" DISABLE TRIGGER ALL;
2  ALTER TABLE "Ticket" DISABLE TRIGGER ALL;
3  ALTER TABLE "Ticket_Productos" DISABLE TRIGGER ALL;
4  ALTER TABLE "Tienda" DISABLE TRIGGER ALL;
5  ALTER TABLE "Tienda_Productos" DISABLE TRIGGER ALL;
6  ALTER TABLE "Trabajador" DISABLE TRIGGER ALL;
```

También, eliminamos todas las FKs.

Antes de hacer la segunda carga (la que vamos a utilizar en el resto de la práctica) ejecutamos lo siguiente:

```
8  ALTER TABLE "Productos" ENABLE TRIGGER ALL;
9  ALTER TABLE "Ticket" ENABLE TRIGGER ALL;
10 ALTER TABLE "Ticket_Productos" ENABLE TRIGGER ALL;
11 ALTER TABLE "Tienda" ENABLE TRIGGER ALL;
12 ALTER TABLE "Tienda_Productos" ENABLE TRIGGER ALL;
13 ALTER TABLE "Trabajador" ENABLE TRIGGER ALL;
```

Tabla	Tiempo sin integridad	Tiempo con integridad
Productos	6.06	22.43s
Ticket	34.123s	484s
Ticket_productos	89s	1598s

Tienda	2.06s	2.48s
Tienda_productos	112.67s	1620s
Trabajador	17s	28.86s

El orden ha sido el siguiente: Productos-> Tienda-> Tienda_productos->Trabajador->Ticket->Ticket_productos. Debe ser este orden o muy parecido debido a que muchas tablas dependen de otras, ya que tiene la FK, y muchas columnas de estas tablas aparecen en otras, por lo que primero hay que insertar los datos en las tablas donde aparecen estas columnas por primera vez como PK.

