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: David Ramos Fernández

ALUMNO 2:

Nombre y Apellidos: Sergio Sánchez Campo

Fecha: 22/04/2020

Profesor Responsable: Santiago Hermira Anchuelo

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?

PostgreSQL tiene una herramienta encargada de elaborar estadísticas sobre las tablas y las almacena en *pg_statistic*. Este módulo almacena la siguiente información:

- ***Stainherit:*** muestra un booleano en función de si existe columnas con herencia.
- ***Stanullfrac:*** muestra la fracción de entradas de la columna que son nulos.
- ***Stawidth:*** muestra la media de entradas no nulas que se guardan en la tabla.
- ***Stadistinct:*** muestra el número de valores distintos que hay en la columna.

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.

Para modificar el *log* de errores debemos acceder al archivo *postgresql.conf* y activar aquellos *log* de errores que estén deshabilitados en la sección *REPORTING AND LOGGING*. Los *logs* de errores que debemos modificar son:

- # log_checkpoints = off a log_checkpoints = on : permite registrar los puntos de control y reinicio del servidor.

- # log_duration= off a log_duration = on : este cambio permitirá mostrar la duracion de cada sentencia completada.
- # log_error_verbosity= default a log_error_verbosity= default: controla la cantidad de detalle escrito en el servidor para cada mensaje que se registra.
- # log_statement= 'none' a log_statement= all : controla las sentencias de SQL que se registran. Poniendolo en *all* registramos todas las sentencias.
- # log_replication_commands= off a log_replication_commands= on : permite almacenar cada comando que se replica.
- # log_temp_files= -1 a log_temp_files= 0: permite controlar el registro del nombre y tamaño de los archivos temporales.

Una vez modificado el archivo guardamos los cambios realizados y reiniciamos el servidor.

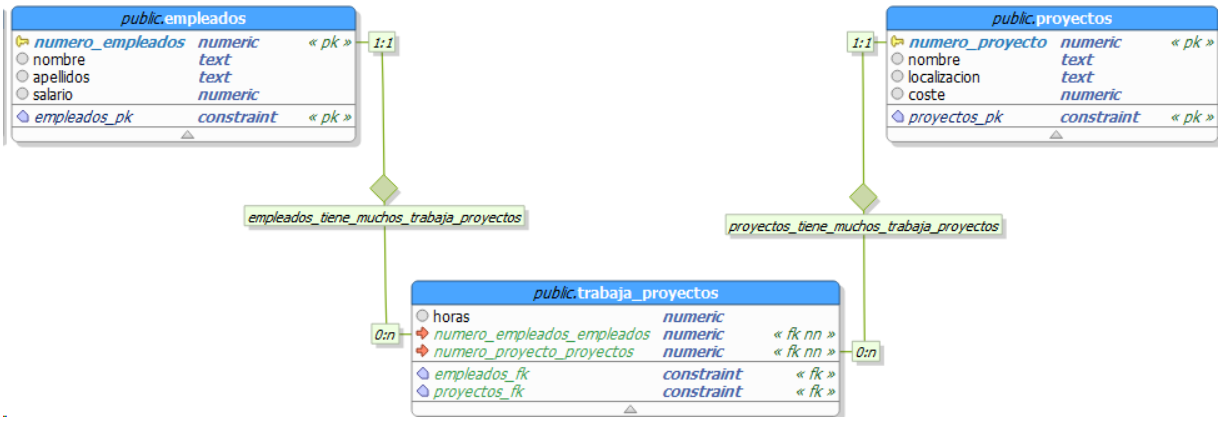
Cuestión 3: Crear una nueva base de datos llamada **empresa** y que tenga las siguientes tablas con los siguientes campos y características:

- 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 realizaremos un modelo relacional con las tablas propuestas en la cuestión haciendo uso de la herramienta pgmodeler dando como resultado la siguiente imagen:



A continuación, exportamos este modelo en formato SQL y lo importamos en nuestra base de datos llamada **empresa**. A continuación, cargamos los datos mediante el comando copy en SQL ya que es el más eficiente, comenzamos con las tablas padre: empleado y proyectos y finalmente cargamos **trabaja_proyectos** porque si no se produciría errores con la foreign key.

```
COPY empleados FROM 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\datos_empleados.csv' DELIMITER ','
```

```
COPY 2000000

Query returned successfully in 21 secs 474 msec.
```

```
COPY proyectos FROM 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\datos_proyectos.csv' DELIMITER ','
```

```
COPY 100000

Query returned successfully in 1 secs 493 msec.
```

```
COPY trabaja_proyectos FROM 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\datos_trabaja_proyectos.csv'  DELIMITER ','
```

```
ERROR:  inserción o actualización en la tabla «trabaja_proyectos» viola la llave foránea «proyectos_fk»
DETAIL:  La llave (numero_proyecto_proyectos)=(0) no está presente en la tabla «proyectos».
SQL state: 23503
```

Debido al error producido por la violación de la clave foránea, decidimos añadir un proyecto con numero_proyecto= 0.

```
INSERT INTO public.proyectos(
    numero_proyecto, nombre, localizacion, coste)
    VALUES (0,'nombre0','localizacion0',1230.00)
```

Una vez insertado el dato volvemos a realizar la operación y nos da el siguiente tiempo de carga:

```
COPY 10000000

Query returned successfully in 7 min 5 secs.
```

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?

Antes de ver las estadísticas necesitamos identificar el oid de las tablas que vamos a analizar. Para ello realizamos la siguiente operación:

```
SELECT * FROM pg_class Where relname= 'empleados'
```

	oid oid	relname name
1	17440	empleados

```
SELECT * FROM pg_class Where relname= 'proyectos'
```

	oid oid	relname name
1	17448	proyectos

```
SELECT * FROM pg_class Where relname= 'trabaja_proyectos'
```

	oid oid	relname name
1	17456	trabaja_proyectos

Una vez obtenidos los *oid* de las tablas vamos a ver las estadísticas de las tablas anteriores ejecutamos la siguiente consulta:

Tabla empleados:

Data Output Explain Messages

	starelid oid	staattnum smallint	stainherit boolean	stanullfrac real	stawidth integer	stadistinct real
1	17440	1	false	0	6	-1
2	17440	2	false	0	13	-1
3	17440	3	false	0	16	-1
4	17440	4	false	0	6	94621

Tabla proyectos:

Data Output Explain Messages

	starelid oid	staattnum smallint	stainherit boolean	stanullfrac real	stawidth integer	stadistinct real
1	17448	1	false	0	6	-1
2	17448	2	false	0	11	-1
3	17448	3	false	0	17	-1
4	17448	4	false	0	6	9858

Tabla trabaja_proyectos:

Data Output Explain Messages

	starelid oid	staattnum smallint	stainherit boolean	stanullfrac real	stawidth integer	stadistinct real
1	17456	1	false	0	6	-0.16248775
2	17456	2	false	0	6	97093
3	17456	3	false	0	4	24

Podemos observar las columnas:

- **stainherit**, que si es true significa que la columna es herencia de otra, en este caso son todas falsas porque ninguna es herencia de otra.
- **stanullfrac** que nos muestra el porcentaje de tuplas nulas que presenta cada campo de cada tabla. Vemos que es 0 en todos los campos porque no hay ningún valor nulo.
- **stawidth** que nos muestra el ancho medio de las entradas no nulas para ese campo.
- **stadistintc** que muestra el número de valores no nulos diferentes que tiene ese campo. Si es 0 indica que es desconocido, si es positivo es el número de valores distintos y si es negativo es el porcentaje de números distintos.

Las estadísticas no son correctas debido a que hemos realizado una inserción de un número muy elevado de datos y al no haber actualizado las estadísticas muchas están obsoletas. Para solucionar esto lo que deberemos hacer es actualizarlas y para ello emplearemos el comando ***analyze***. Tras esto, las estadísticas de las tablas quedarán actualizadas.

Tabla empleados:

Data Output Explain Messages

	starelid oid	staattnum smallint	stainherit boolean	stanullfrac real	stawidth integer	stadistinct real
1	17440	3	false	0	16	-1
2	17440	1	false	0	6	-1
3	17440	2	false	0	13	-1
4	17440	4	false	0	6	97440

Tabla proyectos:

Data Output Explain Messages

	starelid oid	staattnum smallint	stainherit boolean	stanullfrac real	stawidth integer	stadistinct real
1	17448	2	false	0	11	-1
2	17448	3	false	0	17	-1
3	17448	1	false	0	6	-1
4	17448	4	false	0	6	9833

Tabla trabaja_proyectos:

Data Output Explain Messages

	starelid oid	staattnum smallint	stainherit boolean	stanullfrac real	stawidth integer	stadistinct real
1	17456	3	false	0	4	24
2	17456	1	false	0	6	-0.15928216
3	17456	2	false	0	6	98803

Vemos que los datos han variado y ahora sí 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.

Para configurar los costes que tiene en cuenta el comando EXPLAIN debemos acceder al archivo *postgresql.conf* y poner a 0 aquellos costes que no queremos que tenga en cuenta en la sección *PLANNER COST CONSTANTS* en el apartado *QUERY TUNING*. Los *costes* que debemos modificar son:

- # *cpu_tuple_cost* = 0.01 a *cpu_tuple_cost* = 0: coste de procesamiento de cada fila en una consulta.
- # *cpu_index_tuple_cost* = 0.005 a *cpu_index_tuple_cost* = 0: coste de procesamiento de cada entrada de índice en un escaneo de índice.
- # *cpu_operator_cost* = 0.0025 a cpu_operator_cost = 0: coste de procesar cada operador o función ejecutada en una consulta.
- # parallel_tuple_cost= 0.1 a parallel_tuple_cost= 0: coste de transmitir una tupla de un proceso de trabajo paralelo a otro proceso.

