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: Ana Cortés Cercadillo

DNI:

ALUMNO 2:

Nombre y Apellidos: Carlos Javier Hellín Asensio

DNI:

Fecha: 14 de Abril

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 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 cual 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 comando ANALYZE recolecta las estadísticas sobre el contenido de las tablas de una base de datos y lo guarda en pg_statistic. También existe el dominio autovacuuum para cuando el contenido de una tabla ha cambiado suficientemente, entonces automáticamente ejecuta los comandos ANALYZE.

Las estadísticas que se recolectan incluye un listado de algunos de los datos más comunes de cada columna y un histograma mostrando los datos aproximadamente distribuidos en cada columna. Estos tipos de estadísticas son luego usados por el planificador de consultas para generar las consultas más eficientes.

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.

Se edita postgresql.conf quedando de la siguiente forma:

log_statement = 'all'

log_min_duration_statement = 0

Y se reinicia el servidor postgresql para aplicar los cambios.

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.

Se genera la base de datos:

```
postgres=# CREATE DATABASE empresa;
CREATE DATABASE
```

```
empresa=# CREATE TABLE empleados (numero_empleado numeric PRIMARY KEY, nombre text, apellidos text, salario numeric);
CREATE TABLE
empresa=# CREATE TABLE proyectos (numero_proyecto numeric PRIMARY KEY, nombre text, localizacion text, coste numeric);
CREATE TABLE
empresa=# CREATE TABLE trabaja_proyectos(numero_empleado numeric, numero_proyecto numeric, horas numeric, constraint fk_nu
mero_empleado foreign key (numero_empleado) REFERENCES empleados (numero_empleado) ON DELETE RESTRICT ON UPDATE RESTRICT,
constraint fk_numero_proyecto foreign key (numero_proyecto) REFERENCES proyectos (numero_proyecto) ON DELETE RESTRICT ON U
PDATE RESTRICT, PRIMARY KEY (numero_empleado, numero_proyecto));
CREATE TABLE
```

Se carga la información con el comando COPY que está optimizado para cargar un gran número de filas:

```
empresa=# \COPY empleados FROM '/var/lib/postgresql/datos_empleados.csv' DELIMITER ',';
COPY 2000000
empresa=# \COPY proyectos FROM '/var/lib/postgresql/datos_proyectos.csv' DELIMITER ',';
COPY 100000
empresa=# \COPY trabaja_proyectos FROM '/var/lib/postgresql/datos_trabaja_proyectos.csv' DELIMITER ',';
COPY 10000000
```

Y se obtienen los tiempos de carga a partir del LOG que son:

```
2020-04-25 09:47:36.978 PDT [42501] postgres@empresa LOG:  sentencia: COPY  empleados FROM STDIN DELIMITER ',';
2020-04-25 09:47:51.597 PDT [42501] postgres@empresa LOG:  duración: 14618.894 ms
2020-04-25 09:48:04.082 PDT [42501] postgres@empresa LOG:  sentencia: COPY  proyectos FROM STDIN DELIMITER ',';
2020-04-25 09:48:04.504 PDT [42501] postgres@empresa LOG:  duración: 421.695 ms
2020-04-25 09:48:15.226 PDT [42501] postgres@empresa LOG:  sentencia: COPY  trabaja_proyectos FROM STDIN DELIMITER ',';
2020-04-25 09:56:32.424 PDT [42501] postgres@empresa LOG:  duración: 497198.087 ms
```

datos_empleados.csv: 14618,894 ms

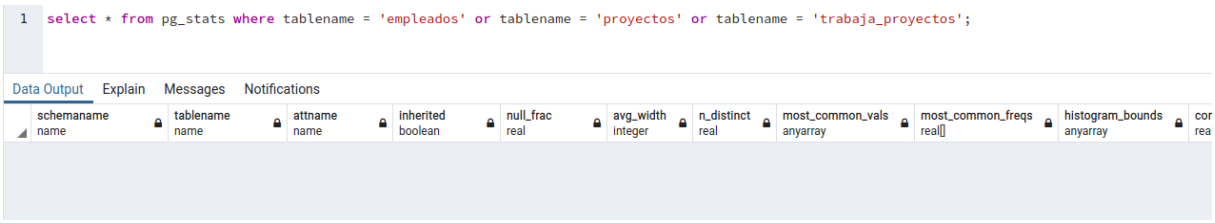
datos_proyectos.csv: 421,695 ms

datos_trabaja_proyectos.csv: 497198,087 ms

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?

Primeramente se ha desactivado el autovacuum para comprobar las estadísticas sin que se realice cambios de forma automática. Se modifica el archivo postgresql.conf y se pone autovacuum = off

Se usa pg_stats que provee acceso a la información almacenada en el catálogo pg_statistic (usado por ANALYZE para almacenar los datos) y se encuentra vacía, por lo tanto no es correcto.



Lo que almacena es:

schemaname: Nombre del esquema conteniendo la tabla

tablename: El nombre de la tabla

attname: Nombre de la columna descrita por esta fila

inherited: Si es verdadero, esta fila incluye columnas secundarias de herencia, no sólo los valores en la tabla especificada.

null_frac: Fracción de entradas de columnas que son nulas

avg_width: Media del ancho en bytes de las entradas de las columnas

n_distinct: Si es mayor que cero, el número estimado de valores distintos en la columna. Si es menor que cero, el negativo del número de valores distintos divididos por el número de filas.

most_common_vals: Una lista de los valores más comunes en la columna

most_common_freqs: Una lista de las frecuencias de los valores más comunes, es decir, el número de ocurrencias de cada uno dividido por el número total de filas.

histogram_bounds: Una lista de valores que dividen los valores de la columna en grupos de población aproximadamente igual. Los valores en most_common_vals, si están presentes, se omiten de este cálculo del histograma.