Los datos insertados sin integridad referencial tardan menos que los insertados con integridad referencial. Esto es debido a que, cuando se insertan los datos cuando tiene integridad, comprueba si la FK (si tiene) está en la otra tabla, por lo que ralentiza la inserción.

El tiempo en el log es muy parecido al que muestra pgAdmin, pero algo menor. Esto puede ser posible a que primero graba en el log cuando acaba la operación y después, manda una señal de finalizado al cliente, y en ese momento es cuando se marca el final de la consulta.

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.

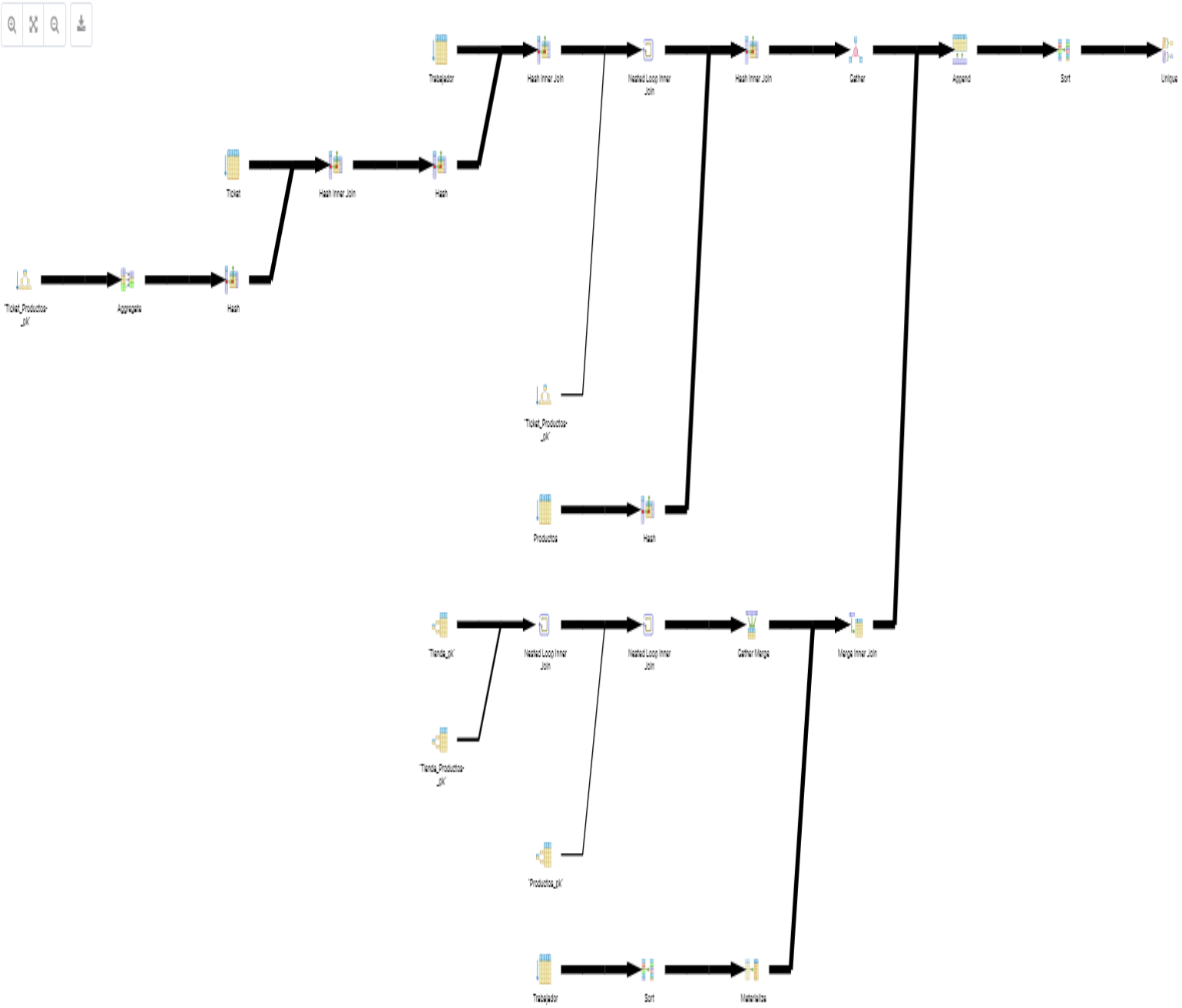
Cuestión 10: Realizar una consulta SQL que muestre "el nombre y DNI de los trabajadores que hayan vendido algún ticket en los cuatro últimos meses del año con más de cuatro productos en los que al menos alguno de ellos tenga un precio de más de 500 euros, junto con los trabajadores que ganan entre 3000 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."

La consulta es la siguiente:

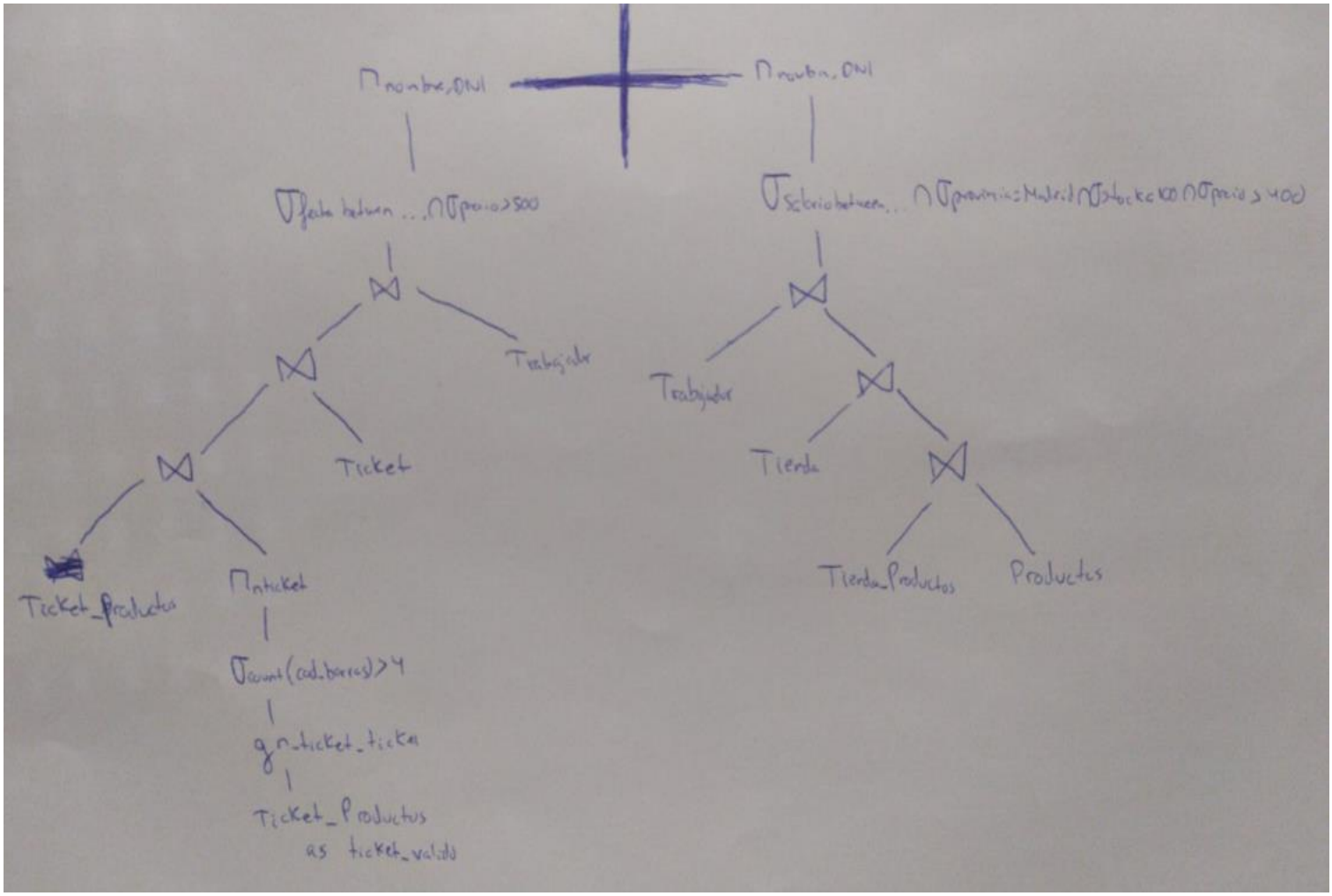
```
--Consulta
select "Trabajador".nombre,dni from "Trabajador"
inner join "Ticket" on "Ticket"."codigo_trabajador_trabajador"="Trabajador".codigo_trabajador
inner join (select nticket_ticket  from "Ticket_Productos" group by nticket_ticket
having count("Ticket_Productos".codigo_de_barras_productos)>4) as "ticket_valido"
on "ticket_valido".nticket_ticket="Ticket".nticket inner join "Ticket_Productos" on