No modificamos el campo *random_page_cost* porque modificaría la preferencia de los índices del sistema dando lugar a errores y el campo *parallel_setup_cost* porque ya que consideramos que al tener un coste muy alto es preferible dejarlo para evitar que las consultas con un coste elevado parezcan una consulta optima además de ocurrir en situaciones excepcionales.

Una vez terminada la modificación de los campos guardamos el archivo y los cerramos.

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.

Realizamos la siguiente consulta aplicando el comando EXPLAIN:

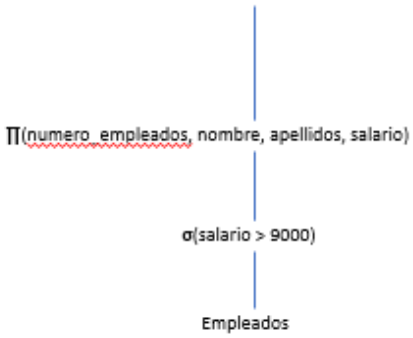
```
EXPLAIN Select * From empleados Where salario > 96000
```

	QUERY PLAN text
1	Seq Scan on empleados (cost=0.00..18673.00 rows=102495 width=41)
2	Filter: (salario > '96000'::numeric)

Observamos en la tercera fila en *Parallel Seq Scan* se ha realizado una búsqueda secuencial con un coste total de 18673 bloques y la longitud media de cada registro de 41 como se indica en la primera fila. Y un número de registros de 102.495.

Ahora realizaremos el cálculo teórico del coste y número de registros:

Primero realizamos un esquema optimizado de la consulta:



El coste de esta consulta es equivalente a la lectura de la tabla Empleados y la materialización del resultado.

Primero obtendremos la Lr (Longitud de registro) y para ello realizaremos las siguientes consultas y sumaremos sus resultados:

```
Select avg(pg_column_size(numero_empleados)) From empleados
```

	avg numeric
1	6.9898010000000000

```
Select avg(pg_column_size(nombre)) From empleados
```

Data Output Explain

	avg numeric
1	13.4444480000000000

```
Select avg(pg_column_size(apellidos)) From empleados
```

Data Output | Explain

	avg numeric
1	16.4444480000000000

```
Select avg(pg_column_size(salario)) From empleados
```

Data Output | Explain

	avg numeric
1	6.8199110000000000

La suma de los resultados es igual 43.70 redondeado a dos decimales. Por tanto, Lr = 43.70.

Fr (Factor de bloque) = Br/Fr = 8192/43.70 = 187 reg/bloq

Nr (número de registro) = 2000000

Coste= Nr/Fr = 2000000/187= **10.695,18717** bloques.

Más el coste de materializar el resultado->

Para ello hay que calcular el número de registros de esta tabla cuyo salario es mayor de 96.000

```
Select count(numero_empleados) from empleados Where salario > 96000
```

count bigint
99523

Realizamos el cálculo del coste:

Fr (Factor de bloque) = Br/Fr = 8192/43.70 = 187 reg/bloq

Nr (número de registro) = 99.523

Coste= Nr/Fr = 99.523/187= 533 bloques.

Coste total= 10.695,18717+533= .11.228,18717.

Registros totales= 99.523

Observamos que el número de registros totales que se obtienen es prácticamente idéntico; en cambio el coste total tiene una diferencia alrededor de los 7.000 bloques. El coste de leer la tabla Trabajadores en teoría es de 10.695 bloques, en cambio con el comando *Explain* nos da un resultado de 18.673 bloques. Resultados dispares, debido a diversos factores como el número de registros que caben en cada bloque ya que estos contienen otros datos aparte de solamente los campos de dicha tabla, como ya vimos en la práctica 1 y por lo tanto el número de bloques que se necesitará para almacenar ese número de registros será bastante superior en la práctica que en la teoría lo que repercutirá en un mayor coste. También influirán otros aspectos como el tamaño de

memoria empleado y el cálculo de la media de la longitud de los campos de la tabla. Pero el proceso que sigue es el mismo que el teórico, primero lee la tabla, luego los costes de la proyección y la selección no existen puesto que con encauzamiento estos procesos no tienen coste y finalmente materializa el resultado.

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.

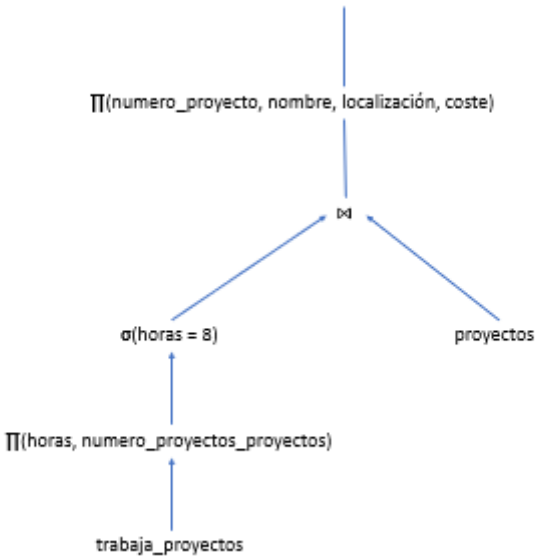
Realizamos la siguiente consulta aplicando el comando EXPLAIN:

```
Explain Select numero_proyecto,nombre,localizacion,coste
From proyectos inner join trabaja_proyectos on numero_proyecto = numero_proyecto_proyectos
Where horas = 8
```

	QUERY PLAN text
1	Nested Loop (cost=0.00..63743.00 rows=6 width=40)
2	-> Seq Scan on trabaja_proyectos (cost=0.00..63695.00 rows=6 width=4)
3	Filter: (horas = '8'::numeric)
4	-> Index Scan using proyectos_pk on proyectos (cost=0.00..8.00 rows=1 width=40)
5	Index Cond: (numero_proyecto = trabaja_proyectos.numero_proyecto_proyectos)

Observamos en la cuarta fila en *Parallel Seq Scan* se ha realizado una búsqueda secuencial para el campo Trabaja proyectos con un coste total de 63.695 bloques. Ahora realizaremos el cálculo teórico:

Primero realizaremos un esquema optimizado de la consulta:



Comenzaremos obteniendo la Lr (Longitud de registro) y para ello realizaremos las siguientes consultas y sumaremos sus resultados:

La suma de los resultados es igual 18,71 redondeado a dos decimales. Por tanto, Lr = 18,71.

```
Select avg(pg_column_size(numero_proyecto)) From proyectos
```

avg numeric
6.7997820021799782

```
Select avg(pg_column_size(nombre)) From proyectos
```

avg numeric
11.8889111108888911

```
Select avg(pg_column_size(localizacion)) From proyectos
```

avg numeric
17.8889111108888911

```
Select avg(pg_column_size(coste)) From proyectos
```

avg numeric
6.9997000029999700

```
Select avg(pg_column_size(horas)) From trabaja_proyectos
```

avg numeric
6.9897550000000000

```
Select avg(pg_column_size(numero_empleados_empleados)) From trabaja_proyectos
```

avg numeric
6.7995492000000000

```
Select avg(pg_column_size(numero_proyecto_proyectos)) From trabaja_proyectos
```

avg numeric
4.9166676000000000

Fr (Factor de bloque) = Br/Fr = 8192/18,71 = 438 reg/bloq

Nr (número de registro) = 100.001

Coste= Nr/Fr = 100.001/ 438 = **228** bloques.

Ahora realizaremos lo mismo con la tabla trabaja_proyectos:

La suma de los resultados es igual 43,58 redondeado a dos decimales. Por tanto, Lr = 43,58.

Fr (Factor de bloque) = Br/Fr = 8192/43,58 = 188 reg/bloq

Nr (número de registro) = 10.000.000

Coste= Nr/Fr = 10.000.000/ 188 = **53.191** bloques.

Vemos que el primer paso seguido es igual en el teórico y en el práctico, pero el costo es prácticamente la mitad en el teórico, que se deberá a las mismas causas que comentamos en el ejercicio anterior.

A continuación, habría que calcular el número de registros de trabaja_proyecto que tienen horas=8 -> 11 registros.

```
Select count(numero_empleados_empleados) From trabaja_proyectos Where horas = 8
```

count bigint
11

Calculamos el coste del *join* que encauzado costaría: 228/256*1=1 bloque.

Calculamos el número de registros que salen del *join* = (11*100.001/max(11, 100.0001) = 11 registros.

Así que el coste de materializarlos será de 1 bloque.

El coste total es: 1+1+53.191=**53.193 bloques.**

Varía 10.000 bloques aproximadamente respecto a los cálculos teóricos del *explain*, debido a las mimas causas que en el ejercicio anterior.

