

TITULACIÓN: GRADO EN INGENIERÍA INFORMÁTICA Y SISTEMAS DE INFORMACIÓN  
CURSO: 2018-2019. CONVOCATORIA ORDINARIA DE ENERO  
ASIGNATURA: BASES DE DATOS AVANZADAS – LABORATORIO

## PRACTICA 2: CARGA MASIVA DE DATOS, PROCESAMIENTO Y OPTIMIZACIÓN DE CONSULTAS

### ALUMNO 1:

Nombre y Apellidos: Marcos Barranquero Fernández

DNI: 51129104N

### ALUMNO 2:

Nombre y Apellidos: Eduardo Graván Serrano

DNI: 03212337L

Fecha: 22/10/2018

Profesor Responsable: Iván Gonzalez

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 práctica como Suspenso – Cero.

### PLAZOS

Tarea en Laboratorio: Semana 22 de Octubre, Semana 29 de Octubre, Semana 5 de Noviembre, semana 12 de Noviembre y semana 19 de Noviembre.

Entrega de práctica: Semana 26 de Noviembre (Lunes). 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 (10.x) y MySQL (V8.x). Se comparará ambos gestores de bases de datos 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 2017 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 24.000.000 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:** ¿PARA QUÉ SIRVE EL LOG DE ERRORES DE POSTGRESQL? ¿TIENE ARRANCADO EL LOG DE ERRORES LA BD? MODIFICAR LA CONFIGURACIÓN DE DICHO LOG PARA QUE QUEDEN REFLEJADAS TODAS LAS OPERACIONES SOLICITADAS A LA BD Y SUS TIEMPOS DE EJECUCIÓN

El log de PostgreSQL captura por defecto solamente errores, y los guarda en "C:\Program Files\PostgreSQL\10\data\logs".

Si deseamos cambiar la configuración para que capture todas las operaciones y sus tiempos de ejecución, debemos editar el archivo postgresql.conf y añadir en la sección de log:

```
log_statement = 'all'
```

Si queremos que además almacene el tiempo que tardan en ejecutarse las diferentes consultas, se añade la línea:

```
log_duration = 'on'
```

Nota: log colector debe estar en on.

**CUESTIÓN 2:** ¿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?

Si: postgresql guarda estadísticas sobre la actividad y estadísticas sobre los datos.

DATOS:

pg\_statistic almacena datos estadísticos sobre el contenido de la base de datos: sobre las tablas y sobre los índices. También guarda info. Sobre la herencia de las tablas. Pg\_statistic está restringido a superusuarios, mientras que pg\_stat está destinado para el resto de los usuarios.

ACTIVIDAD:

Respecto a estadísticas de postgresql, existe un subsistema llamado el colector de estadísticas, que reporta información sobre la actividad del servidor: accesos a tablas, índices, etc.

Una vez más, el sistema se puede activar o desactivar en postgresql.conf. Por defecto recolecta comandos y accesos a tablas e índices.

Las estadísticas de datos se guardan en una tabla asociada a cada base de datos, mientras que las estadísticas de actividad se guardan en pg\_stat y temporalmente en pg\_stat\_temp.

**CUESTIÓN 3:** CREAR UNA NUEVA BASE DE DATOS LLAMADA **LABORATORIO** Y QUE TENGA LAS SIGUIENTES TABLAS CON LOS SIGUIENTES CAMPOS Y CARACTERÍSTICAS:

- investigadores(codigo\_investigador tipo numeric PRIMARY KEY, nombre tipo text, apellidos tipo text, salario tipo numeric)
- proyectos(codigo\_proyecto tipo numeric PRIMARY KEY, nombre tipo text, localización tipo text, coste tipo numeric)
- investigadores\_proyectos(numero\_investigador tipo numeric que sea FOREIGN KEY del campo codigo\_investigador de la tabla investigadores con restricciones de tipo RESTRICT en sus operaciones, numero\_proyecto tipo numeric que sea FOREIGN KEY del campo codigo\_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\_investigador y numero\_proyecto.

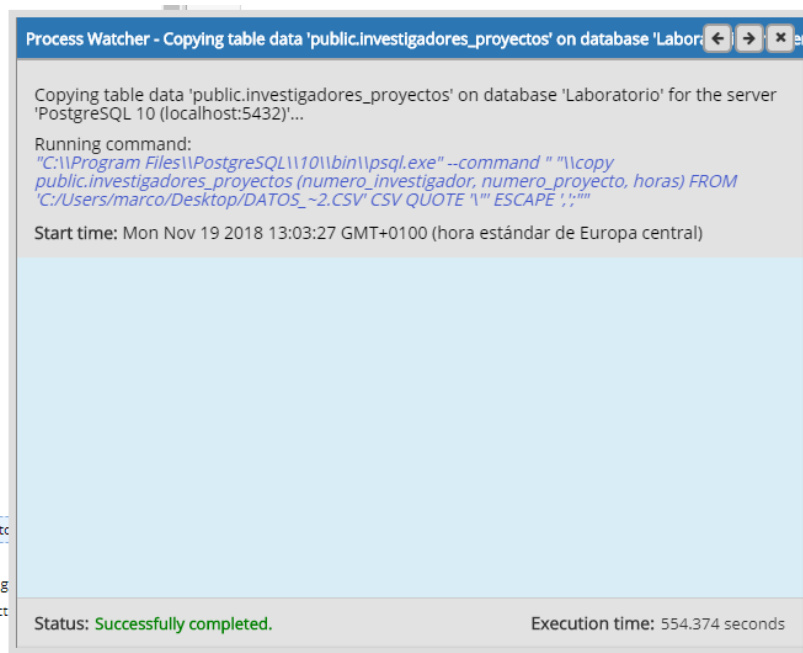
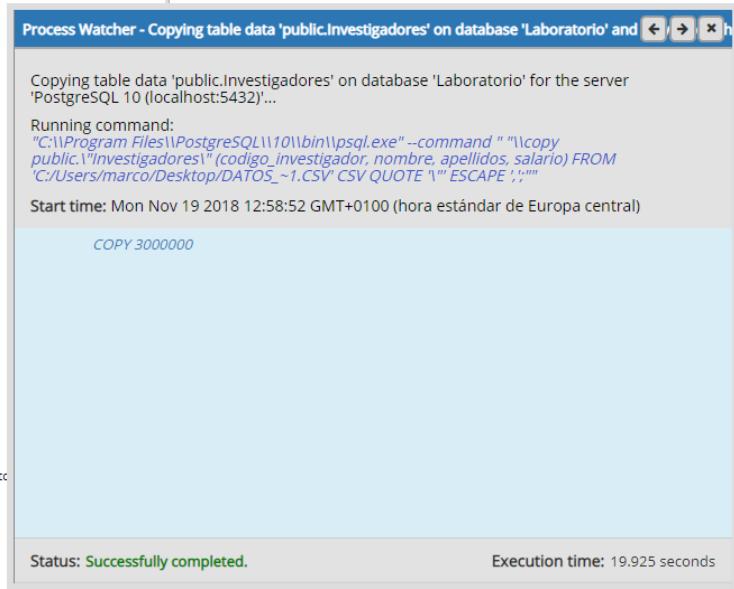
Se pide:

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

Se crea la base de datos a través de la interfaz gráfica del pgAdmin 4. Se crean las tablas tal y como se piden con los atributos y los tipos que se especifican en el enunciado.

A la hora de crear la tabla intermedia, se añaden los constraints de tipo foreign key que hacen referencia a los atributos correspondientes de las otras tablas. Se añade además otro constraint de tipo primary key que engloba a las dos columnas que entran como foreign key para ser una primary key compuesta.

Tras crear las tablas, cargamos los datos:



**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?**

Podemos observar que almacenan los valores más comunes, y la frecuencia de sus apariciones, la correlación entre el orden de filas físico y lógico, entre otros datos:

Investigadores\_proyectos:

name	attname	inherited	null_frac	avg_width	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	correlation
investigadores_pr...	numero_i...	false	0	6	-0.275701	{14339,36212,68892,71...	{6.66667e-005,6.66667e-005}	{124,29480,58471,885...	0.000932887
investigadores_pr...	numero_p...	false	0	6	192380	{2107,52198,107848,11...	{0.001,0.0001,0.0001,0.0001}	{32,2020,3954,6112,8...	0.00993445
investigadores_pr...	horas	false	0	4	24	{19,14,23,22,1,16,3,5,7,...}	{4,0.0397,0.0397,0.0388667}	[null]	0.0316549

Proyectos:

	schemaname	tablename	attname	inherited	null_frac	avg_width	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	correlation	most_common_vals
1	public	Proyectos	codigo_pr...	false	0	6	-1	[null]	[null]	{19,1963,4103,6095,7...	1	[null]
2	public	Proyectos	nombre	false	0	12	-1	[null]	[null]	{nombre100001,nom...	-0.395263	[null]
3	public	Proyectos	localizacion	false	0	18	-1	[null]	[null]	{localizacion100001,lo...	-0.395263	[null]
4	public	Proyectos	coste	false	0	6	-0.138165	{34373.00,11104.00,13...	{6667,0.000166667}	{10000.00,10290.00,1...	-0.00237269	[null]

Investigadores:

	schemaname	tablename	attname	inherited	null_frac	avg_width	n_distinct	most_common_vals	most_common_freqs	histogram_bounds	correlation	most_common_vals
1	public	Investiga...	codigo_in...	false	0	6	-1	[null]	[null]	{70,29910,58446,8809...	1	[null]
2	public	Investiga...	nombre	false	0	13	-1	[null]	[null]	{nombre1000026,no...	-0.127285	[null]
3	public	Investiga...	apellidos	false	0	16	-1	[null]	[null]	{apellidos1000026,ap...	-0.127285	[null]
4	public	Investiga...	salario	false	0	6	185165	{13363.00,174360.0,21...	{0.001,0.0001,0.0001}	{1000.000,2866.000,4...	-0.0136978	[null]

**CUESTIÓN 5: APLICAR EL COMANDO EXPLAIN A UNA CONSULTA QUE OBTENGA LA INFORMACIÓN DE LOS INVESTIGADORES CON SALARIO DE MÁS DE 95000 EUROS. ¿SON CORRECTOS LOS RESULTADOS DEL COMANDO EXPLAIN? ¿POR QUÉ?**

Si ejecutamos:

```
select count(*) from "Investigadores" where salario>95000
```

Esto devuelve 1588314 filas. Sin embargo, al ejecutar:

```
explain (format json) select * from "Investigadores" where salario>95000
```

Vemos que devuelve un valor aproximado de 1589295 filas. Podemos ver que los datos no son correctos, pero se acercan bastante. Por lo que se ha hecho una buena estimación.