correlation: Estadística de correlación entre el orden físico de filas y el orden lógico de los valores de columna. Esto varía de -1 a +1. Cuando el valor está cerca de -1 o +1, se

estimará que una exploración de índice en la columna es más barata que cuando está cerca de cero, debido a la reducción del acceso aleatorio al disco.

most_common_elems: Una lista de valores de elementos no nulos que aparecen con mayor frecuencia dentro de los valores de la columna.

most_common_elem_freqs: Una lista de las frecuencias de los valores de elementos más comunes, es decir, la fracción de filas que contienen al menos una instancia del valor dado. Dos o tres valores adicionales siguen las frecuencias por elemento; estos son el mínimo y el máximo de las frecuencias por elemento anteriores y, opcionalmente, la frecuencia de elementos nulos.

elem_count_histogram: Un histograma de los recuentos de valores distintos de elementos no nulos dentro de los valores de la columna, seguido del número promedio de elementos distintos no nulos.

Como no es correcto, se puede actualizar usando el comando ANALYZE para cada tabla:

ANALYZE empleados;

ANALYZE empleados;

ANALYZE trabaja_proyectos;

Y ya se muestran los contenidos correctos en pg_stats:

```
1  select * from pg_stats where tablename = 'empleados' or tablename = 'proyectos' or tablename = 'trabaja_proyectos';
```

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	co re
1	public	empleados	numero_empleado	false	0	6	-1	[null]	[null]	{9,19089,36650,57223...	
2	public	empleados	nombre	false	0	13	-1	[null]	[null]	{nombre100001,nomb...	
3	public	empleados	apellidos	false	0	16	-1	[null]	[null]	{apellidos100001,apel...	
4	public	empleados	salario	false	0	6	93989	[null]	[null]	{1000.000,2018.000,3...	-0
5	public	proyectos	numero_proyecto	false	0	6	-1	[null]	[null]	{1,959,2019,3050,408...	
6	public	proyectos	nombre	false	0	11	-1	[null]	[null]	{nombre1,nombre110...	
7	public	proyectos	localizacion	false	0	17	-1	[null]	[null]	{localizacion1,localiza...	
8	public	proyectos	coste	false	0	6	9819	{14209.00,10140.00,16...	)003,0.0003,0.0003,0.0003}	{10000.00,10096.00,1...	-0.(
9	public	trabaja_proyectos	numero_empleado	false	0	6	-0.1855669	[null]	[null]	{2,20944,41508,61430...	0
10	public	trabaja_proyectos	numero_proyecto	false	0	6	97508	{45767}	{0.0002}	{8,1030,2076,3122,41...	0
11	public	trabaja_proyectos	horas	false	0	4	24	{14,20,2,7,15,23,21,13,8,...	).040033333,0.038833335}	[null]	

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.

Se modifica el fichero postgresql.conf y se edita los siguientes parámetros:

random_page_cost = 1.0

cpu_tuple_cost = 0.0

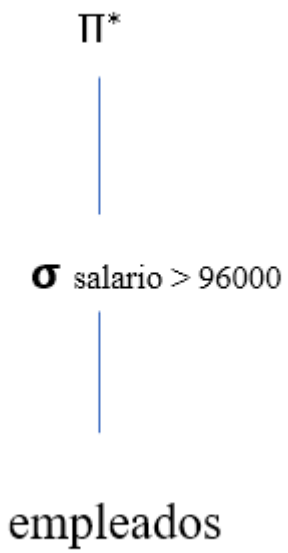
cpu_index_tuple_cost = 0.0

cpu_operator_cost = 0.0

y se reinicia el servidor PostgreSQL para aplicar los cambios.

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.

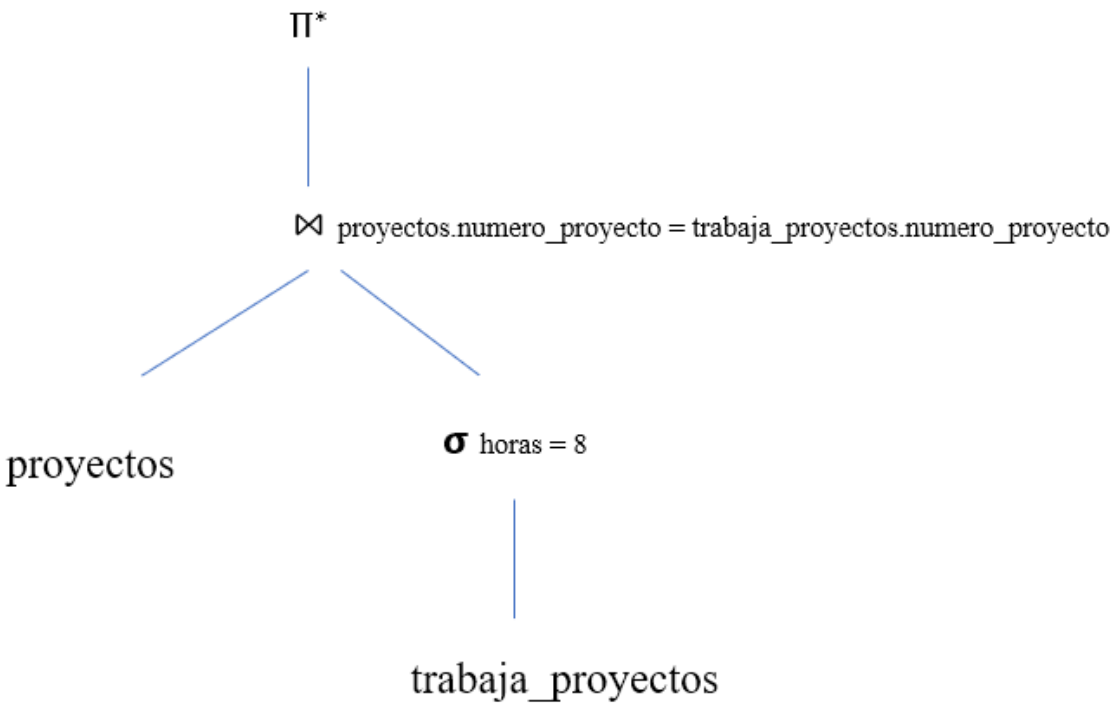
```
empresa=# EXPLAIN SELECT * FROM empleados WHERE salario > 96000;
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on empleados  (cost=0.00..18673.00 rows=97168 width=41)
   Filter: (salario > '96000'::numeric)
(2 filas)
```



El coste es br que es el tamaño de los bloques, por lo tanto los resultados son correctos porque al compararlo con teoría da lo mismo.

