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: Patricia Cuesta Ruiz

DNI: 03211093V

ALUMNO 2:

Nombre y Apellidos: Álvaro Golbano Durán

DNI: 03202759D

Fecha: 26/04/2020

Profesor Responsable: Óscar Gutiérrez

Mediante la entrega de este fichero los alumnos aseguran que cumplen con la normativa de autoría de trabajos de la Universidad de Alcalá, y declaran éste como un trabajo original y propio.

En caso de ser detectada copia, se calificará la asignatura como Suspenso – Cero.

Plazos

Tarea en Laboratorio:	Semana 2 de Marzo, Semana 9 de Marzo, Semana 16 de Marzo, semana 23 de Marzo y semana 30 de Marzo.
Entrega de práctica:	Semana 14 de Abril (Martes). Aula Virtual
Documento a entregar:	Este mismo fichero con las respuestas a las cuestiones planteadas y el programa que genera los datos de carga de la base de datos. No se pide el script de carga de los datos de la base de datos. Se entregará en un ZIP comprimido llamado: **DNI'sdelosAlumnos_PECL2.zip**

AMBOS ALUMNOS DEBEN ENTREGAR EL FICHERO EN LA PLATAFORMA.

Introducción

El contenido de esta práctica versa sobre la monitorización de la base de datos, manipulación de datos, técnicas para una correcta gestión de los mismos, así como tareas de mantenimiento relacionadas con el acceso y gestión de los datos. También se trata el tema de procesamiento y optimización de consultas realizadas por PostgreSQL (12.x). Se analizará PostgreSQL en el proceso de carga masiva y optimización de consultas.

En general, la monitorización de la base de datos es de vital importancia para la correcta implantación de una base de datos, y se suele utilizar en distintos entornos:

- Depuración de aplicaciones: Cuando se desarrollan aplicaciones empresariales no se suele acceder a la base de datos a bajo nivel, sino que se utilizan librerías de alto nivel y mapeadores ORM (Hibernate, Spring Data, MyBatis…) que se encargan de crear y ejecutar consultas para que el programador pueda realizar su trabajo más rápido. El problema en estos entornos está en que se pierde el control de qué están haciendo las librerías en la base de datos, cuántas consultas ejecutan, y con qué parámetros, por lo que la monitorización en estos entornos es vital para saber qué consultas se están realizando y poder optimizar la base de datos y los programas en función de los resultados obtenidos.
- Entornos de prueba y test de rendimiento: Cuando una base de datos ha sido diseñada y se le cargan datos de prueba, una de las primeras tareas a realizar es probar que todos los datos que almacenan son consistentes y que las estructuras de datos dan un rendimiento adecuado a la carga esperada. Para ello se desarrollan programas que simulen la ejecución de aquellas consultas que se consideren de interés para evaluar el tiempo que le lleva a la base de datos devolver los resultados, de cara a buscar optimizaciones, tanto en la estructura de la base de datos como en las propias consultas a realizar.
- Monitorización pasiva/activa en producción: Una vez la base de datos ha superado las pruebas y entra en producción, el principal trabajo del administrador de base de datos es mantener la monitorización pasiva de la base de datos. Mediante esta monitorización el administrador verifica que los parámetros de operación de la base de datos se mantienen dentro de lo esperado (pasivo), y en caso de que algún parámetro salga de estos parámetros ejecuta acciones correctoras (reactivo). Así mismo, el administrador puede evaluar nuevas maneras de acceso para mejorar aquellos procesos y tiempos de ejecución que, pese a estar dentro de los parámetros, muestren una desviación tal que puedan suponer un problema en el futuro (activo).

Para la realización de esta práctica será necesario generar una muestra de datos de cierta índole en cuanto a su volumen de datos. Para ello se generarán, dependiendo del modelo de datos suministrado, para una base de datos denominada **TIENDA**. Básicamente, la base de datos guarda información sobre las tiendas que tiene una empresa en funcionamiento en ciertas provincias. La empresa tiene una serie de trabajadores a su cargo y cada trabajador pertenece a una tienda. Los clientes van a las tiendas a realizar compras de los productos que necesitan y son atendidos por un trabajador, el cuál emite un 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?