**CUESTIÓN 6:** APLICAR EL COMANDO EXPLAIN A UNA CONSULTA QUE OBTENGA LA INFORMACIÓN DE LOS PROYECTOS EN LOS CUALES EL INVESTIGADOR TRABAJA 10 HORAS. ¿SON CORRECTOS LOS RESULTADOS DEL COMANDO EXPLAIN? ¿POR QUÉ?

Si ejecutamos:

```
select count(*) from "investigadores_proyectos" where horas=10
```

Obtenemos 374471 registros. Si ejecutamos:

```
explain (format json) select * from "investigadores_proyectos" where horas=10
```

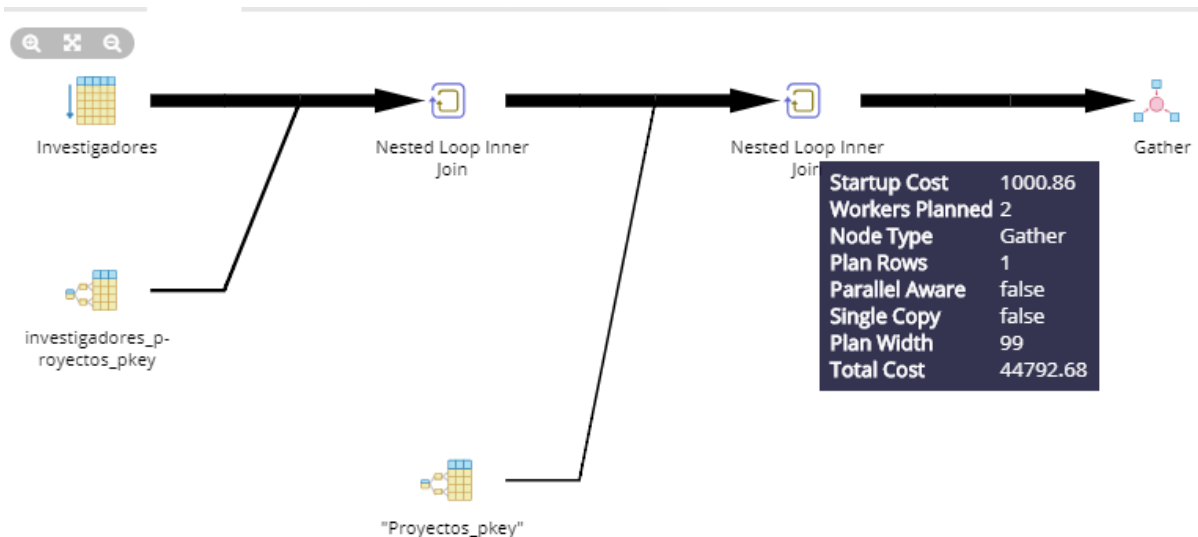
Podemos ver que hace una búsqueda secuencial en paralelo con 2 workers, estimando que va a recuperar un total de 377401 registros.

Esta aproximación es buena y se acerca mucho a la realidad, por lo que se podría decir que los resultados son correctos. Ha hecho una buena estimación.

**CUESTIÓN 7:** APLICAR EL COMANDO EXPLAIN A UNA CONSULTA QUE OBTENGA LA INFORMACIÓN DE LOS PROYECTOS QUE TIENEN UN COSTE MAYOR DE 50000, TIENEN EMPLEADOS DE SALARIO DE 24000 EUROS Y TRABAJAN MENOS DE 2 HORAS EN ELLOS. ¿SON CORRECTOS LOS RESULTADOS DEL COMANDO EXPLAIN? ¿POR QUÉ?

Ejecutando

```
explain (format json) select * from "Proyectos" inner join investigadores_proyectos on  
codigo_proyecto=numero_proyecto inner join "Investigadores" on numero_investigador=codigo_investigador  
where coste>50000 and salario=24000 and horas<2;
```



Vemos que la consulta devuelve 0 tuplas. Sin embargo, el planificador, basándose en las estadísticas, estima que devolverá 1 fila. Este dato no es correcto, pero se aproxima al valor real.

## CUESTIÓN 8: REPETIR LAS CUESTIONES 3,4,5,6 Y 7 CON MYSQL Y COMPARAR LOS RESULTADOS CON LOS OBTENIDOS POR POSTGRESQL.

Para la creación y consultas sobre la base de datos se ha usado la interfaz grafica del programa MySQL Workbench. Desde este programa se han creado las tablas tal y como vienen especificas por el enunciado, añadiendo las claves foráneas y la clave primaria compuesta.

Para la inserción de los datos se ejecuto la siguiente consulta sql, una vez modificado un parámetro en my.ini para permitir la inserción de datos desde archivos de texto.

```
load data infile 'datos_investigadores.csv' into table investigadores
fields terminated by ','
lines terminated by '\r\n'
```

Esta consulta se ejecuta para los 3 archivos, y así rellenamos las 3 tablas.

Se puede comentar que el import de los datos para las tablas investigador y proyecto tardaron prácticamente lo mismo en MySQL a lo que tardaron en PostgreSQL, pero que el import de los datos a la tabla investigadores\_proyectos tardó mucho más. La siguiente captura es de esta última inserción de datos:

#	Time	Action	Message	Duration / Fetch
1	21:53:50	load data infile 'datos_investigadores_proyectos.csv' into table investigadores_proyectos fields terminated by ',' lines terminated by '\r\n'	900000 row(s) affected Records: 900000 Deleted: 0 Skipped: 0 Warnings: 0	24936.953 sec

Esto es debido a las limitaciones de InnoDB con respecto a archivos csv no ordenados por primary key, a archivos que no caben en memoria, y a import de archivos sobre los que tiene que construir índices.

En cuanto a las estadísticas de las tablas, en MySQL estas estadísticas se pueden ver con el comando SQL:

```
SHOW TABLE STATUS FROM db_name LIKE 'table_name';
```

En cuanto a la tabla investigadores, tenemos lo siguiente:

Name	Engine	Version	Row_format	Rows	Avg_row_length	Data_length	Max_data_length	Index_length	Data_free	Auto_increment	Create_time	Update_time	Check_time	Collation
investigadores	InnoDB	10	Dynamic	21297	74	1589248	0	0	4194304	NULL	2018-11-30 18:01:46	2018-11-30 19:30:57	NULL	utf8mb4_0900_ai_ci

Para ver información sobre las columnas, ejecutamos el siguiente comando:

```
SELECT * FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'table_name'
AND table_schema = 'db_name'
```

Y nos devuelve información con este formato:

	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	CHARACTER_MAXIMUM_LENGTH	CHARACTER_OCTET_LENGTH	NUMERIC_PRECISION
def	laboratorio	investigadores	codigo_investigador	1	NULL	NO	decimal	10	10		10
def	laboratorio	investigadores	nombre	2	NULL	YES	text	65535	65535	65535	
def	laboratorio	investigadores	apellidos	3	NULL	YES	text	65535	65535	65535	
def	laboratorio	investigadores	salario	4	NULL	YES	decimal	10	10		10

El resto de las tablas devuelven información con el mismo formato. Se puede observar que la información mostrada por este comando es mucho menor que la que nos suministra postgres.

En cuanto al análisis de las consultas, para la primera se ejecuta la consulta:



```
select count(*) from investigadores where salario>95000
```

Esto devuelve lo mismo que postgres, que son 1588314 registros a devolver. Ahora ejecutamos la consulta que se nos pide con el explain.

```
explain select * from investigadores where salario>95000
```

Al ejecutar la consulta con el explain, una de las columnas que nos devuelve es "rows", que es el estimado de registros que va a tener que examinar. El valor de rows para esta consulta es de 2987423, lo cual es muy superior al numero de columnas que lee realmente, por lo que aproximación no es correcta.

En cuanto a la segunda consulta, si ejecutamos:

```
select count(*) from "investigadores_proyectos" where horas=10
```

Obtenemos 374471 columnas. Ahora ejecutamos la consulta que se nos pide con explain y el valor de rows para esta consulta es de 9238614, este número tampoco es correcto y se encuentra bastante lejos de la realidad. De hecho, el valor es más grande que el tamaño total de la tabla.

Para la ultima consulta, sabemos que el numero de registros que cumplen la condición es 0.

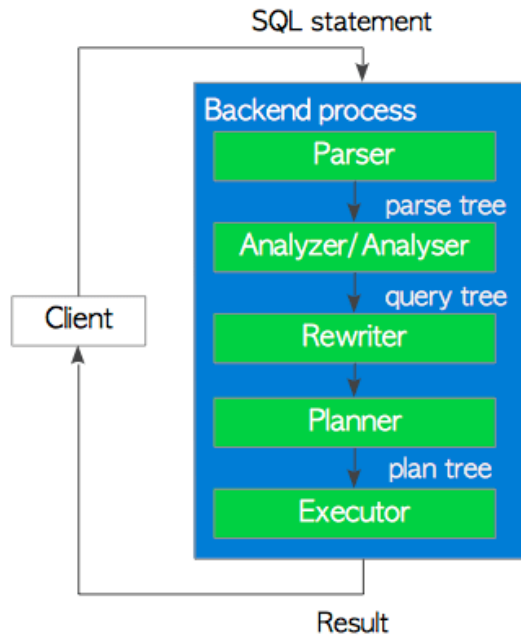
Al ejecutar la consulta con explain, obtenemos que el estimado de registros a recuperar es de 1, que era lo mismo que estimaba postgres, por lo que esta consulta si tiene un resultado correcto ya que se aproxima al valor real.