"ticket_valido".nticket_ticket="Ticket_Productos".nticket_ticket
inner join "Productos"  on "Productos".codigo_de_barras="Ticket_Productos".codigo_de_barras_productos
where (fecha between '2019-09-01' and '2019-12-31') and precio>500
union
select "Trabajador".nombre,dni from "Trabajador" inner join
"Tienda" on "Tienda".id_tienda="Trabajador".id_tienda_tienda
inner join "Tienda_Productos" on "Tienda_Productos".id_tienda_tienda="Tienda".id_tienda
inner join "Productos" on "Productos".codigo_de_barras="Tienda_Productos".codigo_de_barras_productos
where (salario between 3000 and 5000) and
provincia='Madrid' and stock<100 and precio>400
```

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.

El árbol en postgres es el siguiente:



Árbol en álgebra relacional:



Podemos observar que postgres lo hace todo en una misma consulta, mientras que nosotros lo hacemos en dos separadas. Además, postgres utiliza menos joins que los que usamos nosotros.

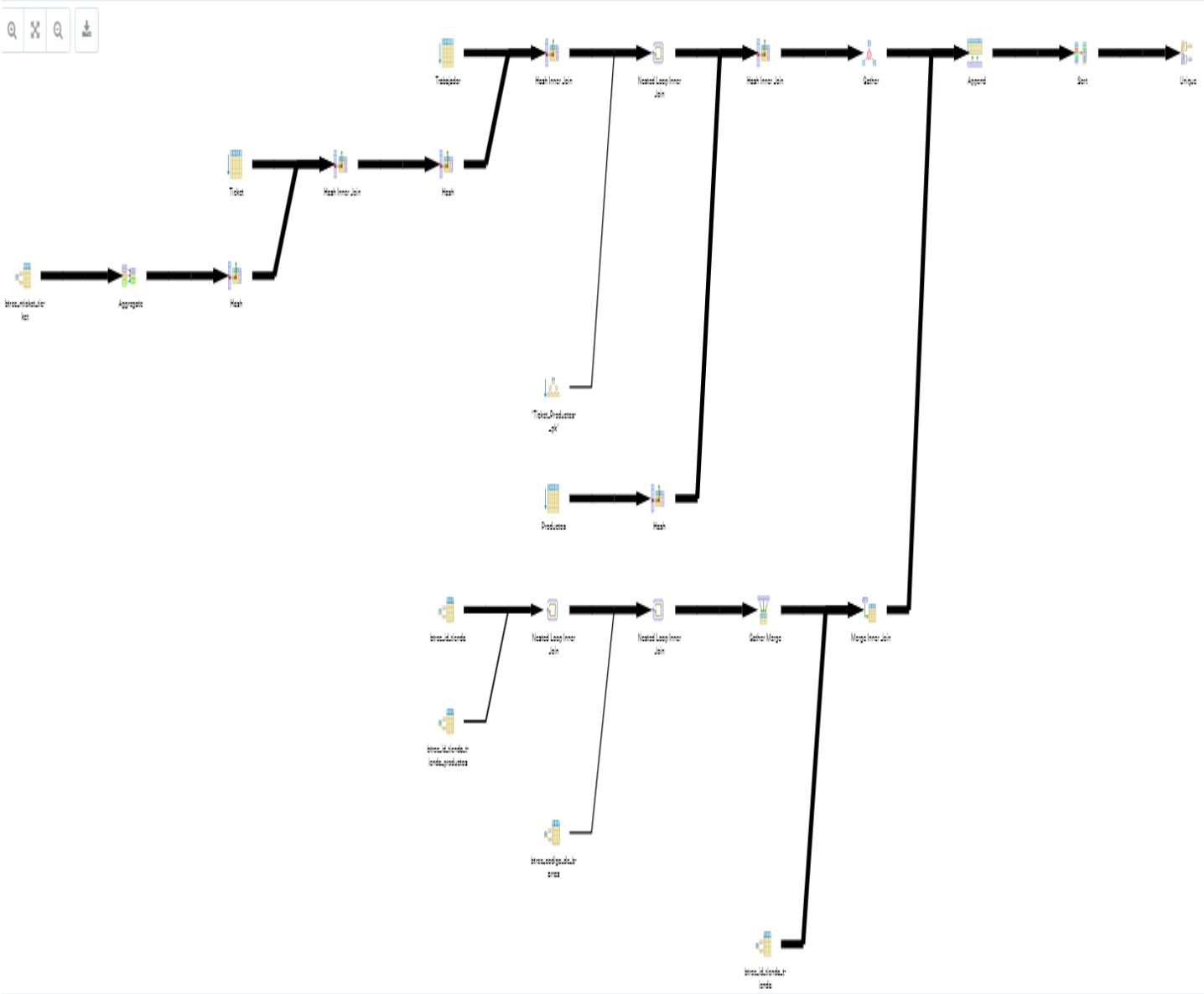
Cuestión 11: Usando PostgreSQL, y a raíz de los resultados de la cuestión anterior, ¿qué modificaciones realizaría para mejorar el rendimiento de la misma y por qué? Obtener la información pedida de la cuestión 10 y explicar los resultados. Obtener el plan de ejecución con el resultado del comando EXPLAIN en forma de árbol de algebra relacional. Comentar los resultados obtenidos y comparar con la cuestión anterior.