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.

```
Explain Select numero_proyecto, proyectos.nombre, localizacion, coste
From proyectos inner join trabaja_proyectos on numero_proyecto = numero_proyecto_proyectos
inner join empleados on numero_empleados_empleados = numero_empleados
Where coste > 15000 and salario = 24000 and horas < 2
```

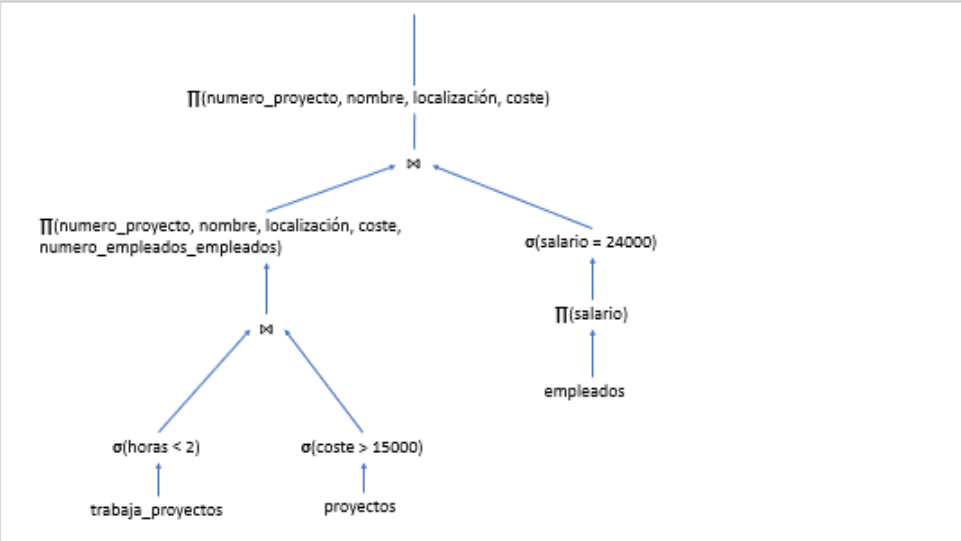
	QUERY PLAN text
1	Merge Join (cost=63695.49..65629.77 rows=1 width=40)
2	Merge Cond: (empleados.numero_empleados = trabaja_proyectos.numero_empleados_empleados)
3	-> Index Scan using empleados_pk on empleados (cost=0.00..40620.00 rows=21 width=6)
4	Filter: (salario = '24000'::numeric)
5	-> Sort (cost=63695.49..63695.49 rows=499 width=46)
6	Sort Key: trabaja_proyectos.numero_empleados_empleados
7	-> Merge Join (cost=63695.00..63695.49 rows=499 width=46)
8	Merge Cond: (proyectos.numero_proyecto = trabaja_proyectos.numero_proyecto_proyectos)
9	-> Index Scan using proyectos_pk on proyectos (cost=0.00..2032.00 rows=49879 width=40)
10	Filter: (coste > '15000'::numeric)
11	-> Sort (cost=63695.00..63695.00 rows=1000 width=10)
12	Sort Key: trabaja_proyectos.numero_proyecto_proyectos
13	-> Seq Scan on trabaja_proyectos (cost=0.00..63695.00 rows=1000 width=10)
14	Filter: (horas < '2'::numeric)

Observamos que el coste total de esta consulta ha sido de **65.629,77 bloques**.

En primer lugar, escanea la tabla Trabaja_Proyectos mediante un escaneo secuencial y ejecuta el filtro de horas<2. A continuación, ordena esta tabla según el campo número de proyectos, el cual empleará más adelante. Luego ejecuta un escaneo por índice en la tabla Proyectos empleando su PK y filtra por el coste>15.000. Prosigue realizando el *join* entre las tablas Proyectos y Trabaja_proyectos, y el coste de este j*oin* se reduce mucho gracias a la ordenación que se realiza anteriormente de la tabla Trabaja_proyectos. Tras realizar este *join*, reordena la tabla resultante según el campo numero_empleados. Después escanea la tabla Empleados empleando como índice su PK y filtra por el salario=24.000. Finalmente, realiza el *join* entre la tabla Empleados y la tabla generada por el anterior *join*, este *join* se realiza según el campo numero_empleados, reduciendo mucho su coste gracias a la ordenación que PostgreSQL había realizado previamente en este campo.

a consulta teóricamente:

Primero realizaremos el esquema optimizado de la consulta:



A continuación, hay que leer las tres tablas que se van a emplear:

Comenzamos por Empleados la cual ya calculamos el coste de leerla en el ejercicio anterior y era de 10.969 bloques. La tabla proyectos presenta un coste de 228 bloques y Trabajo_Proyectos cuesta 53.191 bloques.

A continuación, calculamos el número de registros que quedan tras realizar las proyecciones en cada tabla para decidir cuál es la menor y sobre cual comenzar realizando los *joins*.

```
Select count (distinct numero_proyecto) from proyectos Where coste >15000
```

count bigint
49868

```
Select count (distinct numero_empleados) from empleados Where salario = 24000
```

count bigint
21

```
Select count(numero_empleados_empleados) from trabaja_proyectos Where horas < 2
```

count bigint
9

Como resultado las tablas con menor número de registros y de bloques son empleados y Trabaja_Proyectos así que comenzaremos realizando el *inner join* entre estas dos tablas. El coste total es: (1/256-3) *1= 1 bloques. Y como M-2>1 no hace falta coste para encauzar la tabla.

A continuación, realizamos el segundo *join*, que sería entre las tablas Proyectos y el resultado del anterior *join*. Y su coste sería de: (114/256-3) *1 =1 bloques. Y como M-2>1 no hace falta coste para encauzar la tabla.

Finalmente, el coste de materializar el resultado, que es de 1 bloque.

Coste total=53.191+10.969+228+1+1+1=**64.391**

En este caso el cálculo teórico es muy similar al cálculo realizado por la herramienta Explain, ya que realiza ordenaciones que reducen el coste de los *join*.

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 (segundos)	Tiempo con integridad (segundos)
Tienda	1,701	1,78
Productos	43,196	14,416
Tienda_Productos	301	1904
Trabajador	40,309	57,452
Ticket	78	263
Ticket_Productos	350	2397

Antes de introducir los datos en la tabla, desactivaremos la integridad funcional con las siguientes funciones:

```
ALTER TABLE "Tienda" DISABLE TRIGGER ALL
ALTER TABLE "Productos" DISABLE TRIGGER ALL
ALTER TABLE "Tienda_Productos" DISABLE TRIGGER ALL
ALTER TABLE "Trabajador" DISABLE TRIGGER ALL
ALTER TABLE "Ticket" DISABLE TRIGGER ALL
ALTER TABLE "Ticket_Productos" DISABLE TRIGGER ALL
```

Una vez deshabilitada la integridad funcional, cargamos los datos en las tablas sin importar el orden en la que se cargan dichos datos. Para ello utilizaremos las siguientes funciones:

```
Copy "Tienda" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_tienda.txt' Delimiter ';'
Copy "Productos" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_productos.txt' Delimiter ';'
Copy "Tienda_Productos" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_Tienda_Productos.txt' Delimiter ';'
Copy "Trabajador" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_trabajadores.txt' Delimiter ';'
Copy "Ticket" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_ticket.txt' Delimiter ';'
Copy "Ticket_Productos" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_Ticket_Productos.txt' Delimiter ';'
```

Podemos consultar los tiempos de carga en el archivo *log* donde observaremos el tiempo de carga de los datos de cada sentencia.

- **Sentencia Tienda:** 2020-04-07 11:55:23.207 CEST [10880] LOG: sentencia: Copy "Tienda" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_tienda.txt' Delimiter ';'
  - **Duración:** 1726.757 ms
- **Sentencia Productos:** 2020-04-07 11:55:30.497 CEST [10880] LOG: sentencia: Copy "Productos" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_productos.txt' Delimiter ';'
  - **Duración:** 31810.094 ms
- **Sentencia Tienda_Productos:** 2020-04-07 11:56:06.246 CEST [10880] LOG: sentencia: Copy "Tienda_Productos" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_Tienda_Productos.txt' Delimiter ';'
  - **Duración:** 281386.556 ms

- **Sentencia Trabajador:** 2020-04-07 12:00:55.286 CEST [10880] LOG: sentencia: Copy "Trabajador" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_trabajadores.txt' Delimiter ';'
    - **Duración:** 36512.225 ms
- **Sentencia Ticket:** 2020-04-07 12:01:39.542 CEST [10880] LOG: sentencia: Copy "Ticket" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_ticket.txt' Delimiter ';'
    - **Duracion:** 88806.436 ms
- **Sentencia Ticket_Productos:** 2020-04-07 12:03:54.755 CEST [10880] LOG: sentencia: Copy "Ticket_Productos" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_Ticket_Productos.txt' Delimiter ';'
    - **Duración:** 403209.996 ms

A continuación, volvemos a habilitar la integridad referencial para ejecutar la segunda carga de los datos. En este caso el orden sí que es relevante, comenzaremos cargando las tablas Tienda y Productos, ya que no presentan ninguna FK. A continuación, cargamos la tabla Tienda_Productos, que requiere comprobar la PK de Tienda y Productos. Continuamos con Trabajadores que emplea la PK de Tienda. Y finalmente Ticket que requiere de las PK de Trabajadores y Ticket_Productos que necesita de las PK de Ticket y Productos.

```
ALTER TABLE "Tienda" ENABLE TRIGGER ALL
ALTER TABLE "Productos" ENABLE TRIGGER ALL
ALTER TABLE "Tienda_Productos" ENABLE TRIGGER ALL
ALTER TABLE "Trabajador" ENABLE TRIGGER ALL
ALTER TABLE "Ticket" ENABLE TRIGGER ALL
ALTER TABLE "Ticket_Productos" ENABLE TRIGGER ALL
'
Copy "Tienda" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_tienda' Delimiter ';'
Copy "Productos" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_productos' Delimiter ';'
Copy "Tienda_Productos" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_Tienda_Productos' Delimiter ';'
Copy "Trabajador" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_trabajador' Delimiter ';'
Copy "Ticket" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_ticket' Delimiter ';'
Copy "Ticket_Productos" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_Ticket_Productos' Delimiter ';'
```

A continuación, consultamos los tiempos de carga en el archivo *log*.