**CUESTIÓN 9:** DESCRIBIR EL SISTEMA DE PROCESAMIENTO DE CONSULTAS DE POSTGRESQL. ¿QUÉ ALGORITMOS DE PROCESAMIENTO DE CONSULTAS INCORPORA POSTGRESQL? DESCRIBIRLOS. ¿QUÉ PARÁMETROS SE PUEDEN CONFIGURAR DEL PROCESAMIENTO DE CONSULTAS? ¿PARA QUÉ SIRVEN?

El sistema de procesamiento de consultas de PostgreSQL se divide a su vez en 5 subsistemas:

1. Parser: genera un árbol de parseo de la consulta que se está intentado ejecutar.
2. Analizador: hace un análisis semántico del árbol de parseo generado en el paso anterior y lo transforma en un árbol de consulta.
3. Rewriter: se encarga de reescribir el árbol de consulta del paso anterior siguiendo una serie de reglas almacenadas en el sistema de postgres.
4. Planificador: con el resultado obtenido en el paso anterior, este subsistema se va a encargar de obtener el plan de ejecución óptimo para esta consulta.
5. Ejecutor: este subsistema se encarga de seguir el árbol generado en el paso anterior para hacer el acceso a las tablas e índices para recuperar los datos.

Se puede resumir con el siguiente diagrama:



Este sistema tiene en cuenta que se haya establecido una conexión con el servidor previa para poder realizar la consulta.

Los parámetros configurables son las reglas definidas que son analizadas en el subsistema Rewriter. Sirve para la definición de vistas sobre tablas de la base de datos.

**CUESTIÓN 10:** ¿TIENE POSTGRESQL UN OPTIMIZADOR DE CONSULTAS? SI ES ASÍ, ¿QUÉ TÉCNICA UTILIZA POSTGRESQL PARA OPTIMIZAR LAS CONSULTAS? DESCRIBIRLA. ¿QUÉ PARÁMETROS SE PUEDEN CONFIGURAR DEL OPTIMIZADOR DE CONSULTAS? ¿PARA QUÉ SIRVEN?

Como ya se adelanta en la cuestión anterior, PostgreSQL tiene un subsistema en el procesamiento de consultas que se encarga de optimizar los planes de ejecución de las consultas.

Para hacer esto, el planificador genera posibles planes de ejecución equivalentes para cada consulta y analiza el coste estimado de llevar a cabo cada uno de ellos. Tiene en cuenta el posible uso de índices sobre columnas de tablas, la memoria que tiene el sistema para realizar la consulta y las posibles formas de hacer reuniones entre las tablas de una consulta.

Si se produce una reunión de más de dos relaciones en la consulta, también tiene en cuenta el orden de dichas relaciones para descartar el mayor número de registros lo antes posible. Si hay demasiadas reuniones, el planificador ya no hará una búsqueda exhaustiva, sino que usará la heurística a través de un sistema que PostgreSQL llama GEQO (Genetic Query Optimizer).

Se pueden modificar parámetros relacionados con los Join, aumentando el `geqo_threshold` conseguimos que se haga el análisis exhaustivo del mejor orden de los joins cuando usamos muchas reuniones, pero ralentizamos la velocidad de la consulta exponencialmente.

**CUESTIÓN 11:** 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?

Para cargar los datos, vamos ejecutando la sentencia:

```
BEGIN;
ALTER TABLE "Tabla" DISABLE TRIGGER ALL;
COPY "Tabla" from 'C:\Users\tabla.csv' (format csv);
ALTER TABLE "Tabla" ENABLE TRIGGER ALL;
COMMIT;
```

Sustituyendo Tabla por la tabla a rellenar, y comentando y descomentando los ALTER TABLE con los triggers para ejecutarla con integridad referencial activada y desactivada, respectivamente.

Obtenemos los siguientes tiempos:

Tabla	Tiempo sin integridad	Tiempo con integridad
Grupo	0,692 s	0,701 s
Musicos	9,647 s	18,467 s
Discos	4,92 s	12,864 s
Canciones	49,487 s	2 m 55 s
Conciertos	0,388 s	0,367 s
Entradas	1 m 29 s	5 m 1 s
Grupos_Tocan_Concierto	8,259 s	1 m 4 s

Las tablas se han cargado en el orden en que aparecen en la tabla anterior. Esto es así debido a que hay tablas que dependen de que los valores de sus columnas existan en otras tablas (foreign key), se tiene que hacer en este orden o parecido. Las tablas grupo y/o conciertos deben ser las primeras, discos tiene que ir antes que canciones, conciertos tiene que ir antes que entradas y grupo debe ir antes que músicos. La tabla intermedia entre Grupo y Concierto debe ir después de que estas 2 se hayan insertado ya.

La diferencia de tiempo entre sin integridad y con integridad se debe a que cuando la integridad esta activada, para cada valor que intenta insertar en la tabla y es una foreign key, tiene que comprobar que este valor existe en las otras tablas, lo cual retrasa mucho el proceso de inserción.

El tiempo en el log es de milisegundos menor al tiempo que muestra el pgAdmin. Esto se puede deber a que primero graba en el log cuando acaba la operación y después de hacer esto manda una señal de finalizado al cliente, y en ese momento es cuando se marca el final de la consulta.

**CUESTIÓN 12:** REALIZAR LA CARGA MASIVA DE LOS MISMOS DATOS CON MYSQL, COMPLETAR LA TABLA SIGUIENTE, COMENTAR EL PROCESO SEGUIDO EN LA CARGA DE DATOS Y COMPARAR LOS RESULTADOS CON LOS OBTENIDOS POR POSTGRESQL.

Tabla	Tiempo sin integridad	Tiempo con integridad
Grupo	1,859 s	1,687 s
Musicos	30,125 s	34,297 s
Discos	16,156 s	24,016 s
Canciones	3 m 26 s	5 m 39 s
Conciertos	0,844 s	1,016 s
Entradas	5 m 28 s	8 m 12 s
Grupos_Tocan_Concierto	48,235 s	1 m 34 s

Hay que tener en cuenta que las inserciones en las tablas que contienen datos de tipo date, tienen una llamada a la función de formateo de fechas para que puedan ser almacenadas por MySQL, por lo que van a ser más lentos desde un primer momento.

Como se puede ver, aun teniendo en cuenta el proceso descrito en el paso anterior, los tiempos suelen ser significativamente peores en MySQL.

Para optimizar la velocidad de carga de datos, se han modificado parámetros en el archivo my.ini que tienen que ver con la cantidad de memoria asignada al buffer de InnoDB y al tamaño del log que puede generar también.

Se ha ejecutado el siguiente comando para la inserción de los datos, cambiando el nombre de las tablas y el nombre del archivo a cargar:

```
start transaction;
set foreign_key_checks=0;
load data infile 'canciones.csv' into table canciones
fields terminated by ','
lines terminated by '\n'
(Codigo_cancion,Nombre,Compositor,@Fecha_grabacion,Duracion,Codigo_disco_Discos)
set Fecha_grabacion = STR_TO_DATE(@Fecha_grabacion, '%d-%m-%Y');
set foreign_key_checks=1;
commit;
```

Las líneas que hacen que no se haga el check de las foreign keys se han quitado para las inserciones que se hiciesen con integridad referencial. A parte de esto, la parte de la transformación del formato de la fecha se ha quitado en aquellas tablas donde esto no fuese necesario porque no tuviesen columnas de tipo date.

**CUESTIÓN 13: REALIZAR UNA CONSULTA SQL QUE MUESTRE LOS NOMBRES DE LOS MÚSICOS JUNTO CON EL GRUPO AL QUE PERTENECEN, QUE CUMPLEN QUE REALIZAN CONCIERTOS EN ESPAÑA TENIENDO ENTRADAS CUYO PRECIO VARÍA ENTRE 20 Y 50 EUROS, Y ADEMÁS TIENEN DISCOS DE GÉNERO 'ROCK' CON ALGUNA CANCIÓN DE MÁS DE 3 MINUTOS, Y SON GRUPOS DE MÁS DE 3 COMPONENTES. DIBUJAR EL DIAGRAMA 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.**

Este es el código de la consulta creada:

```
select distinct "Músicos"."Nombre", "Músicos"."Codigo_grupo_Grupo"
from "Músicos" inner join "Grupo" on "Músicos"."Codigo_grupo_Grupo" = "Grupo"."Codigo_grupo"
inner      join      "Grupos_Tocan_Conciertos"      on      "Grupo"."Codigo_grupo"      =
"Grupos_Tocan_Conciertos"."Codigo_grupo_Grupo"
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 "Discos" on "Grupo"."Codigo_grupo" = "Discos"."Codigo_grupo_Grupo"
inner join "Canciones" on "Discos"."Codigo_disco" = "Canciones"."Codigo_cancion"
where "Conciertos"."Pais" = 'Spain'
and ("Entradas"."Precio"::numeric) > 20
and "Entradas"."Precio"::numeric < 50
and "Discos"."Genero" = 'rock'
and "Canciones"."Duracion" > '03:00:00'
and "Músicos"."Codigo_grupo_Grupo" in (select "Codigo_grupo_Grupo"
from "Músicos"
group by "Codigo_grupo_Grupo"
having(count(codigo_musico)>3))
```

De cualquier forma, se entrega el archivo .sql con esta consulta para su mejor visualización y prueba si fuese necesario.