El recopilador de estadísticas se comunica con los servidores que necesitan información (incluido el autovacuum) a través de archivos temporales. Estos archivos se almacenan en el subdirectorio pg_stat_tmp. Cuando el administrador de la base de datos la cierra, se almacena una copia permanente de los datos estadísticos en el subdirectorio global.

El recopilador de estadísticas transmite la información recopilada a otros procesos de PostgreSQL a través de archivos temporales. Estos archivos se almacenan en el directorio nombrado por el parámetro stats_temp_directory, pg_stat_tmp de forma predeterminada.

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

Cuando el servidor se apaga limpiamente, una copia permanente de los datos estadísticos se almacena en el subdirectorio pg_stat, de modo que las estadísticas se pueden conservar en los reinicios del servidor. Cuando la recuperación se realiza al iniciar el servidor (por ejemplo, después del apagado inmediato, el bloqueo del servidor y la recuperación en un punto en el tiempo), todos los contadores de estadísticas se restablecen.

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 errores hay que acceder a la carpeta en la carpeta de postgreSQL/data y buscar el archivo postgresql.conf y se busca lo siguiente:

```
#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
```

Y se pone ese valor a 0:

```
#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
```

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

- empleados(numero_empleado tipo numeric PRIMARY KEY, nombre tipo text, apellidos tipo text, salario tipo numeric)
- proyectos(numero_proyecto tipo numeric PRIMARY KEY, nombre tipo text, localización tipo text, coste tipo numeric)

- trabaja_proyectos(numero_empleado tipo numeric que sea FOREIGN KEY del campo numero_empleado de la tabla empleados con restricciones de tipo RESTRICT en sus operaciones, numero_proyecto tipo numeric que sea FOREIGN KEY del campo numero_proyecto de la tabla proyectos con restricciones de tipo RESTRICT en sus operaciones, horas de tipo numeric. La PRIMARY KEY debe ser compuesta de numero_empleado y numero_proyecto.

Se pide:

- Indicar el proceso seguido para generar esta base de datos.
- Cargar la información del fichero datos_empleados.csv, datos_proyectos.csv y datos_trabaja_proyectos.csv en dichas tablas de tal manera que sea lo más eficiente posible.
- Indicar los tiempos de carga.

Para crear la base de datos y las tablas de la misma hemos ejecutado el siguiente código:

```
CREATE DATABASE "Empresa"
    WITH
    OWNER = postgres
    ENCODING = 'UTF8'
    LC_COLLATE = 'Spanish_Spain.1252'
    LC_CTYPE = 'Spanish_Spain.1252'
    TABLESPACE = pg_default
    CONNECTION LIMIT = -1;
```

```
CREATE TABLE public.empleado
(
    numero_empleado numeric NOT NULL,
    nombre text COLLATE pg_catalog."default" NOT NULL,
    apellidos text COLLATE pg_catalog."default" NOT NULL,
    salario numeric NOT NULL,
    CONSTRAINT empleado_pkey PRIMARY KEY (numero_empleado)
)
```

```
CREATE TABLE public.proyecto
(
    numero_proyecto numeric NOT NULL,
    nombre text COLLATE pg_catalog."default" NOT NULL,
    localizacion text COLLATE pg_catalog."default" NOT NULL,
    coste numeric NOT NULL,
    CONSTRAINT proyecto_pkey PRIMARY KEY (numero_proyecto)
)
```

```
CREATE TABLE public.trabaja_proyecto
(
    numero_empleado numeric NOT NULL,
    numero_proyecto numeric NOT NULL,
    horas numeric NOT NULL,
    CONSTRAINT trabaja_proyecto_pkey PRIMARY KEY (numero_empleado, numero_proyecto),
    CONSTRAINT numero_empleado FOREIGN KEY (numero_empleado)
        REFERENCES public.empleado (numero_empleado) MATCH SIMPLE
        ON UPDATE RESTRICT
        ON DELETE RESTRICT,
    CONSTRAINT numero_proyecto FOREIGN KEY (numero_proyecto)
        REFERENCES public.proyecto (numero_proyecto) MATCH SIMPLE
        ON UPDATE RESTRICT
        ON DELETE RESTRICT
)
```

```
CREATE TABLE public.trabaja_proyecto
(
    numero_empleado numeric NOT NULL,
    numero_proyecto numeric NOT NULL,
    horas numeric NOT NULL,
    CONSTRAINT trabaja_proyecto_pkey PRIMARY KEY (numero_empleado, numero_proyecto),
    CONSTRAINT numero_empleado FOREIGN KEY (numero_empleado)
        REFERENCES public.empleado (numero_empleado) MATCH SIMPLE
        ON UPDATE RESTRICT
        ON DELETE RESTRICT,
    CONSTRAINT numero_proyecto FOREIGN KEY (numero_proyecto)
        REFERENCES public.proyecto (numero_proyecto) MATCH SIMPLE
        ON UPDATE RESTRICT
        ON DELETE RESTRICT
)
```