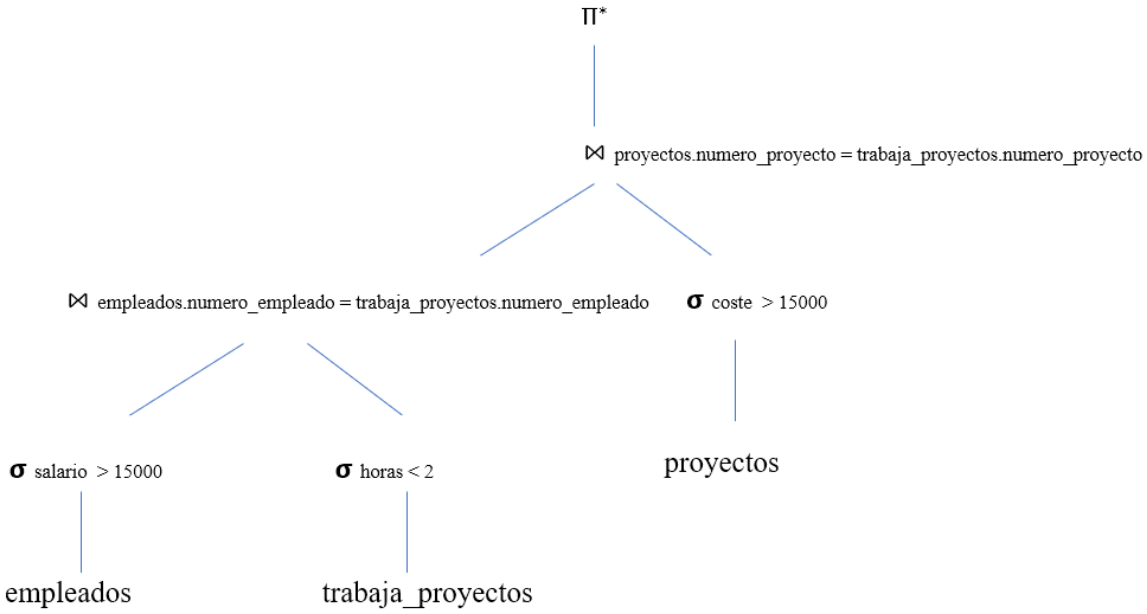
Cuestión 7: Aplicar el comando EXPLAIN a una consulta que obtenga la información de los proyectos en los cuales el empleado trabaja 8 horas. ¿Son correctos los resultados del comando EXPLAIN? ¿Por qué? Comparar con lo que se obtendría con lo visto en teoría.

```
empresa=# EXPLAIN SELECT * FROM proyectos JOIN trabaja_proyectos ON proyectos.numero_proyecto = trabaja_proyectos.numero_proyecto WHERE horas = 8;
-[ RECORD 1 ]-----------------------------------------------------------------------------------
QUERY PLAN | Nested Loop  (cost=0.00..64895.99 rows=413338 width=56)
-[ RECORD 2 ]-----------------------------------------------------------------------------------
QUERY PLAN |   ->  Seq Scan on trabaja_proyectos  (cost=0.00..63695.00 rows=413338 width=16)
-[ RECORD 3 ]-----------------------------------------------------------------------------------
QUERY PLAN |         Filter: (horas = '8'::numeric)
-[ RECORD 4 ]-----------------------------------------------------------------------------------
QUERY PLAN |   ->  Index Scan using proyectos_pkey on proyectos  (cost=0.00..0.00 rows=1 width=40)
-[ RECORD 5 ]-----------------------------------------------------------------------------------
QUERY PLAN |         Index Cond: (numero_proyecto = trabaja_proyectos.numero_proyecto)
```



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.

```
empresa=# EXPLAIN SELECT * FROM proyectos JOIN trabaja_proyectos ON proyectos.numero_proyecto = trabaja_proyectos.numero_proyecto JOIN empleados ON empleados.numero_empleado = t
rabaja_proyectos.numero_empleado WHERE coste > 15000 AND salario = 24000 AND horas < 2;
-[ RECORD 1 ]-----------------------------------------------------------------------------------------------
QUERY PLAN | Nested Loop  (cost=0.00..18820.00 rows=4 width=97)
-[ RECORD 2 ]-----------------------------------------------------------------------------------------------
QUERY PLAN |   ->  Nested Loop  (cost=0.00..18819.99 rows=9 width=57)
-[ RECORD 3 ]-----------------------------------------------------------------------------------------------
QUERY PLAN |         ->  Seq Scan on empleados  (cost=0.00..18673.00 rows=21 width=41)
-[ RECORD 4 ]-----------------------------------------------------------------------------------------------
QUERY PLAN |               Filter: (salario = '24000'::numeric)
-[ RECORD 5 ]-----------------------------------------------------------------------------------------------
QUERY PLAN |         ->  Index Scan using trabaja_proyectos_pkey on trabaja_proyectos  (cost=0.00..7.00 rows=1 width=16)
-[ RECORD 6 ]-----------------------------------------------------------------------------------------------
QUERY PLAN |               Index Cond: (numero_empleado = empleados.numero_empleado)
-[ RECORD 7 ]-----------------------------------------------------------------------------------------------
QUERY PLAN |               Filter: (horas < '2'::numeric)
-[ RECORD 8 ]-----------------------------------------------------------------------------------------------
QUERY PLAN |   ->  Index Scan using proyectos_pkey on proyectos  (cost=0.00..0.00 rows=1 width=40)
-[ RECORD 9 ]-----------------------------------------------------------------------------------------------
QUERY PLAN |         Index Cond: (numero_proyecto = trabaja_proyectos.numero_proyecto)
-[ RECORD 10 ]----------------------------------------------------------------------------------------------
QUERY PLAN |         Filter: (coste > '15000'::numeric)
```



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?

