

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: Luis Alejandro Cabanillas Prudencio

DNI: 04236930P

ALUMNO 2:

Nombre y Apellidos: Álvaro de las Heras Fernández

DNI: 03146833L

Fecha: 02/12/2018

Profesor Responsable: Iván González

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 errores de PostgreSQL sirve para registrar todos los errores que ocurran en el sistema, sin embargo, se puede modificar para que registre otro tipo de acciones elegidas por el usuario. Esto último se consigue modificando el archivo de configuración de PostgreSQL (postgresql.conf).
- Sí, la base de datos tiene arrancado el log de errores cada vez que se inicia el servidor.

```
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

log_statement = 'all'          # none, ddl, mod, all

log_duration = on
```

- Los campos a modificar en postgresql.conf son los mostrados en las imágenes de arriba. El log_min_duration_statement lo ponemos = 0 para que salte cuando realice cualquier acción y lo registre y lo descomentamos (quitamos el #) para que surja efecto. Por otro lado, el log_statement lo ponemos a on para que registre todas las consultas realizadas y no solo los fallos. Por último, ponemos a on el log_duration para registrar el tiempo de la consulta en milisegundos.

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?

- Sí, PostgreSQL cuenta con un recolector de estadísticas basado en tablas y vistas que se almacenan en el catálogo y se actualizan con frecuencia.

- Postgres es capaz de almacenar una gran cantidad de estadísticas sobre las tablas que estén en el sistema como el número de tuplas, el número de bloques accedidos al leer, resultados de operaciones como VACUUM, estadísticas de índices y otros datos relacionados con el usuario; entre otras funciones destacables.
- Estas estadísticas se guardan en el disco de forma similar a las tablas de una base de datos.

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.
- Primero creamos las tres tablas y añadimos sus columnas. Indicamos que en investigadores el codigo_investigador será la primary key (rodeado en rojo):

Inherited from table(s)

Columns						
	Name	Data type	Length	Precision	Not NULL	Primary key
<input checked="" type="checkbox"/>	codigo_investigador	numeric			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	nombre	text			<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	apellidos	text			<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	salario	numeric			<input type="checkbox"/>	<input type="checkbox"/>

- De la misma forma lo hacemos con proyectos, pero esta vez la PK será el campo codigo_proyecto.
- Para investigadores_proyectos necesitaremos una foreign key y una serie de restricciones, que estableceremos de la siguiente manera:

investigadores_proyectos

General Columns Constraints Advanced Parameter Security SQL

Primary Key Foreign Key Check Unique Exclude

Foreign key

Name	Columns
<input checked="" type="checkbox"/> numero_investigador	(numero_investigador) -> (codigo_investigador)
<input checked="" type="checkbox"/> numero_proyecto	(numero_investigador) -> (codigo_proyecto)

General Definition Columns Action

Columns

Local column

References

Referencing

Local	Referenced
numero_investigador	codigo_proyecto

On update

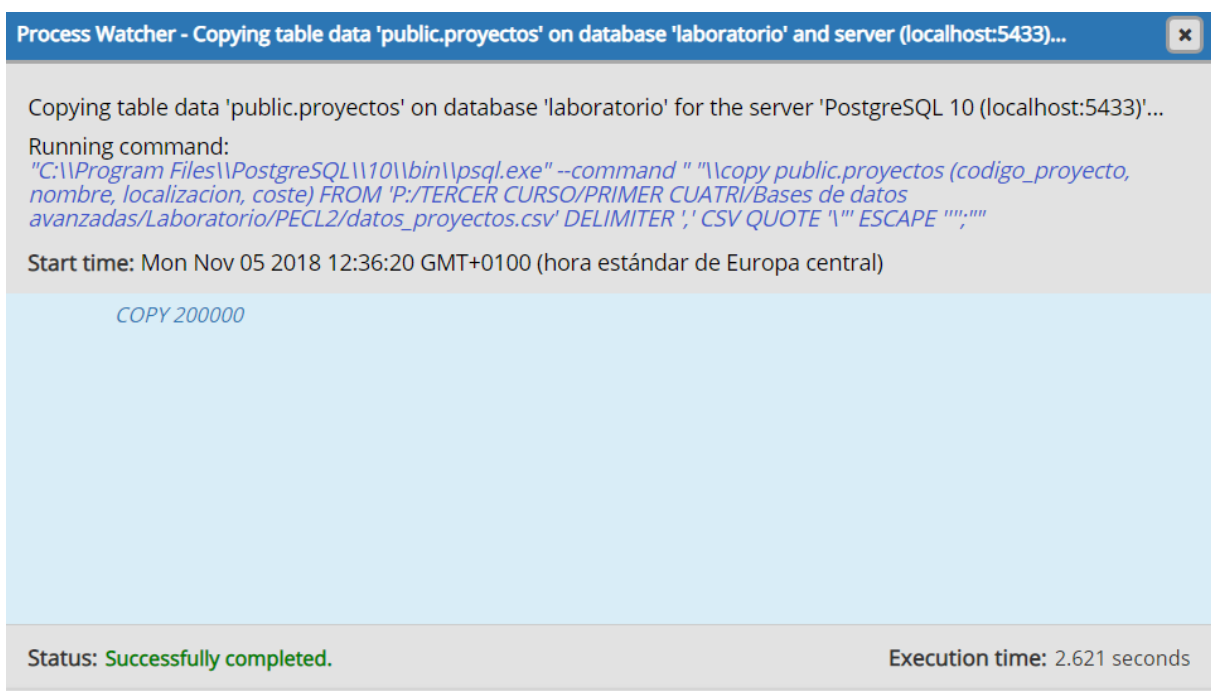
On delete

➤ La primary key la estableceremos de la misma manera que anteriormente, pero conformada por dos campos en vez de uno: numero_investigador y numero_proyecto.

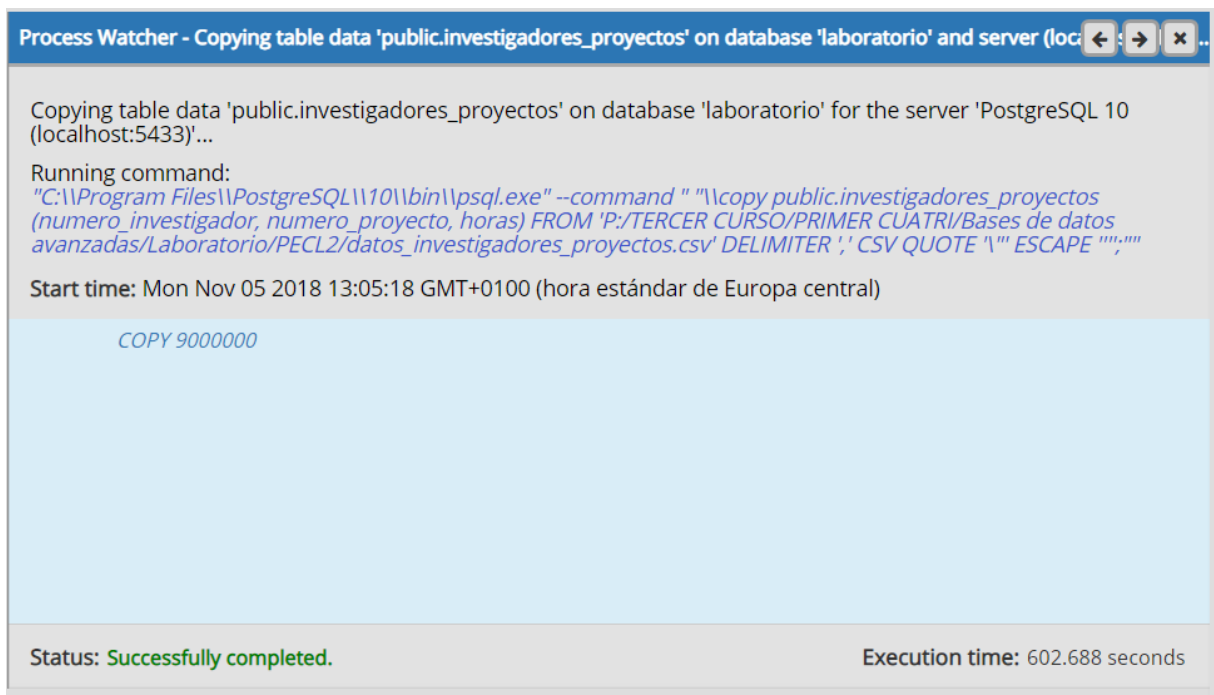
- Cargar la información del fichero datos_investigadores.csv, datos_proyectos.csv y datos_investigadores_proyectos.csv en dichas tablas de tal manera que sea lo más eficiente posible.
- Indicar los tiempos de carga.



➤ Tiempo de carga de investigadores: 31.288 segundos



➤ Tiempo de carga de proyectos: 2.621 segundos



➤ Tiempo de carga investigadores_proyectos: 602.688 segundos

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?

laboratorio on postgres@PostgreSQL 10													
<pre> 1 SELECT * FROM pg_catalog.pg_stat_all_tables 2 WHERE schemaname='public'; </pre>													
Data Output Explain Messages Notifications Query History													
	relid oid	schemaname name	relname name	seq_scan bigint	seq_tup_read bigint	idx_scan bigint	idx_tup_fetch bigint	n_tup_ins bigint	n_tup_upd bigint	n_tup_del bigint	n_tup_hot_upd bigint	n_live_tup bigint	n_dead_tup bigint
1	49236	public	investigadores	1	0	12656577	12656577	3000000	0	0	0	3000000	0
2	49252	public	investigadores_proyectos	5	0	1	0	27000000	0	0	0	9025047	0
3	49244	public	proyectos	1	0	12656576	12656575	200000	0	0	0	200000	0

laboratorio on postgres@PostgreSQL 10									
<pre> 1 SELECT * FROM pg_catalog.pg_statio_all_tables 2 WHERE schemaname='public'; </pre>									
Data Output Explain Messages Notifications Query History									
	relid oid	schemaname name	relname name	heap_blks_read bigint	heap_blks_hit bigint	idx_blks_read bigint	idx_blks_hit bigint	toast_blks_read bigint	toast_blks_hit bigint
1	49236	public	investigadores	2710753	22714546	1056103	45814704	0	0
2	49244	public	proyectos	5665	25314923	1657	38453607	0	0
3	49252	public	investigadores_proy...	663384	36283827	7840427	74083049	0	0

➤ Las estadísticas obtenidas para cada tabla se pueden obtener con la consulta que aparece en la imagen. En ella hacemos uso de las vistas pg_statio_all_tables y pg_stat_all_tables, ambas presentes en pg_catalog. Estas estadísticas no son del todo correctas ya que están desactualizadas. Para actualizarlas tenemos que ejecutar el comando ANALYZE sobre cada tabla.