Podríamos crear índices sobre los campos hemos usado en la consulta, para que evitar múltiples búsquedas secuenciales y así ahorrar registros que leer.

Creamos los índices:

```
Create index btree_dni on "Trabajador"(dni);
Create index btree_nombre on "Trabajador"(nombre);
Create index btree_codigo_trabajador_trabajador on "Ticket"(codigo_trabajador_trabajador);
Create index btree_codigo_trabajador on "Trabajador"(codigo_trabajador);
Create index btree_nticket_ticket on "Ticket_Productos"(nticket_ticket);
Create index btree_nticket on "Ticket"(nticket);
Create index btree_codigo_de_barras_productos on "Ticket_Productos"(codigo_de_barras_productos);
Create index btree_codigo_de_barras on "Productos"(codigo_de_barras);
Create index btree_id_tienda on "Tienda"(id_tienda);
Create index btree_id_tienda_tienda on "Trabajador"(id_tienda_tienda);
Create index btree_id_tienda_tienda_productos on "Tienda_Productos"(id_tienda_tienda);
Create index btree_codigo_de_barras_productos_tienda on "Tienda_Productos"(codigo_de_barras_productos);
```

Árbol:



Como se puede comprobar, se han usado algunos índices, pero no todos, ya que es posible que la lectura secuencial de alguna tabla sea más eficiente que usar el índice.

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.