- **Sentencia Tienda:** 2020-04-07 16:12:22.743 CEST [15312] LOG: sentencia: Copy "Tienda" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_tienda.txt' Delimiter ';'
    - **Duración:** 1040.032 ms
- **Duración: Sentencia Productos:** 2020-04-07 16:12:31.406 CEST [15312] LOG: sentencia: Copy "Productos" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_productos.txt' Delimiter ';'
    - **Duración:** 13852.122 ms
- **Sentencia Tienda_Productos:** 2020-04-07 11:56:06.246 CEST [10880] LOG: sentencia: Copy "Tienda_Productos" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_Tienda_Productos.txt' Delimiter ';'

    - **Duración:** 1900331.213 ms
- **Sentencia Trabajador:** 2020-04-07 12:00:55.286 CEST [10880] LOG: sentencia: Copy "Trabajador" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_trabajadores.txt' Delimiter ';'
    - **Duración:** 56472.754 ms
- **Sentencia Ticket:** 2020-04-07 12:01:39.542 CEST [10880] LOG: sentencia: Copy "Ticket" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_ticket.txt' Delimiter ';'
    - **Duracion:** 263244.077 ms
- **Sentencia Ticket_Productos:** 2020-04-07 18:03:55.022 CEST [15312] LOG: sentencia: Copy "Ticket_Productos" From 'C:\Users\Usuario\Desktop\BDA\Lab\PECL2\Datos Tienda\datos_Ticket_Productos.txt' Delimiter ';'
    - **Duración:** 2396429.197 ms

En conclusión podemos observar que los tiempos de carga son considerablemente menores cuando los cargamos sin la integridad referencial en aquellas tablas con FK que son: Trabajadores, Tickets,Tickets_productos, Tienda_productos. En las que no tiene FK su tiempo de carga es ligeramente inferior debido a que la carga no presenta ninguna variación respecto a la carga sin integridad referencial y además hay datos cargados en caché. Esto se debe a que con la integridad referencial activa PostgreSQL debe comprobar al introducir una FK que existe la PK de ese valor en la tabla correspondiente. Mientras que si está desactivada no se preocupa si el valor de las FK se encuentra en la tabla original.

También podemos observar que los tiempos de carga en el PostgreSQL son muy similares a los del *log,* aunque son un poco menores los de estos últimos, tanto para las cargas con integridad como sin integridad. Esto es debido a que en el log se muestra el tiempo que tarda en ejecutarse nuestra consulta mientras que el tiempo mostrado en el PostgreSQL es el tiempo que tarda hasta que se muestra la salida, lo que incluye otras operaciones que se realizan durante esta consulta como ciertos *debugs*.

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

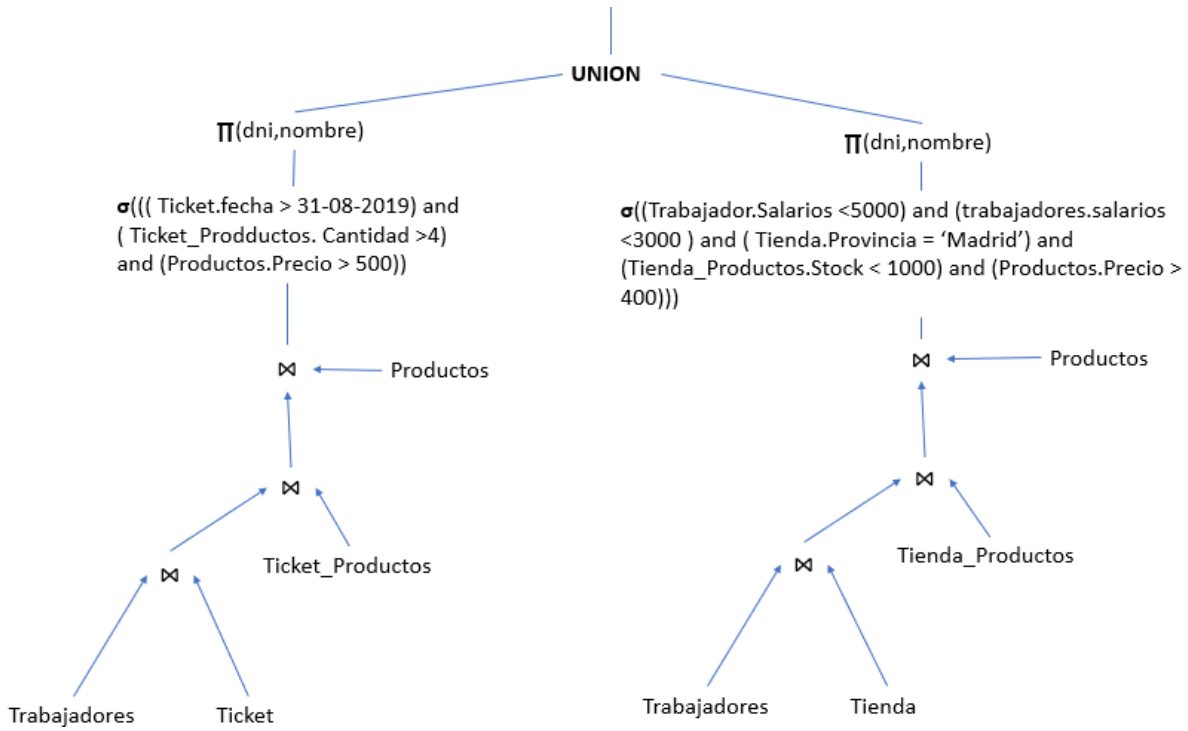
al álgebra relacional y el que obtiene PostgreSQL. Comentar las posibles diferencias entre ambos árboles.

Primero ponemos por defecto los cambios realizados en el archivo postgresql.conf referente a la sección *PLANNER COST CONSTANTS*. Para evitar que el planificador acabe seleccionando rutas de la consulta ineficientes, que incrementen bastante el tiempo de ejecución de la consulta.

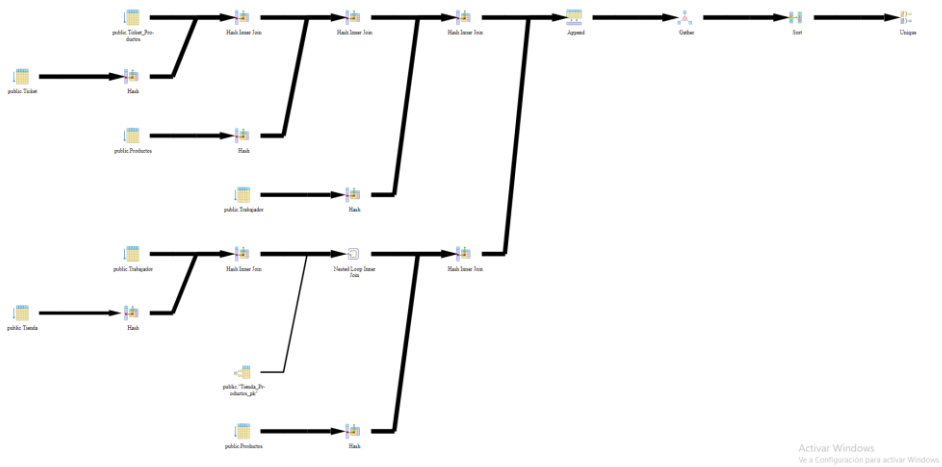
A continuación, creamos la consulta del enunciado

```
SELECT "DNI", "Trabajador"."Nombre" FROM "Trabajador"
INNER JOIN "Ticket" 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 "Productos"."Codigo de barras" = "Ticket_Productos"."Codigo de barras_Productos"
WHERE "Ticket".fecha > '31-08-2019' AND "Ticket_Productos"."Cantidad" > 4 AND "Productos"."Precio" > 500

union

SELECT "DNI", "Trabajador"."Nombre" 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 ("Trabajador"."Salario" between 3000 and 5000 )
and ("Tienda"."Provincia" = 'Madrid') and "Tienda_Productos"."Stock"<1000 and "Productos"."Precio" > 400
```

Al traducir dicha consulta al algebra relacional, obtenemos este árbol (no optimizado heurísticamente):

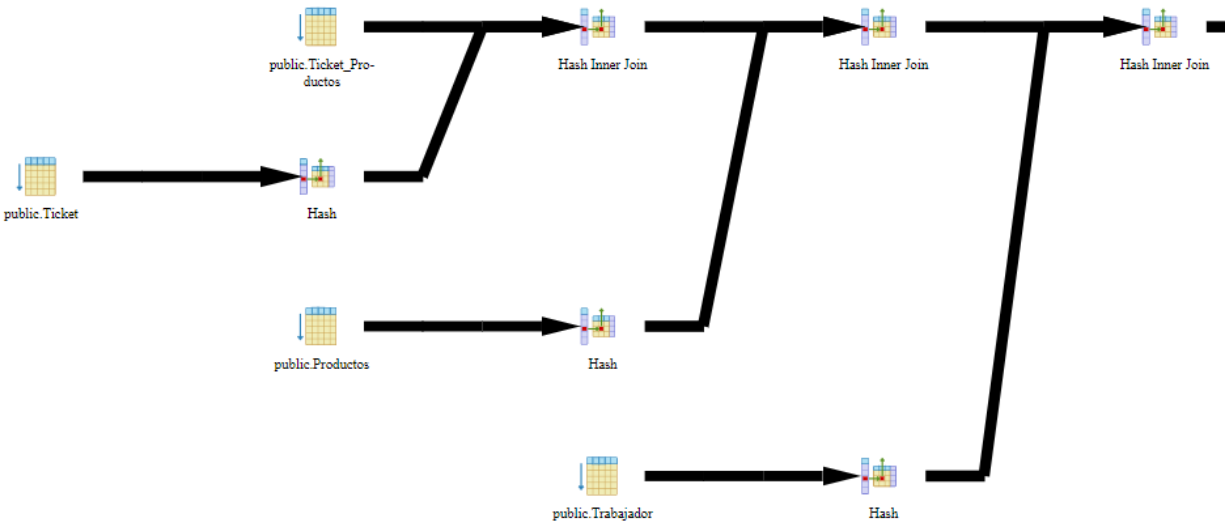


Ahora vamos a aplicar el comando EXPLAIN para obtener el plan de ejecución de PostgreSQL:



Como podemos observar la consulta es de un gran tamaño así que vamos a analizarla paso a paso:

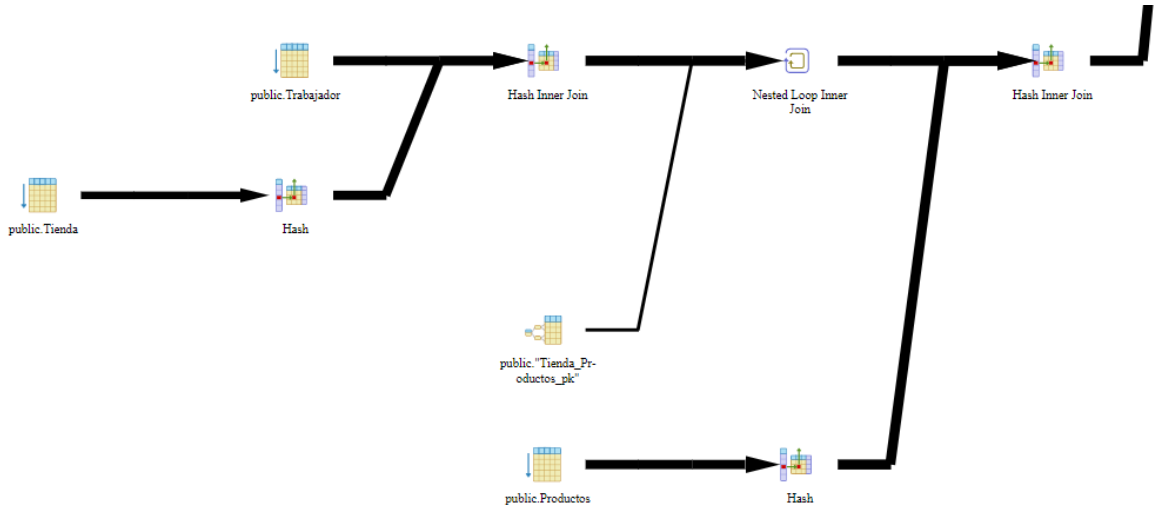
Comenzamos por la primera parte del *Union*:



En primer lugar, PostgreSQL lee las tablas Ticket y Ticket_Productos secuencialmente y ejecuta un *Hash Inner Join* entre ambas, ya que de esta manera puede encauzar ambas tablas y resulta en un menor coste. Comienza realizando el primer *Join* entre las tablas Ticket y Tickect_Productos, esto se debe a que el *Join* entre ambas tablas será el que dé como resultado el menor número de registros. Luego lee la tabla Productos y realiza el *Hash inner Join* con el resultado del *Hash inner Join* anterior. Y finalmente un *Hash Inner Join* con la tabla Trabajador.

Para seleccionar este orden, PostgreSQL analiza todas las posibles maneras de realizar los *Joins* y escoge los que obtengan un menor número de registros.

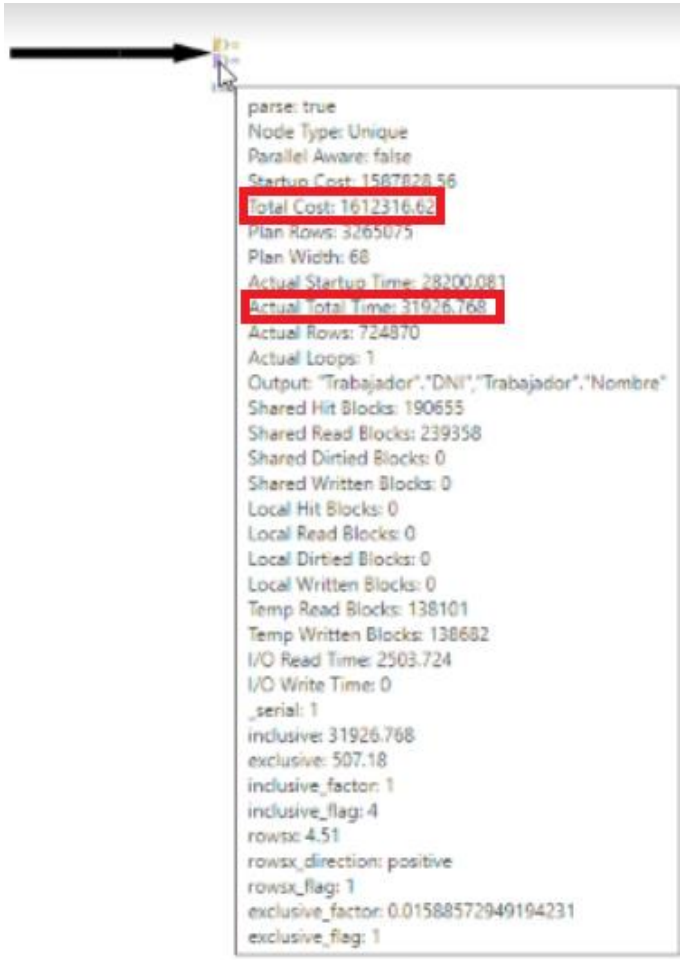
Ahora pasamos con la segunda parte del *Union*:



Observamos que el proceso es muy similar.

Comienza leyendo las tablas Tienda y Trabajador secuencialmente y realizando un *Hash Inner Join* entre ambas. A continuación, lee la tabla Tienda_Productos empleando como índice la PK de la propia tabla y finalmente lee la tabla Productos secuencialmente y realiza el *Hash Inner Join* con el resultado de los *Join* anteriores.

Finalmente realiza el UNION entre ambas consultas, recolecta la información de ambas consultas, las ordena y finalmente elimina aquellas que estén repetidas.



El coste total es de **1.612.316.62** bloques y tarda **31926.768** ms.

Las principales diferencias respecto a nuestro árbol es que: difiere en el orden de los *Joins*, ya que PostgreSQL compara todas las posibilidades para realizar la óptima. Y la parte de las proyecciones también las baja lo máximo posible para realizarlas lo antes posible ya que esto genera una reducción en el coste total de la consulta.

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.

Una buena manera de mejorar el rendimiento sería crear índices sobre todos los campos sobre los que se aplica una condición *Where*, de esta manera reduciríamos el coste de recuperar las tuplas que cumplan la condición, ya que emplearía un índice en vez de una búsqueda secuencial.

Comenzamos realizando índices Hash sobre los campos PK y a las FK de las tablas que utilizaremos para realizar la consulta:

```
Create index "indice_Ticket_NumeroH" ON "Ticket" using hash ("Nº de tickect")
Create index "indice_Productos_CodigoH" ON "Productos" using hash ("Codigo de barras")
Create index "indice_Tienda_Id_tiendaH" ON "Tienda" using hash ("Id_tienda")
Create index "indice_Trabajador_CodigoH" ON "Trabajador" using hash ("codigo_trabajador")
```

```
Create index "indice_Ticket_CodH" ON "Ticket" using hash ("codigo_trabajador_Trabajador")
Create index "indice_Ticket_ProductosCodH" ON "Ticket_Productos" using hash ("Codigo de barras_Productos")
Create index "indice_Ticket_ProductosNumH" ON "Ticket_Productos" using hash ("Nº de tickect_Ticket")
Create index "indice_Tienda_ProductosCodH" ON "Tienda_Productos" using hash ("Codigo de barras_Productos")
Create index "indice_Tienda_ProductosIdH" ON "Tienda_Productos" using hash ("Id_tienda_Tienda")
```

E índices *btree* multicampos para las tablas que presentan varias PK, como son las tablas Tienda_Productos y Ticket_Productos.

```
Create index "indice_Tienda_Productos_allB" ON "Tienda_Productos" using btree ("Id_tienda_Tienda","Stock","Codigo de barras_Productos")
Create index "indice_Ticket_Productos_allB" ON "Ticket_Productos" using btree ("Nº de tickect_Ticket","Codigo de barras_Productos","Cantidad")
```

A continuación, realizaremos índices tipo *btree* para aquellas tablas para las que haya que recuperar registros en un rango de valores:

```
Create index "indice_Productos_Precio_CodigoB" ON "Productos" using btree ("Precio","Codigo de barras")
Create index "indice_Tienda_allB" ON "Tienda" using btree ("Id_tienda","Nombre","Ciudad","Barrio","Provincia")
```

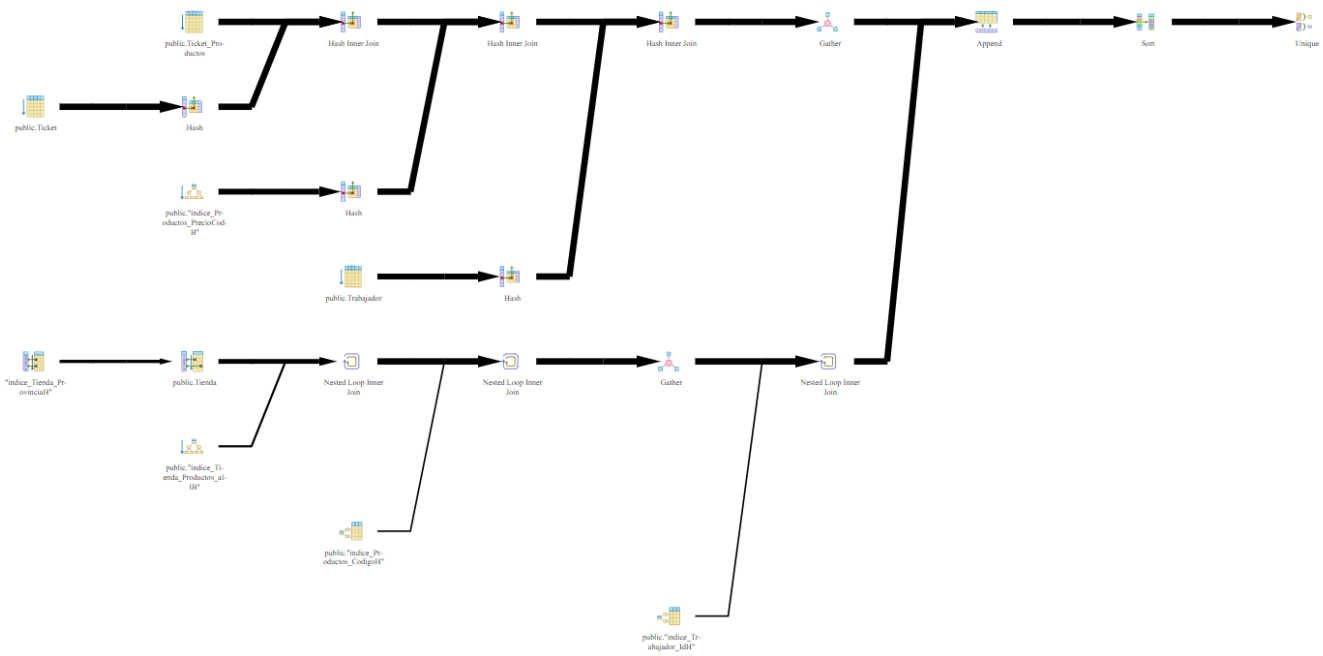
Y finalmente índices hash para aquellos campos en las que haya que recuperar valores concretos o un número muy reducido de valores:

```
Create index "indice_Trabajador_IdH" ON "Trabajador" using hash ("Id_tienda_Tienda")
Create index "indice_Productos_PrecioH" ON "Productos" using hash ("Precio")
Create index "indice_Productos_codigoH" ON "Productos" using hash ("Codigo de barras")
Create index "indice_Tienda_ProvinciaH" ON "Tienda" using hash ("Provincia")
```