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?

laboratorio on postgres@PostgreSQL 10	
1	EXPLAIN SELECT codigo_investigador, nombre, apellidos, salario
2	FROM public.investigadores
3	WHERE salario>95000;

Data Output	Explain	Messages	Notifications	Query History
QUERY PLAN text				
1	Seq Scan on investigadores (cost=0.00..65523.00 rows=1589295 width=41)			
2	Filter: (salario > '95000'::numeric)			

- Los datos que arroja el comando EXPLAIN respecto a las tuplas no son del todo correctos, ya que para obtenerlos realiza una predicción basada en la heurística que es muy cercana al dato real, pero no es exacta. Lo que sí es correcto es el método de búsqueda, que en este caso es secuencial.

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?

laboratorio on postgres@PostgreSQL 10	
1	EXPLAIN SELECT numero_investigador, numero_proyecto, horas
2	FROM public.investigadores_proyectos INNER JOIN proyectos ON (proyectos.codigo_proyecto= investigadores_proyectos.numero_proyecto)
3	WHERE horas=10;

Data Output	Explain	Messages	Notifications	Query History
QUERY PLAN text				
1	Gather (cost=8142.00..206540.78 rows=377247 width=16)			
2	Workers Planned: 2			
3	-> Hash Join (cost=7142.00..167816.08 rows=157186 width=16)			
4	Hash Cond: (investigadores_proyectos.numero_proyecto = proyectos.codigo_proyecto)			
5	-> Parallel Seq Scan on investigadores_proyectos (cost=0.00..157943.45 rows=157186 width=16)			
6	Filter: (horas = '10'::numeric)			
7	-> Hash (cost=3860.00..3860.00 rows=200000 width=6)			
8	-> Seq Scan on proyectos (cost=0.00..3860.00 rows=200000 width=6)			

- De la misma forma que en la cuestión 5, los datos respecto a las tuplas obtenidos por el EXPLAIN no son del todo correctos por la predicción heurística que realiza. Lo que sí es correcto es el método de búsqueda, vemos como utiliza el hash join y la búsqueda secuencial (no tenemos índices) para satisfacer la consulta.

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?

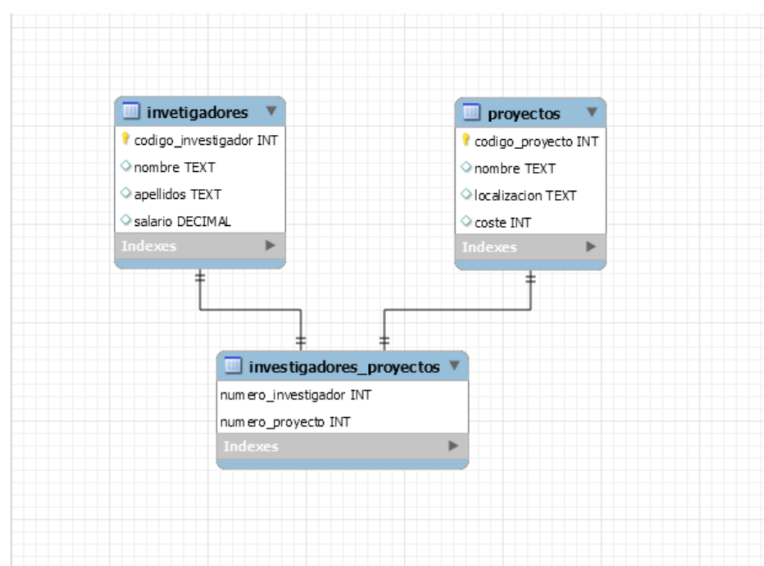
laboratorio on postgres@PostgreSQL 10	
1	EXPLAIN SELECT numero_investigador, numero_proyecto, horas
2	FROM public.investigadores_proyectos INNER JOIN proyectos ON (proyectos.codigo_proyecto=investigadores_proyectos.numero_proyecto)
3	INNER JOIN investigadores ON (investigadores.codigo_investigador=investigadores_proyectos.numero_investigador)
4	WHERE horas<2 AND coste>50000 AND salario=24000;

Data Output	Explain	Messages	Notifications	Query History
QUERY PLAN				
text				
1	Gather (cost=1000.86..44792.68 rows=1 width=16)			
2	Workers Planned: 2			
3	-> Nested Loop (cost=0.85..43792.58 rows=1 width=16)			
4	-> Nested Loop (cost=0.44..43791.67 rows=2 width=16)			
5	-> Parallel Seq Scan on investigadores (cost=0.00..43648.00 rows=7 width=6)			
6	Filter: (salario = '24000'::numeric)			
7	-> Index Scan using investigadores_proyectos_pkey on investigadores_proyectos (cost=0.44..20.51 rows=1 width=16)			
8	Index Cond: (numero_investigador = investigadores.codigo_investigador)			
9	Filter: (horas < '2'::numeric)			
10	-> Index Scan using proyectos_pkey on proyectos (cost=0.42..0.45 rows=1 width=6)			
11	Index Cond: (codigo_proyecto = investigadores_proyectos.numero_proyecto)			
12	Filter: (coste > '50000'::numeric)			

- Los datos que aporta el comando EXPLAIN respecto a las rows no son correctos del todo, ya que, como anteriormente, el comando EXPLAIN realiza una predicción basada en la heurística. Así, los datos no son completamente exactos, pero sí unos valores muy aproximados al resultado final. Tenemos dos INNER JOIN ON y podemos ver que realiza el algoritmo de bucle anidado por bloques, opción óptima en este caso, dado que dispone de memoria suficiente como para reducir coste a cero el join al encauzar los datos. También podemos ver cómo aplica los filtros del WHERE. Otro detalle que podemos ver es que devuelve una fila, que es la fila con los datos del explain únicamente, porque no hay ninguna tupla que satisfaga la búsqueda realizada por la query.

Cuestión 8: Repetir las cuestiones 3,4,5,6 y 7 con MySQL y comparar los resultados con los obtenidos por PostgreSQL.

- Una vez hechas todas las consultas en PostgreSQL hemos vuelto a repetirlas adaptándolas a MySQL.
- Lo primero de todo ha sido rehacer el modelo de la base de datos, este modelo lo hemos hecho con la herramienta que Workbench ofrece para realizar modelos el resultado ha sido el siguiente:



- A partir de este diagrama hemos obtenido un script en SQL con el que hemos generado el esquema para la base de datos. Después hemos procedido a la carga de datos mediante LOAD INFILE dado que el asistente para la carga de datos de MySQL tardaba demasiado en cargar los datos.

✓	10	13:37:52	LOAD DATA INFILE 'datos_proyectos.csv' INTO TABLE laboratorio....	200000 row(s) affected Records: 200000 Deleted: 0 Skipped: 0 Warnings: 0	3.062 sec
✓	16	13:45:32	LOAD DATA INFILE 'datos_investigadores.csv' INTO TABLE laboratorio.investigadores FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'	3000000 row(s) affect...	82.016 sec
✓	21	13:54:47	LOAD DATA INFILE 'datos_investigadores_proyectos.csv' INTO TABLE laboratorio.investigadores_proyectos FIELDS TERMINATED BY ',' LINES TERM...	9000000 row(s) affect...	1254.422 sec

- Según el tamaño del archivo y su integridad referencial el tiempo de carga aumenta o disminuye. Además para conseguir una mayor rapidez hemos modificado los parámetros de my.ini para asignar más memoria a innodb_buffer_pool_size, innodb_log_file_size y innodb_log_buffer_size para así tener más memoria para operaciones y aumentar la memoria para los archivos log que va generando conforme se realizan operaciones, para evitar así cuellos de botella, porque con dar más memoria para las operaciones no es suficiente.
- Para la cuestión 4 nos encontramos con que las estadísticas se almacenaban en el catálogo aunque este tenía mucha menos información comparado con PostgreSQL, la información obtenida es la siguiente:

Query 1

```

SELECT * FROM INFORMATION_SCHEMA.STATISTICS
WHERE table_name='investigadores'
AND table_schema= 'laboratorio'

```

TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	NON_UNIQUE	INDEX_SCHEMA	INDEX_NAME	SEQ_IN_INDEX	COLUMN
laboratorio	laboratorio	investigadores	0	laboratorio	PRIMARY	1	codigo_in

Query 1

```

1 SELECT * FROM INFORMATION_SCHEMA.STATISTICS
2 WHERE table_name='proyectos'
3 AND table_schema= 'laboratorio'

```

TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	NON_UNIQUE	INDEX_SCHEMA	INDEX_NAME	SEQ_IN_INDEX	COLUMN
def	laboratorio	proyectos	0	laboratorio	PRIMARY	1	codigo_p

Query 1

```

1 SELECT * FROM INFORMATION_SCHEMA.STATISTICS
2 WHERE table_name='investigadores_proyectos'
3 AND table_schema= 'laboratorio'

```

TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	NON_UNIQUE	INDEX_SCHEMA	INDEX_NAME
def	laboratorio	investigadores_proyectos	1	laboratorio	fk_investigadores_proyectos_investigadores1_idx
def	laboratorio	investigadores_proyectos	1	laboratorio	fk_investigadores_proyectos_proyectos_idx
def	laboratorio	investigadores_proyectos	0	laboratorio	PRIMARY
def	laboratorio	investigadores_proyectos	0	laboratorio	PRIMARY

Query 1

SHOW TABLE STATUS

	Data_free	Auto_increment	Create_time	Update_time	Check_time	Collation	Checksum	Create_options
6291456	NULL		2018-11-26 13:54:36	2018-11-26 14:15:42	NULL	utf8_general_ci	NULL	row_format=DYNAMIC
6291456	NULL		2018-11-26 13:16:46	2018-11-26 13:46:54	NULL	utf8_general_ci	NULL	
4194304	NULL		2018-11-26 13:16:46	2018-11-26 13:37:55	NULL	utf8_general_ci	NULL	