```
---Pregunta 12
delete top 100000 from "Tienda";
alter table "Tienda_Productos" add foreign key(id_tienda_tienda) references "Tienda"(id_tienda) on delete cascade
alter table "Trabajador" add foreign key(id_tienda_tienda) references "Tienda"(id_tienda) on delete cascade
```

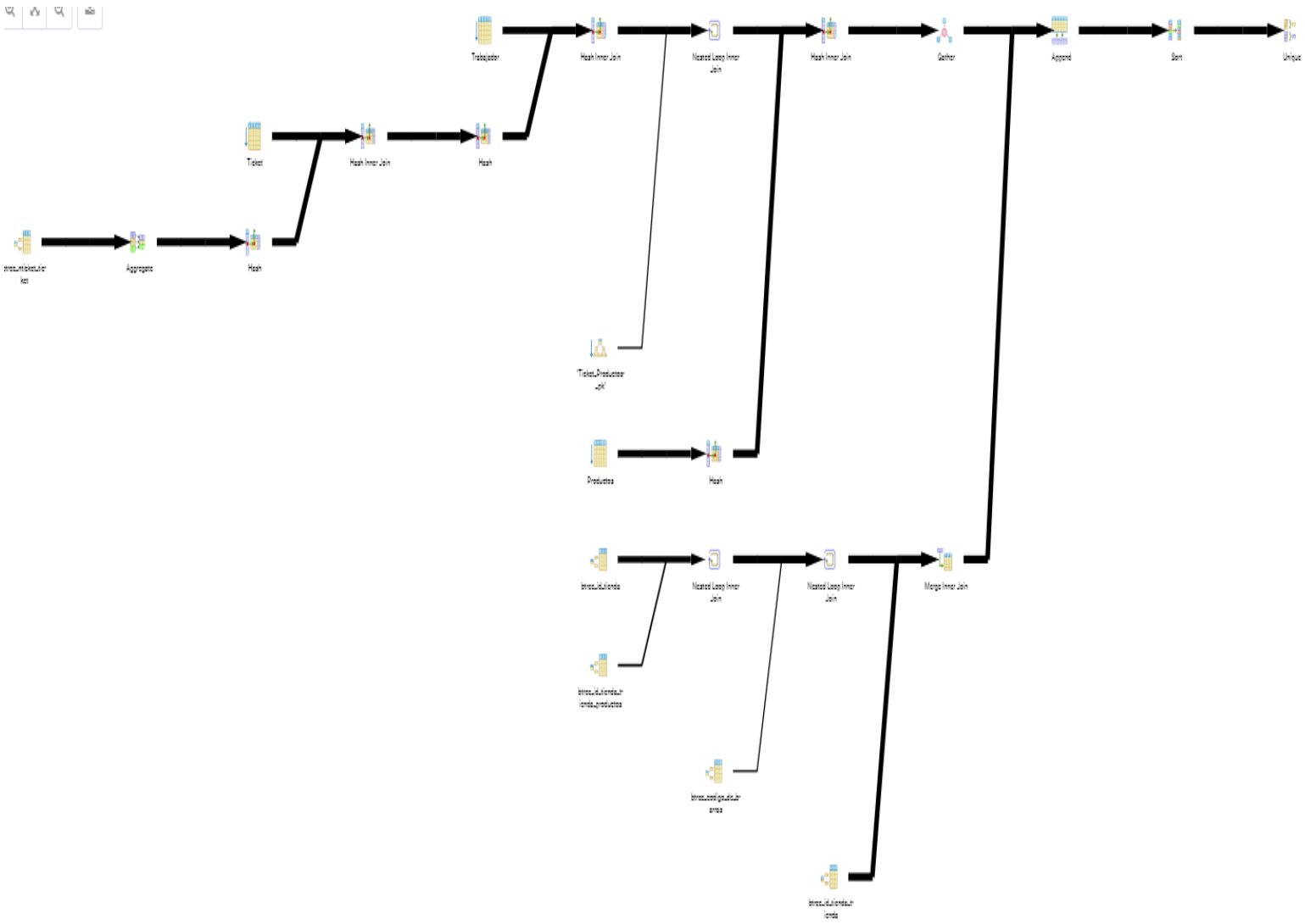
Primero borramos 100000 (el 50%, ya que hay 200000) aleatoriamente con el comando top. Después, al estar relacionada con otras tablas, hay que eliminar de esas tablas las tuplas que se han borrado de la tabla tiendas y eso se hace introduciendo

"alter table "Tienda_Productos" add foreign key(id_tienda_tienda) references "Tienda"(id_tienda) on delete cascade

alter table "Trabajador" add foreign key(id_tienda_tienda) references "Tienda"(id_tienda) on delete cascade"

al ser Trabajdor y Tienda_Productos las tablas relacionadas con tiendas

El tiempo ha sido de hora y media aproximadamente.



Es prácticamente igual que el anterior. Esto es debido a que al borrar estos datos, se han podido quedar tuplas muertas, los índices no han sido actualizados y las estadísticas tampoco.

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 propondría diferentes herramientas.

Vacuum sería una de ellas, ya que permite eliminar las tuplas muertas que se queden al borrar estos datos. Esto hace que se eliminen los registros de forma definitiva y así, el tiempo de acceso a los registros de la consulta

Reindex sería otra, ya que, al eliminar tuplas, los índices quedan desactualizados por lo que esta herramienta les actualiza.