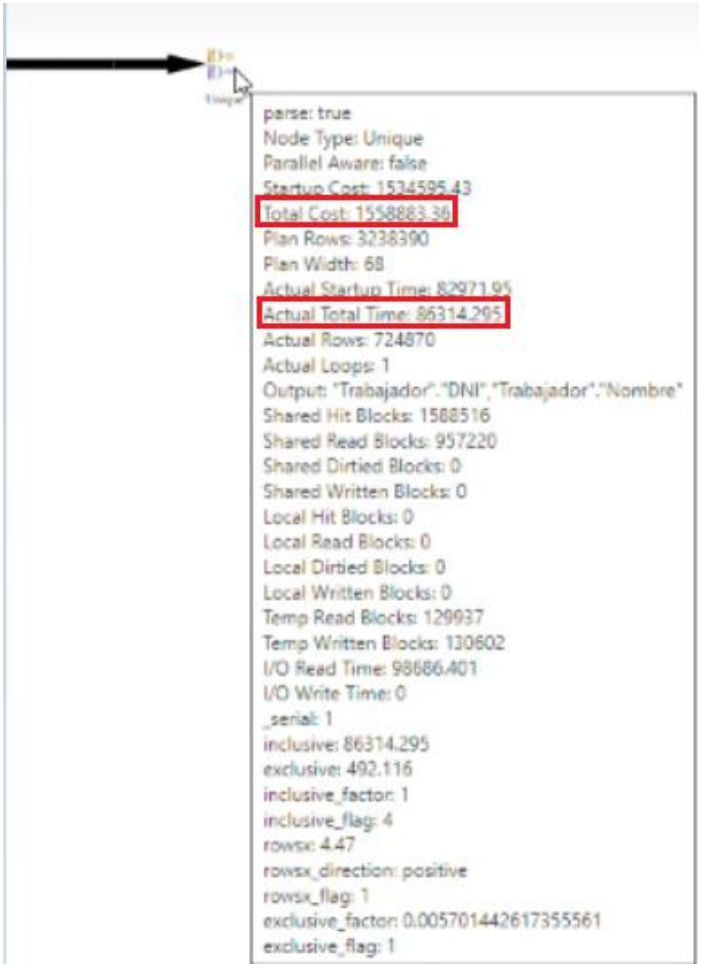
```
Create index "indice_Trabajador_CodoIdB" ON "Trabajador" using btree ("codigo_trabajador","Id_tienda_Tienda")
Create index "indice_Productos_PrecioCodB" ON "Productos" using btree ("Precio","Codigo de barras")
```

(el orden de creación de índices no influye en el resultado final)

A continuación, ejecutamos el plan de ejecución y obtenemos los siguientes resultados:



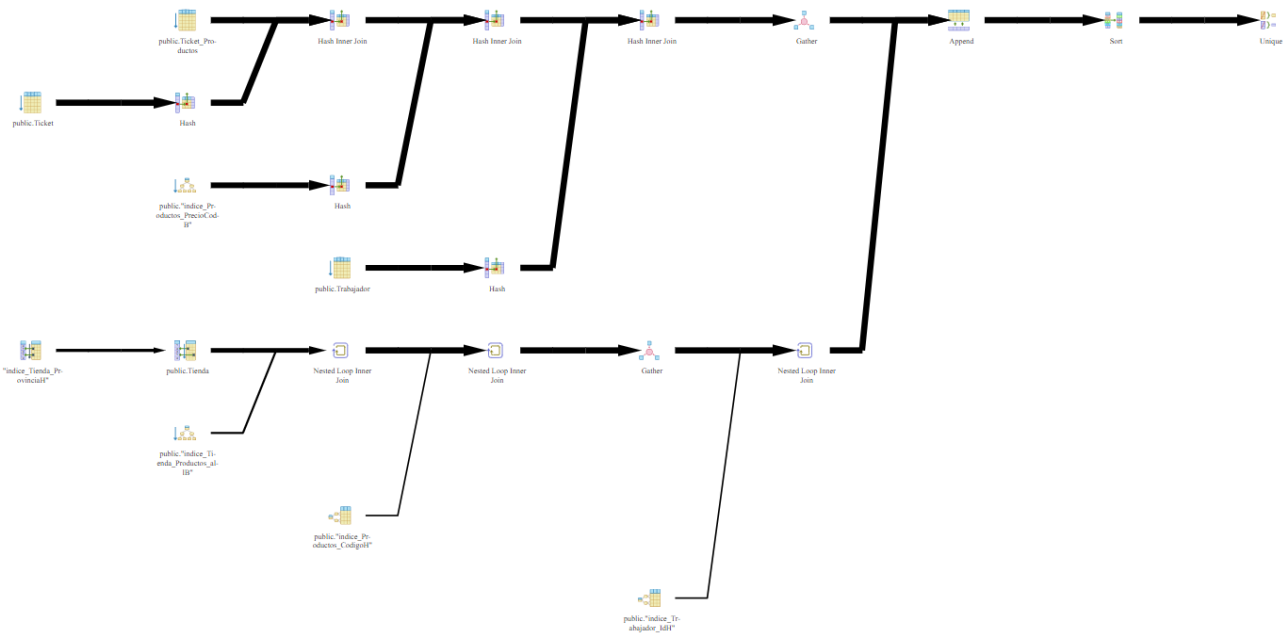
Podemos ver que ha empezado a emplear algunos de los índices creados para realizar la consulta debido a que le resulta óptimo y reduce el coste. Sin embargo, no emplea todos los índices creados, esto se debe a que en algunos casos el coste de leer el índice y luego recuperar los datos es superior al coste de simplemente recuperar los datos directamente sin emplear índices.



Como podemos observar el coste se ha reducido ligeramente, ha pasado a ser de **1.558.883,36** bloques. Pero en cambio, el tiempo se ha reducido drásticamente, ha pasado a ser de tan solo **86.314** ms**.**

Otra buena manera de reducir los costes y el tiempo de ejecución es incrementar los recursos que hardware que utiliza PostgreSQL. Para ello modificaremos el archivo postgresql.conf de la siguiente manera:

- **shared_buffers:** shared_buffers = 128 MB
  a shared_buffers = **4 GB**
- **effective_cache_size**: effective_cache_size = 4GB
  a effective_cache_size = **8 GB**



Como podemos observar, el árbol de ejecución es el mismo que en el caso anterior. El coste prácticamente no se ha reducido y pasa de **1.558.883,36** a **1.560.572,67.** En cambio el tiempo se ha reducido prácticamente un tercio a **29.681,493 ms.**Esto se debe a que al poseer PostgreSQL una mayor capacidad de memoria puede almacenar una mayor cantidad de datos en esta y por lo tanto tiene que gastar menos al no tener que materializar tantos datos en el disco.

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.

Para borrar las tuplas de la tabla Tiendas vamos a ejecutar un DELETE en Cascada. Para ello en primer lugar vamos a convertir los campos que estén en RESTRINCT a CASCADE:

```
ALTER TABLE "Trabajador"
drop CONSTRAINT "Tienda_fk"

ALTER TABLE "Trabajador"
  ADD CONSTRAINT "Tienda_fk"
  FOREIGN KEY ("Id_tienda_Tienda")
  REFERENCES "Tienda"("Id_tienda")
  ON DELETE CASCADE;


ALTER TABLE "Ticket"
drop CONSTRAINT "Trabajador_fk"

ALTER TABLE "Ticket"
  ADD CONSTRAINT "Trabajador_fk"
  FOREIGN KEY ("codigo_trabajador_Trabajador")
  REFERENCES "Trabajador"("codigo_trabajador")
  ON DELETE CASCADE;
```

En segundo lugar, debemos crear índices que faciliten esta operación. Para ello vamos a realizar índices *btrees* en las FK y las PK de los campos que se van a tener que relacionar para el borrado en cascada.

```
Create index "indice_Tienda_Productos_idB" ON "Tienda_Productos" using btree ("Id_tienda_Tienda")
Create index "indice_Tienda_idB" ON "Tienda" using btree ("Id_tienda")
Create index "indice_Trabajador_idB" ON "Trabajador" using btree ("Id_tienda_Tienda")
Create index "indice_Ticket_codigoB" ON "Ticket" using btree ("codigo_trabajador_Trabajador")
Create index "indice_Ticket_Productos_numeroB" ON "Ticket_Productos" using btree ("Nº de tickect_Ticket")
```

```
Delete from "Tienda" cascade
where "Id_tienda" in
(Select "Id_tienda" from "Tienda" order by random() limit 100000)

DELETE 100000

Query returned successfully in 4 min 43 secs.
```

Podemos comprobar que funcionó correctamente y eliminó la mitad de los valores de tiendas en la tabla Tienda y en las tablas que dependían de ese campo.