Query 1

SHOW TABLE STATUS

Name	Engine	Version	Row_format	Rows	Avg_row_length	Data_length	Max_data_length	Index_length	D
investigadores_proyectos	InnoDB	10	Dynamic	8402129	61	517996544	0	373850112	62
investigadores	InnoDB	10	Dynamic	2988328	73	218841088	0	0	62
proyectos	InnoDB	10	Dynamic	199430	65	13123584	0	0	41

- En todas estas estadísticas se puede observar la definición de las tablas en el esquema, las filas, el tamaño que ocupan y el tamaño libre entre otros datos, siendo estos lo más destacados. Mientras que en PostgreSQL disponemos de estadísticas sobre accesos, vistas, módulos propios y muchas más.
- En este caso las estadísticas si son correctas aunque si estuvieran desactualizadas habría que aplicar ANALYZE al igual que en PostgreSQL.
- La cuestión 5, 6 y 7 tienen un planteamiento similar, aquí todas ellas no tenían información relevante únicamente la que ofrecía en una fila el EXPLAIN, pero si nada más acerca de la ejecución ni nada. Sólo esta información se mostraba cuando se realizaba la consulta sin el EXPLAIN lo que suponía un gran coste para comprobarlo. Las únicas diferencias internas que había eran causadas por las tablas a las que accedía la consulta y las selecciones que empleaba. A continuación dejamos unas capturas en las que se puede ver como con EXPLAIN únicamente muestra una fila con pocos datos, solo las filas y el porcentaje de tuplas filtradas. Mientras que una vez realizada la consulta se muestran más estadísticas y hasta un plan de ejecución.

➤ CUESTIÓN 5:

Query 1 x

```
1 EXPLAIN SELECT codigo_investigador, nombre, apellidos, salario
2 FROM investigadores
3 WHERE salario > 95000;
```

Result Grid

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	investigadores	NULL	ALL	NULL	NULL	NULL	NULL	2988328	33.33	Using where

➤ Sin EXPLAIN:

Query 1 x

```
1 SELECT codigo_investigador, nombre, apellidos, salario
2 FROM investigadores
3 WHERE salario > 95000;
```

Visual Explain

Query cost: 302172.05

query_block#1

302172.05 2.99M rows

Full Table Scan

investigadores

Field Types

Query Stats

Execution Plan

Query 1 x

```
1 SELECT codigo_investigador, nombre, apellidos, salario
2 FROM investigadores
3 WHERE salario > 95000;
```

Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:2.84999083
Table lock wait time: 0:00:0.00013300

Errors:
Had Errors: NO
Warnings: 0

Rows Processed:
Rows affected: 0
Rows sent to client: 1588314
Rows examined: 3000000

Joins per Type:
Full table scans (Select_scan): 1
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Sorting:
Sorted rows (Sort_rows): 0
Sort merge passes (Sort_merge_passes): 0
Sorts with ranges (Sort_range): 0
Sorts with table scans (Sort_scan): 0

Index Usage:
No Index used

Form Editor

Field Types

Query Stats

➤ CUESTIÓN 6:

Query 1

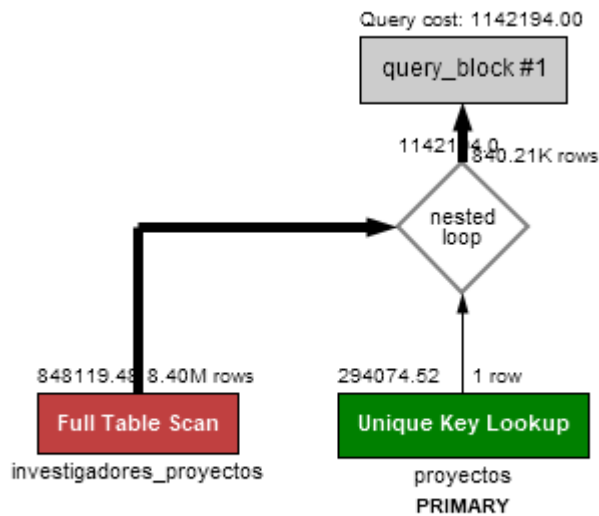
```

1 EXPLAIN SELECT numero_investigador, numero_proyecto, horas
2 FROM investigadores_proyectos INNER JOIN proyectos ON (proyectos.codigo_proyecto=investigadores_proyectos.numero_proyecto)
3 WHERE horas=10;

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	investigadores_proyectos	NONE	ALL	fk_investigadores_proyectos_proyectos_idx	NONE	NONE	NONE	8402129	10.00	Using where
1	SIMPLE	proyectos	NONE	eq_ref	PRIMARY	PRIMARY	4	laboratorio.investigadores_proyectos.numero_...	1	100.00	Using index

➤ Sin EXPLAIN:



Query 1

```

1 SELECT numero_investigador, numero_proyecto, horas
2 FROM investigadores_proyectos INNER JOIN proyectos ON (proyectos.codigo_proyecto=investigadores_proyectos.numero_proyecto)
3 WHERE horas=10;

```

Query Statistics

Timing (as measured at client side): Execution time: 0:00:0.06200000 Timing (as measured by the server): Execution time: 0:00:4.48534305 Table lock wait time: 0:00:0.00011600 Errors: Had Errors: NO Warnings: 0 Rows Processed: Rows affected: 0 Rows sent to client: 374471 Rows examined: 9374471	Joins per Type: Full table scans (Select_scan): 1 Joins using table scans (Select_full_join): 0 Joins using range search (Select_full_range_join): 0 Joins with range checks (Select_range_check): 0 Joins using range (Select_range): 0 Sorting: Sorted rows (Sort_rows): 0 Sort merge passes (Sort_merge_passes): 0 Sorts with ranges (Sort_range): 0 Sorts with table scans (Sort_scan): 0 Index Usage: No Index used
--	---

➤ CUESTIÓN 7:

Query 1

```

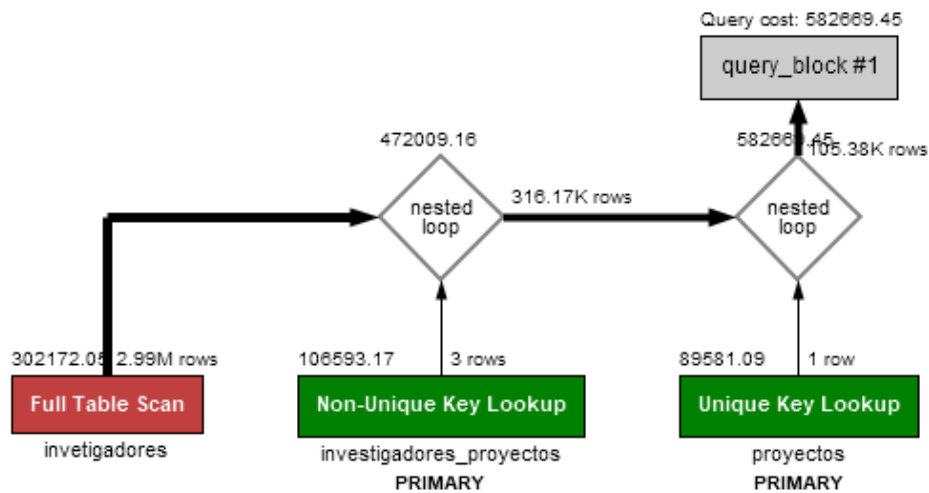
1  ctos.numero_proyecto) INNER JOIN investigadores ON (investigadores.codigo_investigador=investigadores_proyectos.numero_investigador)
2
3

```

Result Grid

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	investigadores	NULL	ALL	PRIMARY	PRIMARY	4	laboratorio.investigadores.codigo_investigador	2988328	10.00	Using where
1	SIMPLE	investigadores_proyectos	NULL	ref	PRIMARY, fk_investigadores_proyectos_proyec...	PRIMARY	4	laboratorio.investigadores.codigo_investigador	3	33.33	Using where
1	SIMPLE	proyectos	NULL	eq_ref	PRIMARY	PRIMARY	4	laboratorio.investigadores_proyectos.numero...	1	33.33	Using where

- Sin EXPLAIN:



Query 1

