Titulación: **Grado en Ingeniería Informática y Sistemas de Información**

Curso: **2019-2020. Convocatoria Extraordinaria de Septiembre**

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: 15/09/2020

Profesor Responsable: Óscar Gutierrez 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 puntuará **TODA** la asignatura como Suspenso – Cero.

Plazos

Entrega de práctica: Día a convenir por el Aula Virtual a primeros de Septiembre. 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. Para ello se generarán, dependiendo del modelo de datos suministrado de una base de datos **MUSICOS**. Básicamente, la base de datos guarda los músicos que pertenecen a grupos musicales, así como los conciertos, discos y canciones que tocan. Además, se almacenan las entradas que se venden para sus conciertos (más información en la lógica de negocio del Aula Virtual). Los datos referidos al año 2019 que hay que generar deben de ser los siguientes:

- 1.000.000 de Discos donde el género musical puede ser clásica, blues, jazz, rock&roll, góspel, soul, rock, metal, funk, disco, techno, pop, reggae, hiphop, salsa. La distribución del campo género musical debe ser aleatoria entre esos valores.
- Cada disco tiene de media 12 canciones con duración de canciones que van entre los 2 minutos y los 7 minutos.
- Hay 23.000.000 de entradas distribuidas de una manera aleatoria entre todos los conciertos y el precio oscila entre 20 y 100 euros de una manera aleatoria.

- Hay 100.000 conciertos en marcha y se realizan entre 20 países donde uno de ellos debe de ser obligatoriamente España. Distribución aleatoria.
- Hay 1.000.000 de músicos.
- Hay 200.000 grupos donde cada uno de ellos debe tener entre 1 y 10 músicos.
- Todos los grupos han realizado por lo menos 10 conciertos y todos los conciertos deben de estar asociados por lo menos a 1 grupo musical.

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

- concesionario (codigo_concesionario tipo numeric PRIMARY KEY, nombre tipo text, marca tipo text)
- automovil (codigo_automovil tipo numeric PRIMARY KEY, matricula tipo text, precio real, codigo_concesionario numeric que sea un FK que referencia a codigo_concesionario de la tabla concesionario)

Se pide:

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

Para crear la base de datos y la tabla hemos ejecutado el siguiente código:

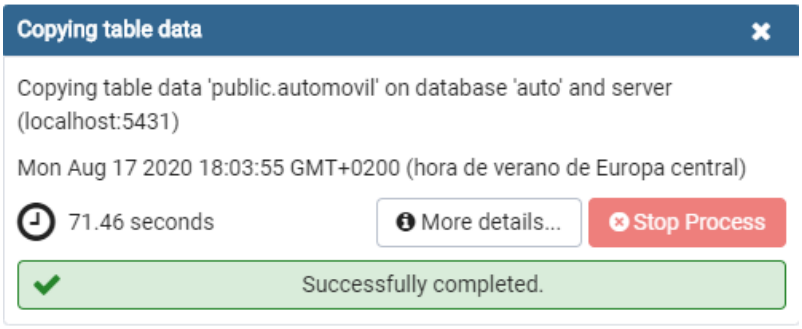
```
CREATE DATABASE auto
    WITH
    OWNER = postgres
    ENCODING = 'UTF8'
    LC_COLLATE = 'Spanish_Spain.1252'
    LC_CTYPE = 'Spanish_Spain.1252'
    TABLESPACE = pg_default
    CONNECTION LIMIT = -1;
```

```
CREATE TABLE public.automovil
(
    codigo_automovil numeric NOT NULL,
    matricula text COLLATE pg_catalog."default",
    precio real,
    codigo_concesionario numeric NOT NULL,
    CONSTRAINT automovil_pkey PRIMARY KEY (codigo_automovil),
    CONSTRAINT codigo_concesionario FOREIGN KEY (codigo_concesionario)
        REFERENCES public.concesionario (codigo_concesionario) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
CREATE TABLE public.concesionario
(
    codigo_concesionario numeric NOT NULL,
    nombre text COLLATE pg_catalog."default",
    marca text COLLATE pg_catalog."default",
    CONSTRAINT concesionario_pkey PRIMARY KEY (codigo_concesionario)
)
```

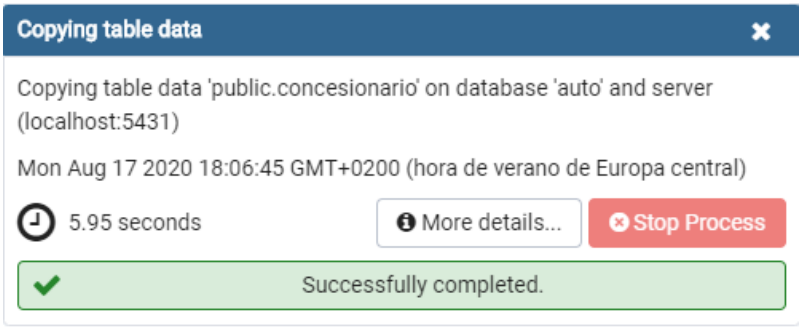
Para realizar la carga de datos de la forma más eficiente posible es necesario deshabilitar la integridad referencial eliminando la foreign key constraint.

```
alter table automovil drop constraint codigo_concesionario;
```