DashboardPropertiesSQLStatisticsDependenciesDependentsQuery - new\_database on postgres@PostgreSQL 10 \*

new\_database on postgres@PostgreSQL 10

```
1 select distinct "Músicos"."Nombre", "Músicos"."Codigo_grupo_Grupo"
2 from "Músicos" inner join "Grupo" on "Músicos"."Codigo_grupo_Grupo" = "Grupo"."Codigo_grupo"
3 inner join "Grupos_Tocan_Conciertos" on "Grupo"."Codigo_grupo" = "Grupos_Tocan_Conciertos"."Codigo_grupo_Grupo"
4 inner join "Conciertos" on "Grupos_Tocan_Conciertos"."Codigo_concierto_Conciertos" = "Conciertos"."Codigo_concierto"
5 inner join "Entradas" on "Conciertos"."Codigo_concierto" = "Entradas"."Codigo_concierto_Conciertos"
6 inner join "Discos" on "Grupo"."Codigo_grupo" = "Discos"."Codigo_grupo_Grupo"
7 inner join "Canciones" on "Discos"."Codigo_disco" = "Canciones"."Codigo_cancion"
8 where "Conciertos"."Pais" = 'Spain'
9 and ("Entradas"."Precio"::numeric) > 20
10 and "Entradas"."Precio"::numeric < 50
11 and "Discos"."Genero" = 'rock'
12 and "Canciones"."Duracion" > '03:00:00'
13 and "Músicos"."Codigo_grupo_Grupo" in (select "Codigo_grupo_Grupo"
14 from "Músicos"
15 group by "Codigo_grupo_Grupo"
16 having(count(codigo_musico)>3))
```

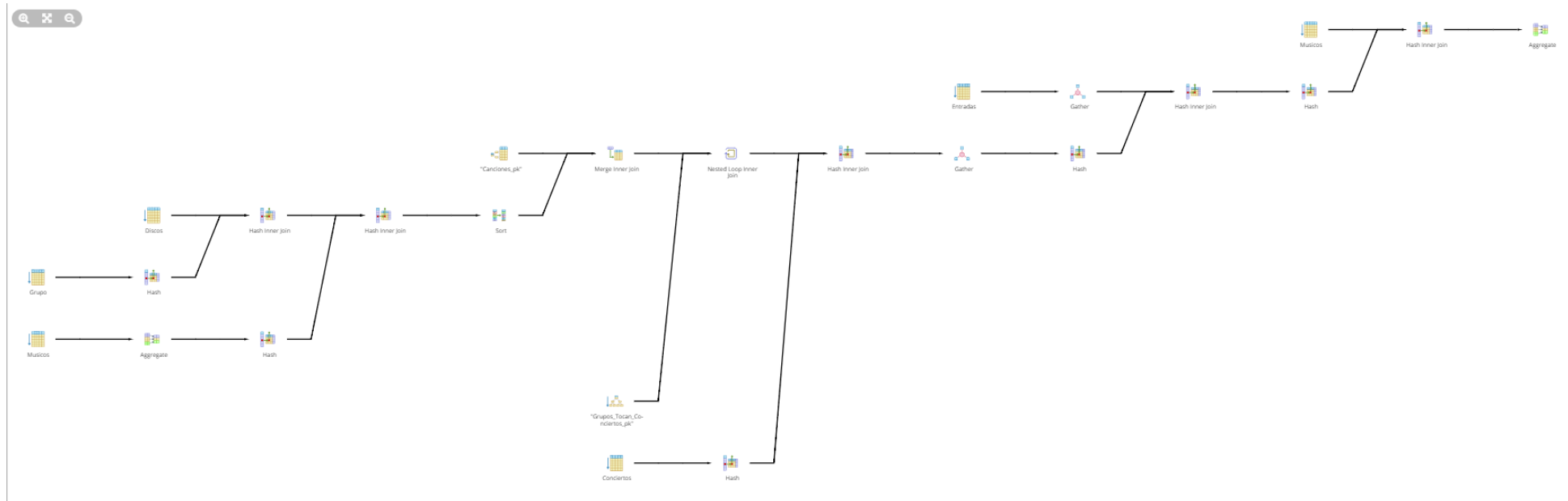
Data OutputExplainMessagesNotificationsQuery History

Successfully run. Total query runtime: 15 secs 91 msec.  
95365 rows affected.

La consulta tarda un total de 15 s y 91 msec en ejecutarse y devuelve un total de 95365 registros .

	Nombre text	Codigo_grupo_Grupo integer
1	nombre8...	180885
2	nombre1...	148630
3	nombre2...	119987
4	nombre7...	17498
5	nombre5...	136069
6	nombre1...	114666
7	nombre5...	117787
8	nombre9...	86688
9	nombre7...	46766
10	nombre1...	65592
11	nombre1...	87108
12	nombre2...	130416
13	nombre2...	155863
14	nombre7...	174483
15	nombre2...	107520
16	nombre6...	130120

Ejecutando el comando explain con formato json, postgre nos muestra el siguiente plan de ejecución:





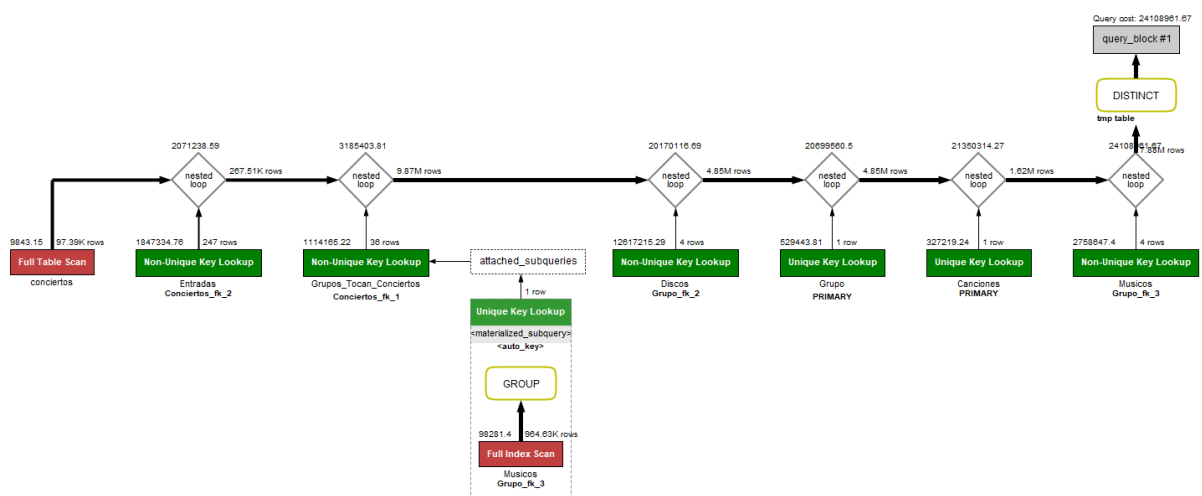


Como se puede apreciar el orden de los joins es diferente, a parte de esto, en vez de hacer una intersección entre el resultado de la consulta y la segunda consulta con la agrupación, postgres ha decidido hacer todo en la misma consulta leyendo la tabla músicos dos veces. Además, postgres decide ordenar la tabla en un punto de la consulta para realizar un merge join. Postgre utiliza funciones de agregado y ejecución en paralelo lanzando 2 workers para dos de las reuniones.

**CUESTIÓN 14: REPETIR LA CUESTIÓN 13 USANDO MYSQL. COMPARAR LOS RESULTADOS ENTRE AMBOS SISTEMAS GESTORES DE BASES DE DATOS.**

Corriendo el mismo script SQL pero modificando la sintaxis para que sea válida para MySQL tenemos que nos devuelve 94257 registros y que ha tardado 1 minuto y 47 segundos en ejecutar la consulta.

Este seria el plan de ejecución que ha desarrollado MySQL:



Como diferencia se puede destacar que MySQL no hace reordenaciones de la tabla mientras esta haciendo la consulta, y que usa bucles anidados para todas las reuniones. El orden de los joins es diferente, aparte de que Postgre no realizaba una subconsulta si no que hacia un join con la tabla Músicos dos veces.

El resultado en cuanto a rendimiento ha sido peor en MySQL a lo que fue en PostgreSQL, teniendo configurado para tener más memoria, por lo que se puede suponer que PostgreSQL realiza un mejor proceso de optimización de consultas que MySQL con InnoDB.

**CUESTIÓN 15: 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? EJECUTAR LA NUEVA CONSULTA Y EXPLICAR LOS RESULTADOS. DIBUJAR UN DIAGRAMA CON EL NUEVO RESULTADO DEL COMANDO EXPLAIN EN FORMA DE ÁRBOL DE ALGEBRA RELACIONAL.**

Se podrían construir índices sobre los campos que van a ser usados durante la consulta. En cuanto al código SQL, no se ve forma de modificarlo para mejorar el rendimiento de esta consulta.