```

1  SELECT numero_investigador, numero_proyecto, horas
2  FROM investigadores_proyectos INNER JOIN proyectos ON (proyectos.codigo_proyecto=investigadores_proyectos.numero_proyecto) INNER JOIN investigadores ON (ir
3  WHERE horas<2 AND coste>50000 AND salario=24000;

```

Query Statistics

Timing (as measured at client side): Execution time: 0:00:0.87500000	Joins per Type: Full table scans (Select_scan): 1 Joins using table scans (Select_full_join): 0 Joins using range search (Select_full_range_join): 0 Joins with range checks (Select_range_check): 0 Joins using range (Select_range): 0
Timing (as measured by the server): Execution time: 0:00:0.87342787 Table lock wait time: 0:00:0.00012600	Sorting: Sorted rows (Sort_rows): 0 Sort merge passes (Sort_merge_passes): 0 Sorts with ranges (Sort_range): 0 Sorts with table scans (Sort_scan): 0
Errors: Had Errors: NO Warnings: 0	Index Usage: No Index used
Rows Processed: Rows affected: 0 Rows sent to client: 0 Rows examined: 3000048	

- Si comparamos estos resultados con PostgreSQL podemos observar la buena implementación que hace PostgreSQL del EXPLAIN ya que para ver el plan de ejecución no hace falta ni siquiera ejecutar la consulta, además tenemos muchos más datos cuando lo realizamos. La solución que ha cogido en la cuestión 5 es la misma para ambos, en la cuestión 6 es diferente la ejecución aplicando un mecanismo más tosco en MySQL con una lectura secuencial y un bucle anidado para el JOIN, en la última cuestión han aplicado una resolución similar con bucles anidados, aunque la búsqueda de las claves y filtros ha sido diferente.
- En definitiva PostgreSQL ofrece mucho mejores soluciones y herramientas en las bases de datos, como se ha podido ver en la comparación.

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 se lleva a cabo en los siguientes pasos:
 1. Se debe establecer una conexión desde un programa de aplicación al servidor PostgreSQL. El programa de aplicación transmite una consulta al servidor y espera recibir los resultados enviados por el servidor.
 2. La etapa del parser comprueba la sintaxis correcta de la consulta transmitida por el programa de aplicación y crea un árbol de consulta.
 3. El sistema de reescritura toma el árbol de consultas creado por la etapa del parser y busca las reglas (almacenadas en los catálogos del sistema) para aplicarlas al árbol de consultas. Realiza las transformaciones dadas en los cuerpos de la regla.

Una aplicación del sistema de reescritura es la realización de *vistas*. Cuando se realiza una consulta contra una vista (es decir, una tabla virtual), el sistema de reescritura reescribe la consulta del usuario a una consulta que accede a las tablas base que figuran en la definición de la vista.

4. El planner/optimizer toma el árbol de consulta (reescrito) y crea un *plan de consulta* que será la entrada al ejecutor.

Lo hace creando primero todas las rutas posibles que conducen al mismo resultado. Por ejemplo, si hay un índice en una relación para ser escaneada, hay dos rutas para la exploración. Una posibilidad es un simple escaneo secuencial y la otra posibilidad es usar el índice. A continuación, se calcula el costo de la ejecución de cada ruta y se elige la ruta más barata. La ruta más barata se expande en un plan completo que el ejecutor puede usar.

5. El ejecutor recorre el árbol del plan recursivamente y recupera filas en la forma representada por el plan. El ejecutor utiliza el sistema de almacenamiento mientras escanea las relaciones, realiza clasificaciones y uniones, evalúa las calificaciones y, finalmente, devuelve las filas derivadas.

- Por otro lado, respecto a los algoritmos de procesamiento de consultas que incorpora PostgreSQL, el planner/optimizer comienza generando planes para analizar cada

tabla utilizada en la consulta. Los posibles planes están determinados por los índices disponibles en cada relación. Siempre existe la posibilidad de realizar una búsqueda secuencial en una relación, por lo que siempre se crea un plan de exploración secuencial. Si tenemos algún join en la consulta tendremos disponibles las siguientes opciones:

- Bucle anidado por bloques: la relación correcta se analiza una vez por cada fila encontrada en la relación izquierda. Esta estrategia es fácil de implementar, pero puede llevar mucho tiempo. (Sin embargo, si la relación correcta se puede escanear con un escaneo de índice, esta puede ser una buena estrategia. Es posible usar valores de la fila actual de la relación izquierda como claves para el escaneo de índice de la derecha).
 - Merge join: cada relación se ordena en los atributos de unión antes de que comience la unión. Luego, las dos relaciones se exploran en paralelo y las filas coincidentes se combinan para formar filas de unión. Este tipo de unión es más atractiva porque cada relación debe ser escaneada una sola vez. La clasificación requerida se puede lograr mediante un paso de clasificación explícito o explorando la relación en el orden correcto utilizando un índice en la clave de unión.
 - Hash join: primero se escanea la relación correcta y se carga en una tabla hash, utilizando sus atributos de unión como claves hash. A continuación, se analiza la relación de la izquierda y los valores apropiados de cada fila encontrada se utilizan como claves hash para ubicar las filas coincidentes en la tabla.
- Algunos parámetros a destacar que podemos configurar son:
- `enable_bitmapscan` (booleano): Activa o desactiva el uso del planificador de consultas de los planes de escaneo de mapas de bits. El valor predeterminado está activado.
 - `enable_hashagg` (booleano): Activa o desactiva el uso del planificador de consultas de los planes de agregación hash. El valor predeterminado está activado.
 - `enable_hashjoin` (booleano): Habilita o deshabilita el uso del planificador de consultas de los planes hash join. El valor predeterminado está activado.
 - `enable_indexscan` (booleano): Activa o desactiva el uso del planificador de consultas de los tipos de planes de exploración de índice. El valor predeterminado está activado.
 - `enable_indexonlyscan` (booleano): Habilita o inhabilita el uso del planificador de consultas de los tipos de planes de escaneo de solo índice. El valor predeterminado está activado.
 - `enable_material` (booleano): Habilita o deshabilita el uso de materialización por parte del planificador de consultas. Es imposible suprimir por completo la materialización, pero desactivar esta variable evita que el planificador inserte nodos de materialización, excepto en los casos en que sea necesario para la corrección. El valor predeterminado está activado.

- `enable_mergejoin` (booleano): Habilita o deshabilita el uso del planificador de consultas de los tipos de planes merge join. El valor predeterminado está activado.
- `enable_nestloop` (booleano): Habilita o deshabilita el uso del planificador de consultas de los planes bucle anidado por bloques. Es imposible suprimir las uniones de bucle anidado por completo, pero desactivar esta variable permite al planificador usar una si hay otros métodos disponibles. El valor predeterminado está activado.
- `enable_seqscan` (booleano): Activa o desactiva el uso del planificador de consultas de los planes de búsqueda secuencial. Es imposible suprimir los escaneos secuenciales por completo, pero desactivar esta variable permite al planificador usar uno si hay otros métodos disponibles. El valor predeterminado está activado.

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?

- Sí, PostgreSQL cuenta con un optimizador de consultas genético conocido como GEQO (The genetic query optimizer). Esta optimización la realiza internamente y se basa en un algoritmo genético que realiza la planificación de consultas mediante la búsqueda heurística porque emplea distintas generaciones que irán mejorando para conseguir mayor fitness, es decir, que sea más eficiente y eficaz cada vez gracias a los datos heurísticos.
- Los parámetros que se pueden modificar del optimizador de consultas son:
 - `geqo` (booleano): Habilita o deshabilita la optimización de consultas. Está activado de forma predeterminada
 - `geqo_effort` (entero): Controla la compensación entre el tiempo de planificación y la calidad del plan de consulta en GEQO. Esta variable debe ser un número entero de 1 a 10. El valor predeterminado es cinco. Los valores más grandes aumentan el tiempo empleado en la planificación de consultas, pero también aumentan la probabilidad de que se elija un plan de consultas eficiente.
 - `geqo_pool_size` (integer): Controla el tamaño del pool utilizada por GEQO. Debe ser al menos dos, y los valores útiles suelen ser de 100 a 1000. Si se establece en cero (la configuración predeterminada), se elige un valor adecuado en función de `geqo_effort` y el número de tablas en la consulta.
 - `geqo_generations` (integer): Controla el número de iteraciones del algoritmo. Debe ser al menos uno. Si se establece en cero (la configuración predeterminada), se elige un valor adecuado en función de `geqo_pool_size`.
 - `geqo_seed` (floating point): Controla el valor inicial del generador de números aleatorios utilizado por GEQO para seleccionar rutas aleatorias a través del espacio de búsqueda de orden de unión. El valor puede variar de cero (el valor predeterminado) a uno. Variar el valor cambia el conjunto de rutas de

unión exploradas, y puede dar como resultado que se encuentre una mejor o peor ruta.

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?

NOTA: Antes de realizar la carga masiva de datos hemos modificado la memoria RAM asignada en el archivo postgresql.conf con el objetivo de reducir los tiempos de carga:

```
# - Memory -
shared_buffers = 2GB          # min 128kB
                                # (change requires restart)
#huge_pages = try             # on, off, or try
                                # (change requires restart)
#temp_buffers = 8MB           # min 800kB
#max_prepared_transactions = 0 # zero disables the feature
                                # (change requires restart)
# Caution: it is not advisable to set max_prepared_transactions nonzero unless
# you actively intend to use prepared transactions.
#work_mem = 4MB                # min 64kB
#maintenance_work_mem = 64MB   # min 1MB
#replacement_sort_tuples = 150000 # limits use of replacement selection sort
#autovacuum_work_mem = -1       # min 1MB, or -1 to use maintenance_work_mem
#max_stack_depth = 2MB         # min 100kB
dynamic_shared_memory_type = windows # the default is the first option
                                    # supported by the operating system:
                                    #   posix
                                    #   sysv
                                    #   windows
                                    #   mmap
                                    # use none to disable dynamic shared memory
                                    # (change requires restart)
```

Tabla	Tiempo sin integridad	Tiempo con integridad
Grupo	0,554 segundos	1,936 segundos
Conciertos	0,543 segundos	1,278 segundos
Músicos	2,909 segundos	48,361 segundos
Discos	2,87 segundos	31,221 segundos
Canciones	34,076 segundos	386,446 segundos
Entradas	44,832 segundos	695,052 segundos
Grupos_Tocan_Conciertos	2,017 segundos	107,87 segundos

- El orden a la hora de la carga de datos sin integridad referencial no afecta, ya que no se comprueba nada durante la carga. A la hora de cargar los datos con la integridad referencial activada, tenemos que tener en cuenta el orden en el que insertamos los datos en las tablas, ya que se irán comprobando las constraints entre tablas. Por ello deberemos cargar antes los datos de las tablas más independientes, es decir, que no tengan FK's.
- Así, el orden que seguiremos a la hora de cargar los datos con la integridad referencial será: grupo, conciertos, músicos, discos, canciones, entradas y grupos_tocan_conciertos.
- En cuanto a los tiempos de carga con la integridad referencial desactivada, vemos que el que más tarda es entradas y el que menos conciertos. Esto es así simplemente porque entradas es la que más registros tiene y conciertos la que menos. Así, llegamos a la conclusión de que sin la integridad referencial los tiempos de carga serán proporcionales a los datos a insertar. Cuantos más datos a insertar, más tiempo se tardará en cargarlos y viceversa.
- Por otro lado, respecto a los tiempos de carga con la integridad referencial activada, podemos ver que es mucho mayor, ya que ahora no sólo importa los datos a insertar que tengamos, sino también las dependencias entre tablas. Así, el tiempo de carga dependerá de los datos a introducir y del grado de dependencia de la tabla con respecto de otras tablas.
- Si nos vamos al log, veremos que existe una pequeña diferencia con respecto a los tiempos que mide el pgadmin. Esto es debido a la propia arquitectura cliente-servidor de pgadmin y PostgreSQL. Los tiempos en el log siempre son algo más pequeños porque estos tiempos los escribe primero el servidor, PostgreSQL, y luego el cliente, el pgadmin.

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	2.656 sec	3,109 sec
Conciertos	1.531 sec	1,965 sec
Músicos	65.875 sec	77.656 sec
Discos	27.343 sec	35.312 sec
Canciones	371.203 sec	428.094 sec
Entradas	704.906 sec	801.719 sec
Grupos_tocan_conciertos	73.610 sec	79.906 sec

- Lo primero que hicimos fue pasar el modelo de Pgmodeller a MySQL, para ello usamos un módulo conversor, pg2mysql, aunque hubo que realizar unas modificaciones a los nombres. Una vez hecho eso obtuvimos nuestro modelo y de ahí nuestro esquema para la base de datos.
- Para realizar la carga masiva de datos usamos el script LOAD INFILE y para desactivar la integridad referencial a la hora de insertar datos el parámetro SET foreign_key_checks=0. Para comprobar si se había desactivado correctamente la integridad referencial probamos primero a insertar las tablas hijas, haciéndolo sin problema alguno sin integridad, aunque cuando la volvimos a activar ya daba problemas.
- Comparando estos datos con los de PostgreSQL nos queda muy claro que son unos tiempos realmente altos tanto con integridad referencial como sin integridad referencial. Especialmente si la cantidad de datos a insertar es muy grande como ocurre con la tabla entradas o la tabla canciones. Al igual que PostgreSQL los tiempos de carga dependen de la cantidad de datos a insertar y de las FK que tenga.
- En definitiva, PostgreSQL es una mejor opción para la carga de datos, tanto por su facilidad de carga de datos, como por su velocidad de carga que supera a MySQL, mejorando ampliamente contra mayor cantidad de datos haya que insertar.

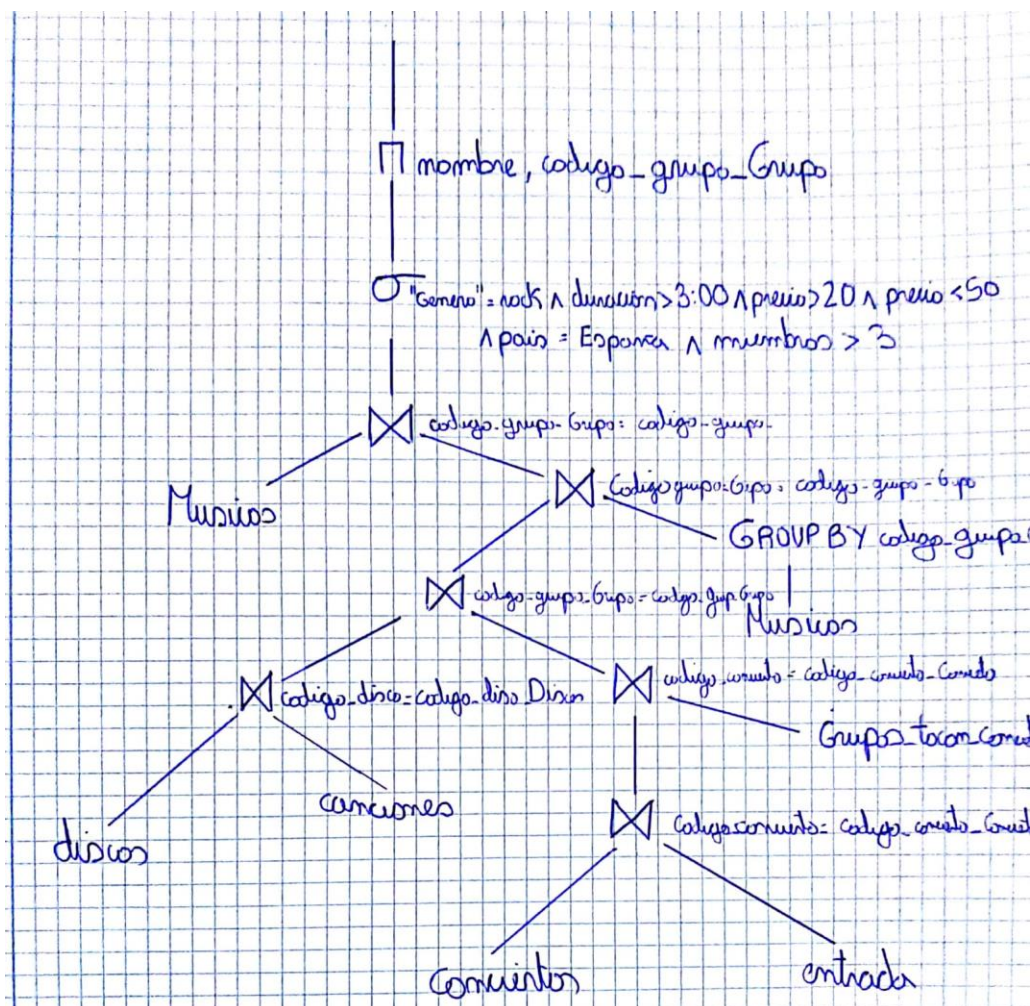
A partir de este momento en adelante, se deben de realizar las siguientes cuestiones con la base de datos que tiene la integridad referencial activada.

Cuestión 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.

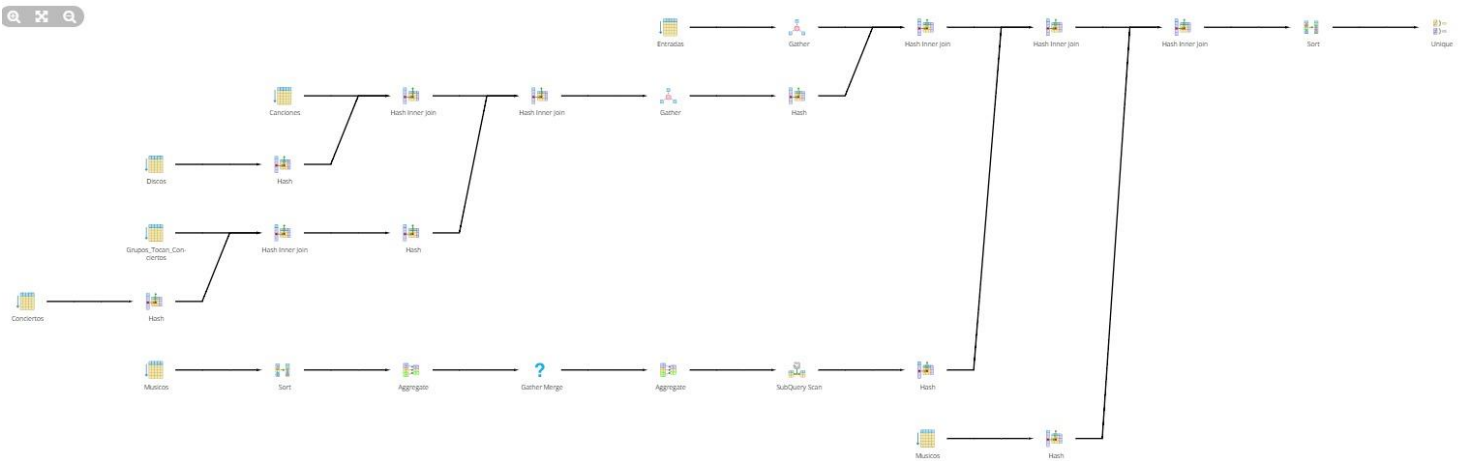
➤ Nuestra consulta será:

```
EXPLAIN SELECT DISTINCT "Músicos"."Nombre", "Músicos"."Codigo_grupo_Grupo", "Músicos"."codigo_musico"
FROM "Músicos" INNER JOIN (SELECT "Codigo_grupo_Grupo", COUNT("Codigo_grupo_Grupo")
FROM "Músicos"
GROUP BY("Codigo_grupo_Grupo")) AS miembros ON miembros."Codigo_grupo_Grupo"= "Músicos"."Codigo_grupo_Grupo"
INNER JOIN "Discos" ON "Músicos"."Codigo_grupo_Grupo"="Discos"."Codigo_grupo_Grupo"
INNER JOIN "Canciones" ON "Discos"."Codigo_disco"="Canciones"."Codigo_disco_Discos"
INNER JOIN "Grupos_Tocan_Conciertos" ON "Discos"."Codigo_grupo_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"
WHERE "Genero"='rock' AND "Duracion">'3:00' AND (CAST ("Precio" AS decimal(10,2)) BETWEEN 20.00 AND 50.00) AND "Conciertos"."Pais"='España' AND (miembros.count>3)
```

➤ El árbol obtenido por nosotros a partir de esta consulta será:



➤ El árbol obtenido por postgresql será:



➤ El árbol obtenido con EXPLAIN es totalmente diferente al que realizaría nuestra query. Esto se debe a que PostgreSQL internamente realiza una optimización de la consulta conforme a cálculos heurísticos que se realizan empleando las estadísticas disponibles de la tabla. Por eso es capaz de conmutar los JOIN de tal forma que se consiga hacer subir el menor número de tuplas posible, para así hacer la consulta empleando la menor cantidad de memoria posible

Cuestión 14: Repetir la cuestión 13 usando MySQL. Comparar los resultados entre ambos sistemas gestores de bases de datos.

id	select_type	table	partiti	type	possible_keys	key	key_ref	rows	filtered	Extra
1	PRIMARY	grupos_Tocan_Conciertos	INDEX	ref	PRIMARY, fk_Grupos_Tocan_Conciertos_Con...	fk_Grupos_Tocan_Conciertos_Grupo1_idx	4	12	100.00	Using index
1	PRIMARY	discos	INDEX	ref	PRIMARY, fk_Discos_Grupo1_idx	fk_Discos_Grupo1_idx	4	5	10.00	Using where
1	PRIMARY	conciertos	INDEX	eq_ref	PRIMARY	PRIMARY	4	1	10.00	Using where
1	PRIMARY	musicos	INDEX	ref	fk_Musicos_Grupo1_idx	fk_Musicos_Grupo1_idx	4	4	100.00	Using index
2	DERIVED	musicos	INDEX	index	fk_Musicos_Grupo1_idx	fk_Musicos_Grupo1_idx	4	994080	100.00	Using index
1	PRIMARY	entradas	INDEX	ref	fk_Entradas_Conciertos_idx	fk_Entradas_Conciertos_idx	4	211	11.11	Using where
1	PRIMARY	canciones	INDEX	ref	fk_Canciones_Discos1_idx	fk_Canciones_Discos1_idx	4	11	33.33	Using where
1	PRIMARY	<derived2>	INDEX	ALL			4	994080	33.33	Using where

➤ Lo primero de todo es hacer compatible nuestra primera consulta en PostgreSQL con la nueva en MySQL. Para ello modificaremos las comillas, operaciones y algunos nombres de tablas y columnas que se modificaron al crear el esquema en MySQL.

➤ Con la consulta una vez hecha, lo siguiente era que ejecutásemos esta con EXPLAIN, en este caso nos muestra una entrada por cada ajord que el optimizador emplee. Aunque no nos permite acceder al árbol de ejecución sin lanzar la consulta sin EXPLAIN, a diferencia de PostgreSQL.

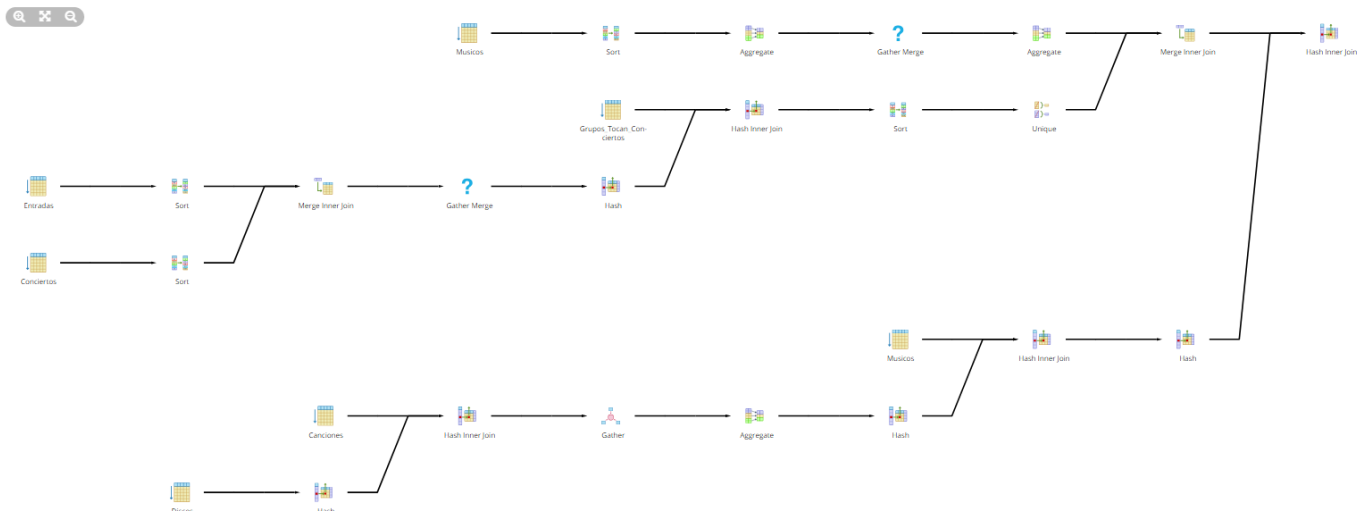
- Analizando los datos mostrados por el EXPLAIN, ajord ver como el optimizador es capaz de crear tablas intermedias e índices derivados, para agilizar la consulta, pero mayoritariamente utiliza selecciones para filtrar los datos. También las filas que se devuelven en cada momento no tienen mucho sentido dado el gran conjunto de datos que hay y el bajo porcentaje en algunos casos del filtro.
- Comparando con PostgreSQL la única ventaja que aporta es la creación de índices y tablas derivadas para agilizar las consultas; pero esto no es capaz de contrarrestar las desventajas como una mala implementación del EXPLAIN, un mal uso de la heurística y una mala optimización al no usar la técnica más eficiente ajorda en cada momento para ajorda la consulta ajorda primero las que menos tuplas devuelven.
- En definitive, PostgreSQL posee un mejor planificador y optimizador de consultas que MySQL

- Para optimizar la consulta lo primero que haremos será aplicar las normas de la optimización heurística al árbol obtenido anteriormente, de igual modo que en clase:

- Así, podemos ver cómo se han dividido las selecciones (podemos hacerlo ya que la condición estaba expresada con AND's) y se han añadido proyecciones. De esta forma, hemos optimizado el árbol y ahora deberemos obtener la consulta SQL a partir del árbol creado:

```
SELECT joinDiscosMusicos."Codigo_grupo_Grupo", "Nombre"
FROM
  (SELECT "Nombre", musicos."Codigo_grupo_Grupo", "codigo_musico"
  FROM (SELECT miembros."Codigo_grupo_Grupo"
  FROM
    (SELECT DISTINCT "Codigo_grupo_Grupo"
    FROM
      (SELECT "Codigo_disco", "Codigo_grupo_Grupo"
      FROM
        (SELECT "Codigo_disco", "Codigo_grupo_Grupo", "Genero"
        FROM "Discos") AS disco
        WHERE "Genero"='rock') AS seleccionDisco
      INNER JOIN
        (SELECT "Codigo_disco_Discos"
        FROM
          (SELECT "Codigo_disco_Discos", "Duracion"
          FROM "Canciones") AS canciones
          WHERE "Duracion">'3:00' ) AS seleccionCancion ON "Codigo_disco"="Codigo_disco_Discos") AS joinDiscosCanciones
      INNER JOIN
        (SELECT "Codigo_grupo_Grupo"
        FROM "Musicos"
        GROUP BY("Codigo_grupo_Grupo")
        HAVING COUNT("Codigo_grupo_Grupo") > 3) AS miembros ON miembros."Codigo_grupo_Grupo"=joinDiscosCanciones."Codigo_grupo_Grupo") AS joinMiembrosDiscos
      INNER JOIN
        (SELECT "Nombre", "Codigo_grupo_Grupo", "codigo_musico"
        FROM "Musicos") AS musicos ON joinMiembrosDiscos."Codigo_grupo_Grupo" = musicos."Codigo_grupo_Grupo") AS joinDiscosMusicos
    )
  )
  INNER JOIN
    (SELECT DISTINCT "Codigo_grupo_Grupo"
    FROM
      "Grupos_Tocan_Conciertos"
    INNER JOIN
      (SELECT "Codigo_concierto"
      FROM
        (SELECT "Codigo_concierto"
        FROM
          (SELECT "Codigo_concierto", "Pais"
          FROM "Conciertos") AS subConsConciertos
          WHERE "Pais"='España') AS conciertos
        INNER JOIN
          (SELECT "Codigo_concierto_Conciertos"
          FROM
            (SELECT "Codigo_concierto_Conciertos", "Precio"
            FROM "Entradas") AS subConsEntradas
            WHERE (CAST ("Precio" AS decimal(10,2)) BETWEEN 20.00 AND 50.00)) AS entradas ON "Codigo_concierto"="Codigo_concierto_Conciertos") AS joinConciertosEntradas
          ON "Codigo_concierto_Conciertos"=joinConciertosEntradas."Codigo_concierto") AS joinConciertosEntradasGruposTocan
      ON joinConciertosEntradasGruposTocan."Codigo_grupo_Grupo"= joinDiscosMusicos."Codigo_grupo_Grupo"
```

- El árbol de ejecución de esta consulta en PostgreSQL será:

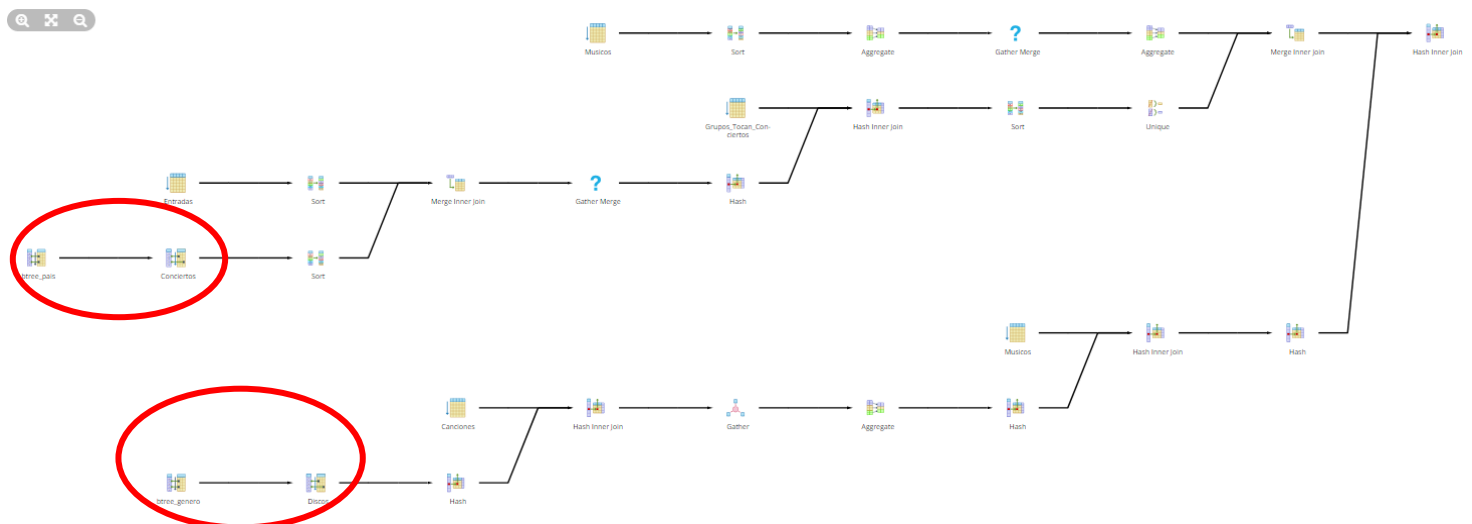


- Así, vemos como hemos optimizado la consulta inicial, pero aún se puede optimizar más con la creación de índices. De esta forma, hemos decidido crear dos índices btree sobre los campos genero (tabla discos) y país (tabla conciertos) para lograr la máxima optimización posible:

```
Create - Index
General Definition SQL
1 CREATE INDEX btree_genero
2 ON public."Discos" USING btree
3 ("Genero" ASC NULLS LAST)
4 TABLESPACE pg_default;
5
6 ALTER TABLE public."Discos"
7 CLUSTER ON btree_genero;
```