Tiempo de carga de la tabla automóvil:



Tiempo de carga de la tabla concesionario:



Una vez subidos los datos volvemos a crear la foreign key constraint.

```
alter table automovil add constraint codigo_concesionario foreign key (codigo_concesionario)
    references public.concesionario(codigo_concesionario) on delete restrict on update restrict
```

De esta manera, al quitar la IR nos aseguramos de que al realizar la carga de datos no se compruebe la integridad referencial en ningún momento, por lo que el tiempo de carga se reduce drásticamente.

Cuestión 3: 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 para mostrar los datos de la base de datos auto.

```
select *
from pg_stats
where (tablename = 'automovil') or (tablename = 'concesionario')
```

Como resultado obtenemos lo siguiente:

	schemaname name	tablename name	attname name	inherited boolean	null_frac real	avg_width integer	n_distinct real	most_common_vals anyarray	most_common_freqs real[]	histogram_bounds anyarray	correlation real	most_compo anyarray	most_compo real[]	elem_count_hi real[]
1	public	concesionario	codigo_conc...	false	0	6	-1	[null]	[null]	{173,20335,40034,61206...	1	[null]	[null]	[null]
2	public	concesionario	nombre	false	0	20	-1	[null]	[null]	{concesionario1000035,...	-0.39153662	[null]	[null]	[null]
3	public	concesionario	marca	false	0	8	1000	{marca228,marca46...	14666667,0.0014666667)	{marca0,marca106,marc...	0.0025658407	[null]	[null]	[null]
4	public	automovil	codigo_auto...	false	0	6	-1	[null]	[null]	{228,107980,212854,307...	1	[null]	[null]	[null]
5	public	automovil	matricula	false	0	16	-1	[null]	[null]	{matricula1000461,matri...	0.8208749	[null]	[null]	[null]
6	public	automovil	precio	false	0	4	-0.17636462	[null]	[null]	{10000.72,10200.3,1041...	0.003734499	[null]	[null]	[null]
7	public	automovil	codigo_conc...	false	0	6	-0.15618916	[null]	[null]	{130,19341,39907,60938...	-0.0007390465	[null]	[null]	[null]

Tal y como se observa en la tabla, los resultados de las estadísticas son correctos. En el caso de no serlo se debería actualizar cada una de las tablas usando el comando ANALYZE, ya que por defecto está activado el AUTOVACUUM y las estadísticas, a pesar de que se actualizan de forma automática, podrían no estarlo.

Cuestión 4: ¿Para qué sirve el comando EXPLAIN? Aplicarlo a una consulta que obtenga la información del automóvil con código número 8.134.468. ¿Son correctos los resultados del comando EXPLAIN? ¿Por qué? Comparar con lo que se obtendría al aplicar el procedimiento visto en teoría.

El comando EXPLAIN se usa para mostrar una tabla en la que se muestra la forma en que serán escaneadas las tablas referenciadas sin ejecutar la consulta.

Ejecutamos la siguiente consulta:

```
explain analyze
select *
from automovil
where codigo_automovil = 8134468
```

Obteniendo:

	QUERY PLAN text
1	Index Scan using automovil_pkey on automovil (cost=0.43..8.45 rows=1 width=32) (actual time=0.015..0.016 rows=1 loops=1)
2	Index Cond: (codigo_automovil = '8134468'::numeric)
3	Planning Time: 0.053 ms
4	Execution Time: 0.027 ms

La consulta ha salido que solo devuelve 1 tupla. Al ser una selección de igualdad con una distribución uniforme de valores, la fórmula para calcularlo teóricamente es la siguiente:

$$n_{rc} = \frac{n_r}{\mathrm{V}(A)}$$

Sabiendo que n_r = 10.000.000 y V(código_automovil) = 10.000.000:

$$n_{rc} = \frac{10.000.000}{10.000.000} = 1\ tuplas$$

El resultado es 1, por lo que concuerda con lo obtenido mediante el comando EXPLAIN, lo cual es obvio ya que código_automovil es la PK de la tabla automóvil, por lo que cada valor es único.

Cuestión 5: Aplicar el comando EXPLAIN a una consulta que obtenga la información de los precios de los automóviles cuyo concesionario sea de la marca marca600. ¿Son correctos los resultados del comando EXPLAIN? ¿Por qué? Comparar con lo que se obtendría al aplicar el procedimiento visto en teoría.

