

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: Laura Mambrilla Moreno

DNI:

ALUMNO 2:

Nombre y Apellidos: Isabel Blanco Martínez

DNI:

Fecha: 3 de marzo de 2020

Profesor Responsable: Óscar Gutiérrez Blanco

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 Suspense – Cero.

Plazos

Tarea en Laboratorio: Semana 2 de Marzo, Semana 9 de Marzo, Semana 16 de Marzo, semana 23 de Marzo y semana 30 de Marzo.

Entrega de práctica: Semana 26 de Abril (Domingo). 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 ticket en una fecha determinada con los productos que ha comprado el cliente, reflejando el importe total de la compra. Cada tienda tiene registrada los productos que pueden suministrar.

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

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

Actividades y Cuestiones

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

Sí. El colector de estadísticas de PostgreSQL's es un subsistema que admite la recopilación y el informe de información sobre la actividad del servidor.

El colector puede contar accesos a tablas e índices, tanto en términos de bloque de disco como de fila individual. Además, información de cada VACUUM y ANALYZE para cada tabla. También cuenta las funciones definidas por el usuario y el tiempo empleado con cada una.

Dicho recopilador de estadísticas se comunica con los servidores que necesitan información (incluido el VACUUM) a través de archivos temporales. Estos archivos se almacenan en el subdirectorio pg_stat_tmp. Cuando el cerramos PostgreSQL, se almacena una copia permanente de los datos estadísticos en el subdirectorio global.

Para un mejor rendimiento, stats_temp_directory se puede apuntar a un sistema de archivos basado en RAM, lo que disminuye los requisitos físicos de E / S.

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.

1. Accedemos al archivo *postgresql.conf*, que se encuentra en dirección C:\Program Files\PostgreSQL\12\data.
2. En la sección *REPORTING AND LOGGING* podemos ver que *log_min_duration_statement* realiza lo que se nos pide en el enunciado.
3. Cambiamos *log_min_duration_statement* de -1 a 0.

Antes:

```
#log_min_duration_statement = -1          # -1 is disabled, 0 logs all statements
                                           # and their durations, > 0 logs only
                                           # statements running at least this number
                                           # of milliseconds
```

Después:

```
log_min_duration_statement = 0           # -1 is disabled, 0 logs all statements
                                           # and their durations, > 0 logs only
                                           # statements running at least this number
                                           # of milliseconds
```

Cambiando dicha configuración lo que conseguimos es que se modifique el log de errores de todas las operaciones, así como su duración.

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)




```
CREATE TABLE public.empleados(  
    numero_empleado numeric NOT NULL,  
    nombre text NOT NULL,  
    apellidos text NOT NULL,  
    salario numeric NOT NULL,  
  
    CONSTRAINT empleados_pk PRIMARY KEY (numero_empleado)  
);
```

- proyectos(numero_proyecto tipo numeric PRIMARY KEY, nombre tipo text, localización tipo text, coste tipo numeric)

```
CREATE TABLE public.proyectos(  
    numero_proyecto numeric NOT NULL,  
    nombre text NOT NULL,  
    localizacion text NOT NULL,  
    coste numeric NOT NULL,  
  
    CONSTRAINT proyectos_pk PRIMARY KEY (numero_proyecto)  
);
```

- 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.

Para crear esta tabla hemos decidido usar la propia interfaz de pgAdmin. Crearemos una tabla llamada *trabaja_proyectos*. En el apartado de *Columns* podemos ver:

Columns							+
	Name	Data type	Length	Precision	Not NULL?	Primary key?	
	numero_empleado	numeric			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
	numero_proyecto	numeric			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
	horas	numeric			<input checked="" type="checkbox"/>	<input type="checkbox"/>	

Si nos vamos al apartado de *Constraints* podremos ver dos cosas. La información referente a las Primary Key (en este caso *numero_empleado* y *numero_proyecto*):

Primary Key Foreign Key Check Unique Exclude		
	Name	Columns
 	trabaja_proyectos_pkey	numero_empleado,numero_proyecto

Y la información de las Foreign Keys:

Primary Key Foreign Key Check Unique Exclude		
	Name	Columns
 	trabaja_proyectos_numero_empleado_fkey	(numero_empleado) -> (numero_empleado)
 	trabaja_proyectos_numero_proyecto_fkey	(numero_proyecto) -> (numero_proyecto)

Se pide:

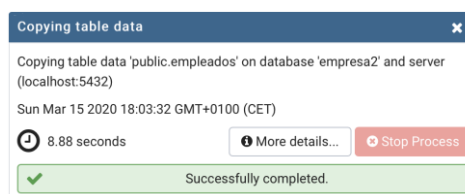
- **Indicar el proceso seguido para generar esta base de datos.**

Para crear esta base de datos hemos creado las tres tablas requeridas, tal y como lo hemos explicado por fotos anteriormente, cargando sus datos con la opción Import en cada tabla.

- **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.**

Realizaremos tres cargas de datos para demostrar cuál es la forma más eficiente para subir los datos de las tablas. La primera, será sin modificar absolutamente nada de lo hecho anteriormente, para ver lo que tarda en completarse la operación.

Tiempo de carga de empleados:



Tiempo de carga de proyectos:



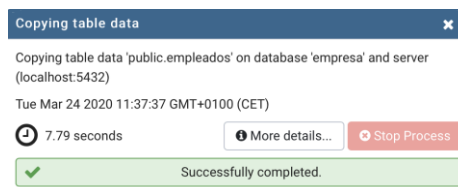
Tiempo de carga de trabaja_proyectos:



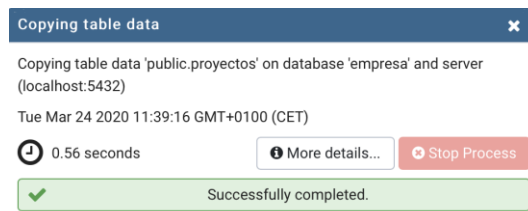
Ahora, para subir los datos de una manera eficiente vamos a eliminar las foreign key constraints, ya que se subirá antes la información en conjunto.

```
ALTER TABLE trabaja_proyectos DROP CONSTRAINT trabaja_proyectos_numero_empleado_fkey;  
ALTER TABLE trabaja_proyectos DROP CONSTRAINT trabaja_proyectos_numero_proyecto_fkey;
```

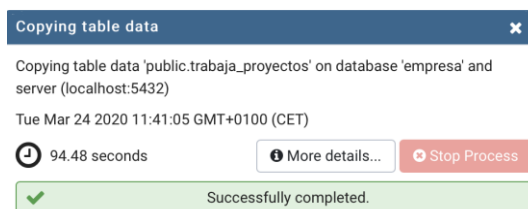
Tiempo de carga empleados: Como podemos ver se reduce un poco el tiempo de carga.



Tiempo de carga de proyectos: El tiempo de carga es el mismo.



Tiempo de carga de trabaja proyectos: Como podemos ver se ha reducido considerablemente, ya que de forma original la operación tardó 442 segundos mientras que ahora ha tardado 94 segundos.



Una vez subidos los datos volvemos a crear las foreign keys constraints.

```
ALTER TABLE trabaja_proyectos ADD CONSTRAINT empleado_fkey FOREIGN KEY (numero_empleado)
REFERENCES public.empleados(numero_empleado) ON DELETE RESTRICT ON UPDATE RESTRICT;
```

```
ALTER TABLE trabaja_proyectos ADD CONSTRAINT proyecto_fkey FOREIGN KEY (numero_proyecto)
REFERENCES public.proyectos(numero_proyecto) ON DELETE RESTRICT ON UPDATE RESTRICT;
```

En la tercera prueba, volveremos a crear las tablas como en el apartado anterior pero en este caso vamos a modificar el archivo de postgresql.conf. Según la documentación, para mejorar la eficiencia a la hora de subir una carga masiva de datos hay que incrementar el valor de *max_wal_size* y el *checkpoint_timeout*. Esto reducirá su ocurrencia minimizando las escrituras de disco. Por lo tanto, si vamos al archivo de configuración encontramos lo siguiente en WRITE-AHEAD LOG:

```
# - Checkpoints -
#checkpoint_timeout = 5min           # range 30s-1d
#max_wal_size = 1GB
#min_wal_size = 80MB
#checkpoint_completion_target = 0.5  # checkpoint target duration, 0.0 - 1.0
#checkpoint_flush_after = 0          # measured in pages, 0 disables
#checkpoint_warning = 30s            # 0 disables
```

Nosotros vamos a probar a descomentar *checkpoint_timeout* e incrementamos *max_wal_size* a 2 GB para comprobar si se producen mejoras. Esto podría variar dependiendo de cada ordenador, pero nosotros probaremos con estos datos. Queda así:

```
# - Checkpoints -
checkpoint_timeout = 5min           # range 30s-1d
max_wal_size = 2GB
min_wal_size = 80MB
#checkpoint_completion_target = 0.5 # checkpoint target duration, 0.0 - 1.0
#checkpoint_flush_after = 0         # measured in pages, 0 disables
#checkpoint_warning = 30s           # 0 disables
```

Probamos a importar de nuevo los datos en cada tabla:

Tiempo de carga empleados: Como podemos ver el tiempo de carga ha aumentado casi el doble.