Tabla	Tiempo sin integridad	Tiempo con integridad
Tienda	734,839 ms	792,729 ms
Productos	10246,971 ms	29680,175 ms
Tienda_Productos	382198,670 ms	635334,344 ms
Trabajador	7281,674 ms	32422,535 ms
Ticket	71068,999 ms	74791,727 ms
Ticket_Productos	74020,050 ms	833504,267 ms

El orden es el que se especifica en la tabla, dicho orden es debido a que para insertar datos en tablas como Tienda_Productos o Ticket_Productos es necesario que existan datos en otros tablas que están relacionadas

Al comparar las diferencias entre los tiempos obtenidos y los que aparecen en el LOG se aprecian unos pocos milisegundos de diferencia. Al usar \timing para medir el tiempo de los COPY, lo que hace es medir el tiempo en el cliente, mientras que el LOG

contiene la duración en la parte del servidor. El tiempo del cliente incluye la sobrecarga de transferir datos al servidor.

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.

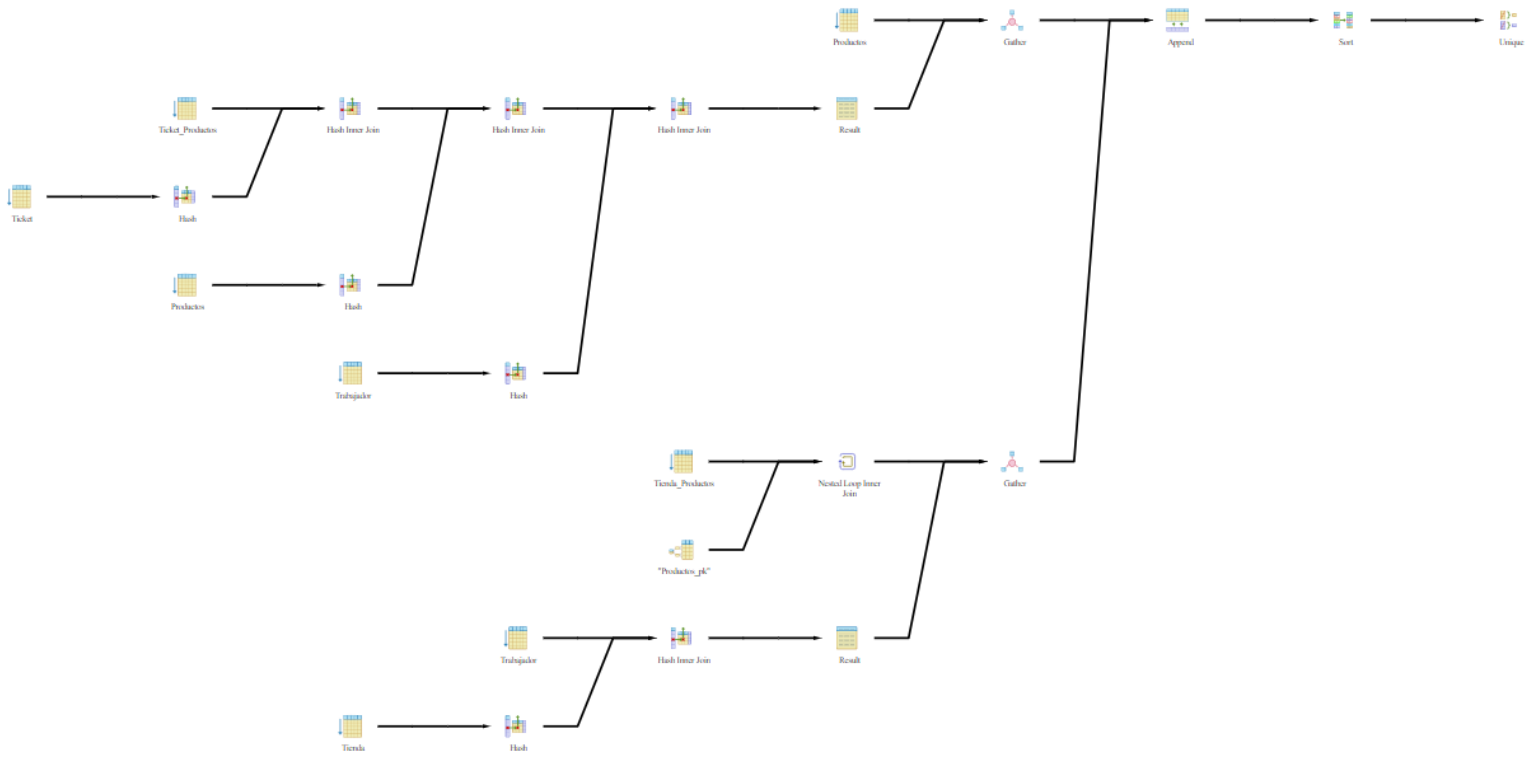
Cuestión 10: Realizar una consulta SQL que muestre “el nombre y DNI de los trabajadores que hayan vendido algún ticket en los cuatro últimos meses del año con más de cuatro productos en los que al menos alguno de ellos tenga un precio de más de 500 euros, junto con los trabajadores que ganan entre 3000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