Ejecutamos la siguiente consulta:

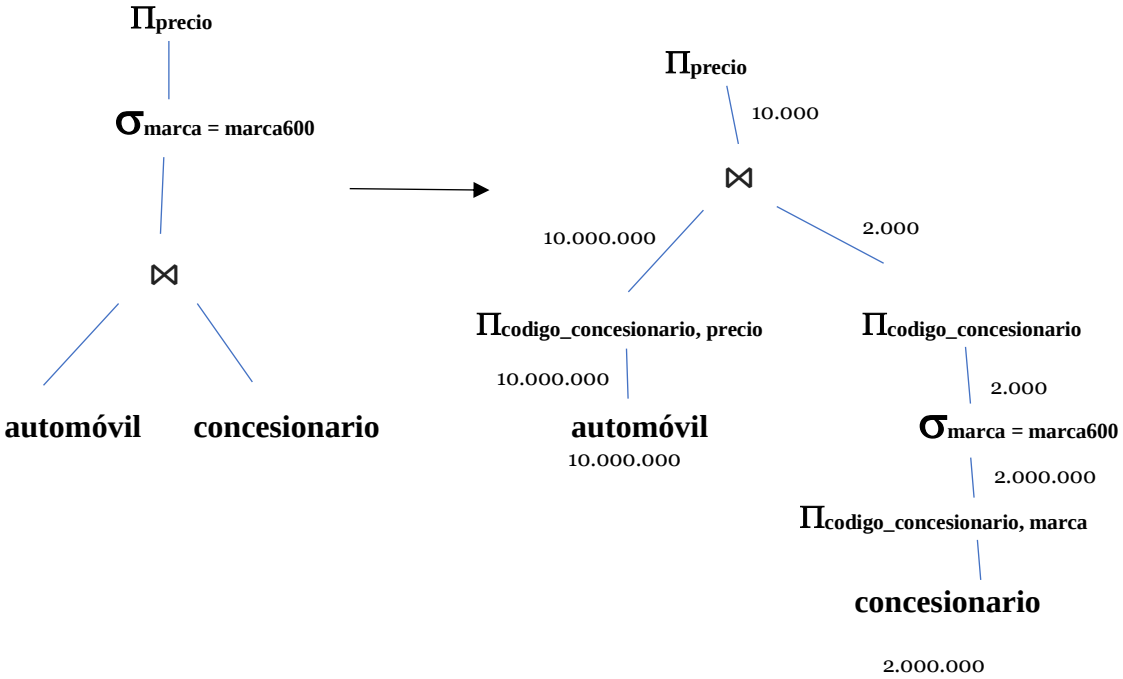
```
explain analyze
select precio
from automovil inner join concesionario on automovil.codigo_concesionario = concesionario.codigo_concesionario
where marca = 'marca600'
```

Obteniendo lo siguiente:

	QUERY PLAN text
1	Gather (cost=28084.02..165006.36 rows=9935 width=4) (actual time=122.361..928.831 rows=10034 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Hash Join (cost=27084.02..163012.86 rows=4140 width=4) (actual time=96.613..895.638 rows=3345 loops=3)
5	Hash Cond: (automovil.codigo_concesionario = concesionario.codigo_concesionario)
6	-> Parallel Seq Scan on automovil (cost=0.00..124991.18 rows=4166718 width=10) (actual time=0.167..323.568 rows=3333333 loops=3)
7	-> Parallel Hash (cost=27073.67..27073.67 rows=828 width=6) (actual time=95.548..95.548 rows=659 loops=3)
8	Buckets: 2048 Batches: 1 Memory Usage: 112kB
9	-> Parallel Seq Scan on concesionario (cost=0.00..27073.67 rows=828 width=6) (actual time=0.341..95.120 rows=659 loops=3)
10	Filter: (marca = 'marca600'::text)
11	Rows Removed by Filter: 666007
12	Planning Time: 0.160 ms
13	Execution Time: 929.246 ms

Esta consulta devuelve 9.935 tuplas.

Para calcularlo teóricamente es necesario usar un árbol:



Para poder saber el número de valores distintos de la columna marca hemos realizado la siguiente consulta:

```
select count(distinct(marca))
from concesionario
```

Obteniendo:

	count bigint
1	1000

Por lo tanto, podemos calcular ya el número de tuplas teóricamente:

$$T(\sigma_{marca\ =marca600}) = \frac{n_r}{V(marca, concesionario)} = \frac{2.000.000}{1.000} = 2.000\ tuplas$$

$$T(\bowtie) = \frac{n_r * n_s}{\max\{V(cod\_con, automovil), V(cod\_con, concesionario)} = \frac{10.000.000 * 2.000}{2.000.000} = 10.000\ tuplas$$

Como podemos observar, el resultado teórico se aproxima mucho al resultado dado por el comando EXPLAIN por lo que podemos decir que el resultado del EXPLAIN es correcto.