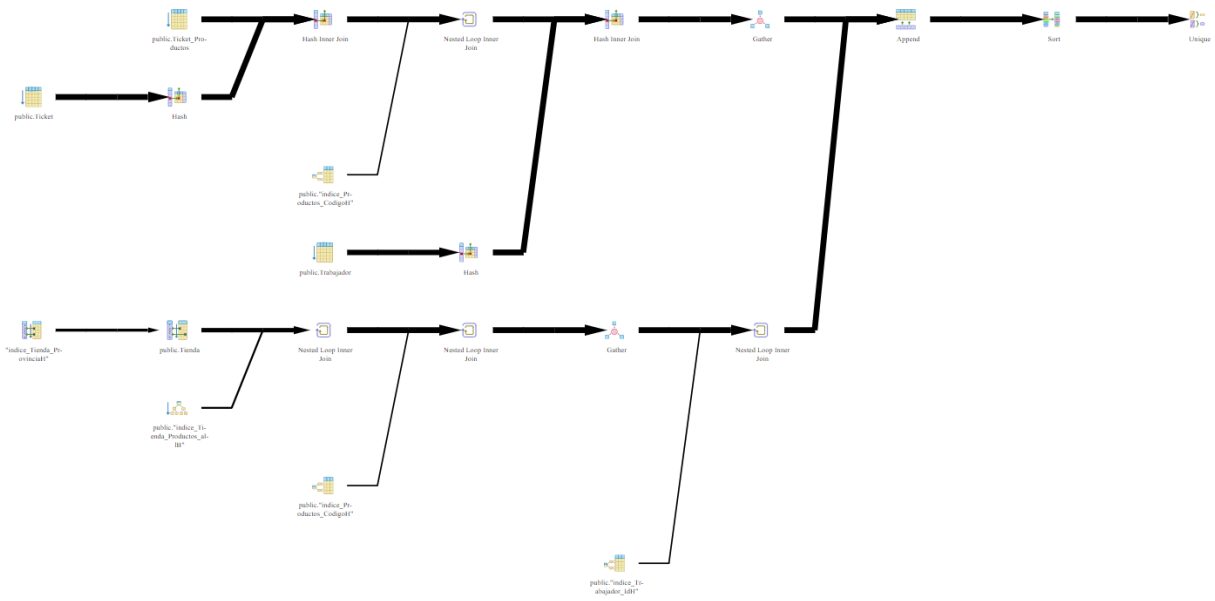
```
Select count ( "Id_tienda")
from "Tienda"
```

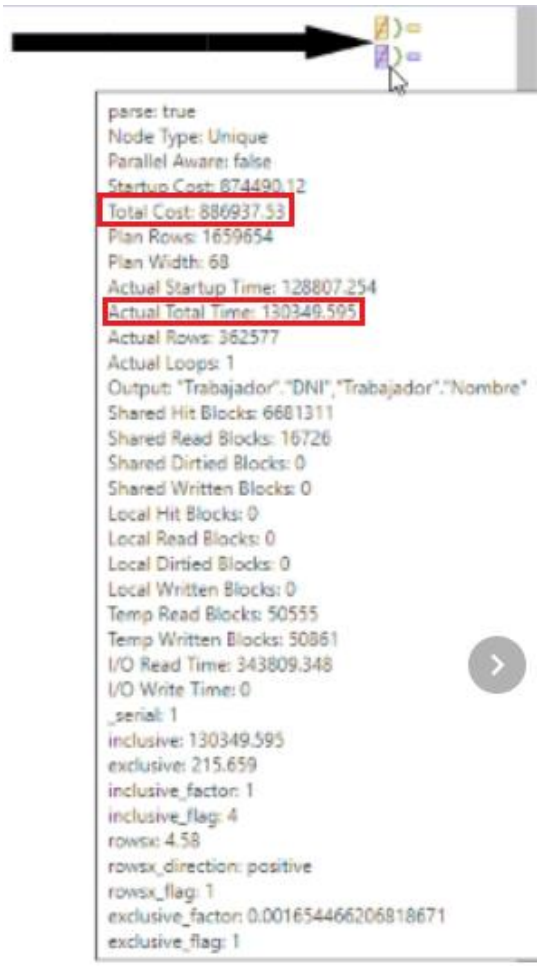
100000

```
Select count (distinct "Id_tienda_Tienda")
from "Tienda_Productos"
```

100000

A continuación, volvemos a ejecutar la consulta y obtenemos el siguiente árbol:





Con un **coste** de 886.937,53 y un tiempo de 130.349,595 **ms.**

Vemos que el árbol es prácticamente idéntico, respecto al coste es bastante menor que los anteriores debido a que el número de tuplas a leer y a enviar es la mitad. En cambio, el coste no se ha reducido, sino que además ha aumentado, esto se debe a que los índices se han quedado desfasados, por lo que ha bajado bastante su rendimiento y por lo tanto su utilidad.

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?

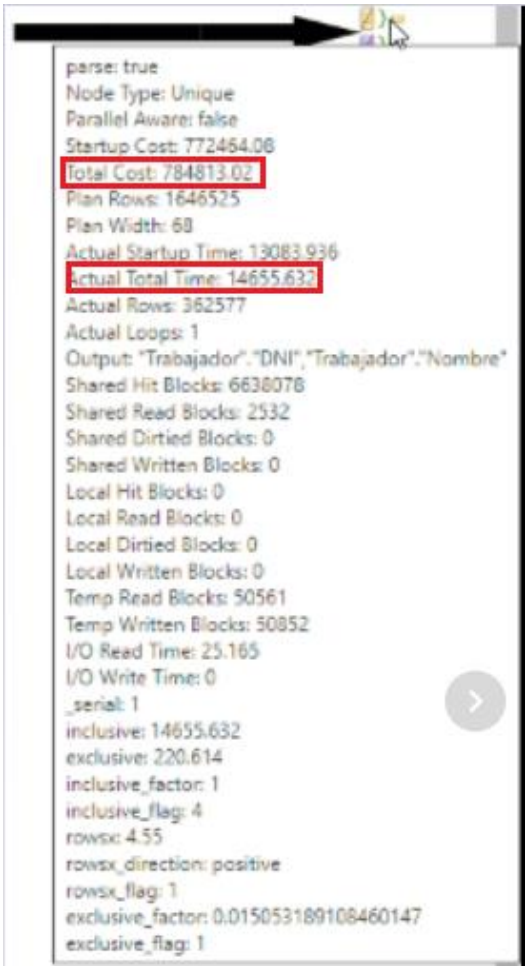
En primer lugar, podríamos emplear el comando VACUUM o VACUUM FULL, este nos permitiría recuperar el espacio tanto de las filas que se hayan marcado para borrado como de aquellas que se hayan actualizado. También actualiza las estadísticas del catálogo del sistema (que son las que emplea el optimizador de consultas de PostgreSQL y si estuviera desfasada el optimizador podría generar consultas que no fuesen lo más eficientes posibles) como el mapa de visibilidad (que contiene el rastro de las páginas que solo contiene tuplas visibles para todas las transacciones activas). Además, también limpia los registros de todas las transacciones que ya hayan sido realizadas.

A continuación del VACUUM sería recomendable emplear el comando ANALYZE para actualizar las estadísticas del catálogo pg_statistics.

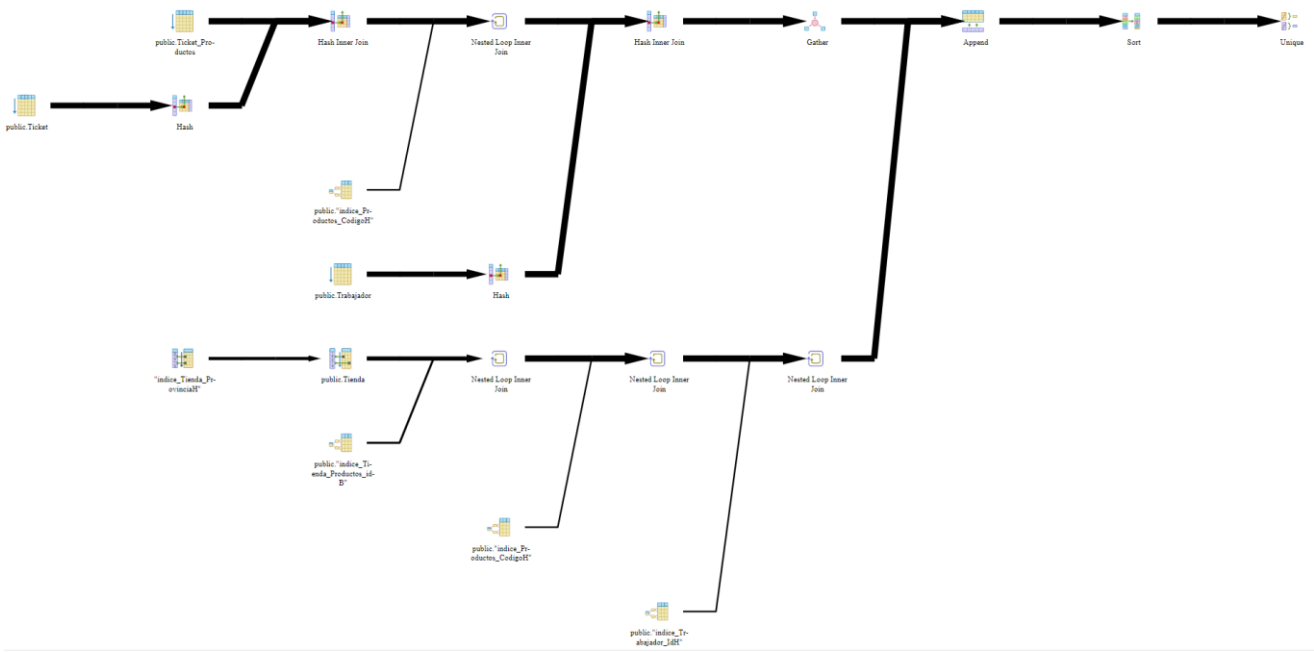
Finalmente, como empleamos índices, podríamos utilizar el comando REINDEX que sirve para reconstruir uno o más índices, reemplazando las versiones anteriores y desactualizadas del índice, que podrían estar ocupando espacio innecesario con bloques vacíos.

```
REINDEX DATABASE "Tienda"
```

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.