Se crean los siguientes índices:

```

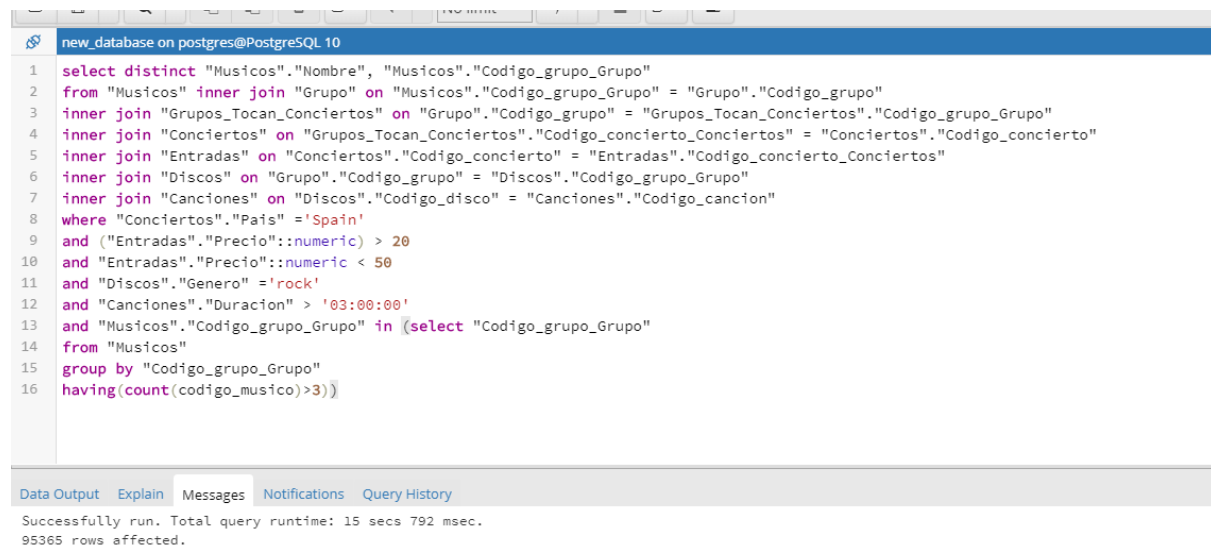
BEGIN;
create index "btree_nombre" on "Músicos"("Nombre");
create index "btree_cod_grupo" on "Músicos"("Codigo_grupo_Grupo");
create index "btree_cod_grupo_Grupo" on "Grupo"("Codigo_grupo");
create index "btree_cod_grupo_tocan_conciertos" on "Grupos_Tocan_Conciertos"("Codigo_grupo_Grupo");
create index "btree_cod_concierto_tocan_conciertos" on
"Grupos_Tocan_Conciertos"("Codigo_concierto_Conciertos");
create index "btree_cod_concierto_conciertos" on "Conciertos"("Codigo_concierto");
create index "btree_pais_conciertos" on "Conciertos"("Pais");
create index "btree_cod_concierto_entradas" on "Entradas"("Codigo_concierto_Conciertos");
create index "btree_precio_entradas" on "Entradas"("Precio");
create index "btree_cod_grupo_discos" on "Discos"("Codigo_grupo_Grupo");
create index "btree_genero_discos" on "Discos"("Genero");
create index "btree_cod_discos_canciones" on "Canciones"("Codigo_disco_Discos");
create index "btree_duracion_canciones" on "Canciones"("Duracion");
COMMIT;

```

Al ejecutar el commit que crea los índices, se tarda:

Query returned successfully in 2 min 11 secs.

Ejecutando la misma consulta:



```

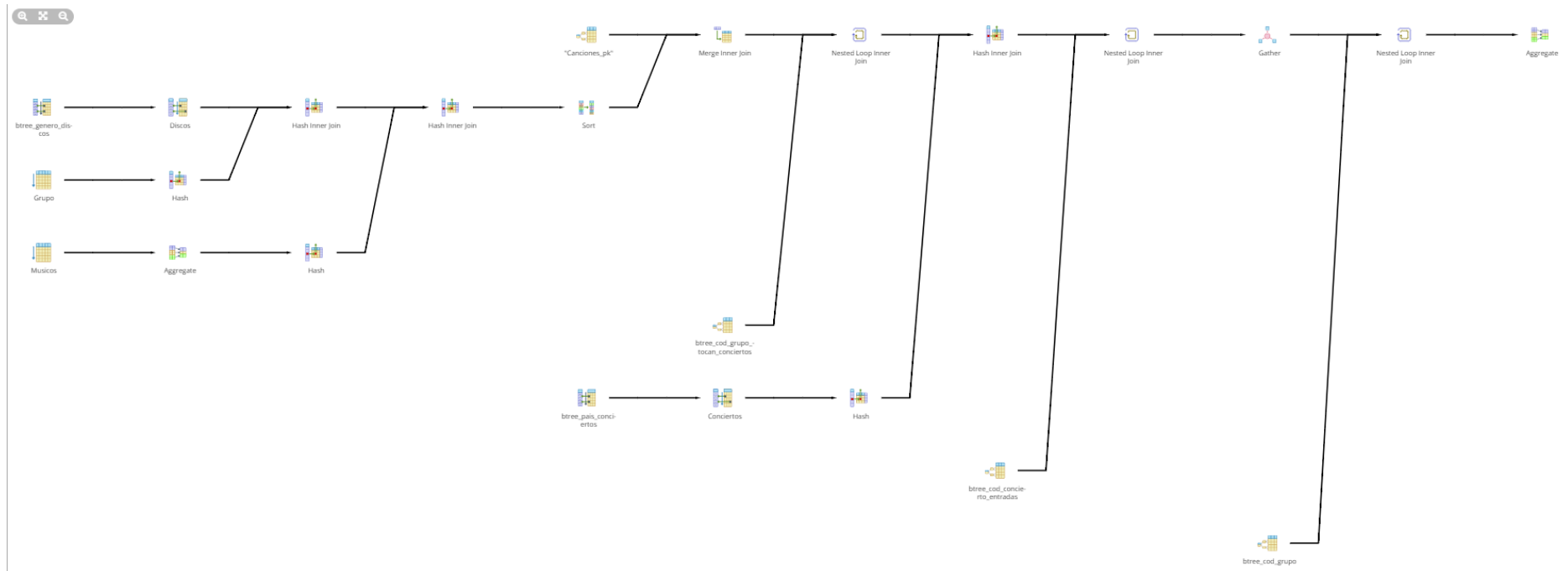
new_database on postgres@PostgreSQL 10
1 select distinct "Músicos"."Nombre", "Músicos"."Codigo_grupo_Grupo"
2 from "Músicos" inner join "Grupo" on "Músicos"."Codigo_grupo_Grupo" = "Grupo"."Codigo_grupo"
3 inner join "Grupos_Tocan_Conciertos" on "Grupo"."Codigo_grupo" = "Grupos_Tocan_Conciertos"."Codigo_grupo_Grupo"
4 inner join "Conciertos" on "Grupos_Tocan_Conciertos"."Codigo_concierto_Conciertos" = "Conciertos"."Codigo_concierto"
5 inner join "Entradas" on "Conciertos"."Codigo_concierto" = "Entradas"."Codigo_concierto_Conciertos"
6 inner join "Discos" on "Grupo"."Codigo_grupo" = "Discos"."Codigo_grupo_Grupo"
7 inner join "Canciones" on "Discos"."Codigo_disco" = "Canciones"."Codigo_cancion"
8 where "Conciertos"."Pais" = 'Spain'
9 and ("Entradas"."Precio"::numeric) > 20
10 and "Entradas"."Precio"::numeric < 50
11 and "Discos"."Genero" = 'rock'
12 and "Canciones"."Duracion" > '03:00:00'
13 and "Músicos"."Codigo_grupo_Grupo" in ((select "Codigo_grupo_Grupo"
14 from "Músicos"
15 group by "Codigo_grupo_Grupo"
16 having(count(codigo_musico)>3))

```

Data Output Explain Messages Notifications Query History

Successfully run. Total query runtime: 15 secs 792 msec.  
95365 rows affected.

Se ejecuta la consulta con el comando explain y el formato json y pgadmin nos devuelve esto:

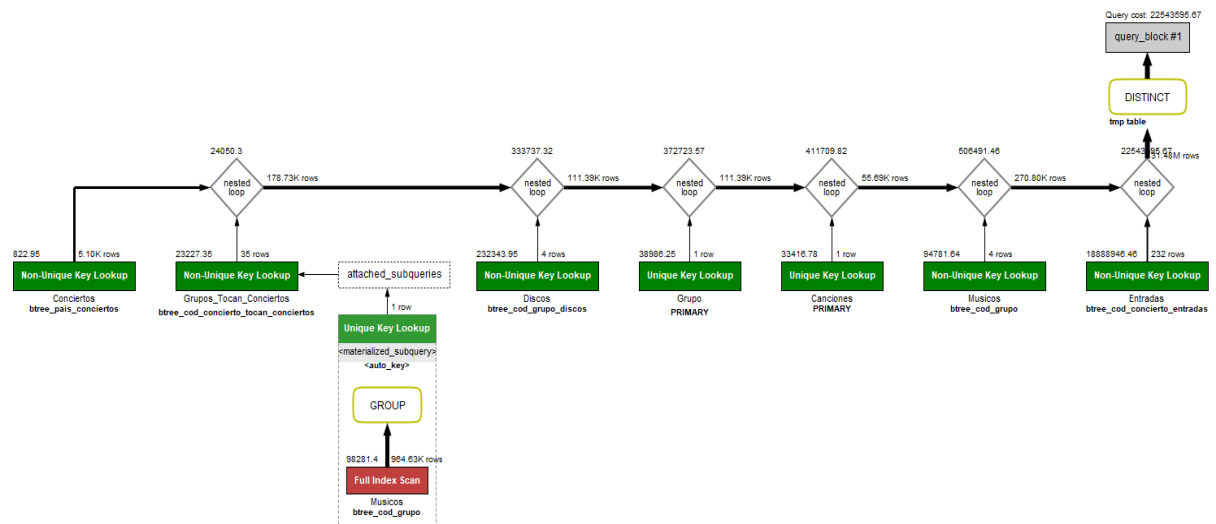


Como se puede ver postgre ha decidido usar alguno de los índices que se han creado, pero no todos, ya que en cuando tiene que hacer lectura de muchos datos en una tabla, le viene mejor usar la lectura secuencial.

El orden en el que se realizan alguna de las reuniones ha cambiado también.