Cuestión 6: Aplicar el comando EXPLAIN a una consulta que obtenga la información de los automóviles cuya marca sea del concesionario marca600 y un precio de más de 16.000 euros. ¿Son correctos los resultados del comando EXPLAIN? ¿Por qué? Comparar con lo que se obtendría al aplicar el procedimiento visto en teoría.

Ejecutamos la siguiente consulta:

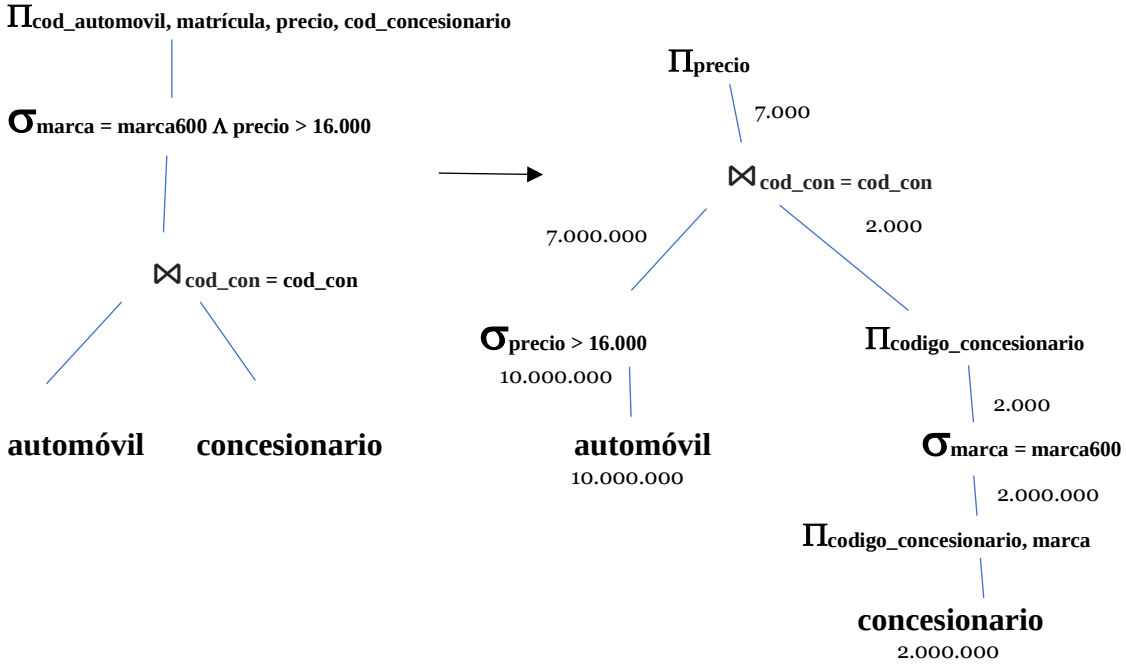
```
explain analyze
select codigo_automovil, matricula, precio, automovil.codigo_concesionario
from automovil inner join concesionario on automovil.codigo_concesionario = concesionario.codigo_concesionario
where marca = 'marca600' and precio > 16000
```

Obteniendo:

	QUERY PLAN text
1	Gather (cost=28084.02..171856.49 rows=6965 width=66) (actual time=181.382..7469.235 rows=6995 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Hash Join (cost=27084.02..170159.99 rows=2902 width=66) (actual time=146.202..7406.458 rows=2332 loops=3)
5	Hash Cond: (automovil.codigo_concesionario = concesionario.codigo_concesionario)
6	-> Parallel Seq Scan on automovil (cost=0.00..135407.98 rows=2921136 width=32) (actual time=4.800..6873.887 rows=2332872 loops=3)
7	Filter: (precio > '16000'::double precision)
8	Rows Removed by Filter: 1000462
9	-> Parallel Hash (cost=27073.67..27073.67 rows=828 width=34) (actual time=95.765..95.765 rows=659 loops=3)
10	Buckets: 2048 Batches: 1 Memory Usage: 240kB
11	-> Parallel Seq Scan on concesionario (cost=0.00..27073.67 rows=828 width=34) (actual time=0.574..95.341 rows=659 loops=3)
12	Filter: (marca = 'marca600'::text)
13	Rows Removed by Filter: 666007
14	Planning Time: 79.645 ms
15	Execution Time: 7470.227 ms

Esta consulta devuelve 6965 tuplas.