Tiempo de carga de proyectos: El tiempo de carga también aumenta respecto al original.

Copying table data

Copying table data 'public.proyectos' on database 'empresa' and server (localhost:5432)

Tue Mar 24 2020 12:26:37 GMT+0100 (CET)

0.69 seconds

More details...

Stop Process

Successfully completed.

Tiempo de carga de trabaja proyectos: En comparación con la carga original ha tardado aproximadamente 100 segundos menos. Esto se debe a que al ser una carga masiva de datos hemos mejorado su rendimiento, mientras que para las tablas anteriores al no ser cargas masivas ha provocado que tarden más en completar la importación.

Copying table data

Copying table data 'public.trabaja_proyectos' on database 'empresa' and server (localhost:5432)

Tue Mar 24 2020 12:29:46 GMT+0100 (CET)

352.09 seconds

More details...

Stop Process

Successfully completed.

Como conclusión, aunque también se podrían juntar estos dos métodos, el más eficiente sería eliminar las foreign keys constraints, después introducir los datos y por último añadir de nuevo las constraints.

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?

Para acceder a las estadísticas utilizamos el comando `pg_stats`. Mostraremos las estadísticas de todas las tablas de la base de datos *empresa*.

```
SELECT *
FROM pg_stats
WHERE (tablename = 'empleados') OR (tablename = 'proyectos') OR (tablename = 'trabaja_proyectos');
```

Como resultado de ejecutar dicha consulta obtenemos lo siguiente:

	schema name	tablename	attname	inherited	null_frac	avg_width	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	correlation	most_c	most_e	elem_co
				boolean	real	integer	real	anyarray	real[]	anyarray	real	anyarray	real	real[]
1	public	empleados	numero_em...	false	0	6	-1	[null]	[null]	{130,20937,41518,60851...		1	[null]	[null]
2	public	empleados	nombre	false	0	13	-1	[null]	[null]	{nombre1000054,nombr...	-0.39567083	[null]	[null]	[null]
3	public	empleados	apellidos	false	0	16	-1	[null]	[null]	{apellidos1000054,apelli...	-0.39567083	[null]	[null]	[null]
4	public	empleados	salario	false	0	6	97240	[null]	[null]	{1000.000,2056.000,303...	0.013944376	[null]	[null]	[null]
5	public	proyectos	numero_pro...	false	0	6	-1	[null]	[null]	{8,1082,2093,3112,4050...	1	[null]	[null]	[null]
6	public	proyectos	nombre	false	0	11	-1	[null]	[null]	{nombre100,nombre109...	0.8204347	[null]	[null]	[null]
7	public	proyectos	localizacion	false	0	17	-1	[null]	[null]	{localizacion100,localiza...	0.8204347	[null]	[null]	[null]
8	public	proyectos	coste	false	0	6	9828	{11976.00,19966.00,1127...	{0.0003,0.0003,0.0003,0.0003}	{10000.00,10098.00,102...	-0.0018162596	[null]	[null]	[null]
9	public	trabaja_proyect...	numero_em...	false	0	6	-0.16043773	[null]	[null]	{66,18812,38765,60613...	0.006361798	[null]	[null]	[null]
10	public	trabaja_proyect...	numero_pro...	false	0	6	99164	[null]	[null]	{8,1096,2162,3161,4164...	0.0061302097	[null]	[null]	[null]
11	public	trabaja_proyect...	horas	false	0	4	24	{23,20,9,4,19,11,0,10,8,5,15...	{0.0003,0.0003,0.0003,0.0003}	{0.0003,0.0003,0.0003,0.0003}	0.053881817	[null]	[null]	[null]

Tal y como podemos ver en las tablas los resultados de las estadísticas son correctos.

En caso de no ser correctas, deberíamos actualizar cada una de las tablas haciendo uso de ANALYZE, ya que por defecto nuestro archivo de configuración tiene activado el AUTOVACUUM y las estadísticas, aunque se actualizan de forma automática, podrían no estar actualizadas.

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.

1. Accedemos al archivo *postgresql.conf*, que se encuentra en dirección C:\Program Files\PostgreSQL\12\data.
2. En la sección *QUERY TUNING* podemos ver en el apartado de *Planner Cost Constants*, donde se encuentran todos los costes de PostgreSQL.
3. Dejamos *seq_page_cost* con 1.0 y cambiamos *random_page_cost* a 1.0, todos los demás los cambiaremos a 0.0.

Configuración previa:

```
# - Planner Cost Constants -

#seq_page_cost = 1.0                # measured on an arbitrary scale
#random_page_cost = 4.0             # same scale as above
#cpu_tuple_cost = 0.01              # same scale as above
#cpu_index_tuple_cost = 0.005       # same scale as above
#cpu_operator_cost = 0.0025         # same scale as above
#parallel_tuple_cost = 0.1           # same scale as above
#parallel_setup_cost = 1000.0       # same scale as above

#jit_above_cost = 100000            # perform JIT compilation if available
                                   # and query more expensive than this;
                                   # -1 disables
#jit_inline_above_cost = 500000     # inline small functions if query is
                                   # more expensive than this; -1 disables
#jit_optimize_above_cost = 500000   # use expensive JIT optimizations if
                                   # query is more expensive than this;
                                   # -1 disables

#min_parallel_table_scan_size = 8MB
#min_parallel_index_scan_size = 512kB
#effective_cache_size = 4GB
```

Configuración final:

```
# - Planner Cost Constants -

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

jit_above_cost = 0.0               # perform JIT compilation if available
                                   # and query more expensive than this;
                                   # -1 disables
jit_inline_above_cost = 0.0        # inline small functions if query is
                                   # more expensive than this; -1 disables
jit_optimize_above_cost = 0.0      # use expensive JIT optimizations if
                                   # query is more expensive than this;
                                   # -1 disables

#min_parallel_table_scan_size = 8MB
#min_parallel_index_scan_size = 512kB
#effective_cache_size = 4GB
```

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.

Transcribimos la consulta del enunciado, la cual sería la siguiente.

```
EXPLAIN ANALYZE SELECT salario
FROM empleados
WHERE salario > 96000;
```

Al ejecutar la consulta obtenemos:

	QUERY PLAN	
	text	
1	Gather (cost=1000.00..40070.87 rows=99812 width=6) (actual time=0.132..124.915 rows=99523 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Parallel Seq Scan on empleados (cost=0.00..29089.67 rows=41588 width=6) (actual time=0.062..113.175 rows=331...	
5	Filter: (salario > '96000'::numeric)	
6	Rows Removed by Filter: 633492	
7	Planning Time: 0.632 ms	
8	Execution Time: 128.774 ms	

Lo que hace esta consulta es una búsqueda secuencial en empleados, donde la condición es que salario sea mayor que 96000.

Podemos ver que el número de tuplas totales que lee es de 99523 rows. Si calculamos el resultado teóricamente con la fórmula:

$$n_{rc} = n_r * \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$$

$n_r \rightarrow$ número de tuplas totales

$\max(A,r)$ y $\min(A,r)$ lo calculamos con las siguientes consultas:

```
SELECT min(salario)
FROM empleados
```

```
SELECT max(salario)
FROM empleados
```

donde $\min(A,r) \rightarrow 100000$ y $\max(A,r) \rightarrow 100999$

Obteniendo que las tuplas a recuperar serían 99981, se aproxima a 99523, que es el resultado obtenido con PostgreSQL.

Por lo tanto, el resultado mostrado por el comando EXPLAIN es correcto.

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.