Una vez realizado lo primero en lo que nos damos cuenta es el tiempo, que ha bajado drásticamente hasta **14.655 ms,** se ha reducido prácticamente a un décimo**.** Por otra parte, el **coste** se ha reducido ligeramente, ha pasado a ser **784.813,02.**



Fijándonos en el árbol observamos que ha cambiado el índice empleado para la lectura de Tienda_Productos de un *btree* con todos los campos a un *btree* que contiene solo el campo ID. El resto de la consulta se mantiene igual, pero al actualizar los índices, estos son rehechos eliminando las tuplas muertas. Lo que supone en esto caso reducir el número de valores en cada índice alrededor de un 50% y permite encontrar cada valor mucho más rápido, mejorando mucho la eficiencia de todos los índices, lo que repercute de gran manera en el tiempo empleado en la consulta.

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.

Mediante el *log* podemos obtener información sobre las consultas, para ello buscamos pg_stat_database dentro del archivo *log*, ya que cada vez que se realiza una consulta se produce una consulta a este catálogo.

```
    (SELECT count(*) FROM pg_stat_activity WHERE datname = (SELECT datname FROM pg_database WHERE oid = 17717)) AS "Total",
    (SELECT count(*) FROM pg_stat_activity WHERE state = 'active' AND datname = (SELECT datname FROM pg_database WHERE oid = 17717))  AS "Active",
    (SELECT count(*) FROM pg_stat_activity WHERE state = 'idle' AND datname = (SELECT datname FROM pg_database WHERE oid = 17717))  AS "Idle"
) t
UNION ALL
SELECT 'tps_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
    (SELECT sum(xact_commit) + sum(xact_rollback) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 17717)) AS "Transactions",
    (SELECT sum(xact_commit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 17717)) AS "Commits",
    (SELECT sum(xact_rollback) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 17717)) AS "Rollbacks"
) t
UNION ALL
SELECT 'ti_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
    (SELECT sum(tup_inserted) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 17717)) AS "Inserts",
    (SELECT sum(tup_updated) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 17717)) AS "Updates",
    (SELECT sum(tup_deleted) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 17717)) AS "Deletes"
) t
UNION ALL
SELECT 'to_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
    (SELECT sum(tup_fetched) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 17717)) AS "Fetched",
    (SELECT sum(tup_returned) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 17717)) AS "Returned"
) t
UNION ALL
SELECT 'bio_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
    (SELECT sum(blks_read) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 17717)) AS "Reads",
    (SELECT sum(blks_hit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 17717)) AS "Hits"
) t

2020-04-11 10:56:34.088 CEST [22468] LOG:  duración: 25.455 ms
2020-04-11 10:56:35.065 CEST [22468] LOG:  sentencia: /*pga4dash*/
```

Podemos observar en la imagen de arriba la siguiente información:

- Comenzamos obteniendo los guardados de los cambios realizados (Commit) y los cambios deshechos (Rollback).
- A continuación, obtenemos el número de tuplas que han sido insertadas, actualizadas y eliminadas en ese momento, indicado por la fecha y hora.
- También podemos obtener el número de filas recuperadas (F*etched*) y el número de filas devueltas (R*eturned*)
- Por último, obtenemos información de los bloques leídos en disco (*Read)* y RAM (*Hits*)

Respecto a los tiempos medios de las operaciones realizadas en esta práctica obtenidos consulta de cuestiones anteriores observando el *log* son los siguientes:

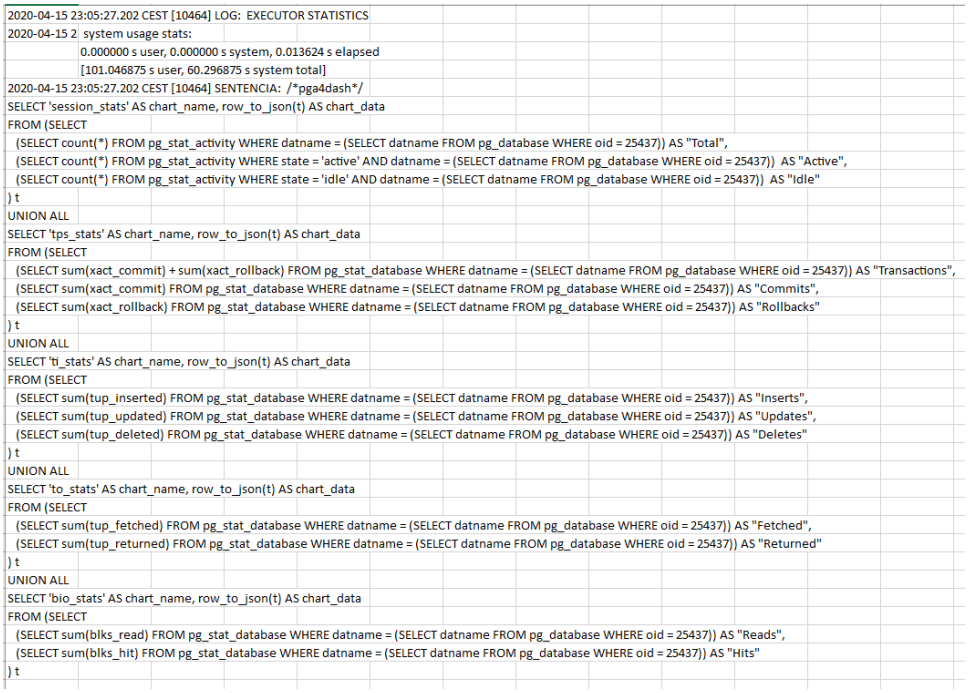
- **Tiempo medio de carga de datos sin integridad referencial:** 140.575,344 ms
- **Tiempo medio de carga de datos con integridad referencial:** 792.672,8992 ms
- **Tiempo de ejecución de la consulta antes del borrado:** 152.364 ms
- **Tiempo de ejecución de la consulta después del borrado:** 21.613 ms

El número de sentencias que ha ejecutado PostgreSQL ha sido de 4.122 realizando un conteo con la herramienta proporcionada por Excel.

Las más empleadas han sido:

Inserts, Up-dates, Deletes que van siempre en fila cada vez que se ejecutan cascada.

Por último, podemos ver en la siguiente foto información de la consulta ejecutada como la fecha y hora de ejecución, la consulta que se realizó, los pasos que sigue PostgreSQL para realizar la consulta y el tiempo de ejecución de dicha consulta.



```
2020-04-15 23:05:27.202 CEST [10464] LOG:  EXECUTOR STATISTICS
2020-04-15 2  system usage stats:
              0.000000 s user, 0.000000 s system, 0.013624 s elapsed
              [101.046875 s user, 60.296875 s system total]
2020-04-15 23:05:27.202 CEST [10464] SENTENCIA:  /*pga4dash*/
SELECT 'session_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
  (SELECT count(*) FROM pg_stat_activity WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25437)) AS "Total",
  (SELECT count(*) FROM pg_stat_activity WHERE state = 'active' AND datname = (SELECT datname FROM pg_database WHERE oid = 25437))  AS "Active",
  (SELECT count(*) FROM pg_stat_activity WHERE state = 'idle' AND datname = (SELECT datname FROM pg_database WHERE oid = 25437))  AS "Idle"
) t
UNION ALL
SELECT 'tps_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
  (SELECT sum(xact_commit) + sum(xact_rollback) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25437)) AS "Transactions",
  (SELECT sum(xact_commit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25437)) AS "Commits",
  (SELECT sum(xact_rollback) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25437)) AS "Rollbacks"
) t
UNION ALL
SELECT 'ti_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
  (SELECT sum(tup_inserted) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25437)) AS "Inserts",
  (SELECT sum(tup_updated) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25437)) AS "Updates",
  (SELECT sum(tup_deleted) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25437)) AS "Deletes"
) t
UNION ALL
SELECT 'to_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
  (SELECT sum(tup_fetched) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25437)) AS "Fetched",
  (SELECT sum(tup_returned) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25437)) AS "Returned"
) t
UNION ALL
SELECT 'bio_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
  (SELECT sum(blks_read) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25437)) AS "Reads",
  (SELECT sum(blks_hit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25437)) AS "Hits"
) t
```

2020-04-15 23:05:27.202 CEST [10464] LOG: duraciÃ³n: 21.613 ms	
2020-04-15 23:05:29.192 CEST [10464] LOG: PARSER STATISTICS	
2020-04-15 2	system usage stats:
	0.000000 s user, 0.000000 s system, 0.000085 s elapsed
	[101.046875 s user, 60.296875 s system total]
2020-04-15 23:05:29.192 CEST [10464] SENTENCIA: /*pga4dash*/	

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 en PostgreSQL funciona de la siguiente manera:

En primer lugar, cuando ejecutamos una consulta PostgreSQL comprueba que dicha consulta es correcta. Si es así, genera el árbol de consultas a partir de la consulta. Tras haber generado el árbol aplica las reglas de optimización heurística para seleccionar el camino óptimo que tenga un menor coste. En este proceso el optimizador evalúa todas alternativas para seleccionar la de mayor eficiencia. A continuación, genera el árbol de planificación junto con los algoritmos que se han utilizado en el plan de ejecución. Para finalizar, interviene el *executor* que se encarga de extraer el conjunto de filas requeridas a partir del árbol de planificación.

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.