```
Create - Index
General Definition SQL
1 CREATE INDEX btree_pais
2 ON public."Conciertos" USING btree
3 ("Pais" ASC NULLS LAST)
4 TABLESPACE pg_default;
```

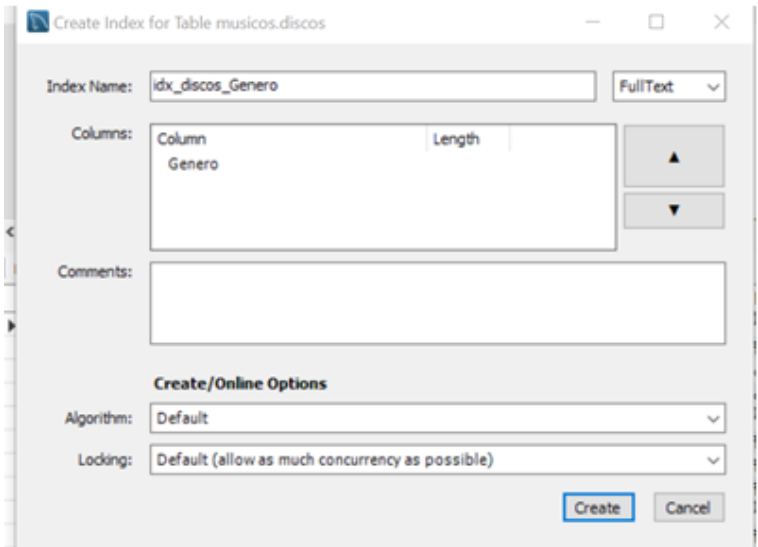
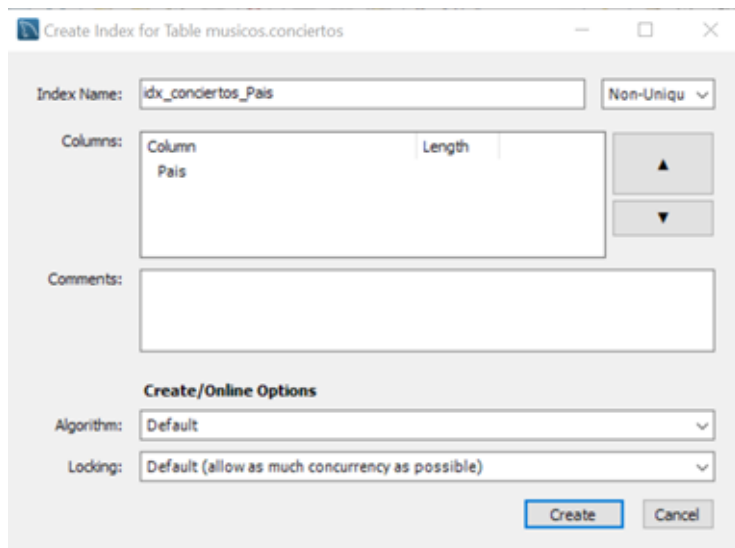
- Para comprobar que se usan los índices, ejecutamos de nuevo la consulta con el comando EXPLAIN para obtener el árbol de ejecución de PostgreSQL:



- Como podemos ver, los índices creados han evitado la búsquedas secuenciales que realizaba planificador de consultas antes en las tablas conciertos y discos. En lugar de estas se han usado los índices.

Cuestión 16: Usando MySQL, repita la cuestión 15 y compare los resultados obtenidos con PostgreSQL.

- Para optimizar la consulta, al igual que en el caso de PostgreSQL, hemos creado índices sobre los campos de las selecciones para mejorar el tiempo de carga de datos al emplear directamente este índice en vez de una lectura secuencial seguida de una condición:



- Una vez hecho esto hemos adaptado la consulta de PostgreSQL a MySQL haciendo los cambios pertinentes como las comillas. Finalmente hemos obtenido la tabla de EXPLAIN que como se ve comparando a la anterior sí que hace un mayor uso de índices y tablas derivadas que se han obtenido de las subconsultas:

id	select_type	table	part	type	possible_keys	key	key_rows	ref	filtered	Extra
1	PRIMARY	<derived4>		ALL			5		100.00	
1	PRIMARY	musicos		ref	fk_musicos_grupo1_idx	fk_musicos_grupo1_idx	4	joinDiscosCanciones.Grupo...	100.00	
1	PRIMARY	<derived11>		ref	<auto_key0>	<auto_key0>	4	joinDiscosCanciones.Grupo...	100.00	
1	PRIMARY	<derived9>		ref	<auto_key0>	<auto_key0>	4	joinDiscosCanciones.Grupo...	100.00	Using index
11	DERIVED	conciertos		ALL	PRIMARY_idx_conciertos_Pais		9940		0.00	Using where; Using temporary
11	DERIVED	entradas		ref	fk_entradas_conciertos_idx_entradas_P...	fk_entradas_conciertos_idx	4	musicos.conciertos.Codigo...	50.00	Using where
11	DERIVED	grupos_tocan_conciertos		ref	PRIMARY_fk_grupos_tocan_conciertos_Con...	fk_grupos_tocan_conciertos_Con...	4	musicos.conciertos.Codigo...	100.00	Using index
9	DERIVED	musicos		index	fk_musicos_grupo1_idx	fk_musicos_grupo1_idx	4	994080	100.00	Using index
4	DERIVED	discos		ALL	PRIMARY_fk_discos_grupo1_idx_discos_...		995683		0.00	Using where; Using temporary
4	DERIVED	canciones		ref	fk_canciones_discos1_idx_canciones_D...	fk_canciones_discos1_idx	4	musicos.discos.Codigo_discos	50.00	Using where

- Con respecto a PostgreSQL, se crean muchas más tablas intermedias para las operaciones de JOIN, también se emplean más índices, aunque los filtrados son mucho menores porque los carga con la condición casi directos de la tabla. Tampoco hace la mejor optimización posible en el orden de realizar los JOIN como sí hace PostgreSQL con su algoritmo genético

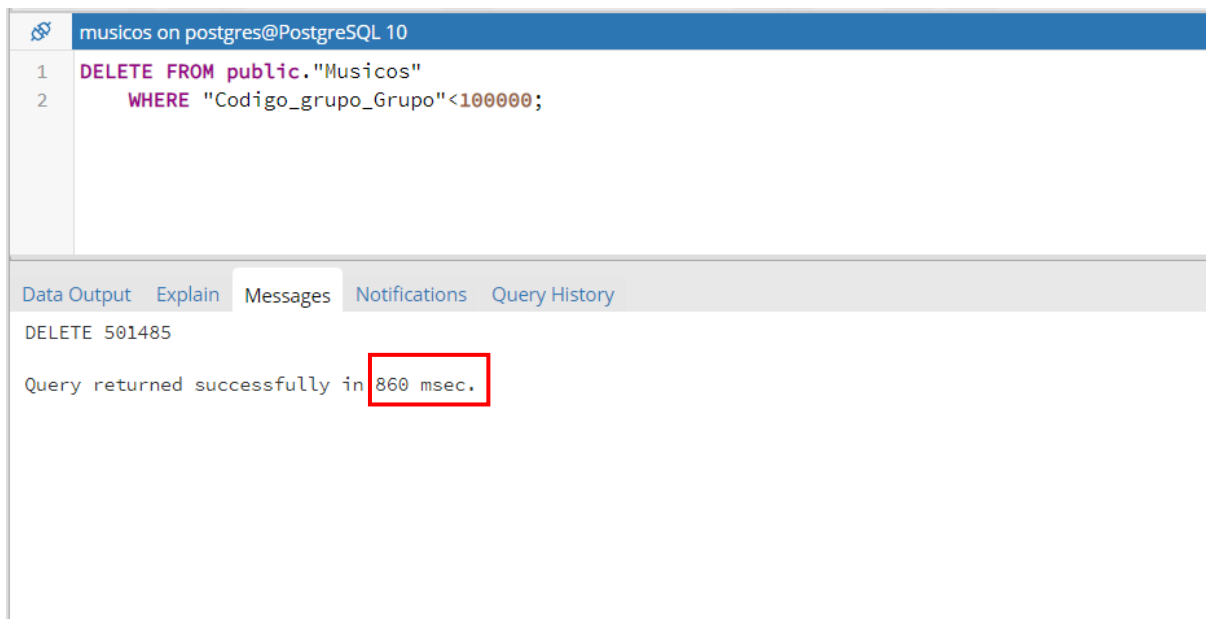
Cuestión 17: Usando PostgreSQL, borre el 50% de los datos (aproximadamente). ¿Cuál ha sido el proceso seguido? ¿Y el tiempo empleado en el borrado? Obtenga el plan de ejecución de la Cuestión 13. Dibujar un diagrama con el nuevo resultado del comando EXPLAIN en forma de árbol de álgebra relacional. Comparar con los resultados anteriores.

- Para borrar la mitad de los datos el proceso que hemos seguido ha sido el siguiente.
- Lo primero que hemos hecho ha sido trabajar con una base de datos de pruebas igual que la que tenemos, pero con un conjunto más pequeño de datos para ir probando con integridad referencial consultas de borrado hasta obtener una que permitiera borrar la mitad de los datos y mantener la consistencia que tiene dos partes una para grupos y otra para conciertos, ya que borrando todas las hijas de una sola no era suficiente.
- Después de tener esta query lo siguiente fue probarla en la base de datos grande con el comando EXPLAIN para comprobar si tenía que devolver las tuplas que esperábamos, una vez se comprobó eso desactivamos la integridad referencial e hicimos las consultas directamente. La forma en la que se desactivó la integridad referencial fue la siguiente:

```
musicos on postgres@PostgreSQL 10
1 ALTER TABLE "Canciones" DISABLE TRIGGER ALL;
2 ALTER TABLE "Conciertos" DISABLE TRIGGER ALL;
3 ALTER TABLE "Discos" DISABLE TRIGGER ALL;
4 ALTER TABLE "Grupo" DISABLE TRIGGER ALL;
5 ALTER TABLE "Entradas" DISABLE TRIGGER ALL;
6 ALTER TABLE "Músicos" DISABLE TRIGGER ALL;
7 ALTER TABLE "Grupos_Tocan_Conciertos" DISABLE TRIGGER ALL;
```

- El resultado que se obtuvo fue muy similar al esperado por lo que volvimos a activar de nuevo la integridad referencial en cada tabla activando de nuevo los disparadores, que es el mecanismo que implementa PostgreSQL para comprobar la integridad.

- El tiempo empleado en el borrado ha sido alto en algunos casos, pero no mucho al desactivar la integridad referencial y es el que se ve en las siguientes capturas:



The screenshot shows a PostgreSQL query execution window. The title bar indicates the connection is to 'postgres@PostgreSQL 10'. The query editor contains two lines of SQL: 'DELETE FROM public."Musicos"' and 'WHERE "Codigo_grupo_Grupo"<100000;'. Below the query editor, there are tabs for 'Data Output', 'Explain', 'Messages', 'Notifications', and 'Query History'. The 'Messages' tab is selected, showing the output 'DELETE 501485' and 'Query returned successfully in 860 msec.'. The value '860 msec.' is highlighted with a red rectangular box.