Transcribimos la consulta del enunciado, la cual sería la siguiente.

```
EXPLAIN ANALYZE SELECT proyectos.numero_proyecto, nombre, localizacion, coste
FROM proyectos INNER JOIN trabaja_proyectos ON proyectos.numero_proyecto=trabaja_proyectos.numero_proyecto
WHERE trabaja_proyectos.horas = 8;
```

Al ejecutar la consulta obtenemos:

	QUERY PLAN
	text
1	Gather (cost=4957.00..165482.29 rows=421328 width=40) (actual time=70.621..3147.913 rows=416698 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Hash Join (cost=3957.00..122349.49 rows=175553 width=40) (actual time=58.259..3072.111 rows=138899 loops=3)
5	Hash Cond: (trabaja_proyectos.numero_proyecto = proyectos.numero_proyecto)
6	-> Parallel Seq Scan on trabaja_proyectos (cost=0.00..115777.64 rows=175553 width=6) (actual time=14.940..2914.882 rows=138899 loops=3)
7	Filter: (horas = '8'::numeric)
8	Rows Removed by Filter: 3194434
9	-> Hash (cost=1925.00..1925.00 rows=100000 width=40) (actual time=42.689..42.689 rows=100000 loops=3)
10	Buckets: 65536 Batches: 4 Memory Usage: 2347kB
11	-> Seq Scan on proyectos (cost=0.00..1925.00 rows=100000 width=40) (actual time=0.043..13.681 rows=100000 loops=3)
12	Planning Time: 9.863 ms
13	JIT:
14	Functions: 33
15	Options: Inlining false, Optimization false, Expressions true, Deforming true
16	Timing: Generation 5.773 ms, Inlining 0.000 ms, Optimization 3.179 ms, Emission 38.650 ms, Total 47.602 ms
17	Execution Time: 3210.281 ms

Con este resultado vemos que se han leído un total de 416698 tuplas.

Si calculamos el resultado teóricamente con la fórmula:

$$n_{rc} = n_r * \frac{v - \min(A, r)}{\max(A, r) - \min(A, r)}$$

Veríamos que el resultado es $\frac{1}{24}$ de la tabla *trabaja_proyectos*. Esto es así ya que hay 24 horas en un día y nosotros estamos poniendo de condición que horas = 8. El resultado teórico es de 416666 tuplas a recuperar .

El resultado teórico (416666) y el resultado real (416698) son muy próximos, por lo que podemos decir que es correcto el resultado del comando EXPLAIN.

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.

Transcribimos la consulta del enunciado, la cual sería la siguiente.

```
EXPLAIN ANALYZE SELECT proyectos.numero_proyecto, proyectos.nombre, localizacion, coste
FROM proyectos INNER JOIN trabaja_proyectos
ON proyectos.numero_proyecto=trabaja_proyectos.numero_proyecto
INNER JOIN empleados ON trabaja_proyectos.numero_empleado = empleados.numero_empleado
WHERE (trabaja_proyectos.horas < 2 AND empleados.salario = 24000) AND proyectos.coste > 15000;
```

Al ejecutar la consulta obtenemos:

	QUERY PLAN	
	text	
1	Gather (cost=1000.73..30348.42 rows=4 width=40) (actual time=3.992..128.977 rows=2 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Nested Loop (cost=0.73..29348.02 rows=2 width=40) (actual time=50.172..115.892 rows=1 loops=3)	
5	-> Nested Loop (cost=0.43..29346.75 rows=4 width=6) (actual time=24.148..115.212 rows=2 loops=3)	
6	-> Parallel Seq Scan on empleados (cost=0.00..29089.67 rows=9 width=6) (actual time=16.941..109.557 rows=7 loops=3)	
7	Filter: (salario = '24000'::numeric)	
8	Rows Removed by Filter: 666660	
9	-> Index Scan using trabaja_proyectos_pkey on trabaja_proyectos (cost=0.43..28.55 rows=1 width=12) (actual time=0.798..0.802 rows=0 loops=21)	
10	Index Cond: (numero_empleado = empleados.numero_empleado)	
11	Filter: (horas < '2'::numeric)	
12	Rows Removed by Filter: 5	
13	-> Index Scan using proyectos_pk on proyectos (cost=0.29..0.32 rows=1 width=40) (actual time=0.288..0.288 rows=0 loops=7)	
14	Index Cond: (numero_proyecto = trabaja_proyectos.numero_proyecto)	
15	Filter: (coste > '15000'::numeric)	
16	Rows Removed by Filter: 1	
17	Planning Time: 6.011 ms	
18	Execution Time: 129.009 ms	

Con este resultado vemos que se han leído 2 tuplas.

Si calculamos el resultado teóricamente con la fórmula:

$$n_{rc} = n_r * \frac{v - \min(A, r)}{\max(A, r) - \min(A, r)}$$

Vemos que el resultado teórico es de 21 tuplas. Podemos ver que los resultados no son compatibles, por lo que en este caso el resultado mostrado por el comando EXPLAIN podría no ser correcto.

Cuestión 9: Realizar la carga masiva de los datos mencionados en la introducción con la integridad referencial deshabilitada (tomar tiempos) utilizando uno de los mecanismos que proporciona postgresQL. Realizarlo sobre la base de datos suministrada TIENDA. Posteriormente, realizar la carga de los datos con la integridad referencial habilitada (tomar tiempos) utilizando el método propuesto. Especificar el orden de carga de las tablas y explicar el porqué de dicho orden. Comparar los tiempos en ambas situaciones y explicar a qué es debida la diferencia. ¿Existe diferencia entre los tiempos que ha obtenido y los que aparecen en el LOG de operaciones de postgresQL? ¿Por qué?

Para **habilitar** la integridad referencial usaremos una consulta que lo que hace es crearnos una sentencia de *ENABLE* por cada tabla.

```
SELECT 'ALTER TABLE ' || TABLE_NAME || ' ENABLE TRIGGER ALL;' FROM information_schema.tables WHERE table_schema = 'public';
```

Para **deshabilitarlo** usaremos la misma consulta cambiando *ENABLE* por *DISABLE*.

```
SELECT 'ALTER TABLE ' || TABLE_NAME || ' DISABLE TRIGGER ALL;' FROM information_schema.tables WHERE table_schema = 'public';
```

Tabla	Tiempo sin integridad	Tiempo con integridad
Productos	5.41 seconds	4.75 seconds
Tienda	0.81 seconds	0.79 seconds
Trabajador	15.89 seconds	15.5 seconds
Ticket	54.81 seconds	57.84 seconds
Tienda_Productos	415.85 seconds	430.56 seconds
Ticket_Productos	507.8 seconds	899.88 seconds

Hemos seguido dicho orden ya que algunas de las tablas necesitan datos de otras. Por ejemplo, Ticket necesita datos de la tabla Trabajador, mientras que trabajador necesita datos de la tabla Tienda. Ticket_Productos y Tienda_Productos necesitan datos de Ticket, Productos y Tienda. Sin embargo, el orden de insertar Productos y Tienda no hubiese supuesto ningún inconveniente en caso de que se insertase uno antes que el otro o viceversa.

Como podemos apreciar el tiempo empleado en la subida de los datos que hemos generado con Java aumentan considerablemente cuando está la integridad referencial activada. Esto se debe a que cuando insertamos los datos de una tabla que tienen foreign keys de otras tienen que recorrer las otras tablas para comprobar que no se repitan, por lo cual aumenta el trabajo que tiene que hacer la consulta. En el caso de las tablas con la integridad referencial desactivada, lo único que hacen es un simple COPY sin considerar los datos de las otras tablas.

El tiempo que aparece en el LOG de operaciones difiere al tiempo que podemos ver desde pgAdmin. Esto hace que sea mucho más fiable contrastar los datos relacionados con tiempo de ejecución desde el propio archivo de LOG antes que con el tiempo que nos aparece desde la pgAdmin.

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.