Para calcularlo teóricamente es necesario hacer la optimización heurística del siguiente árbol:



Para poder usar la fórmula que calcula el tamaño de una selección de comparación es necesario conocer el número de valores que satisfacen la condición de precio > 16.000 y ya que el precio máximo es 30.000 y el mínimo 10.000 (obtenidos mediante la función max y min de postgres), el número de valores que la satisfacen son $\frac{14}{20}$.

$$T(\sigma_{precio>16.000}) = \frac{n_v * n_r}{V(precio, automovil)} = \frac{14 * 10.000.000}{20} = 7.000.000\ tuplas$$

$$T(\sigma_{marca\ =marca600}) = \frac{n_r}{V(marca, concesionario)} = \frac{2.000.000}{1.000} = 2.000\ tuplas$$

$$T(\bowtie) = \frac{n_r * n_s}{\max\{V(cod\_con, automovil), V(cod\_con, concesionario)} = \frac{7.000.000 * 2.000}{2.000.000} = 7.000\ tuplas$$

Como podemos observar, el número de tuplas proporcionado por el EXPLAIN se aproxima mucho al calculado teóricamente, por lo que podemos deducir que el número de tuplas es concordante con el obtenido.

Cuestión 7: 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 **MUSICOS**. 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 en este último paso 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
Canciones	105.34 segundos	202.77 segundos
Conciertos	0.48 segundos	0.54 segundos
Discos	9.91 segundos	15.81 segundos
Entradas	193.14 segundos	327.39 segundos
Grupos	0.56 segundos	0.88 segundos
Musicos	23.23 segundos	41.05 segundos

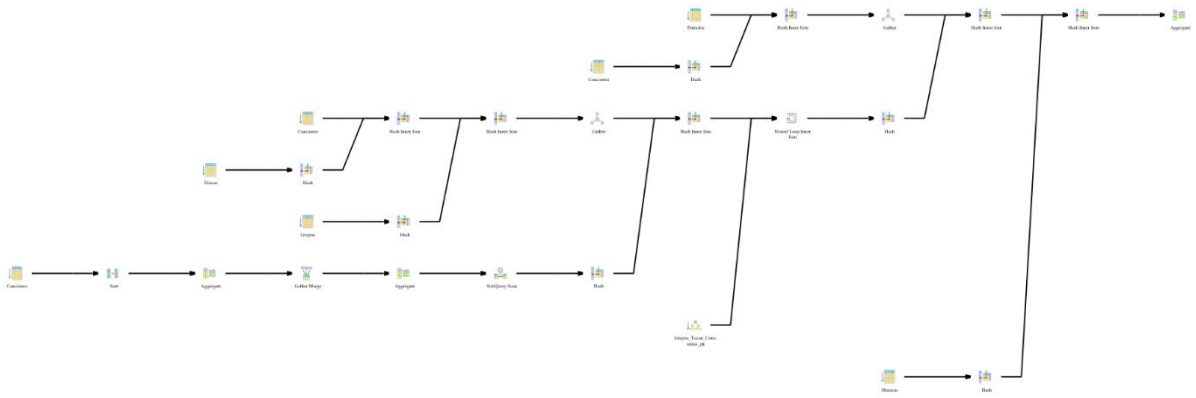
El orden de carga es necesario ya que, si insertamos de otro orden, en el cual, si las tablas con una FK se intentan insertar antes que las tablas que tienen su PK, saltaría un error por la integridad referencial de la base de datos.

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.

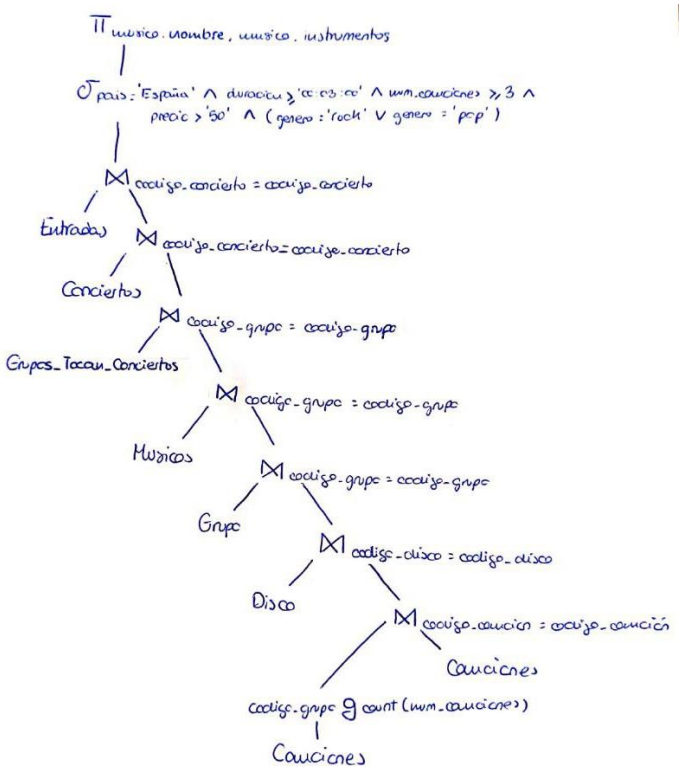
A partir de este momento en adelante, se deben de realizar las siguientes cuestiones con la base de datos que tiene la integridad referencial activada.