Analyze, también la usaría para actualizar las estadísticas de las tablas. Al actualizarlas, la planificación de las consultas se harán sobre las estadísticas actualizadas y así, las consultas se optimizarán.

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.

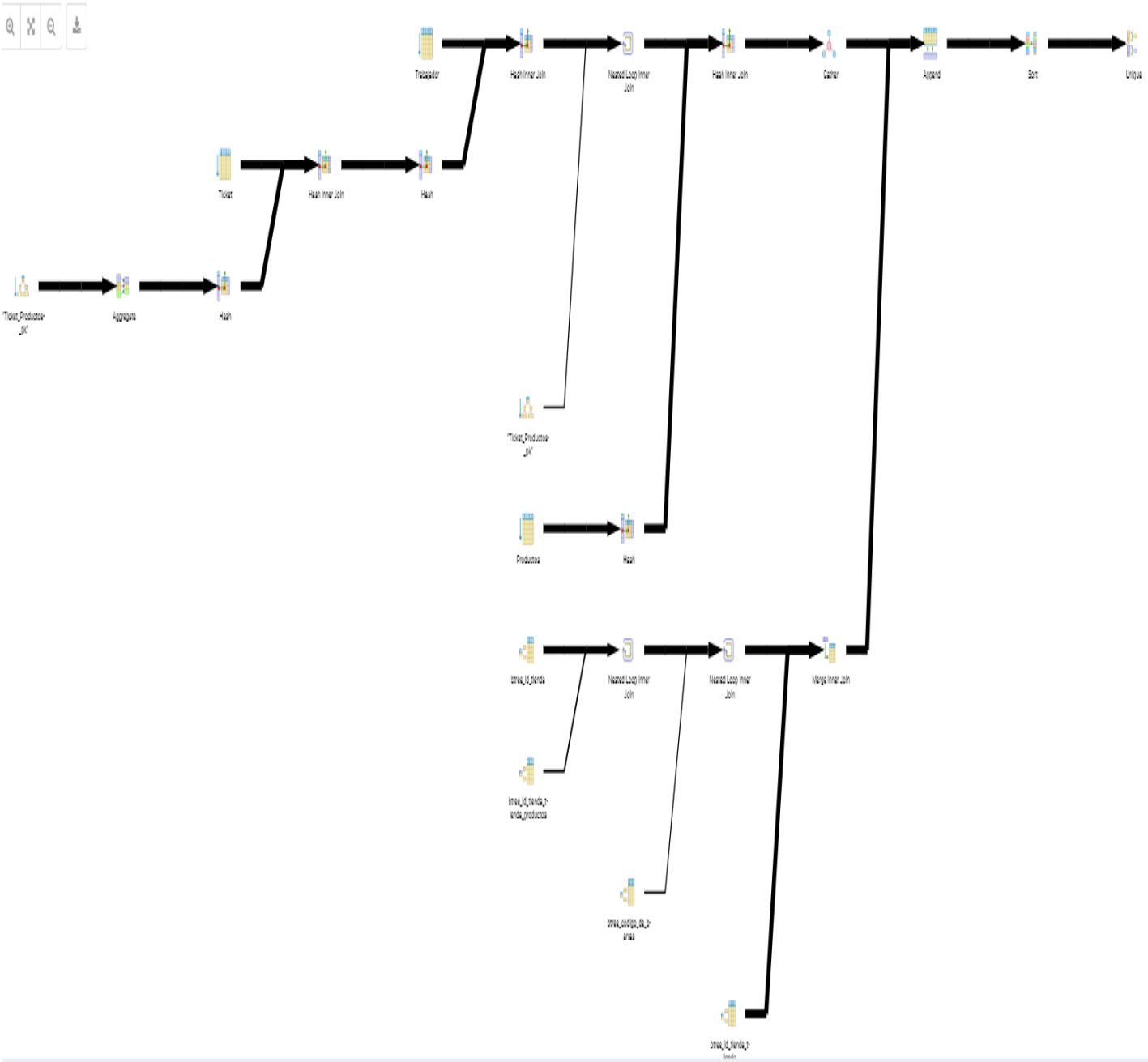
Ejecutamos vacuum y analyze para cada tabla:

```
vacuum analyze "Productos";
vacuum analyze "Ticket";
vacuum analyze "Ticket_Productos";
vacuum analyze "Tienda";
vacuum analyze "Tienda_Productos";
vacuum analyze "Trabajador";
```

Después utilizamos la herramienta reindex para actualizar los índices

```
Reindex index btree_dni;
Reindex index btree_codigo_trabajador_trabajador;
Reindex index btree_nombre;
Reindex index btree_codigo_trabajador;
Reindex index btree_nticket_ticket;
Reindex index btree_nticket;
Reindex index btree_codigo_de_barras_productos;
Reindex index btree_codigo_de_barras;
Reindex index btree_id_tienda;
Reindex index btree_id_tienda_tienda;
Reindex index btree_id_tienda_tienda_productos;
Reindex index btree_codigo_de_barras_productos_tienda;
```

Árbol:



Es exactamente igual que el anterior.