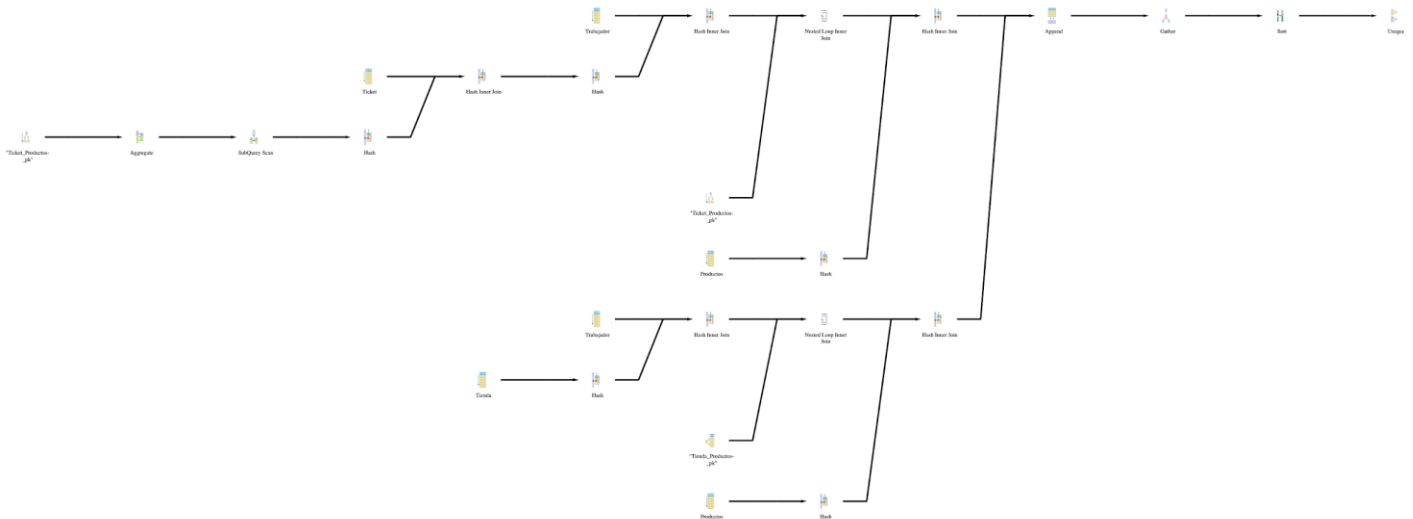
La consulta para obtener lo solicitado por el enunciado es la siguiente:

```
SELECT "Trabajador"."Nombre", "Trabajador"."DNI"
FROM "Trabajador" INNER JOIN "Ticket" ON "Trabajador".codigo_trabajador = "Ticket"."codigo_trabajador_Trabajador"
INNER JOIN "Ticket_Productos" ON "Ticket"."N_de_ticket" = "Ticket_Productos"."N_de_ticket_Ticket"
INNER JOIN "Productos" ON "Ticket_Productos"."Codigo_de_barras_Productos" = "Productos"."Codigo_de_barras"
INNER JOIN
    (SELECT count("Ticket_Productos"."Codigo_de_barras_Productos"), "Ticket_Productos"."N_de_ticket_Ticket"
     FROM "Ticket_Productos"
     GROUP BY "Ticket_Productos"."N_de_ticket_Ticket" HAVING COUNT
("Codigo_de_barras_Productos") > 4 ) AS ticket_productos
    ON "Ticket"."N_de_ticket" = ticket_productos."N_de_ticket_Ticket"
WHERE ("Ticket".fecha between '2019-09-01' and '2019-12-31' AND "Productos"."Precio">500)

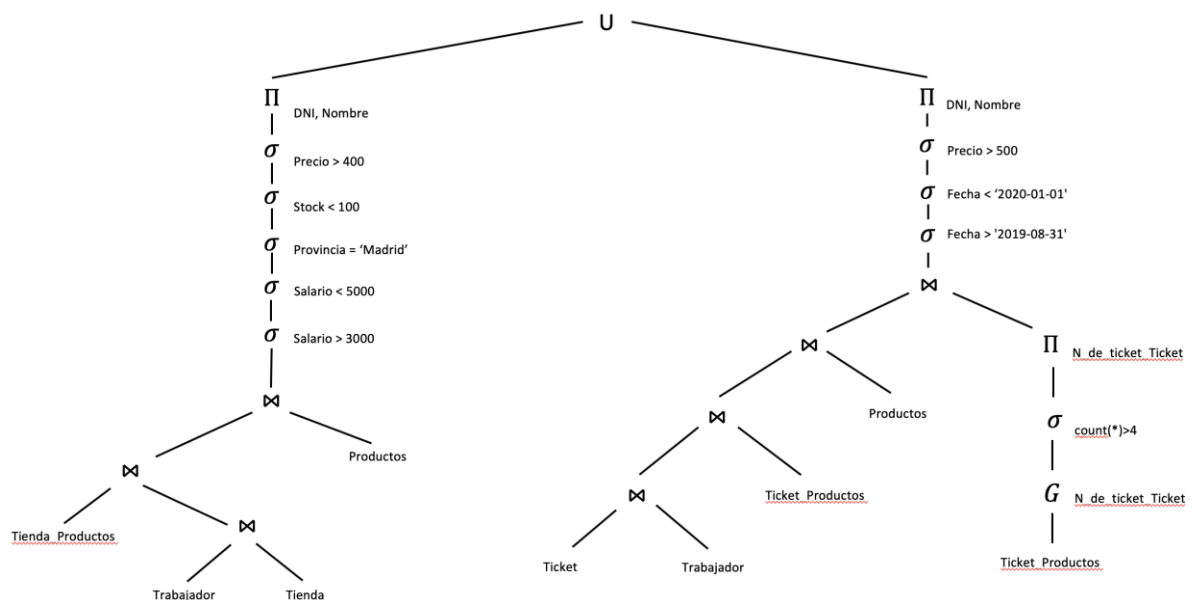
UNION

SELECT "Trabajador"."Nombre", "Trabajador"."DNI"
FROM "Trabajador" INNER JOIN "Tienda" ON "Trabajador"."Id_tienda_Tienda" ="Tienda"."Id_tienda"
INNER JOIN "Tienda_Productos" ON "Tienda"."Id_tienda"="Tienda_Productos"."Id_tienda_Tienda"
INNER JOIN "Productos" ON "Tienda_Productos"."Codigo_de_barras_Productos" = "Productos"."Codigo_de_barras"
WHERE (("Trabajador"."Salario">3000 AND "Trabajador"."Salario"<5000)AND "Tienda"."Provincia"='Madrid' AND
("Tienda_Productos"."Stock"<100) and "Productos"."Precio">400);
```

Obtenemos el árbol de dicha consulta de PostgreSQL:



Haciendo el árbol según lo que hemos visto teóricamente sería lo siguiente:



***Hemos sustituido el between de la fecha de la consulta por '>' y '<' para que quedase mejor expresado en el árbol.**

El árbol consta de dos ramas principales:

La primera rama comienza con Ticket_Productos_pk, seguido de aggregate, subquery scan y hash, este último unido a Ticket en hash inner join. Lo sigue hash y su unión en un hash inner join con Trabajador. Ese resultado se una a Ticket_Productos_pk en un nested loop inner join. Encontramos una sub-rama: Productos, hash; la cual se una con el resultado anterior en un hash inner join, dando fin a la primera rama.

La segunda rama se compone de lo siguiente: Tienda, hash, esto se une con Trabajador en un hash inner join que a su vez se une con Tienda_Productos_pk en un nested loop inner join. Aquí también encontramos una sub-rama (Productos, hash) que se une con lo anterior en un hash inner join.

El árbol finaliza con la unión de las dos ramas en un append, seguido de un gather, de un sort y finalizando con un unique.

Como podemos ver, una de las diferencias más notorias que surgen entre árboles generados por PostgreSQL y el que hacemos al igual que lo aprendido teóricamente es que en el árbol de pgAdmin podemos ver los tipos de nodos que se realiza en cada momento. También podemos ver que PostgreSQL respeta el orden de los INNER JOIN que hemos establecido en la consulta. Entre los nodos utilizados por PostgreSQL se encuentran:

- **Hash aggregate:** Usa una tabla hash temporal para agrupar los registros. La operación HashAggregate no requiere un conjunto de datos preordenado; en su lugar usa una gran cantidad de memoria para materializar el resultado intermedio. La salida no está ordenada.
- **Nested loop join:** Une dos tablas es buscar el resultado desde una tabla y después seleccionar la otra tabla por cada fila de la primera.
- **Merge join:** Merge join (ordenado) combina dos listas ordenadas como una cremallera de pantalón. Ambos lados de la unión debe ser preordenados.
- **Hash Join:** La unión hash carga los registros candidatos desde un lado de la unión, dentro de la tabla hash (marcado con Hash dentro del plan), los cuales se prueban, para cada registro, contra el otro lado de la unión.
- **Subquery Scan:** Se utiliza un operador Subquery Scan para satisfacer una cláusula UNION.
- **Group Aggregate:** Agrega un conjunto preordenado de acuerdo con la cláusula group by. La operación no pone en memoria una gran cantidad de datos (en pipeline).