**CUESTIÓN 16:** USANDO MYSQL, REPITA LA CUESTIÓN 15 Y COMPARE LOS RESULTADOS OBTENIDOS CON POSTGRESQL.

Se crean exactamente los mismos índices y se ejecuta la consulta con el comando explain:



Se puede decir que el tiempo de la consulta ha sido mucho menor. Al igual que pasaba con postgre, MySQL ha usado los nuevos índices que se han creado cuando lo ha visto conveniente, pero no siempre. También se ha modificado el orden en el que se hacían las reuniones.

**CUESTIÓN 17:** USANDO POSTGRESQL, BORRE EL 50% DE LOS DATOS (APROXIMADAMENTE). ¿CUÁL HA SIDO EL PROCESO SEGUIDO? ¿Y EL TIEMPO EMPLEADO EN EL BORRADO? OBTENGA EL PLAN DE EJECUCIÓN DE LA CUESTIÓN 13. DIBUJAR UN DIAGRAMA CON EL NUEVO RESULTADO DEL COMANDO EXPLAIN EN FORMA DE ÁRBOL DE ALGEBRA RELACIONAL. COMPARAR CON LOS RESULTADOS ANTERIORES.

Para que la integridad referencial no suponga problemas, desactivamos primero los triggers utilizando:

`Alter table nombre_tabla disable trigger all`

Tras esto, para eliminar tablas utilizaremos la sentencia:

`Delete from nombre_tabla`

`Where nombre_tabla.codigo > (nº de codigos/2)`

Utilizando el comando EXPLAIN para eliminar la mitad de los grupos se obtiene el siguiente resultado:

```

explain (FORMAT json)

delete from "Grupo"

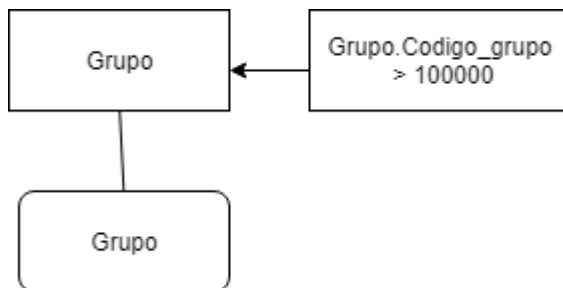
WHERE "Grupo"."Codigo_grupo" > 100000

```



QUERY PLAN	
	text
1	Delete on "Grupo" (cost=0.42..3837.82 rows=99623 width=6)
2	-> Index Scan using "Grupo_pk" on "Grupo" (cost=0.42..3837.82 rows=99623 width=6)
3	Index Cond: ("Codigo_grupo" > 100000)

El diagrama quedaría así:



La siguiente tabla refleja los tiempos que se ha tardado en ejecutar cada sentencia eliminando la mitad de filas según cada tabla:

Tabla	Sentencia SQL	Tiempo tomado
<b>Grupo(200K)</b>	BEGIN; Alter table "Grupo" disable trigger all; delete from "Grupo" WHERE "Grupo"."Codigo_grupo " > 100000; Alter table "Grupo" enable trigger all; COMMIT;	343 msec
<b>Músicos(1M)</b>	BEGIN; Alter table "Musicos" disable trigger all; delete from "Musicos" WHERE "Musicos"."codigo_musico" > 500000; Alter table "Musicos" enable trigger all; COMMIT;	1 sec 926 msec
<b>Entradas(24M)</b>	BEGIN; Alter table "Entradas" disable trigger all; delete from "Entradas" WHERE "Entradas"."Codigo_entrada" > 12000000; Alter table "Entradas" enable trigger all; COMMIT;	47 secs 556 msec
<b>Grupos_Tocan_Conciertos</b>	BEGIN; Alter table "Grupos_Tocan_Conciertos" disable trigger all; delete from "Grupos_Tocan_Conciertos" WHERE "Grupos_Tocan_Conciertos"."Codigo_grupo_Grupo"	3 secs 77 msec

	> 100000; Alter table "Grupos_Tocan_Conciertos" enable trigger all; COMMIT;	
<b>Discos(1M)</b>	BEGIN; Alter table "Discos" disable trigger all; delete from "Discos" WHERE "Discos"."Codigo_disco" > 500000; Alter table "Discos" enable trigger all; COMMIT;	1 secs 498 msec
<b>Conciertos(100K)</b>	BEGIN; Alter table "Conciertos" disable trigger all; delete from "Conciertos" WHERE "Conciertos"."Codigo_concierto" > 50000; Alter table "Conciertos" enable trigger all; COMMIT;	202 msec
<b>Canciones (12M)</b>	BEGIN; Alter table "Canciones" disable trigger all; delete from "Canciones" WHERE "Canciones"."Codigo_cancion" > 6000000; Alter table "Canciones" enable trigger all; COMMIT;	20 secs 525 msec

Si ejecutamos la consulta de la cuestión 13 de nuevo, obtenemos los siguientes resultados:

The screenshot shows a PostgreSQL query editor interface. The query is as follows:

```

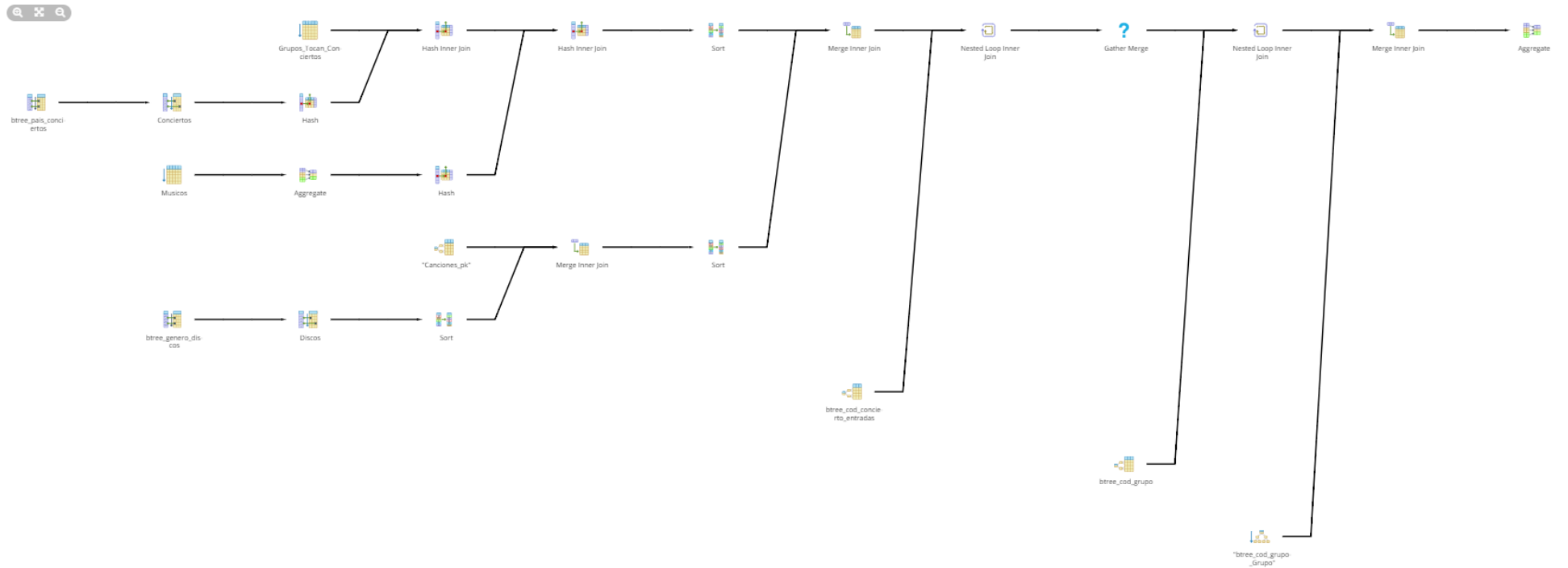
1 select distinct "Músicos"."Nombre", "Músicos"."Codigo_grupo_grupo"
2 from "Músicos" inner join "Grupo" on "Músicos"."Codigo_grupo_grupo" = "Grupo"."Codigo_grupo"
3 inner join "Grupos_Tocan_Conciertos" on "Grupo"."Codigo_grupo" = "Grupos_Tocan_Conciertos"."Codigo_grupo_grupo"
4 inner join "Conciertos" on "Grupos_Tocan_Conciertos"."Codigo_concierto_conciertos" = "Conciertos"."Codigo_concierto"
5 inner join "Entradas" on "Conciertos"."Codigo_concierto" = "Entradas"."Codigo_concierto_Conciertos"
6 inner join "Discos" on "Grupo"."Codigo_grupo" = "Discos"."Codigo_grupo_grupo"
7 inner join "Canciones" on "Discos"."Codigo_disco" = "Canciones"."Codigo_cancion"
8 where "Conciertos"."Pais" = 'Spain'
9 and ("Entradas"."Precio"::numeric > 20
10 and "Entradas"."Precio"::numeric < 50 and "Discos"."Genero" = 'rock'
11 and "Canciones"."Duracion" > '03:00:00'
12 and "Músicos"."Codigo_grupo_grupo" in (select "Codigo_grupo_grupo"
13 from "Músicos"
14 group by "Codigo_grupo_grupo"
15 having(count(codigo_musico)>3))

```

Below the query, the execution status is shown: "Successfully run. Total query runtime: 18 secs 791 msec. 3861 rows affected."

Vemos que tarda más tiempo que antes, lo cual podría sorprender teniendo en cuenta que ahora solo están la mitad de los datos. Sin embargo, podemos entender que esto es así debido a que, aunque hayamos borrado los datos, están como tuplas muertas en los archivos de POSTGRES. Los índices no han sido actualizados y las estadísticas tampoco, por lo que al sistema le cuesta más esfuerzo devolver el resultado de la consulta.