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.

El archivo del log lo encontramos en la ruta: "C:\Program Files\PostgreSQL\12\data\log"

A continuación mostramos el archivo del log, donde se muestran los errores que han podido surgir, las consultas ejecutadas, la fecha de cada una, las horas, el tiempo de ejecución… :

```
2020-04-03 14:11:22.044 CEST [6568] LOG:  el sistema de bases de datos fue apagado en 2020-04-03 14:08:29 CEST
2020-04-03 14:11:23.967 CEST [5784] LOG:  el sistema de bases de datos está listo para aceptar conexiones
2020-04-03 14:28:48.168 CEST [7160] LOG:  usando estadísticas añejas en vez de actualizadas porque el recolector de estadísticas no está respondiendo
2020-04-03 14:29:00.663 CEST [7160] LOG:  usando estadísticas añejas en vez de actualizadas porque el recolector de estadísticas no está respondiendo
2020-04-03 14:29:03.669 CEST [10184] LOG:  usando estadÃ-sticas aÃ±ejas en vez de actualizadas porque el recolector de estadÃ-sticas no estÃ¡ respondiendo
2020-04-03 14:29:10.991 CEST [7160] LOG:  usando estadísticas añejas en vez de actualizadas porque el recolector de estadísticas no está respondiendo
2020-04-03 14:29:33.451 CEST [7160] LOG:  usando estadísticas añejas en vez de actualizadas porque el recolector de estadísticas no está respondiendo
2020-04-03 14:51:46.223 CEST [9520] ERROR:  no existe la relaciÃ³n Â«Tickets_ProductosÂ» en carÃ¡cter 35
2020-04-03 14:51:46.223 CEST [9520] SENTENCIA:  select count(nticket_ticket) from "Tickets_Productos"
2020-04-03 14:54:17.081 CEST [9520] ERROR:  error de sintaxis en o cerca de Â«(Â» en carÃ¡cter 28
2020-04-03 14:54:17.081 CEST [9520] SENTENCIA:  select nticket_ticket count(codio_de_barras_productos) from "Ticket_Productos" group by codio_de_barras_productos
        having count(codio_de_barras_productos)>1
2020-04-03 14:54:30.727 CEST [9520] ERROR:  error de sintaxis en o cerca de Â«(Â» en carÃ¡cter 29
2020-04-03 14:54:30.727 CEST [9520] SENTENCIA:
        select nticket_ticket count(codigo_de_barras_productos) from "Ticket_Productos" group by codigo_de_barras_productos
        having count(codigo_de_barras_productos)>1
2020-04-03 14:54:40.752 CEST [9520] ERROR:  la columna Â«Ticket_Productos.nticket_ticketÂ» debe aparecer en la clÃ¡usula GROUP BY o ser usada en una funciÃ³n de agregaciÃ³n en carÃ¡cter 8
2020-04-03 14:54:40.752 CEST [9520] SENTENCIA:  select nticket_ticket, count(codigo_de_barras_productos) from "Ticket_Productos" group by codigo_de_barras_productos
        having count(codigo_de_barras_productos)>1
2020-04-03 15:53:19.788 CEST [9520] ERROR:  error de sintaxis en o cerca de Â«whereÂ» en carÃ¡cter 568
2020-04-03 15:53:19.788 CEST [9520] SENTENCIA:  select nombre,dni from "Trabajador" inner join "Ticket" on "Ticket"."codigo_trabajador_trabajador"="Trabajador".codigo_trabajdor
        inner join "Ticket_productos" on "Ticket_productos".nticket_ticket="Ticket".nticket inner join "Productos"  on
        "Productos".codigo_de_barras="Ticket_Productos".codigo_de_barras_productos inner join "Tienda_Productos" on
        "Productos".codigo_de_barras="Tienda_Productos".codigo_de_barras_productos inner join "Tienda" on
        "Tienda_Productos".id_tienda_tienda="Tienda".id_tienda
        group by nticket_ticket having count(codigo_de_barras_productos)
        where fecha>=2019-09-01 and "Productos".precio>500 and ("Trabajador".salario>=3000 or "Trabajador".salario<=5000) and
        provincia='Madrid' and stock>100 and precio>400
2020-04-03 15:53:52.820 CEST [9520] ERROR:  no existe la columna Ticket.codigo_trabajador_trabajador en carÃ¡cter 60
2020-04-03 15:53:52.820 CEST [9520] HINT:  Probablemente quiera hacer referencia a la columna Â«Ticket.codigo_trabajador_TrabajadorÂ».