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 crearemos una serie de índices que optimizarán la búsqueda. En este caso, lo indicado sería crear un índice por cada uno de los atributos cuya búsqueda es especificada en la cláusula WHERE. En este caso, si revisamos la consulta, podemos ver que hemos de crear un índice para la fecha del ticket, el precio de los productos, el salario del trabajador, la provincia de la tienda y el stock de los productos en las tiendas.

Dicha consulta sería la siguiente:

```
CREATE INDEX fecha_index ON "Ticket"("fecha");  
CREATE INDEX precio_index ON "Productos"("Precio");  
CREATE INDEX salario_index ON "Trabajador"("Salario");  
CREATE INDEX provincia_index ON "Tienda"("Provincia");  
CREATE INDEX stock_index ON "Tienda_Productos"("Stock");
```

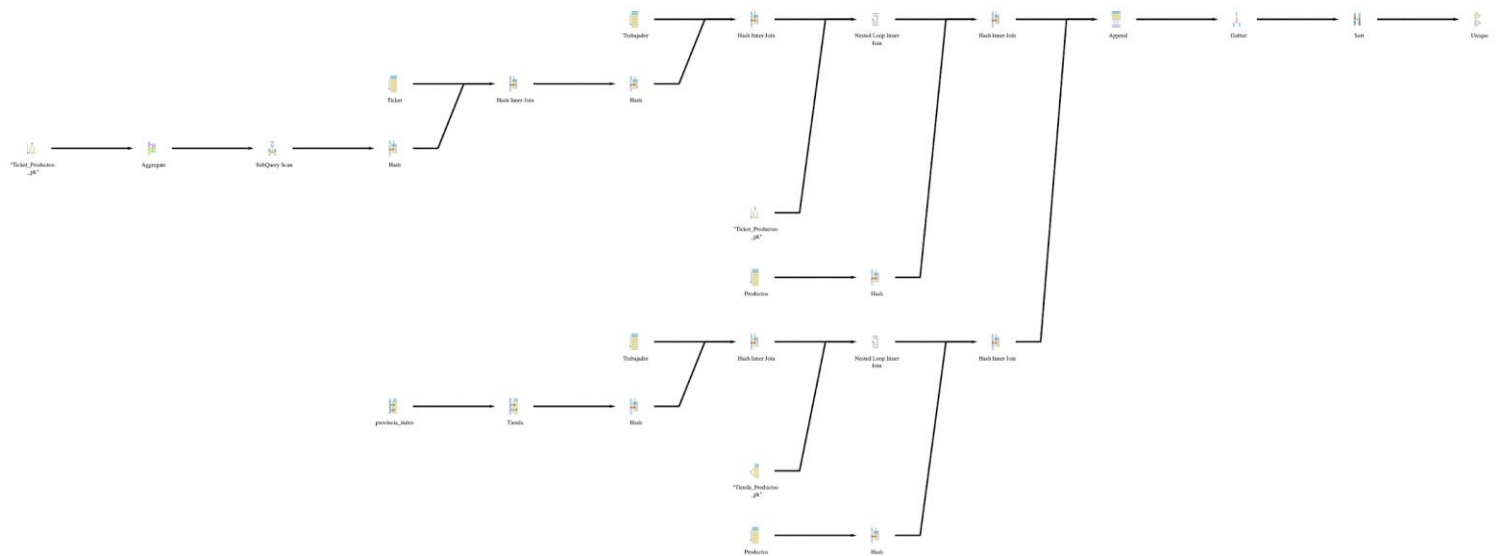
Antes de aplicar las modificaciones podemos comprobar que nuestra consulta sin optimizar tardaba lo siguiente:

```
Successfully run. Total query runtime: 19 secs 37 msec.  
568364 rows affected.
```

Mientras que la consulta una vez aplicadas las modificaciones previamente indicadas tarda lo siguiente:

```
Successfully run. Total query runtime: 13 secs 145 msec.  
568364 rows affected.
```

Obtenemos el árbol de la consulta del ejercicio 10 con las modificaciones aplicadas:



En la primera rama encontramos Ticket_Productos_pk, aggregate, subquery scan, hash y un hash inner join que une lo anterior con Ticket. Le sigue hash y otro hash inner join que lo une con Trabajador. Ticket_Productos_pk se junta con lo previo e un nested loop inner join que, para finalizar la rama, se une a una sub-rama (Productos, hash) en un hash inner join.

La segunda rama: provincia_index, Tienda, hash, un hash inner join que une lo anterior con Trabajador y esto a su vez con Tienda_Productos_pk en un nested loop inner join. También encontramos una sub-rama (Productos, hash) que de une en un hash inner join con lo previo. Y así finaliza la segunda rama.

La unión de las dos ramas se realiza en un append seguido de gather, sort y unique, finalmente.

Como podemos ver la única diferencia que podemos apreciar es que para optimizar el tiempo de la consulta, PostgreSQL utiliza provincia_Index de entre todos los nuevos índices que hemos añadido. Esto debe ser a que ya de por si el tiempo que tarda en ejecutarse la consulta es relativamente corto, por lo que puede que no encuentre necesario hacer uso de los demás índices pues pueden hacer que en vez de reducirse el tiempo este mismo aumentase.

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.

Como tenemos 200.000 registros en la tabla Tienda, sabemos que el 50% son 100.000 tuplas.

Para hacer el borrado más eficiente se crean unos índices btree, ya que será mucho más rápido buscar de esa manera los elementos a eliminar.

```
CREATE INDEX indice_ticket ON "Ticket" USING btree("codigo_trabajador_Trabajador");
CREATE INDEX indice_tickproductos1 ON "Ticket_Productos" USING btree ("Codigo_de_barras_Productos");
CREATE INDEX indice_tickproductos2 ON "Ticket_Productos" USING btree("N_de_ticket_Ticket");
CREATE INDEX indice_tiadaproductos1 ON "Tienda_Productos" USING btree ("Codigo_de_barras_Productos");
CREATE INDEX indice_tiadaproductos2 ON "Tienda_Productos" USING btree ("Id_tienda_Tienda");
CREATE INDEX indice_trabajador ON "Trabajador" USING btree ("Id_tienda_Tienda");
```

Esta consulta ha tardado:

✓ Query returned successfully in 5 min 59 secs.

Después, se realiza el borrado eliminando las relaciones entre las tablas.

```
ALTER TABLE "Trabajador" DROP CONSTRAINT "Tienda_fk";

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

ALTER TABLE "Ticket" DROP CONSTRAINT "Trabajador_fk";

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

DELETE FROM "Tienda" WHERE "Tienda"."Id_tienda" IN (SELECT "Tienda"."Id_tienda" from "Tienda" order by random() limit 100000);
```

Del cual el DELETE ha tardado:

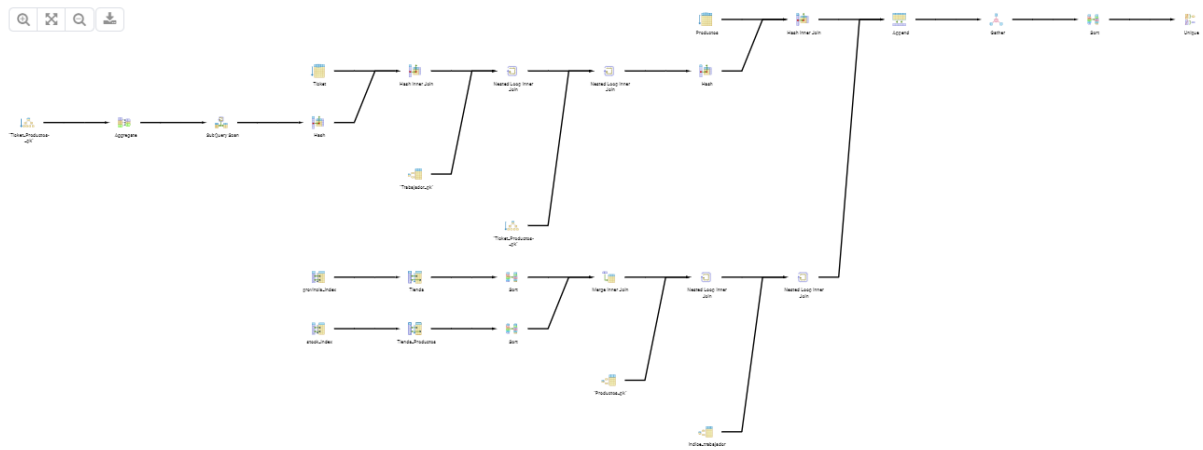
DELETE 100000

Query returned successfully in 8 min 3 secs.