Ejecutando explain obtenemos el siguiente diagrama:



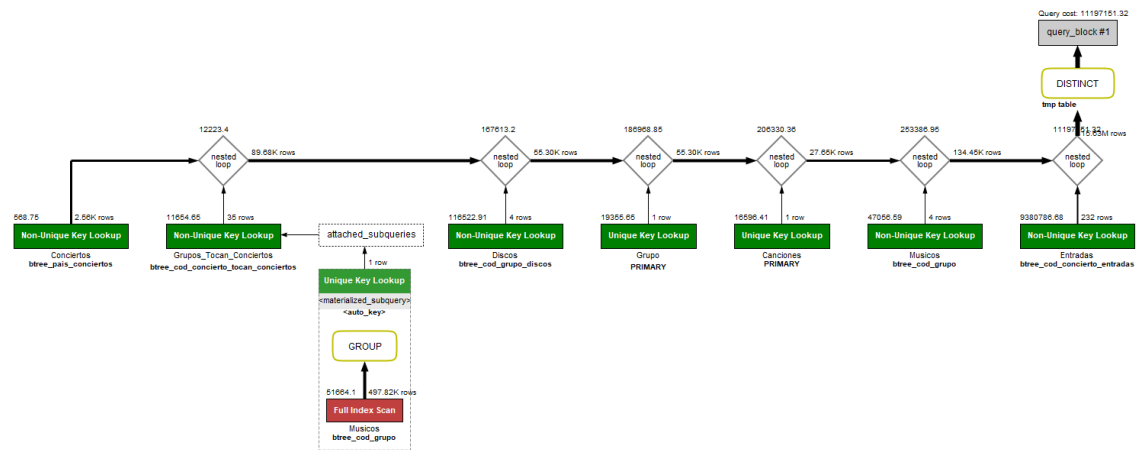
**CUESTIÓN 18:** REPITA LA CUESTIÓN 17 CON MYSQL Y COMPARE LOS RESULTADOS CON POSTGRESQL.

Se sigue el mismo principio para el borrado en MySQL.

Tabla	Sentencia SQL	Tiempo tomado
<b>Grupo(200K)</b>	start transaction; set foreign_key_checks=0; delete from grupo where grupo.Codigo_grupo > 100000; set foreign_key_checks=1; commit;	1,328 s
<b>Músicos(1M)</b>	start transaction; set foreign_key_checks=0; delete from musicos where musicos.codigo_musico > 500000; set foreign_key_checks=1; commit;	34,054 s
<b>Entradas(24M)</b>	start transaction; set foreign_key_checks=0; delete from entradas where entradas.Codigo_entrada > 12000000; set foreign_key_checks=1; commit;	2 m 51 s
<b>Grupos_Tocan_Conciertos</b>	start transaction; set foreign_key_checks=0; delete from grupos_tocan_conciertos where grupos_tocan_conciertos.Codigo_grupo_grupo > 100000; set foreign_key_checks=1; commit;	20,453 s
<b>Discos(1M)</b>	start transaction; set foreign_key_checks=0; delete from discos where discos.Codigo_disco > 500000; set foreign_key_checks=1; commit;	8,531 s
<b>Conciertos(100K)</b>	start transaction; set foreign_key_checks=0; delete from conciertos where conciertos.Codigo_concierto > 50000; set foreign_key_checks=1; commit;	0,875 s
<b>Canciones (12M)</b>	start transaction; set foreign_key_checks=0; delete from canciones where canciones.Codigo_cancion > 6000000; set foreign_key_checks=1; commit;	1 m 31 s

Después del borrado se corre la consulta y MySQL nos muestra este plan de ejecución:





El plan de ejecución es exactamente igual al generado en la pregunta 16 una vez hechos los índices, pero ajustando la estimación de registros a recuperar y el coste de la consulta.

### CUESTIÓN 19: ¿QUÉ TÉCNICAS DE MANTENIMIENTO DE LA BD PROPONDRÍA PARA MEJORAR LOS RESULTADOS DE DICHO PLAN SIN MODIFICAR EL CÓDIGO DE LA CONSULTA? ¿POR QUÉ?

Para mejorar los resultados se podrían ejecutar las siguientes acciones:

- Usar VACUUM: Esto permite eliminar tuplas muertas. Así eliminamos los registros definitivamente, de forma que reducimos el tamaño del archivo y por tanto el tiempo de acceso a los registros de la consulta.
- Usar REINDEX: Esto permite actualizar los índices. Debido a que hemos eliminado registros, debemos reconstruir los índices ya que también estarán desactualizados.
- Usar ANALYZE: Esto permite actualizar las estadísticas por cada tabla. Al actualizar las estadísticas, la planificación de consultas se hará sobre la nueva cantidad de datos, de forma que las consultas se optimizarán más adecuadamente que si no actualizamos los datos.

**CUESTIÓN 20:** 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.

**Propuesta 1:** ejecutamos VACUUM y ANALYZE en cada tabla:

**VACUUM ANALYZE** nombre\_tabla

Se ejecutan las siguientes instrucciones SQL:

```
VACUUM ANALYZE "Grupos";  
VACUUM ANALYZE "Musicos";  
VACUUM ANALYZE "Entradas";  
VACUUM ANALYZE "Discos";  
VACUUM ANALYZE "Canciones";  
VACUUM ANALYZE "Conciertos";  
VACUUM ANALYZE "Grupos_Tocan_Conciertos";
```

**Propuesta 2:** reconstruimos los índices usando reindex:

**REINDEX INDEX** nombre\_indice

Ejecutándolo como una transacción:

```
BEGIN;
reindex index "btree_nombre";
reindex index "btree_cod_grupo";
reindex index "btree_cod_grupo_grupo";
reindex index "btree_cod_grupo_tocan_conciertos";
reindex index "btree_cod_concierto_tocan_conciertos";
reindex index "btree_cod_concierto_conciertos";
reindex index "btree_pais_conciertos";
reindex index "btree_cod_concierto_entradas";
reindex index "btree_precio_entradas";
reindex index "btree_cod_grupo_discos";
reindex index "btree_genero_discos";
reindex index "btree_cod_discos_canciones";
reindex index "btree_duracion_canciones";
COMMIT;
```

Ejecutando obtenemos lo siguiente:

Copy All

```
BEGIN;
reindex index "btree_nombre";
reindex index "btree_cod_grupo";
reindex index "btree_cod_grupo_grupo";
reindex index "btree_cod_grupo_tocan_conciertos";
reindex index "btree_cod_concierto_tocan_conciertos";
reindex index "btree_cod_concierto_conciertos";
reindex index "btree_pais_conciertos";
reindex index "btree_cod_concierto_entradas";
reindex index "btree_precio_entradas";
reindex index "btree_cod_grupo_discos";
reindex index "btree_genero_discos";
reindex index "btree_cod_discos_canciones";
reindex index "btree_duracion_canciones";
COMMIT;
```

Messages

COMMIT

Query returned successfully in 57 secs 573 msec.

La transacción toma 57 segundos y 573 msec. Podemos ver que en la cuestión 14, donde creábamos los índices, se tardaron 120 segundos aproximadamente en generarse todos los índices, y ahora se ha tardado la mitad, lo cual es congruente debido a que tenemos aproximadamente la mitad de datos.

Ejecutando la consulta del ejercicio 13 obtenemos los siguientes resultados:

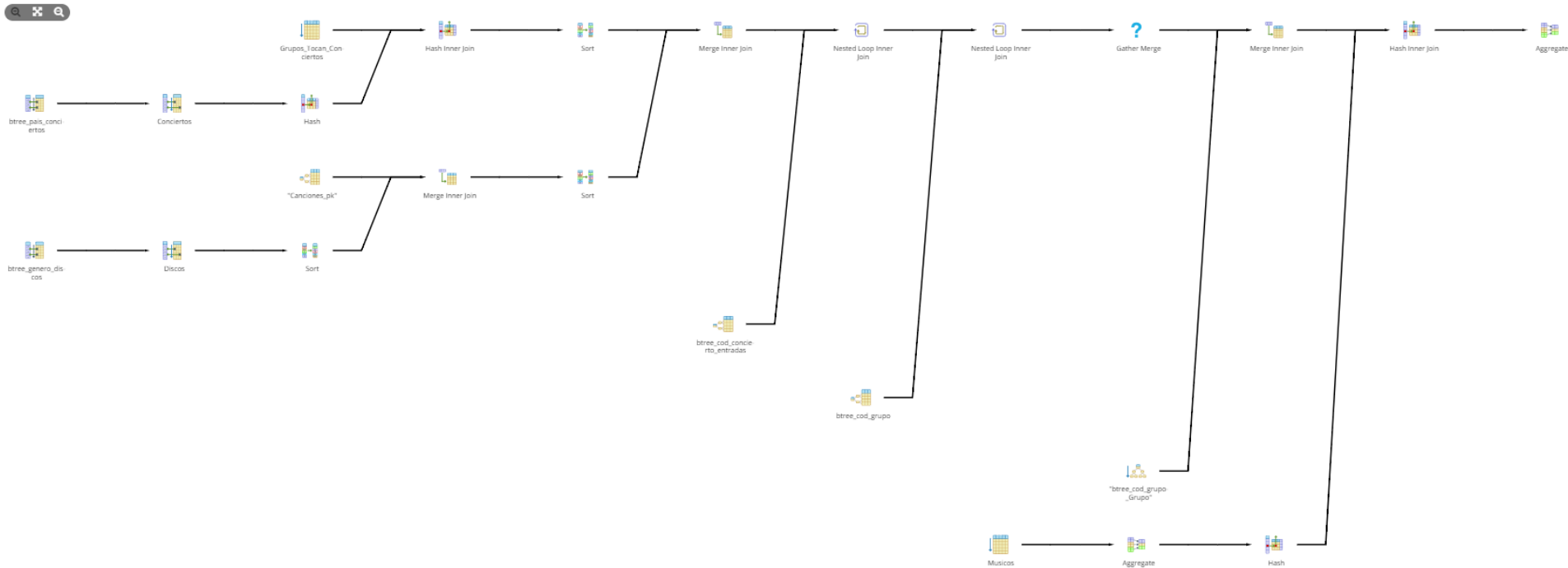
```
new_database on postgres@PostgreSQL 10

1 select distinct "Músicos"."Nombre", "Músicos"."Codigo_grupo_grupo"
2 from "Músicos" inner join "Grupo" on "Músicos"."Codigo_grupo_grupo" = "Grupo"."Codigo_grupo"
3 inner join "Grupos_Tocan_Conciertos" on "Grupo"."Codigo_grupo" = "Grupos_Tocan_Conciertos"."Codigo_grupo_grupo"
4 inner join "Conciertos" on "Grupos_Tocan_Conciertos"."Codigo_concierto_Conciertos" = "Conciertos"."Codigo_concierto"
5 inner join "Entradas" on "Conciertos"."Codigo_concierto" = "Entradas"."Codigo_concierto_Conciertos"
6 inner join "Discos" on "Grupo"."Codigo_grupo" = "Discos"."Codigo_grupo_grupo"
7 inner join "Canciones" on "Discos"."Codigo_disco" = "Canciones"."Codigo_cancion"
8 where "Conciertos"."Pais" = 'Spain'
9 and ("Entradas"."Precio)::numeric > 20
10 and "Entradas"."Precio)::numeric < 50
11 and "Discos"."Genero" = 'rock'
12 and "Canciones"."Duracion" > '03:00:00'
13 and "Músicos"."Codigo_grupo_grupo" in (select "Codigo_grupo_grupo"
14 from "Músicos"
15 group by "Codigo_grupo_grupo"
16 having(count(codigo_musico)>3))

Data Output Explain Messages Notifications Query History
Successfully run. Total query runtime: 2 secs 478 msec.
3861 rows affected.
```