Para cargar los datos se ejecutan los siguientes comandos con sus respectivos tiempos de carga:

```
copy empleado from 'D:\postgreSQL\base\datos_empleados.csv' delimiter ','
```

```
COPY 2000000

Query returned successfully in 15 secs 783 msec.
```

```
copy proyecto from 'D:\postgreSQL\base\datos_proyectos.csv' delimiter ','
```

```
COPY 100000

Query returned successfully in 747 msec.
```

```
copy trabaja_proyecto from 'D:\postgreSQL\base\datos_trabaja_proyectos.csv' delimiter ','
```

```
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?

Para poder ver las estadísticas de la tabla empleado:

```
select * from pg_stats where tablename = 'empleado'
```

	schemaname name	tablename name	attname name	inherited boolean	null_frac real	avg_width integer	n_distinct real	most_common_vals anyarray	most_common_freqs real[]
1	public	empleado	numero_em...	false	0	6	-1	[null]	[null]
2	public	empleado	nombre	false	0	13	-1	[null]	[null]
3	public	empleado	apellidos	false	0	16	-1	[null]	[null]
4	public	empleado	salario	false	0	6	96133	[null]	[null]

histogram_bounds anyarray	correlation real	most_common_elems anyarray	most_common_elem_freqs real[]	elem_count_histogram real[]
{130,21312,39214,61518...	1	[null]	[null]	[null]
{nombre1000072,nombr...	-0.39836875	[null]	[null]	[null]
{apellidos1000072,apelli...	-0.39836875	[null]	[null]	[null]
{1003.000,2022.000,299...	0.0018903745	[null]	[null]	[null]

Para poder ver las estadísticas de la tabla proyecto:

```
select * from pg_stats where tablename = 'proyecto'
```

	schemaname name	tablename name	attname name	inherited boolean	null_frac real	avg_width integer	n_distinct real	most_common_vals anyarray	most_common_freqs real[]
1	public	proyecto	numero_pro...	false	0	6	-1	[null]	[null]
2	public	proyecto	nombre	false	0	11	-1	[null]	[null]
3	public	proyecto	localizacion	false	0	17	-1	[null]	[null]
4	public	proyecto	coste	false	0	6	9853	{12555.00,15735.00,1095...	,0.0003,0.0003,0.0003,0.0003}

histogram_bounds anyarray	correlation real	most_common_elems anyarray	most_common_elem_freqs real[]	elem_count_histogram real[]
{1,972,1917,2942,4023,5...	1	[null]	[null]	[null]
{nombre1,nombre10923,...	0.8222929	[null]	[null]	[null]
{localizacion1,localizaci...	0.8222929	[null]	[null]	[null]
{10000.00,10094.00,102...	-0.00084630505	[null]	[null]	[null]

Para poder ver las estadísticas de la tabla proyecto:

```
select * from pg_stats where tablename = 'trabaja_proyecto'
```

	schemaname name	tablename name	attname name	inherited boolean	null_frac real	avg_width integer	n_distinct real	most_common_vals anyarray	most_common_freqs real[]
1	public	trabaja_proyecto	numero_em...	false	0	6	-0.17921287	[null]	[null]
2	public	trabaja_proyecto	numero_pro...	false	0	6	101437	[null]	[null]
3	public	trabaja_proyecto	horas	false	0	4	24	{21,23,11,7,12,9,1,6,3,18,2...	)40466666,0.04,0.039566666}

histogram_bounds anyarray	correlation real	most_common_elems anyarray	most_common_elem_freqs real[]	elem_count_histogram real[]
{78,19130,39204,60818,...	0.0015313218	[null]	[null]	[null]
{1,989,1971,2943,3960,4...	-0.0071529443	[null]	[null]	[null]
[null]	0.055593494	[null]	[null]	[null]

Las estadísticas son correctas, tal y como muestran las tablas. Si estas estadísticas no lo estuvieran, simplemente habría que usar ANALYZE en cada tabla para que las estadísticas de la misma se actualizasen.

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 cambiar el comando explain hay que modificar los siguientes valores:

```
#seq_page_cost = 1.0
#random_page_cost = 4.0
#cpu_tuple_cost = 0.01
#cpu_index_tuple_cost = 0.005
#cpu_operator_cost = 0.0025
#parallel_tuple_cost = 0.1
#parallel_setup_cost = 1000.0
```

Simplemente tuvimos que cambiar las constantes para el cálculo de costes y poner a 1 las que tuvieran que ver con el acceso a bloques y las que no ponerlas a 0, además de descomentar las líneas correspondientes.

```
seq_page_cost = 1.0
random_page_cost = 1.0
cpu_tuple_cost = 1.0
cpu_index_tuple_cost = 0.0
cpu_operator_cost = 0.0
parallel_tuple_cost = 0.0
parallel_setup_cost = 0.0
```

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.

Hemos ejecutado el comando EXPLAIN con la consulta:

```
explain (analyze TRUE, COSTS TRUE, BUFFERS TRUE,FORMAT text)
    select *
    from empleado
    where salario > 96000
```

Obteniendo lo siguiente:

	QUERY PLAN text
1	Gather (cost=0.00..852006.33 rows=98799 width=41) (actual time=1.095..180.227 rows=99523 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	Buffers: shared hit=1216 read=17457
5	-> Parallel Seq Scan on empleado (cost=0.00..852006.33 rows=41166 width=41) (actual time=0.287..129.268 rows=33174 loops=3)
6	Filter: (salario > '96000'::numeric)
7	Rows Removed by Filter: 633492
8	Buffers: shared hit=1216 read=17457
9	Planning Time: 0.093 ms
10	Execution Time: 183.104 ms

Obtenemos que el resultado de la consulta ha sido 99.523. Este resultado es correcto ya que al calcularlo teóricamente con la fórmula:

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

Siendo n_r el número de tuplas totales y obteniendo el máximo y el mínimo con las consultas:

```
select min(salario)
from empleado
```

```
select max(salario)
from empleado
```

Obteniendo que teóricamente las tuplas a recuperar serían 99.981 aproximándose al resultado obtenido con el comando EXPLAIN.

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.

Hemos ejecutado el comando EXPLAIN con la consulta:

```
explain (analyze TRUE, COSTS TRUE, BUFFERS TRUE, FORMAT text)
    select proyecto.*
    from trabaja_proyecto
        inner join proyecto on trabaja_proyecto.numero_proyecto = proyecto.numero_proyecto
    where horas = 8
```

Obteniendo lo siguiente:

	QUERY PLAN text
1	Gather (cost=0.00..4407441.33 rows=423672 width=40) (actual time=0.910..1337.708 rows=416698 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	Buffers: shared hit=1250370 read=63421 dirtied=33695 written=33486
5	-> Nested Loop (cost=0.00..4407441.33 rows=176530 width=40) (actual time=0.514..1218.828 rows=138899 loops=3)
6	Buffers: shared hit=1250370 read=63421 dirtied=33695 written=33486
7	-> Parallel Seq Scan on trabaja_proyecto (cost=0.00..4230412.92 rows=176530 width=6) (actual time=0.313..697.781 rows=138899 loops=3)
8	Filter: (horas = '8'::numeric)
9	Rows Removed by Filter: 3194434
10	Buffers: shared hit=275 read=63420 dirtied=33695 written=33486
11	-> Index Scan using proyecto_pkey on proyecto (cost=0.00..1.00 rows=1 width=40) (actual time=0.003..0.003 rows=1 loops=416698)
12	Index Cond: (numero_proyecto = trabaja_proyecto.numero_proyecto)
13	Buffers: shared hit=1250095 read=1
14	Planning Time: 1.450 ms
15	Execution Time: 1349.437 ms

Con este resultado podemos ver que se han leído 416.698 tuplas, aproximándose al valor correcto calculado de forma teórica ya que si se calcula sale que se obtendrán 1/24 parte de la tabla trabaja_proyecto, la que corresponde a horas = 8 (ya que hay 24h). Siendo el resultado teórico 416.666 tuplas a recuperar muy parecido al resultado real.

Cuestión 8: Aplicar el comando EXPLAIN a una consulta que obtenga la información de los proyectos que tienen un coste mayor de 15000, y tienen empleados de salario de 24000 euros y trabajan menos de 2 horas en ellos. ¿Son correctos los resultados del comando EXPLAIN? ¿Por qué? Comparar con lo que se obtendría con lo visto en teoría.

Hemos ejecutado el comando EXPLAIN con la consulta:

```
explain (analyze TRUE, COSTS TRUE, BUFFERS TRUE, FORMAT text)
    select proyecto.*
    from trabaja_proyecto
        inner join proyecto on proyecto.numero_proyecto = trabaja_proyecto.numero_proyecto
        inner join empleado on trabaja_proyecto.numero_empleado = empleado.numero_empleado
    where coste > 15000 and salario = 24000 and horas < 2
```

Obteniendo lo siguiente:

	QUERY PLAN text
1	Gather (cost=0.00..857622.34 rows=4 width=40) (actual time=6.615..222.717 rows=2 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	Buffers: shared hit=12763 read=16659 written=1
5	-> Nested Loop (cost=0.00..857622.34 rows=2 width=40) (actual time=75.303..181.738 rows=1 loops=3)
6	Buffers: shared hit=12763 read=16659 written=1
7	-> Nested Loop (cost=0.00..857618.33 rows=4 width=6) (actual time=33.817..181.683 rows=2 loops=3)
8	Buffers: shared hit=12740 read=16659 written=1
9	-> Parallel Index Scan using empleado_pkey on empleado (cost=0.00..857492.33 rows=9 width=6) (actual time=33.027..180.339 rows=7 loops=3)
10	Filter: (salario = '24000'::numeric)
11	Rows Removed by Filter: 666660
12	Buffers: shared hit=12716 read=16516 written=1
13	-> Index Scan using trabaja_proyecto_pkey on trabaja_proyecto (cost=0.00..13.00 rows=1 width=12) (actual time=0.182..0.187 rows=0 loops=21)
14	Index Cond: (numero_empleado = empleado.numero_empleado)
15	Filter: (horas < '2'::numeric)
16	Rows Removed by Filter: 5
17	Buffers: shared hit=24 read=143
18	-> Index Scan using proyecto_pkey on proyecto (cost=0.00..1.00 rows=1 width=40) (actual time=0.020..0.020 rows=0 loops=7)
19	Index Cond: (numero_proyecto = trabaja_proyecto.numero_proyecto)
20	Filter: (coste > '15000'::numeric)
21	Rows Removed by Filter: 1
22	Buffers: shared hit=23
23	Planning Time: 1.775 ms
24	Execution Time: 222.750 ms

El resultado obtenido mediante el comando EXPLAIN es de 2 tuplas a recuperar y el calculado teóricamente es de 21. Este valor no cuadra, aunque cabe considerar que una cosa es el cálculo teórico de una consulta y otro es la consulta real de la misma.

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

Tabla	Tiempo sin integridad	Tiempo con integridad
Productos	10 sec	23.08 sec
Tienda	1.62 sec	1.88 sec
Tienda_Productos	1078.20 sec	2400.23 sec
Ticket	37.87 sec	197.95 sec
Trabajadores	16.01 sec	41.16 sec
Ticket_Productos	856.37 sec	2029.54 sec

El orden de carga corresponde al orden de la tabla de arriba, este orden es necesario ya que, si insertamos de otro orden, en el cual las tablas Tienda_Productos y Ticket_Productos fueran antes de las tablas que tienen su PK, saltaría un error por la integridad referencial de la base de datos.

Por supuesto que existe una gran diferencia de tiempos en la carga de los datos ya que al desactivar todos los triggers de las tablas, la integridad referencial ya no se comprueba, reduciendo así los tiempos a prácticamente la mitad o en algunos casos algo más.

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

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

Para esta cuestión hemos ejecutado la siguiente consulta:

```
explain(analyze TRUE, buffers TRUE, format TEXT)
select "Trabajador"."Nombre", "Trabajador"."DNI"
from "Trabajador"
    inner join "Ticket" on "Trabajador"."Codigo_trabajador" = "Ticket"."Codigo_trabajador_Trabajador"
    inner join "Ticket_Productos" on "Ticket"."No_de_ticket" = "Ticket_Productos"."No_de_ticket_Ticket"
    inner join "Productos" on "Ticket_Productos"."Codigo_de_barras_Productos" = "Productos"."Codigo_de_barras"
    inner join "Tienda_Productos" on "Productos"."Codigo_de_barras" = "Tienda_Productos"."Codigo_de_barras_Productos"
    inner join "Tienda" on "Tienda_Productos"."ID_Tienda_Tienda" = "Tienda"."ID_Tienda"
where "Ticket"."Fecha" between '01-09-2019' and '31-12-2019'
    and "Ticket_Productos"."Cantidad" > 4
    and "Productos"."Precio" = any (select "Precio"
                                    from "Productos"
                                    where "Precio" > 500)
    and "Trabajador"."Salario" > 3000 and "Trabajador"."Salario" < 5000
    and "Tienda"."Provincia" = 'Madrid'
    and "Tienda_Productos"."Stock" = any (select "Stock"
             from "Tienda_Productos"
             where "Stock" < 100)
    and "Productos"."Precio" > 400
```

Siendo el resultado:

	QUERY PLAN text
1	Nested Loop Semi Join (cost=425897.67..11501043.00 rows=2 width=12) (actual time=6052.507..6052.507 rows=0 loops=1)
2	Join Filter: ("Tienda_Productos"."Stock" = "Tienda_Productos_1"."Stock")
3	Buffers: shared hit=89 read=9228
4	-> Nested Loop Semi Join (cost=425897.67..3058426.81 rows=2 width=14) (actual time=6052.505..6052.506 rows=0 loops=1)
5	Join Filter: ("Productos"."Precio" = "Productos_1"."Precio")
6	Buffers: shared hit=89 read=9228
7	-> Gather (cost=425897.67..2627186.05 rows=2 width=18) (actual time=6052.503..6073.974 rows=0 loops=1)
8	Workers Planned: 2
9	Workers Launched: 2
10	Buffers: shared hit=89 read=9228
11	-> Nested Loop (cost=425897.67..2627186.05 rows=1 width=18) (actual time=5975.631..5975.631 rows=0 loops=3)
12	Buffers: shared hit=89 read=9228
13	-> Nested Loop (cost=425897.67..2627153.04 rows=33 width=22) (actual time=5975.627..5975.627 rows=0 loops=3)
14	Buffers: shared hit=89 read=9228
15	-> Nested Loop (cost=425897.67..2611103.82 rows=2 width=28) (actual time=5975.627..5975.627 rows=0 loops=3)