La consulta del ejercicio 10 ahora ha tardado:

✓ Successfully run. Total query runtime: 23 secs 776 msec. 282471 rows affected.

...y el árbol resultante es este:



Como podemos observar, hay dos ramas principales;

La primera rama empieza con Ticket_Productos_pk y continua con aggregate, subquery scan y hash. Este resultado se une con Ticket en un hash inner join, y el resultado de este se une con Trabajador_pk en un nested loop inner join. En un nested loop inner join se unen Ticket_Productos_pk y el resultado de lo anterior. Esta rama termina con un hash inner join que une Productos y lo previo.

La segunda rama se divide al principio en dos sub-ramas. La primera de ellas comienza con provincia_index y le siguen Tienda, sort. La segunda tiene la siguiente secuencia: stock_index, Tienda_Productos y sort. Las sub-ramas se unen con un merge inner join, uniéndose a Productos_pk con un nested loop inner join. La rama acaba con la unión de lo anterior con indice_Trabajador con un nested loop inner join.

Ambas ramas se juntan con un append. El árbol acaba con un gather, un sort y un unique.

Observamos que este árbol difiere tanto del árbol del ejercicio 10 como el del ejercicio 11, los cuales son explicados en sus respectivos ejercicios previos. Aquí dejamos las comparaciones:

Como podemos ver, la primera rama del árbol del ejercicio 10 es más larga, utilizando Productos y, de nuevo, Ticket_Productos_pk; la del ejercicio 11 vemos que de Productos hace un hash. La segunda rama, en el árbol de este ejercicio observamos que hay dos sub-ramas principales (la primera empieza con provincia_index y la segunda con stock_index, mientras que en el árbol del ejercicio 10 empieza con Tienda (solo hay una rama principal); en el del ejercicio 11 también vemos que de Productos hace un hash.

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?

1. Incrementar los segmentos de control (checkpoint_segments), lo que reduciría su ocurrencia minimizando las estructuras al disco.
2. Ejecutar ANALYZE siempre que haya alterado significativamente la distribución de los datos en una tabla, para mantener actualizadas las estadísticas.
3. REINDEX: En algunas situaciones merece la pena reconstruir los índices periódicamente con el comando REINDEX.

4. AUTOVACCUUM: el proceso realiza una limpieza de tuplas muertas que han sido marcadas como borradas o modificadas, ya que el motor de base de datos no las borra inmediatamente de la parte física para no sobrecargar las operaciones normales.

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. Coméntalos.

1. Checkpoints

Incrementaremos el número de checkpoints desde el archivo de configuración ya que así se nos recomienda en el archivo de LOG. Pasa de tener max_wal_size de 1GB a 2GB

```
# - Checkpoints -
#checkpoint_timeout = 5min           # range 30s-1d
max_wal_size = 2GB
min_wal_size = 80MB
#checkpoint_completion_target = 0.5  # checkpoint target duration, 0.0 - 1.0
#checkpoint_flush_after = 0          # measured in pages, 0 disables
#checkpoint_warning = 30s            # 0 disables
```

2. REINDEX

Haremos el REINDEX de toda la base de datos completa. El comando usado es el siguiente:

```
REINDEX DATABASE tienda;
```

3. VACUUM & ANALYZE

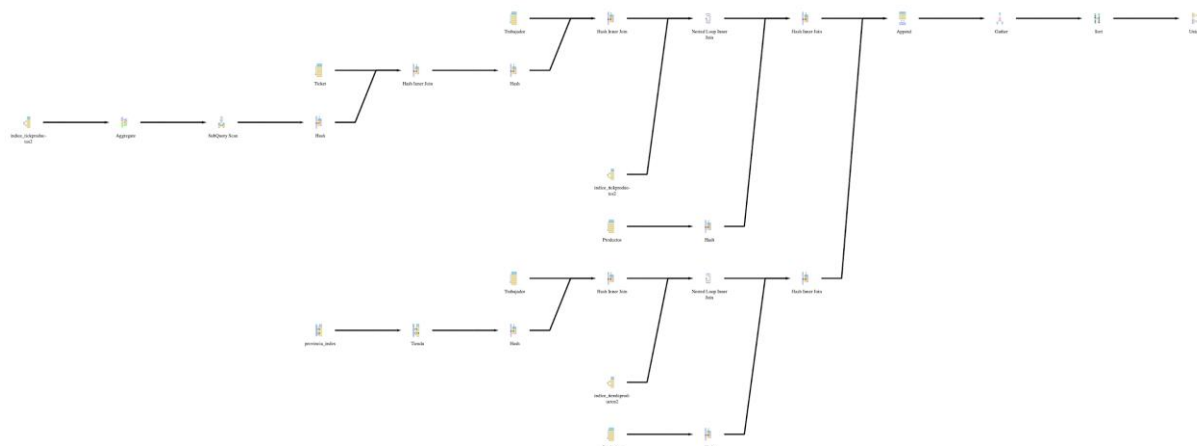
Puesto que autovacuum automatiza la ejecución de los comandos VACUUM y ANALYZE realizaremos una consulta juntando estos dos comandos para asegurarnos de que se realiza el vacuum. Después haremos un VACUUM FULL.

```
VACUUM ANALYZE;
VACUUM FULL;
```

La consulta del ejercicio 10 ahora ha tardado:

```
Successfully run. Total query runtime: 10 secs 520 msec.
284325 rows affected.
```

El árbol de la consulta después de aplicar las técnicas de mantenimiento queda así:



Este árbol tiene dos ramas principales:

La primera de ellas comienza con `indice_tickproductos2`, seguido de `aggregate`, `subquery scan` y `hash`, que se unirá con `Ticket` en un `hash inner join`. Continuamos con otro `hash`, que se une con `Trabajador` en un `hash inner join`. Este resultado se une con `indice_tickproductos2` en un `nested loop inner join`. El resultado de la sub-rama (`Productos`, `hash`) se une con lo anterior en un `hash inner join`. Y hasta aquí la primera rama.

La segunda rama: `provincia_index`, `Tienda` y `hash`, que se une con `Trabajador` en un `hash inner join`, cuyo resultado se une con `indice_tiadaproductos2` en un `need loop inner join`. Este resultado se une con la subrama (`Productos`, `hash`) en un `hash inner join`, finalizando la segunda rama.

Ambas ramas se unen en un `append`. Lo siguen `gather`, `sort` y `unique`, acabando así el árbol.

Como podemos ver, la ejecución de la consulta ha cambiado respecto a las anteriores, viendo la descripción de los árboles anteriores en sus respectivos ejercicios y la comparación de los previo en el ejercicio 12.

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.

Accedemos a la carpeta de PostgreSQL que contiene los log. Dentro, podemos ver que en total hemos realizado un total de 35 consultas. Entre las cuales se encuentran `SELECT`, `EXPLAIN`, `DELETE`, `REINDEX`, `COPY`, `CREATE INDEX`, `ALTER TABLE`, `VACUUM`, etc.

De estas 35 consultas, algunas se han usado más que otras. La que más se ha utilizado ha sido `CREATE INDEX` ya que lo hemos usado reiteradas veces para crear índices con la función de optimizar la base de datos, seguido de `COPY TABLE`, que lo usamos para copiar la información en cada tabla Otro comando que también se ha usado en diversas ocasiones es `SELECT`, ya que hemos tenido que . El número de uso de cada función lo definiremos a continuación.