Cuestión 8: Realizar una consulta SQL que muestre el nombre de los músicos junto con el instrumento que tocan en los grupos que tienen discos de género rock o pop teniendo por lo menos 3 canciones de más de 3 minutos en cada disco; y **que además hayan** realizado conciertos en España con entradas de más de 50 euros. Obtener el plan de ejecución realizado 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.

```
select distinct "Musicos"."nombre", "Musicos"."instrumentos"
from "Musicos"
    inner join "Grupos" on "Musicos"."codigo_grupo_Grupos" = "Grupos"."codigo_grupo"
    inner join "Discos" on "Grupos"."codigo_grupo" = "Discos"."codigo_grupo_Grupos"
    inner join "Canciones" on "Discos"."codigo_disco" = "Canciones"."codigo_disco_Discos"
    inner join "Grupos_Tocan_Conciertos" on "Grupos"."codigo_grupo" = "Grupos_Tocan_Conciertos"."codigo_grupo_Grupos"
    inner join "Conciertos" on "Grupos_Tocan_Conciertos"."codigo_concierto_Conciertos" = "Conciertos"."codigo_concierto"
    inner join "Entradas" on "Conciertos"."codigo_concierto" = "Entradas"."codigo_concierto_Conciertos"
    inner join (select "Canciones"."codigo_disco_Discos", count("Canciones"."codigo_disco_Discos") as "num_canciones"
                from "Canciones"
                group by "Canciones"."codigo_disco_Discos") as "canciones_query"
                on "Canciones"."codigo_disco_Discos" = "canciones_query"."codigo_disco_Discos"
where "Conciertos"."pais" = 'España'
    and ("Discos"."genero" = 'rock' or "Discos"."genero" = 'pop')
    and "Canciones"."duracion" >= '00:03:00'
    and "canciones_query"."num_canciones" >= 3
    and "Entradas"."precio" > '50'
```



Como podemos observar el árbol de ejecución realiza en las distintas ramas, los procedimientos necesarios para obtener la consulta. Empieza el árbol procesando las canciones para poder hacer un hash con ellas y así poder hacer un *inner join* con otras tres tablas que son las de Discos, Grupos y Conciertos (los cuales ya han sido juntados con otros dos *inner join*, uno para los Discos y las Canciones y el otro para el resultado de este con los Grupos). Una vez procesadas las tablas, PostgreSQL lo que va haciendo es las reuniones con las tablas que tienen atributos comunes o dependencias relacionales para que al final del todo, una vez tenga reunidas todas las tablas pueda hacer la consulta con las condiciones necesarias.



El árbol que hemos estimado no se corresponde del todo con el obtenido con PostgreSQL ya que nosotros hemos realizado los cálculos estadísticamente, mientras que PostgreSQL ha realizado las operaciones sobre las estadísticas reales de los datos que almacena. El motivo por el que las consultas aparecen como en ramas es porque PostgreSQL es capaz de realizar consultas de manera paralela.

La consulta devuelve un total de 121 tuplas.

Cuestión 9: 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 9 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.

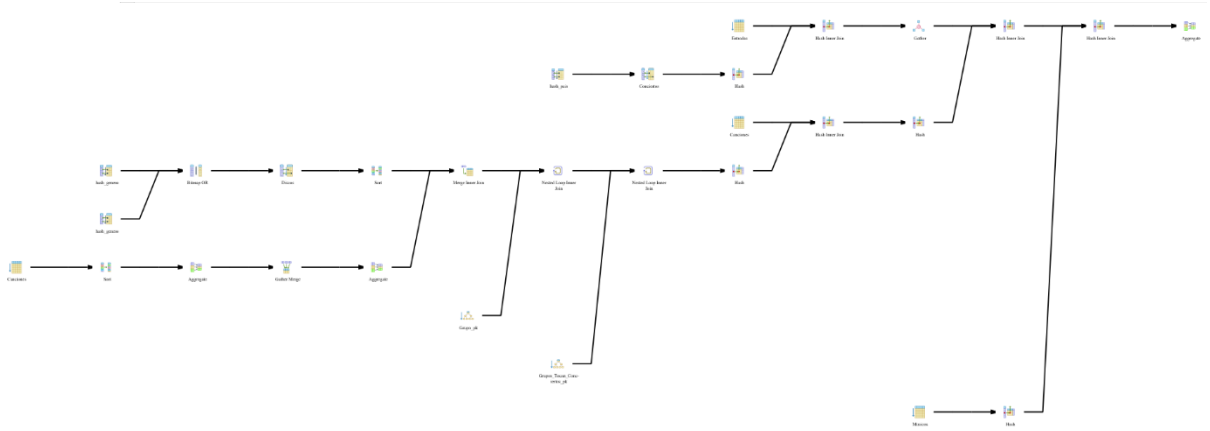
Para intentar mejorar el rendimiento una opción es añadir índices hash sobre los campos que buscamos, en este caso los campos son:

- **País de la tabla Conciertos.**
- **Género de la tabla Discos.**
- **Duración de la tabla Canciones.**

Al igual que también haremos un índice árbol B sobre el atributo precio de la tabla entradas, ya que no es compatible hacer un índice hash sobre un campo tipo *money*.

```
create index hash_pais on "Conciertos" using hash(pais);
create index hash_genero on "Discos" using hash(genero);
create index hash_duracion on "Canciones" using hash(duracion);
create index arbolb_precio on "Entradas"(precio);
```

Realizamos la consulta retornando este árbol:



Cuando no existen índices la consulta tarda menos, ya que Postgres realiza una búsqueda secuencial paralela, lo que supone que la consulta con los índices sea más costosa que la consulta sin índices.

También se podría crear índices de árbol B para cada tabla sobre las PK, reduciendo aún más el tiempo de búsqueda, como también con el uso de las estadísticas de la propia base de datos que usa el comando ANALYZE al realizar la consulta.

Cuestión 10: Usando PostgreSQL, borre el 50% de los grupos almacenados de manera aleatoria y todos sus datos relacionados ¿Cuál ha sido el proceso seguido? ¿Y el tiempo empleado en el borrado? Obtenga el plan de ejecución al ejecutar la consulta de la Cuestión 9. Dibujar un diagrama con el nuevo resultado del comando EXPLAIN en forma de árbol de algebra relacional. Comparar con los resultados anteriores.

Sabemos que la tabla Grupos tiene 200.000 registros, por lo que es necesario borrar 100.000 tuplas.

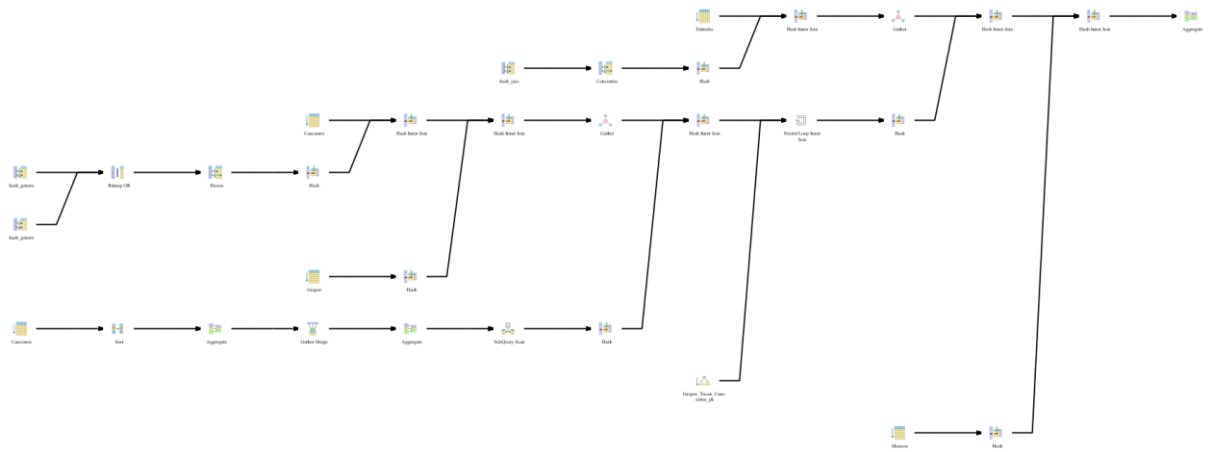
A continuación, hay que deshabilitar la integridad referencial y realizar el borrado aleatorio:

```
alter table "Musicos" drop constraint "Grupos_fk";
alter table "Discos" drop constraint "Grupos_fk";
alter table "Grupos_Tocan_Conciertos" drop constraint "Grupos_fk";

delete from "Grupos" where "Grupos"."codigo_grupo" in (select "Grupos"."codigo_grupo"
                                                       from "Grupos"
                                                       order by random()
                                                       limit 100000)
```

El borrado ha tardado:

```
DELETE 100000

Query returned successfully in 542 msec.
```

El árbol de la cuestión 9 tras el borrado es el siguiente:



A pesar de que se aprecian cambios, la consulta devuelve el mismo número de tuplas ya que en la consulta no usamos la tabla "Grupos" más que para hacer un inner join, por lo que el resultado no se ve afectado en gran medida por el borrado.