```
16                    Buffers: shared hit=89 read=9228
17                    ->  Nested Loop  (cost=425897.67..2611100.82 rows=3 width=18) (actual time=5975.626..5975.626 rows=0 loops=3)
18                        Buffers: shared hit=89 read=9228
19                        ->  Parallel Hash Join  (cost=425897.67..2536260.42 rows=1 width=16) (actual time=5975.625..5975.629 rows=0 loops=3)
20                            Hash Cond: ("Ticket"."Codigo_trabajador_Trabajador" = "Trabajador"."Codigo_trabajador")
21                            Buffers: shared hit=89 read=9228
22                            ->  Parallel Seq Scan on "Ticket"  (cost=0.00..2110361.75 rows=691858 width=8) (never executed)
23                                Filter: (("Fecha" >= '2019-09-01'::date) AND ("Fecha" <= '2019-12-31'::date))
24                            ->  Parallel Hash  (cost=425896.67..425896.67 rows=1 width=16) (actual time=5975.372..5975.372 rows=0 loops=3)
25                                Buckets: 1024  Batches: 1  Memory Usage: 0kB
26                                Buffers: shared hit=2 read=9228
27                                ->  Parallel Seq Scan on "Trabajador"  (cost=0.00..425896.67 rows=1 width=16) (actual time=5975.227..5975.227 rows=0 loops...
28                                    Filter: (("Salario" > 3000) AND ("Salario" < 5000))
29                                    Rows Removed by Filter: 333333
30                                    Buffers: shared hit=2 read=9228
31                        ->  Index Scan using "Ticket_Productos_pk" on "Ticket_Productos"  (cost=0.00..74823.40 rows=17 width=10) (never executed)
32                            Index Cond: ("No_de_ticket_Ticket" = "Ticket"."No_de_ticket")
33                            Filter: ("Cantidad" > 4)
34                    ->  Index Scan using "Productos_pk" on "Productos"  (cost=0.00..1.00 rows=1 width=10) (never executed)
35                        Index Cond: (("Codigo_de_barras")::text = ("Ticket_Productos"."Codigo_de_barras_Productos")::text)
36                        Filter: ("Precio" > 400)
37                ->  Index Scan using "Tienda_Productos_pk" on "Tienda_Productos"  (cost=0.00..8002.61 rows=22 width=12) (never executed)
38                    Index Cond: (("Codigo_de_barras_Productos")::text = ("Ticket_Productos"."Codigo_de_barras_Productos")::text)
39                ->  Index Scan using "Tienda_pk" on "Tienda"  (cost=0.00..1.00 rows=1 width=4) (never executed)
40                    Index Cond: ("ID_Tienda" = "Tienda_Productos"."ID_Tienda_Tienda")
41                    Filter: ("Provincia" = 'Madrid'::text)
42            ->  Gather  (cost=0.00..425913.67 rows=524181 width=4) (never executed)
43                Workers Planned: 2
44                Workers Launched: 0
45                ->  Parallel Seq Scan on "Productos" "Productos_1"  (cost=0.00..425913.67 rows=218409 width=4) (never executed)
46                    Filter: ("Precio" > 500)
47    ->  Gather  (cost=0.00..8441512.33 rows=9471454 width=2) (never executed)
48        Workers Planned: 2
49        Workers Launched: 0
50        ->  Parallel Seq Scan on "Tienda_Productos" "Tienda_Productos_1"  (cost=0.00..8441512.33 rows=3946439 width=2) (never executed)
51            Filter: ("Stock" < 100)
52    Planning Time: 658.247 ms
53    Execution Time: 6075.825 ms
```

El árbol mostrado con el comando explain es el de arriba, indicando las operaciones que realizaría PostgreSQL para realizar la consulta y el procedimiento de filtros a aplicar en la consulta, con nuestro árbol teórico realizado la gran diferencia es la falta de optimización entre el nuestro y el que haría PostgreSQL. Mientras que el nuestro al no aplicar ninguna optimización tardaría más tiempo en realizar la consulta, PostgreSQL las realiza de forma previa para reducir tiempos de consulta mejorando así la eficiencia de este.

En teoría, con lo visto en el árbol de consultas del comando EXPLAIN no habría ninguna coincidencia o ningún resultado que cumpla todas estas condiciones ya que todos los datos se han insertado con valores aleatorios y ha dado la casualidad de que ninguno ha coincidido. También nos muestra este comando explain que hay unos cuantos filtros o condiciones que ni siquiera las ha llegado a aplicar ya que al parecer antes de llegar a esas, ya no quedaban tuplas para filtrar (never executed).

Cuestión 11: Usando PostgreSQL, y a raíz de los resultados de la cuestión anterºior, ¿qué modificaciones realizaría para mejorar el rendimiento de esta 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.