- ❑ La consulta de **CREATE INDEX** se ha utilizado en 11 ocasiones diferentes. Para calcular el tiempo medio de ejecución, revisamos el log de dichas consultas y calculamos la media:

```
2020-04-26 13:23:05.315 CEST [27586] LOG: duration: 33327.487 ms statement: CREATE INDEX fecha_index ON "Ticket"("fecha");
CREATE INDEX precio_index ON "Productos"("Precio");
CREATE INDEX salario_index ON "Trabajador"("Salario");
CREATE INDEX provincia_index ON "Tienda"("Provincia");
CREATE INDEX stock_index ON "Tienda_Productos"("Stock");
2020-04-26 13:23:06.124 CEST [21157] LOG: duration: 1007.714 ms statement: /*pga4dash*/

2020-04-26 13:26:52.881 CEST [27586] LOG: duration: 101443.329 ms statement: CREATE INDEX indice_ticket ON "Ticket" USING btree("codigo_trabajador_Trabajador");
CREATE INDEX indice_tickproductos1 ON "Ticket_Productos" USING btree ("Codigo_de_barras_Productos");
CREATE INDEX indice_tickproductos2 ON "Ticket_Productos" USING btree("N_de_ticket_Ticket");
CREATE INDEX indice_tiendaproductos1 ON "Tienda_Productos" USING btree ("Codigo_de_barras_Productos");
CREATE INDEX indice_tiendaproductos2 ON "Tienda_Productos" USING btree ("Id_tienda_Tienda");
CREATE INDEX indice_trabajador ON "Trabajador" USING btree ("Id_tienda_Tienda");
```

$$T_{medio} = (33327.487 \text{ ms} + 101443.329 \text{ ms}) / 2 = 67385.409 \text{ ms}$$

- ❑ La consulta de **COPY TABLE** se ha utilizado en 6 ocasiones diferentes. Vemos el tiempo de ejecución de las consultas:

```
2020-04-26 12:35:13.935 CEST [21516] LOG: duration: 21084.450 ms statement: COPY public."Trabajador" ( codigo_trabajador, "DNI" , "Nombre" , "Apellidos" ,
"Puesto" , "Salario" , "Id_tienda_Tienda" ) FROM STDIN DELIMITER ',' CSV QUOTE '"' ESCAPE '\';

2020-04-26 12:34:34.582 CEST [21474] LOG: duration: 382.802 ms statement: COPY public."Tienda" ( "Id_tienda" , "Nombre" , "Ciudad" , "Barrio" , "Provincia" ) FROM
STDIN DELIMITER ',' CSV QUOTE '"' ESCAPE '\';

2020-04-26 12:37:56.495 CEST [21904] LOG: duration: 4306.788 ms statement: COPY public."Productos" ( "Codigo_de_barras" , "Nombre" , "Tipo" , "Descripcion" ,
"Precio" ) FROM STDIN DELIMITER ',' CSV QUOTE '"' ESCAPE '\';

2020-04-26 12:40:55.439 CEST [22074] LOG: duration: 103985.517 ms statement: COPY public."Ticket" ( "N_de_ticket" , "Importe" , fecha,
"codigo_trabajador_Trabajador" ) FROM STDIN DELIMITER ',' CSV QUOTE '"' ESCAPE '\';

2020-04-26 12:50:08.900 CEST [22145] LOG: duration: 625560.003 ms statement: COPY public."Tienda_Productos" ( "Id_tienda_Tienda" , "Codigo_de_barras_Productos" ,
"Stock" ) FROM STDIN DELIMITER ',' CSV QUOTE '"' ESCAPE '\';

2020-04-26 12:56:45.622 CEST [22622] LOG: duration: 799601.788 ms statement: COPY public."Ticket_Productos" ( "Cantidad" , "Codigo_de_barras_Productos" ,
"N_de_ticket_Ticket" ) FROM STDIN DELIMITER ',' CSV QUOTE '"' ESCAPE '\';
```

$$T_{medio} = (382.802 \text{ ms} + 21084.450 \text{ ms} + 4306.788 \text{ ms} + 103985.517 \text{ ms} + 625560.003 \text{ ms} + 799601.788 \text{ ms}) = 125886.5933 \text{ ms}$$

- ❑ La consulta de **ALTER TABLE** se ha utilizado en 6 ocasiones diferentes. Cuando accedemos al log podemos ver el tiempo de ejecución de cada una:

```
2020-04-26 13:42:40.828 CEST [29887] LOG: duration: 4.021 ms statement: ALTER TABLE "Ticket" DROP CONSTRAINT "Trabajador_fk";

2020-04-26 13:28:29.647 CEST [27586] LOG: duration: 14.086 ms statement: ALTER TABLE "Trabajador" DROP CONSTRAINT "Tienda_fk";

2020-04-26 12:57:44.880 CEST [21243] LOG: duration: 2.051 ms statement: select 'ALTER TABLE ' || table_name || ' ENABLE TRIGGER ALL;'
from information_schema.tables
where table_schema = 'public';

2020-04-26 12:38:42.572 CEST [21243] LOG: duration: 3.689 ms statement: select 'ALTER TABLE ' || table_name || ' DISABLE TRIGGER ALL;'
from information_schema.tables
where table_schema = 'public';
```

$$T_{medio} = (9.182 + 4.021 + 8.285 + 14.086 + 2.051 + 3.689) \text{ ms} / 6 = 37.293 \text{ ms}$$

- ❑ La consulta de **SELECT** se ha utilizado en 4 ocasiones diferentes. El tiempo medio de ejecución de una de las consultas es:

```
2020-04-26 13:24:21.499 CEST [21243] LOG: duration: 48402.005 ms statement: SELECT "Trabajador"."Nombre", "Trabajador"."DNI"
FROM "Trabajador" INNER JOIN "Ticket" ON "Trabajador".codigo_trabajador = "Ticket"."codigo_trabajador_Trabajador"
INNER JOIN "Ticket_Productos" ON "Ticket"."N_de_ticket" = "Ticket_Productos"."N_de_ticket_Ticket"
INNER JOIN "Productos" ON "Ticket_Productos"."Codigo_de_barras_Productos" = "Productos"."Codigo_de_barras"
INNER JOIN
(SELECT count("Ticket_Productos"."Codigo_de_barras_Productos"), "Ticket_Productos"."N_de_ticket_Ticket"
FROM "Ticket_Productos"
GROUP BY "Ticket_Productos"."N_de_ticket_Ticket"
HAVING COUNT ("Codigo_de_barras_Productos") > 4 ) AS ticket_productos
ON "Ticket"."N_de_ticket" = ticket_productos."N_de_ticket_Ticket"
WHERE ("Ticket".fecha between '2019-09-01' and '2019-12-31' AND "Productos"."Precio">500)
UNION
SELECT "Trabajador"."Nombre", "Trabajador"."DNI"
FROM "Trabajador" INNER JOIN "Tienda" ON "Trabajador"."Id_tienda_Tienda" = "Tienda"."Id_tienda"
INNER JOIN "Tienda_Productos" ON "Tienda"."Id_tienda"="Tienda_Productos"."Id_tienda_Tienda"
INNER JOIN "Productos" ON "Tienda_Productos"."Codigo_de_barras_Productos" = "Productos"."Codigo_de_barras"
WHERE ((("Trabajador"."Salario">3000 AND "Trabajador"."Salario"<5000)AND "Tienda"."Provincia"='Madrid' AND ("Tienda_Productos"."Stock"<100) and
"Productos"."Precio">400);
```

$$T_{medio} = (48402.005 \text{ ms} + 39226.001 \text{ ms} + 46112.087 \text{ ms} + 45325.071 \text{ ms}) / 4 = 44766.291 \text{ ms}$$

- ❑ La consulta de **EXPLAIN** se ha utilizado en 4 ocasiones diferentes.

$$T_{medio} = (253 \text{ msec} + 66 \text{ msec} + 211 \text{ ms} + 178 \text{ ms}) / 4 = 177 \text{ ms}$$