2020-04-03 15:53:52.820 CEST [9520] SENTENCIA:  select nombre,dni from "Trabajador" inner join "Ticket" on "Ticket"."codigo_trabajador_trabajador"="Trabajador".codigo_trabajdor
        inner join "Ticket_productos" on "Ticket_productos".nticket_ticket="Ticket".nticket inner join "Productos"  on
        "Productos".codigo_de_barras="Ticket_Productos".codigo_de_barras_productos inner join "Tienda_Productos" on
        "Productos".codigo_de_barras="Tienda_Productos".codigo_de_barras_productos inner join "Tienda" on
        "Tienda_Productos".id_tienda_tienda="Tienda".id_tienda
        group by nticket_ticket having count(codigo_de_barras_productos)
2020-04-03 15:54:17.215 CEST [9520] ERROR:  no existe la columna Trabajador.codigo_trabajdor en carÃ¡cter 100
2020-04-03 15:54:17.215 CEST [9520] HINT:  Probablemente quiera hacer referencia a la columna Â«Trabajador.codigo_trabajadorÂ».
2020-04-03 15:54:17.215 CEST [9520] SENTENCIA:  select nombre,dni from "Trabajador" inner join "Ticket" on "Ticket"."codigo_trabajador_trabajador"="Trabajador".codigo_trabajdor
        inner join "Ticket_productos" on "Ticket_productos".nticket_ticket="Ticket".nticket inner join "Productos"  on
        "Productos".codigo_de_barras="Ticket_Productos".codigo_de_barras_productos inner join "Tienda_Productos" on
        "Productos".codigo_de_barras="Tienda_Productos".codigo_de_barras_productos inner join "Tienda" on
        "Tienda_Productos".id_tienda_tienda="Tienda".id_tienda
        group by nticket_ticket having count(codigo_de_barras_productos)
2020-04-03 15:54:43.330 CEST [9520] ERROR:  no existe la relaciÃ³n Â«Ticket_productosÂ» en carÃ¡cter 142
2020-04-03 15:54:43.330 CEST [9520] SENTENCIA:  select nombre,dni from "Trabajador" inner join "Ticket" on "Ticket"."codigo_trabajador_trabajador"="Trabajador".codigo_trabajador
        inner join "Ticket_productos" on "Ticket_productos".nticket_ticket="Ticket".nticket inner join "Productos"  on
        "Productos".codigo_de_barras="Ticket_Productos".codigo_de_barras_productos inner join "Tienda_Productos" on
        "Productos".codigo_de_barras="Tienda_Productos".codigo_de_barras_productos inner join "Tienda" on
        "Tienda_Productos".id_tienda_tienda="Tienda".id_tienda
        group by nticket_ticket having count(codigo_de_barras_productos)
```

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.

El procesamiento se desarrolla en una serie de fases:

- Parser: encargado de reconocer los identificadores y palabras claves del lenguaje SQL, realizar la interpretación semántica necesaria para comprender que tablas, funciones y operadores son referenciados en la consulta y además construye el árbol de traducción
- Rewriter: módulo en el cual se toma el árbol devuelto por el Parser y se realiza la búsqueda de reglas presentes dentro de la consulta
- Optimizer: crear un plan de ejecución óptimo. Primero, combina las posibles formas de recorrer y unir las relaciones que aparecen en la consulta (caminos). Todos los caminos creados llevan al mismo resultado y la tarea del optimizador es estimar el costo de ejecución de cada camino y encontrar cuál de éstos es el más barato
- Executor: Ejecuta la consulta utilizando el plan elegido por el Planner/Optimizer, este plan representa lógicamente los pasos necesarios para satisfacer la consulta, y físicamente es un árbol cuyos nodos representan operaciones básicas.

Optimización:

Para la optimización, PostgreSQL utiliza un plan (árbol) que se divide en diferentes subplanes. Para los accesos a tablas, postgres utiliza escaneo secuencial, escaneo con índice o escaneo de índice de mapa de bits. Para el join utiliza el bucle anidado, el ordenamiento por mezcla o el hash.

Cuando la consulta incluye más de dos relaciones, el planificador examina diferentes secuencias de acoples posibles para encontrar la menos costosa

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.