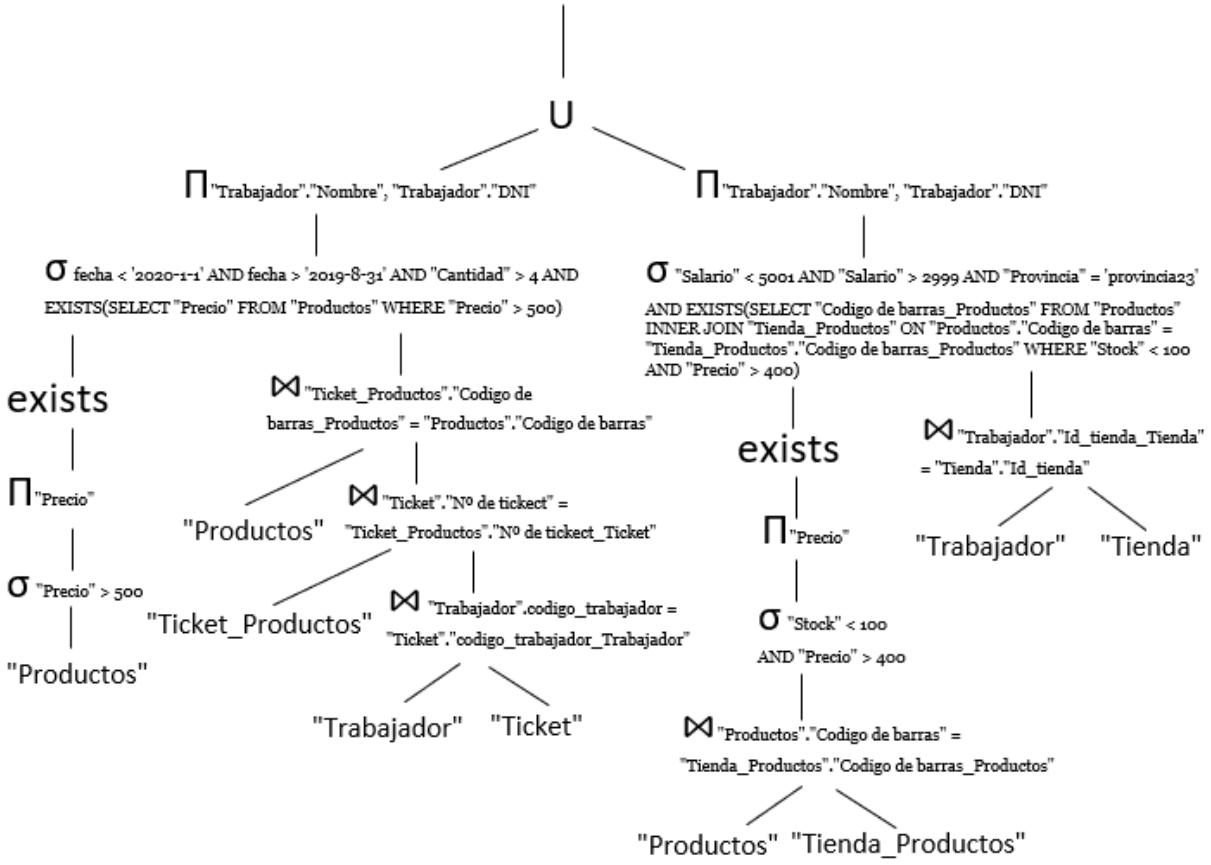
En la consulta realizada se usa la provincia “provincia23” en vez de “Comunidad de Madrid” por la forma en la que están generados los datos. Se podría usar cualquier otra.

```
SELECT "Trabajador"."Nombre", "Trabajador"."DNI" FROM "Ticket" INNER JOIN "Trabajador" ON "Trabajador".codigo_trabajador = "Ticket"."codigo_trabajador_Trabajador" INNER JOIN "Ticket_Productos" ON "Ticket"."Nº de tickect" = "Ticket_Productos"."Nº de tickect_Ticket" INNER JOIN "Productos" ON "Ticket_Productos"."Codigo de barras_Productos" = "Productos"."Codigo de barras" WHERE fecha < '2020-1-1' AND fecha > '2019-8-31' AND "Cantidad" > 4 AND EXISTS(SELECT "Precio" FROM "Productos" WHERE "Precio" > 500) UNION SELECT "Trabajador"."Nombre", "Trabajador"."DNI" FROM "Trabajador" INNER JOIN "Tienda" ON "Trabajador"."Id_tienda_Tienda" = "Tienda"."Id_tienda" WHERE "Salario" < 5001 AND "Salario" > 2999 AND "Provincia" = 'provincia23' AND EXISTS(SELECT "Codigo de barras_Productos" FROM "Productos" INNER JOIN "Tienda_Productos" ON "Productos"."Codigo de barras" = "Tienda_Productos"."Codigo de barras_Productos" WHERE "Stock" < 100 AND "Precio" > 400);
```

El árbol de álgebra relacional del plan de ejecución del resultado de la consulta que obtiene postgreSQL es el siguiente:



El árbol obtenido por nosotros al traducir la consulta original al álgebra relacional:



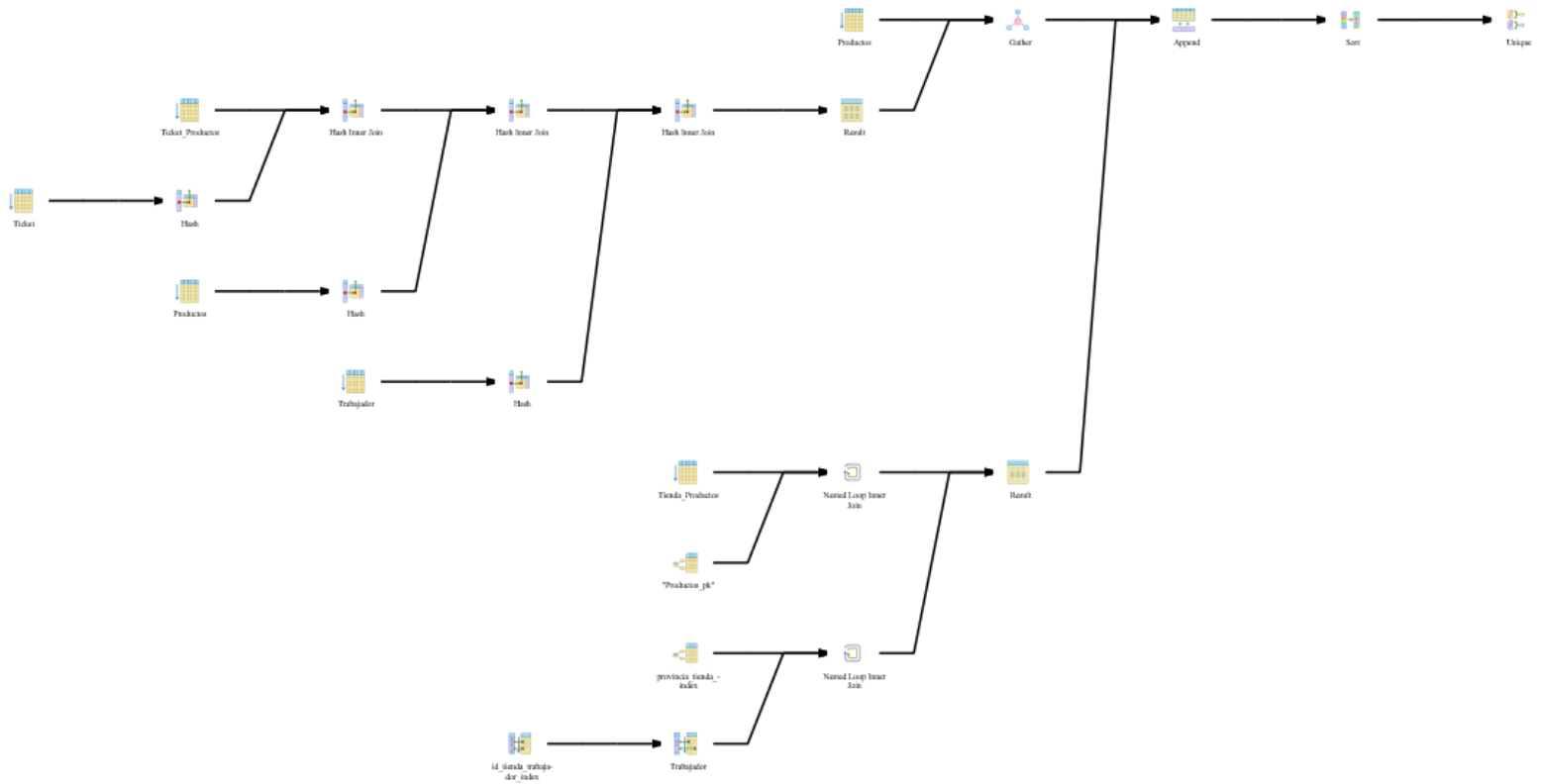
La principal diferencia que se encuentra en la rama de la izquierda es el orden en el que se hacen los inner join en las tablas. Finalmente se hace en los dos árboles la selección del resultado de todos los join con la condición de la tabla Productos. En la rama de la izquierda, la condición de la selección hace un join en las tablas Productos y Tienda_Productos que según el árbol de postgreSQL se hace mediante el índice Productos_pk en la tabla Productos. Por último, se observa que la salida se ordena (sort) y se usa la sentencia unique.

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 álgebra relacional. Comentar los resultados obtenidos y comparar con la cuestión anterior.

Para mejorar el rendimiento de la consulta anterior, se crean índices en los atributos que hacen que el coste de la consulta se reduzca ("Id_tienda_Tienda" de la tabla "Trabajador" y "Provincia" de la tabla "Tienda").

También se modifica los parámetros del fichero postgresql.conf de max_parallel_workers_per_gather a 8 que es el valor máximo de procesos que se pueden hacer de forma paralela (max_parallel_workers = 8). Otra forma de disminuir considerablemente el coste es cambiando parallel_tuple_cost a 0.001 y parallel_setup_cost a 1.

El árbol de álgebra relacional de PostgreSQL quedaría así:



Se puede ver como en la parte más baja del árbol aparece el uso de estos índices que se han creado.

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 álgebra relacional. Comparar con los resultados anteriores.

Primero se deshabilitan los triggers de las tablas que vamos a modificar. Se realiza el borrado de la tabla Tienda de forma aleatoria y después de las tablas asociadas directamente. Finalmente se vuelven a habilitar los triggers.

```
alter table "Tienda" disable trigger all;
alter table "Tienda_Productos" disable trigger all;
alter table "Trabajador" disable trigger all;

delete from "Tienda" where "Id_tienda" in(select "Id_tienda" from "Tienda" order by random() limit 100000);
delete from "Tienda_Productos" where not exists (select null from "Tienda" where "Tienda"."Id_tienda"="Tienda_Productos"."Id_tienda_Tienda");
delete from "Trabajador" where not exists (select null from "Tienda" where "Tienda"."Id_tienda"="Trabajador"."Id_tienda_Tienda");

alter table "Tienda" enable trigger all;
alter table "Tienda_Productos" enable trigger all;
alter table "Trabajador" enable trigger all;
```

```
DELETE 100000

Query returned successfully in 1 secs 980 msec.
```