- ❑ La consulta de **DELETE** se ha utilizado en 1 única ocasión. Accedemos al log para ver su tiempo de ejecución:

```
2020-04-26 13:55:37.218 CEST [29887] LOG: duration: 735294.054 ms statement:
DELETE FROM "Tienda" WHERE "Tienda"."Id_tienda" IN (SELECT "Tienda"."Id_tienda" from "Tienda" order by random() limit 100000);
```

$$\text{Al ser una única consulta, } T_{medio} = 735294.054 \text{ ms.}$$

- ❑ La consulta de **REINDEX** se ha utilizado en 1 única ocasión. Accedemos al log para ver su tiempo de ejecución:

```
2020-04-26 14:01:45.834 CEST [29887] LOG: duration: 122221.476 ms statement: reindex database tienda2;
2020-04-26 14:01:46.436 CEST [29871] LOG: duration: 35.408 ms statement: /*pga4dash*/
```

$$\text{Al ser una única consulta, } T_{medio} = 122221.476 \text{ ms.}$$

- ❑ La consulta de **VACUUM ANALYZE** se ha utilizado en 1 única ocasión.

```
2020-04-26 14:05:18.042 CEST [29887] LOG: duration: 68780.946 ms statement: VACUUM ANALYZE;
```

$$\text{Al ser la única, el tiempo es de } 68780.946 \text{ ms}$$

- ❑ La consulta de **VACUUM FULL** se ha utilizado en 1 única ocasión.

```
2020-04-26 14:08:33.999 CEST [29887] LOG: duration: 176611.090 ms statement: VACUUM FULL;
```

$$\text{Al ser la única, el tiempo es de } 176611.090 \text{ ms}$$

Otro comando que parece digno de destacar ya que aparece constantemente en el LOG, es el siguiente:

```
2020-04-26 14:05:36.565 CEST [29871] LOG: duration: 23.669 ms statement: /*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 = 25191)) AS "Total",
  (SELECT count(*) FROM pg_stat_activity WHERE state = 'active' AND datname = (SELECT datname FROM pg_database WHERE oid = 25191)) AS "Active",
  (SELECT count(*) FROM pg_stat_activity WHERE state = 'idle' AND datname = (SELECT datname FROM pg_database WHERE oid = 25191)) 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 = 25191)) AS "Transactions",
  (SELECT sum(xact_commit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25191)) AS "Commits",
  (SELECT sum(xact_rollback) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25191)) 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 = 25191)) AS "Inserts",
  (SELECT sum(tup_updated) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25191)) AS "Updates",
  (SELECT sum(tup_deleted) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25191)) 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 = 25191)) AS "Fetched",
  (SELECT sum(tup_returned) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25191)) 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 = 25191)) AS "Reads",
  (SELECT sum(blks_hit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 25191)) AS "Hits"
) t
```

Podría ser un comando que PostgreSQL está ejecutando constantemente para mantenerse actualizado.

Y por último, debemos tener en cuenta un comando que nos avisa de que tenemos que aumentar el parámetro `max_wal_size`, el cual por defecto viene predeterminado con 1GB en el archivo de config de PostgreSQL.

```
2020-04-15 15:45:01.129 CEST [300] LOG: checkpoints are occurring too frequently (26 seconds apart)
2020-04-15 15:45:01.129 CEST [300] HINT: Consider increasing the configuration parameter "max_wal_size".
```

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.

Después de la realización de esta práctica, podemos decir que los objetivos del procesamiento de las consultas es transformar una consulta SQL en una estrategia de ejecución eficaz y ejecutar dicha estrategia para recuperar los datos requeridos minimizando el uso de los recursos. En general, hemos visto que no se garantiza que la estrategia elegida sea la óptima, pero sí una estrategia razonablemente eficiente.

Los pasos del procesamiento de una consulta son análisis (léxico, sintáctico y validación), optimización, generación de código y ejecución.

Nos centraremos en los dos primeros:

- **Análisis.**

El análisis léxico identifica los componentes (léxicos) en el texto de la consulta SQL.

El análisis sintáctico revisa la sintaxis de la consulta (corrección gramática).

La validación semántica verifica la validez de los nombres de las tablas, vistas, columnas, etc. y si tienen sentido.

La traducción de la consulta a una representación interna eliminando peculiaridades del lenguaje de alto nivel empleado .

- **Optimización.**

El Optimizador de Consultas suele combinar varias técnicas. Las técnicas principales son las siguientes: optimización heurística y estimación de costes.

- **Optimización heurística**

Ordena las operaciones de la consulta para incrementar la eficiencia de su ejecución.

Aplica reglas de transformación y heurísticas para modificar la representación interna de una consulta (Álgebra Relacional o Árbol de consulta) a fin de mejorar su rendimiento.

Varias expresiones del Álgebra Relacional pueden corresponder a la misma consulta

Lenguajes de consulta como SQL permiten expresar una misma consulta de muchas formas diferentes, pero el rendimiento no debe depender de cómo sea expresada la consulta.

El Analizador Sintáctico genera árbol de consulta inicial (si no hay optimización, la ejecución es ineficiente).

El Optimizador de Consultas transforma el árbol de consulta inicial en árbol de consulta final equivalente y eficiente. Se aplican reglas de transformación guiadas por reglas heurísticas.

Se convierte la consulta en su forma canónica equivalente.

Obtenida la forma canónica de la consulta, el Optimizador decide cómo evaluarla.

- **Estimación de costes**

Estima sistemáticamente el costo de cada estrategia de ejecución y elige la estrategia con el menor costo estimado.

El punto de partida es considerar la consulta como una serie de operaciones elementales interdependientes (join, proyección, restricción, unión, intersección...).

El Optimizador tiene un conjunto de técnicas para realizar cada operación. Por ejemplo, las técnicas para implementar la operación de restricción σ son: búsqueda lineal, búsqueda Binaria, empleo de índice primario o clave de dispersión, empleo de índice de agrupamiento, empleo de índice secundario.

La información estadística que nos proporciona es:

Para cada tabla

- Cardinalidad (n^0 de filas)
- Factor de bloques (n^0 de filas que caben en un bloque)
- N^0 de bloques ocupados

- Método de acceso primario y otras estructuras de acceso (hash, índices, etc.)
- Columnas indexadas, de dispersión, de ordenamiento (físico o no), etc.

Para cada columna

- Nº de valores distintos almacenados,
- Valores máximo y mínimo, etc.

Nº de niveles de cada índice de múltiples niveles

- El éxito de la estimación del tamaño y del coste de las operaciones incluidas en una consulta, depende de la cantidad y actualidad de la información estadística almacenada en el diccionario de datos del SGBD

El optimizador genera varios planes de ejecución, que son combinaciones de técnicas candidatas (una técnica por cada operación elemental de la consulta). Cada técnica tendrá asociada una estimación del coste (número de accesos a bloque de disco necesarios). La estimación precisa de costes es difícil, pues para estimar el nº de accesos a bloque es necesario estimar el tamaño de las tablas (base o generadas como resultados intermedios), lo cual depende de los valores actuales de los datos.

El Optimizador elige el plan de ejecución más económico. Para ello formula una función de coste que se debe minimizar. En general, existen muchos posibles planes de ejecución para una consulta. Se suele hacer uso de técnicas heurísticas para mantener el conjunto de planes de consulta generados dentro de unos límites razonables: reducción del “espacio de evaluación”.

La optimización se puede llevar a cabo de varias maneras, entre ellas:

- a través de índices

Un índice permite al servidor de base de datos encontrar y recuperar filas específicas más rápidamente.

Al crear una PK (al definir un campo como UNIQUE) automáticamente se crea un índice. A parte se pueden crear índices en los campos que sean necesarios para mejorar el rendimiento de las consultas.

- partición de tablas

Es una forma de organizar los datos clasificándolos según criterios de agrupación de manera que cada transacción realizada en una tabla padre se dirija automáticamente a un menor grupo de datos que están agrupados en las tablas hijas, el interés radica a la hora de realizar las consultas con un ahorro significativo en el tiempo de respuesta. Consiste en segmentación de la información mediante criterios. Debe aplicarse cuando existen tablas con gran volumen de datos y para asignar permisos a un grupo de datos específico de una tabla.

Reduce la cantidad de datos a recorrer en cada consulta y aumenta el rendimiento.

Nos aporta ciertas ventajas, así como que trabajamos con segmentos de datos más pequeños, obtenemos índices más pequeños y realizamos los backups más rápidos.

- uso del comando EXPLAIN

Este comando muestra el plan de ejecución que el planificador Postgres genera para la consulta dada. Este muestra la manera en que serán escaneadas las tablas referenciadas; ya sea escaneo secuencial plano, escaneo por índice, etc.

EXPLAIN ANALIZE muestra el plan de ejecución que PostgreSQL genera para la declaración suministrada. El plan de ejecución muestra cómo se escaneará la tabla de referencia en la declaración. Permite visualizar el costo estimado de ejecución de la sentencia, que es la suposición del planificador en el tiempo que se necesita para ejecutar la sentencia.

ANALIZE recoge información sobre el contenido de las tablas de la base de datos y almacena los resultados en la tabla del sistema pg_statistic. El optimizador de consultas utiliza estas estadísticas para determinar los planes de ejecución más eficientes para 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.