Vemos que el tiempo es significativamente menor que antes de optimizar.

Ejecutando explain sobre la consulta, obtenemos el siguiente árbol:



**CUESTIÓN 21:** REPITA LA CUESTIÓN 20 CON MYSQL. COMPARE LOS RESULTADOS CON LOS OBTENIDOS PARA POSTGRESQL.

En MySQL no hay un comando Vacuum como tal, pero la sentencia SQL OPTIMIZE TABLE hace las veces de Vacuum.

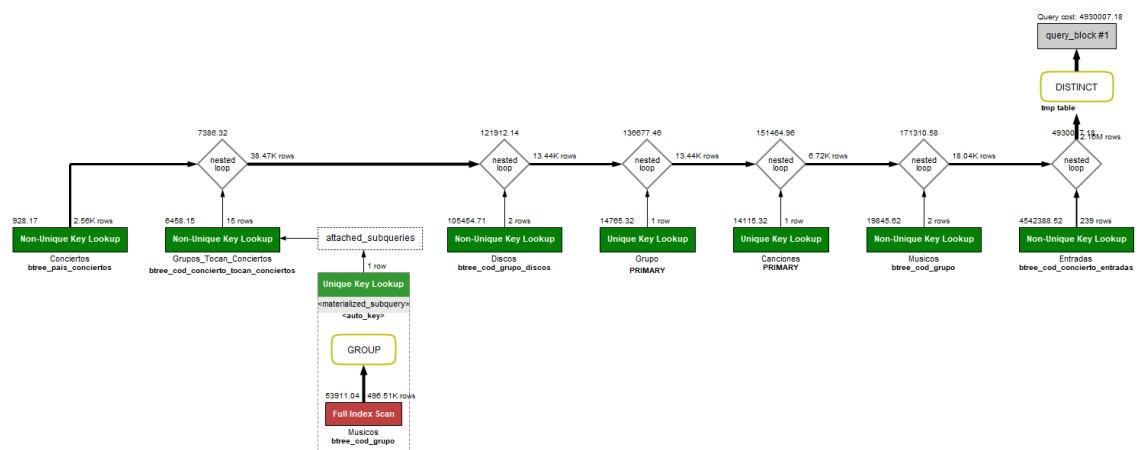
Este comando reorganiza el almacenamiento físico de la tabla y los índices asociados a la misma. Además, ejecuta un analyze sobre la tabla, lo cual actualiza las estadísticas.

Se ejecuta el siguiente script:

```
START TRANSACTION;
OPTIMIZE TABLE canciones;
OPTIMIZE TABLE conciertos;
OPTIMIZE TABLE discos;
OPTIMIZE TABLE entradas;
OPTIMIZE TABLE grupo;
OPTIMIZE TABLE grupos_tocan_conciertos;
OPTIMIZE TABLE musicos;
COMMIT;
```

Puesto que estamos usando InnoDB como motor, el comando optimize table se traduce internamente a un RECREATE TABLE y un ANALYZE, lo cual tiene un efecto similar al comando original.

Tras esto, se ejecuta la consulta y obtenemos el siguiente plan de ejecución:



El plan de ejecución es el mismo que en preguntas anteriores, pero la estimación de registros y costes ha bajado considerablemente.

**CUESTIÓN 22:** 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.

El directorio de archivos de log lo podemos encontrar en:

“C:\Program Files\PostgreSQL\10\data\log”

Consultando los logs más recientes:

- Vemos que ocupan entre 0 KB y 1 KB las consultas con errores o logs vacíos por distintas consultas simples o cambios pequeños en la BBDD.
- Vemos que ocupan entre 3 y 10 MB las consultas grandes de inserción, borrado o acceso a datos.

Si accedemos a uno de estos logs más grandes, comprobamos que son abundantes los mensajes de este tipo:

```
2504 SELECT
2505 (SELECT sum(tup_fetched) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Fetched",
2506 (SELECT sum(tup_returned) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Returned"
2507 LOG:  sentencia: /*pga4dash*/
2508 SELECT
2509 (SELECT sum(xact_commit) + sum(xact_rollback) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Transactions",
2510 (SELECT sum(xact_commit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Commits",
2511 (SELECT sum(xact_rollback) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Rollbacks"
2512 LOG:  2018-12-01 14:37:03.627 CET [11092] LOG:  sentencia: /*pga4dash*/
2513 SELECT
2514 (SELECT sum(blks_read) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Reads",
2515 (SELECT sum(blks_hit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Hits"
2516 LOG:  2018-12-01 14:37:04.670 CET [11092] LOG:  sentencia: /*pga4dash*/
2517 SELECT
2518 (SELECT count(*) FROM pg_stat_activity WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Total",
2519 (SELECT count(*) FROM pg_stat_activity WHERE state = 'active' AND datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Active",
2520 (SELECT count(*) FROM pg_stat_activity WHERE state = 'idle' AND datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Idle"
2521 LOG:  2018-12-01 14:37:04.724 CET [11092] LOG:  sentencia: /*pga4dash*/
2522 SELECT
2523 (SELECT sum(tup_inserted) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Inserts",
2524 (SELECT sum(tup_updated) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Updates",
2525 (SELECT sum(tup_deleted) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Deletes"
2526 LOG:  2018-12-01 14:37:04.736 CET [11092] LOG:  sentencia: /*pga4dash*/
2527 SELECT
2528 (SELECT sum(tup_fetched) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Fetched",
2529 (SELECT sum(tup_returned) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Returned"
2530 LOG:  2018-12-01 14:37:04.737 CET [11092] LOG:  sentencia: /*pga4dash*/
2531 SELECT
2532 (SELECT sum(xact_commit) + sum(xact_rollback) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Transactions",
2533 (SELECT sum(xact_commit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Commits",
2534 (SELECT sum(xact_rollback) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Rollbacks"
2535 LOG:  2018-12-01 14:37:04.739 CET [11092] LOG:  sentencia: /*pga4dash*/
2536 SELECT
2537 (SELECT sum(blks_read) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Reads",
2538 (SELECT sum(blks_hit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Hits"
2539 LOG:  2018-12-01 14:37:05.640 CET [11092] LOG:  sentencia: /*pga4dash*/
2540 SELECT
2541 (SELECT count(*) FROM pg_stat_activity WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Total",
2542 (SELECT count(*) FROM pg_stat_activity WHERE state = 'active' AND datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Active",
2543 (SELECT count(*) FROM pg_stat_activity WHERE state = 'idle' AND datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Idle"
2544 LOG:  2018-12-01 14:37:05.642 CET [11092] LOG:  sentencia: /*pga4dash*/
2545 SELECT
2546 (SELECT sum(xact_commit) + sum(xact_rollback) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Transactions",
2547 (SELECT sum(xact_commit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Commits",
2548 (SELECT sum(xact_rollback) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Rollbacks"
2549 LOG:  2018-12-01 14:37:06.700 CET [11092] LOG:  sentencia: /*pga4dash*/
2550 SELECT
2551 (SELECT sum(tup_inserted) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Inserts",
2552 (SELECT sum(tup_updated) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Updates",
2553 (SELECT sum(tup_deleted) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24673)) AS "Deletes"
```

Donde podemos observar que se trabaja con pg\_stat\_database, y se van logueando y cambiando los estados y acciones de la consulta (“Fetched”, “Returned”, “Reads”, “Hits” ...).

## BIBLIOGRAFÍA

### PostgreSQL.

- Capítulo 14: Performance Tips.
- Capítulo 15: Parallel Query.
- Capítulo 24: Routine Database Maintenance Tasks.
- Capítulo 50: Overview of PostgreSQL Internals.
- Capítulo 59: Genetic Query Optimizer
- Capítulo 68: How the Planner Uses Statistics.

### MySQL.

- <http://dev.mysql.com/downloads/>
- <http://dev.mysql.com/doc/refman/8.0/en/>
- <http://dev.mysql.com/doc/refman/8.0/en/optimization.html>
- <http://dev.mysql.com/doc/refman/8.0/en/table-maintenance-sql.html>