En el borrado de la tabla "Tienda" se tarda 1 segundo y 980 milisegundos.

```
DELETE 10000000

Query returned successfully in 4 min 19 secs.
```

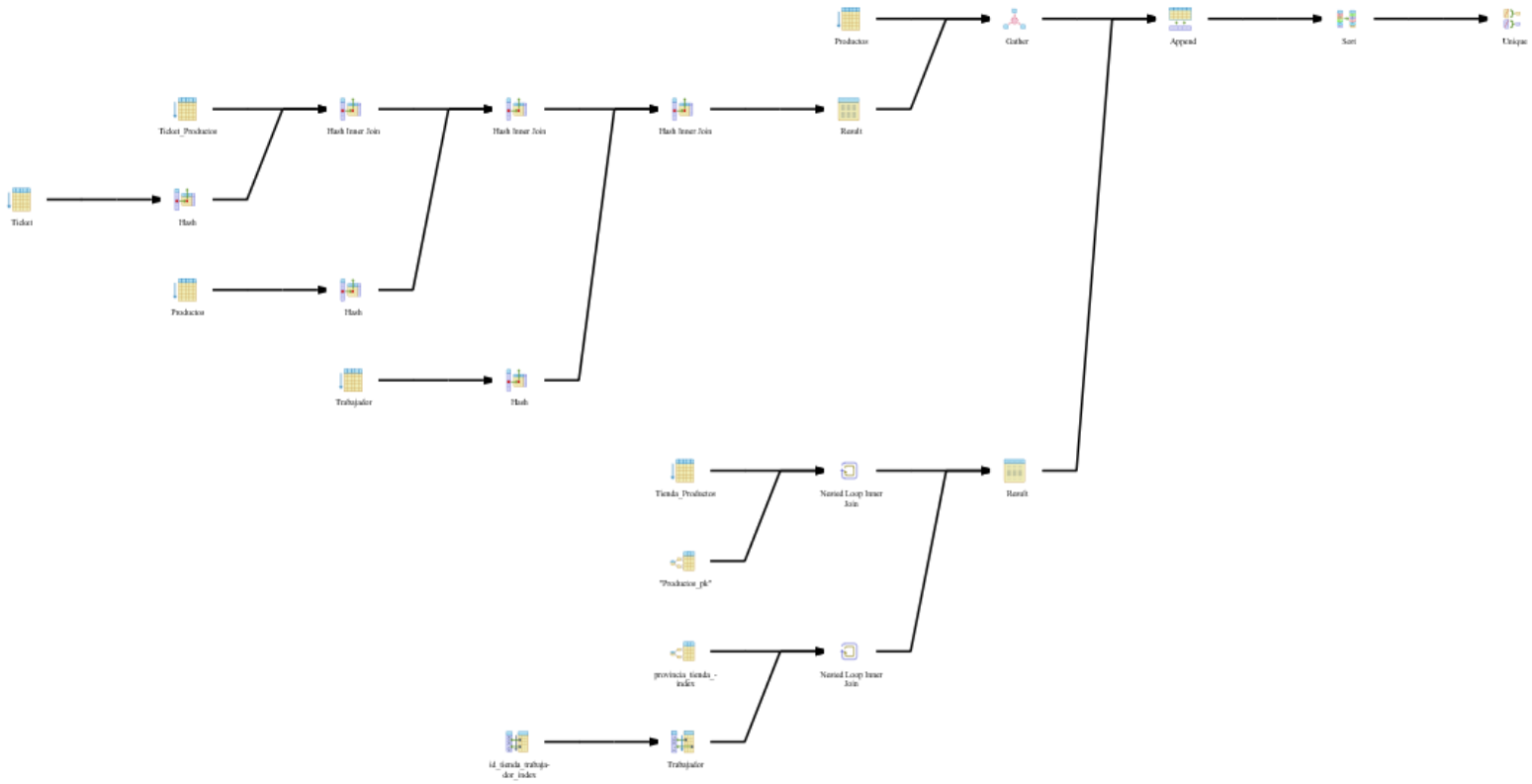
En el borrado de la tabla "Tienda_Productos" se tarda 4 minutos y 19 segundos.

```
DELETE 499665

Query returned successfully in 18 secs 434 msec.
```

En el borrado de la tabla "Trabajador" se tarda 18 segundos y 434 milisegundos.

El árbol de la consulta sería este:



La forma de ejecutar la consulta sería igual que en el anterior ejercicio. Lo que se reduce es el coste de la consulta respecto a anteriormente ya que se tienen que leer menos datos.