```
explain(analyze TRUE, buffers TRUE, format TEXT)
select "Trabajador"."Nombre", "Trabajador"."DNI"
from "Tienda"
    inner join "Trabajador" on "Tienda"."ID_Tienda" = "Trabajador"."ID_Tienda_Tienda"
where "Trabajador"."Salario" between 3000 and 5000
    and "Tienda"."Provincia" = 'Madrid'
    and "Tienda"."ID_Tienda"
    in
    (select "Tienda_Productos"."ID_Tienda_Tienda"
     from "Tienda_Productos"
        inner join "Productos" on "Productos"."Codigo_de_barras" = "Tienda_Productos"."Codigo_de_barras_Productos"
     where "Tienda_Productos"."Stock" < 100 and "Productos"."Precio" > 400)
```

	QUERY PLAN text
1	Gather (cost=0.00..426693.79 rows=1 width=12) (actual time=185.322..199.567 rows=0 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	Buffers: shared hit=577 read=8653
5	-> Nested Loop Semi Join (cost=0.00..426693.79 rows=1 width=12) (actual time=139.899..139.899 rows=0 loops=3)
6	Buffers: shared hit=577 read=8653
7	-> Nested Loop (cost=0.00..426118.46 rows=2 width=20) (actual time=139.898..139.899 rows=0 loops=3)
8	Buffers: shared hit=577 read=8653
9	-> Parallel Seq Scan on "Trabajador" (cost=0.00..425896.67 rows=77 width=16) (actual time=139.897..139.897 rows=0 loops=3)
10	Filter: (("Salario" >= 3000) AND ("Salario" <= 5000))
11	Rows Removed by Filter: 333333
12	Buffers: shared hit=577 read=8653
13	-> Index Scan using "Tienda_pk" on "Tienda" (cost=0.00..2.88 rows=1 width=4) (never executed)
14	Index Cond: ("ID_Tienda" = "Trabajador"."ID_Tienda_Tienda")
15	Filter: ("Provincia" = 'Madrid'::text)
16	-> Nested Loop (cost=0.00..471.25 rows=92 width=4) (never executed)
17	-> Index Scan using "Tienda_Productos_pk" on "Tienda_Productos" (cost=0.00..324.96 rows=146 width=10) (never executed)
18	Index Cond: ("ID_Tienda_Tienda" = "Tienda"."ID_Tienda")
19	Filter: ("Stock" < 100)
20	-> Index Scan using "Productos_pk" on "Productos" (cost=0.00..1.00 rows=1 width=6) (never executed)
21	Index Cond: (("Codigo_de_barras")::text = ("Tienda_Productos"."Codigo_de_barras_Productos")::text)
22	Filter: ("Precio" > 400)
23	Planning Time: 1.488 ms
24	Execution Time: 199.632 ms

Para poder optimizar la consulta hemos pensado en paralelizar la misma y así conseguir más Workers en el planner y en la consulta, viendo cómo se reduce al final del todo de la consulta de arriba de 6075 ms de Execution time a 200 ms en esta, pudiendo reducir el coste de la consulta. Aunque también se podría haber hecho índices de las tablas y reducir aún más los costes de la consulta, pero teniendo en cuenta que se ha reducido a estados 0.2 segundos, no es necesario usarlos y gastar más memoria en 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.

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?

Las herramientas que podemos utilizar para mejorar los resultados del plan de ejecución son vacuum, que se encarga de limpiar y analizar una base de datos, analyze, que se encarga de actualizar las estadísticas de las columnas usadas por el optimizador para determinar la manera más eficiente de ejecutar una consulta, y reindex, que se encarga de rehacer los índices que se habían creado con los datos restantes.

Estas opciones las podemos encontrar en la opción de mantenimiento de pgAdmin, para optimizar nuestra base de datos.

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.

```
VACUUM (VERBOSE, ANALYZE) "Nombre_tabla"
```

El plan de ejecución de la consulta no cambia, ya que tras realizar el mantenimiento vuelve a tener la estructura que tenía cuando se realizó la consulta antes de hacer el borrado. Ahora simplemente almacena un 50% menos de registros ya que hemos borrado anteriormente el 50% de las tablas.

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.

Las consultas más utilizadas han sido las del ejercicio 10 y 11 ya que son las únicas que se han hecho a parte del borrado de la tabla Tienda. El número de consultas es (insertar el numero después del borrado) y el tiempo de ejecución nos ha sido imposible encontrarlo ya que en el log no aparecen nada de tiempos, pero si en el comando explain que ya hemos mostrado en las capturas anteriores.

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 proceso de procesamiento y optimización que realiza PostgreSQL es muy amplio y variado. Puedes desde la creación de índices para optimizar la búsqueda de datos, como también la paralelización de las consultas como en el ejercicio 11, pero también puedes realizar unas tareas de mantenimiento muy útiles ya implementadas. Que solo a golpe de click se pueden hacer para que el plan de consultas sea más rápido que lo usual.

Sabiendo esto se podría hacer un trigger o alguna automatización del mantenimiento de la base de datos como también al realizar consultas o implementarlas, intentar buscar que se realizan de la forma más concurrente posible y así utilizar más recursos de CPU y ahorrar tiempo de espera para los clientes que usen la base de datos, como también siempre buscar la indexación de las tablas actualizada para mantener los tiempos de búsqueda en lo mínimo posible.

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.