Cuestión 11: ¿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 técnicas de mantenimiento que se pueden usar para mejorar los resultados sin modificar el código de la consulta son:

- **Vacuum: es un proceso que realiza la base de datos para recuperar el espacio de disco perdido en borrados y actualizaciones de datos y protección ante la perdida de datos por reutilizar identificadores de transacción.**
- **Reindex (reindexación): no es una tarea muy habitual, pero puede mejorar considerablemente la velocidad de las consultas complejas en tablas que se utilizan a menudo. Para utilizarlo, solo tenemos que activarlo encima de las tablas como REINDEX TABLE, y se reindexarán todos los índices que posea la tabla.**
- **Analyze: es colector de estadísticas de las tablas de la base de datos, y las almacena en el catálogo del sistema pg_statistic. El planificador de consultas utiliza estas estadísticas para ayudar a determinar la ejecución del plan más eficiente para las consultas.**
- **Cluster: cuando una tabla está agrupada, esta físicamente reordenada en función de la información de su índice. Es un proceso lento y costoso pero que en ocasiones puede resultar muy beneficioso (como puede ser en el nuestro caso, en el que la cantidad de información borrada es bastante considerable).**

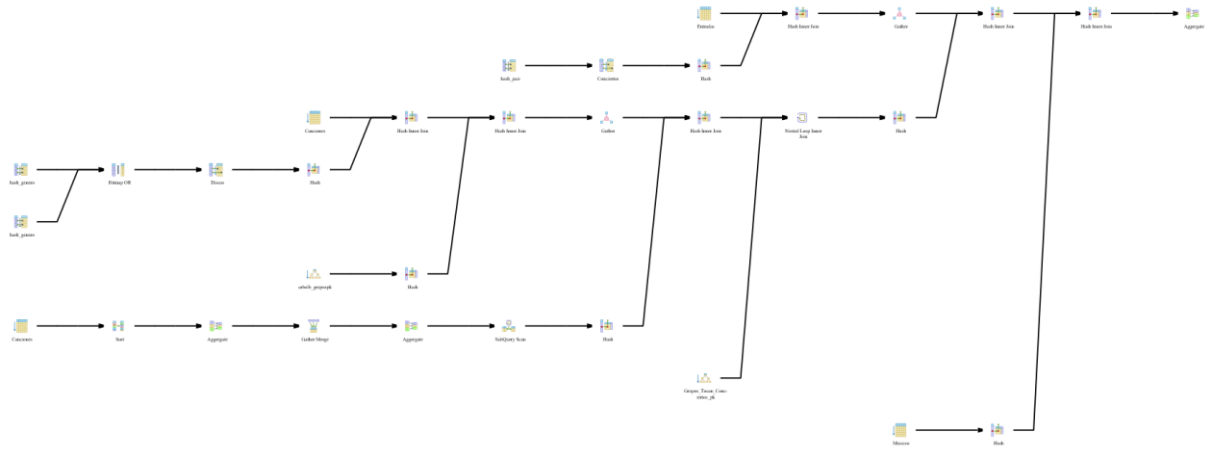
Cuestión 12: Usando PostgreSQL, lleve a cabo las operaciones propuestas en la cuestión anterior y ejecute el plan de ejecución de la misma consulta. Dibujar un diagrama con el nuevo resultado del comando EXPLAIN en forma de árbol de algebra relacional. Compare los resultados del plan de ejecución con los de los apartados anteriores. Coméntelos.

Las opciones de mantenimiento de postgres son:

También se podrían hacer las consultas en cada una de las tablas, por ejemplo:

```
vacuum full "Grupos";
analyze "Grupos";
reindex table "Grupos";
cluster "Grupos" using arbolb_gruposPK;
```

El árbol de ejecución de la consulta de la cuestión 8 es el siguiente:



Tras ejecutar la consulta después de haber realizado el mantenimiento, la mejora que se observa es mínima, tan solo hemos obtenido 20 segundos de mejora.

Cuestión 13: 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.

Tras la realización de la práctica, podemos decir que los objetivos del procesamiento de las consultas es transformar una consulta SQL, en una estrategia de ejecución eficaz y recuperar los datos requeridos minimizando el uso de los recursos del ordenador o servidor. En general, se puede ver que no se garantiza que la estrategia elegida sea la óptima, pero sí una estrategia razonablemente eficiente que cumpla con los objetivos establecidos.

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 sintáctico revisa la sintaxis de la consulta (corrección gramática). La validación semántica verifica la validez de los nombres de las tablas, vistas, columnas, etc. La traducción de la consulta a una representación interna eliminando peculiaridades del lenguaje de alto nivel empleado.

- **Optimización y optimización heurística:**

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

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, por ejemplo: la 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 (nº de filas).**
- **Factor de bloques (nº de filas que caben en un bloque).**
- **Nº 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.**

Y 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), y cada técnica tendrá asociada una estimación del coste (número de accesos a bloque de disco necesarios).

Finalmente, el optimizador elige el plan de ejecución más económico formulando 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 pueden ser: a través de índices, partición de tablas, uso del comando EXPLAIN, el uso de EXPLAIN ANALYZE, etc. Todas estas funciones nos pueden ayudar a mejorar nuestras consultas en la base de datos y así alcanzar un gran rendimiento, o al menos el rendimiento óptimo 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.