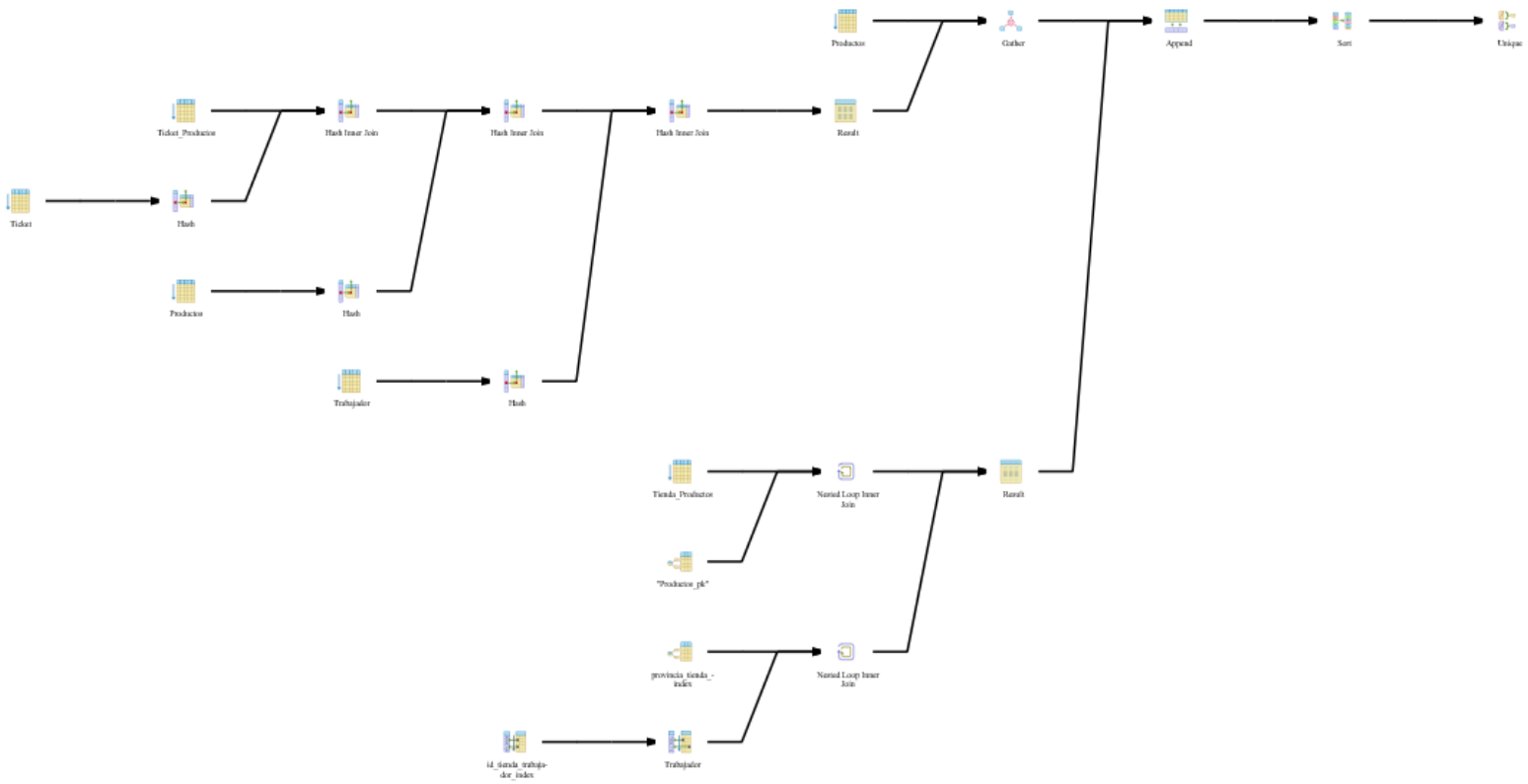
Cuestión 13: ¿Qué técnicas de mantenimiento de la BD propondría para mejorar los resultados de dicho plan sin modificar el código de la consulta? ¿Por qué?

Para mejorar los resultados del plan sin modificar el código de la consulta, se llevan a cabo técnicas de mantenimiento de la base de datos como la ejecución del comando VACUUM. El comando VACUUM se utiliza para recuperar o reutilizar el espacio en disco ocupado por filas actualizadas o eliminadas, para actualizar las estadísticas de datos utilizadas por el planificador de consultas PostgreSQL, para actualizar el mapa de visibilidad, que acelera los escaneos de solo índice o para protegerse contra la pérdida de datos muy antiguos debido a la identificación de transacción envolvente o la identificación de múltiples activos envolvente.

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 álgebra relacional. Compare los resultados del plan de ejecución con los de los apartados anteriores. Coméntelos.

Se ejecuta el comando VACUUM: vacuum;

El resultado en forma de árbol de álgebra relacional es el mismo que en los ejercicios anteriores. El cambio que se ha producido con respecto a antes de la ejecución del



comando es una reducción en el coste al haber liberado memoria y actualizado las estadísticas.

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.

Primero se modifica el fichero postgresql.conf: shared_preload_libraries = 'pg_stat_statements'.

Después se ejecuta la instrucción: CREATE EXTENSION pg_stat_statements;.

Finalmente para saber las 5 consultas más utilizadas, se lleva a cabo la siguiente consulta: SELECT query, calls, total_time, rows, 100.0 * shared_blks_hit / nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent FROM pg_stat_statements WHERE dbid = (select oid from pg_database where datname = 'tienda') ORDER BY total_time DESC LIMIT 5;

	query text	calls bigint	total_time double precision	rows bigint	hit_percent numeric
1	/*pga4dash*/	115	2091.228400000001	575	100.0000000000000000
2	SELECT set_config($1,$2,$3) FROM pg_settings WHERE name = $4	3	858.7189	3	[null]
3	SELECT DISTINCT dep.deptype, dep.classid, cl.relkind, ad.adbin, pg_get_expr(ad.adbin, ad.adrelid) as adsrc,	3	25.5185	0	69.6808510638297872
4	SELECT at.attname, ty.typname, at.attnum	1	18.6593	23	68.4210526315789474
5	SELECT query, calls, total_time, rows, $1 * shared_blks_hit / nullif(shared_blks_hit + shared_blks_read, $2) A...	3	3.4414	6	100.0000000000000000

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.

PostgreSQL realiza una serie de pasos para mostrar el resultado de una consulta. Primero analiza la consulta y la traduce a una expresión del álgebra relacional. Esta expresión la podemos obtener en forma de árbol como se ha visto en los ejercicios anteriores. Con esta expresión y las estadísticas guardadas de los datos, se consigue el plan de ejecución. Se puede saber el plan de ejecución a través del comando EXPLAIN sin llegar a realizar la consulta. Una vez se tenga el plan de ejecución, el motor de evaluación de PostgreSQL la lleva a cabo y obtiene el resultado final de la consulta. Para optimizar la consulta, antes de obtener el plan de ejecución, PostgreSQL contiene comandos para liberar espacio en disco y actualizar las estadísticas. Esto ayuda a disminuir el coste de las consultas.

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.