```
musicos on postgres@PostgreSQL 10
```

```
1 DELETE FROM public."Musicos"
2 WHERE "Codigo_grupo_Grupo"<100000;
```

Data Output Explain Messages Notifications Query History

DELETE 501485

Query returned successfully in 860 msec.

musicos on postgres@PostgreSQL 10

```
1 DELETE FROM public."Canciones"
2   WHERE "Codigo_disco_Discos" IN (
3     SELECT "Codigo_disco"
4     FROM "Discos"
5     WHERE "Codigo_grupo_grupo"<100000);
```

Data Output Explain Messages Notifications Query History

DELETE 6009532

Query returned successfully in 2 min 2 secs.

musicos on postgres@PostgreSQL 10

```
1 DELETE FROM public."Grupo"
2   WHERE "Codigo_grupo"<100000;
```

Data Output Explain Messages Notifications Query History

DELETE 100000

Query returned successfully in 197 msec.

musicos on postgres@PostgreSQL 10

```
1 DELETE
2 FROM "Conciertos"
3 WHERE "Codigo_concierto" > 50000;
```

Data Output Explain Messages Notifications Query History

DELETE 49998

Query returned successfully in 112 msec.

musicos on postgres@PostgreSQL 10

```
1 DELETE FROM public."Discos"
2 WHERE "Codigo_grupo_Grupo"<100000;
```

Data Output Explain Messages Notifications Query History

DELETE 500804

Query returned successfully in 1 secs 460 msec.

musicos on postgres@PostgreSQL 10

```
1 DELETE
2 FROM "Entradas"
3 WHERE "Codigo_concierto_Conciertos" > 50000;
4
```

Data Output Explain Messages Notifications Query History

DELETE 12001616

Query returned successfully in 39 secs 199 msec.

musicos on postgres@PostgreSQL 10

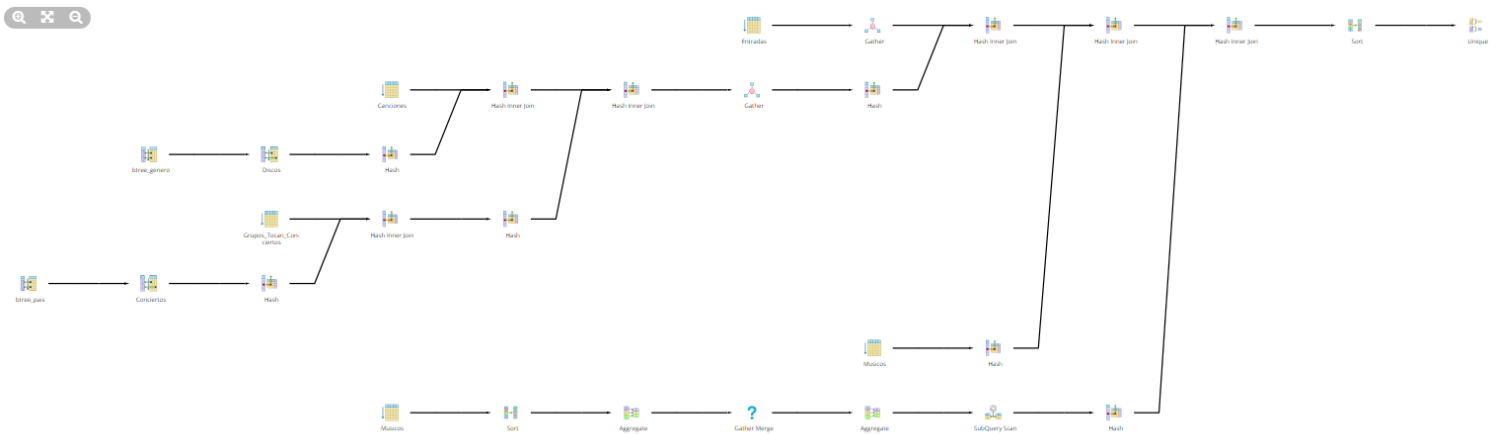
```
1 DELETE
2 FROM "Grupos_Tocan_Conciertos"
3 WHERE "Codigo_concierto_Conciertos" > 50000;
4
5
```

Data Output Explain Messages Notifications Query History

DELETE 649487

Query returned successfully in 857 msec.

- Aplicando ahora el plan de ejecución de la cuestión 13 conseguimos obtener el siguiente árbol además de un gran coste que luego se verá reducido drásticamente al aplicar un mantenimiento. La única diferencia obtenida en el dibujo del árbol es el cambio de orden en los dos últimos JOIN en el que se ha invertido, debido a las nuevas estadísticas obtenidas tras el borrado.



- En definitiva, tendremos un árbol muy similar al de la cuestión 13 aunque con un gran coste para recuperar menos datos que en la primera

Cuestión 18: Repita la cuestión 17 con MySQL y compare los resultados con PostgreSQL.

- El proceso seguido para el borrado de datos ha sido similar al de PostgreSQL. En primer lugar, se ha modificado la integridad referencial para facilitar el borrado con SET foreign_key_checks=0, aunque esto solamente lo hace para la sesión activa. El siguiente paso ha sido hacer el borrado con las queries que teníamos de PostgreSQL, con sus respectivas modificaciones para las comillas.
- El tiempo de borrado ha sido el que se muestra en la siguiente tabla:

Tabla	Tiempo de borrado
Canciones	515,219 segundos
Discos	80,484 segundos
Entradas	1417,5 segundos
Grupos_Tocan_Conciertos	177,969 segundos
Grupo	4,968 segundos
Conciertos	12,391 segundos
Musicos	65,375 segundos

➤ Algunos ejemplos del borrado en mysql:

SQL File 4* x musicos.canciones

```

1 DELETE FROM `Discos`
2 WHERE `Grupo_Codigo_grupo` > 100000;
3

```

Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.

Context Help Snippets

Output

Action Output

#	Time	Action	Message	Duration / Fetch
16	20:11:43	KILL 16	Error Code: 1317. Query execution was interrupted	0.000 sec
17	20:11:51	SHOW FULL processlist	3 row(s) returned	0.000 sec / 0.000 sec
18	20:14:14	ANALYZE TABLE `musicos`.`canciones`	OK	0.000 sec
19	20:14:21	ANALYZE TABLE `musicos`.`canciones`	OK	0.000 sec
20	20:17:57	SELECT `canciones`.`Codigo_cancion`, `canciones`.`Nombre`, ...	100 row(s) returned	0.015 sec / 0.000 sec
21	20:19:39	DELETE FROM `Discos` WHERE `Grupo_Codigo_grupo` > 100000	499194 row(s) affected	80.484 sec

SQL File 4* x musicos.canciones

```

1 DELETE FROM `grupos_tocan_conciertos`
2 WHERE `Grupo_Codigo_grupo` > 100000;

```

Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.

Context Help Snippets

Output

Action Output

#	Time	Action	Message	Duration / Fetch
19	20:14:21	ANALYZE TABLE `musicos`.`canciones`	OK	0.000 sec
20	20:17:57	SELECT `canciones`.`Codigo_cancion`, `canciones`.`Nombre`, ...	100 row(s) returned	0.015 sec / 0.000 sec
21	20:19:39	DELETE FROM `Discos` WHERE `Grupo_Codigo_grupo` > 100000	499194 row(s) affected	80.484 sec
22	20:22:10	DELETE FROM `Musicos` WHERE `Grupo_Codigo_grupo` > 100000	498514 row(s) affected	65.375 sec
23	20:26:32	DELETE FROM `grupos_tocan_conciertos` WHERE `Grupo_Codig...	1250568 row(s) affected	177.969 sec

- Al aplicar la consulta de la pregunta 13 hemos obtenido el siguiente resultado con el EXPLAIN:

SQL File 4+ musicos.canciones musicos.entradas

```
1 EXPLAIN SELECT `musicos`.`Nombre`,`musicos`.`Grupo_Codigo_grupo`
2 FROM `musicos` INNER JOIN (SELECT `Grupo_Codigo_grupo`, COUNT(`Grupo_Codigo_grupo`) AS cuenta
3 FROM `musicos`
4 GROUP BY(`musicos`.`Grupo_Codigo_grupo`) ) AS miembros ON miembros.`Grupo_Codigo_grupo`= `musicos`.`Grupo_Codigo_grupo`
5 INNER JOIN `discos` ON `musicos`.`Grupo_Codigo_grupo`= `discos`.`Grupo_Codigo_grupo`
6 INNER JOIN `canciones` ON `discos`.`Codigo_disco`= `canciones`.`Discos_Codigo_disco`
7 INNER JOIN `grupos_Tocan_Conciertos` ON `discos`.`Grupo_Codigo_grupo`= `grupos_Tocan_Conciertos`.`Grupo_Codigo_grupo`
8 INNER JOIN `conciertos` ON `grupos_Tocan_Conciertos`.`Conciertos_Codigo_concierto` = `conciertos`.`Codigo_concierto`
9 INNER JOIN `entradas` ON `conciertos`.`Codigo_concierto`= `entradas`.`Conciertos_Codigo_concierto`
10 WHERE `Genero`='rock' AND `Duracion`>'3:00' AND `Precio` BETWEEN 20.00 AND 50.00 AND `conciertos`.`Pais`='España' AND (cuenta>3);
```

<

Result Grid Filter Rows: Export: Wrap Cell Content: 15

	id	select_type	table	part	type	possible_keys	key	key_ref	rows	filtered	Extra
▶	1	PRIMARY	conciertos	ALL	ALL	PRIMARY,idx_conciertos_Pais	PRIMARY	NULL	49861	0.00	Using where
	1	PRIMARY	grupos_Tocan_Conciertos	ref	ref	PRIMARY,R_Grupos_Tocan_Conciertos_Con...	PRIMARY	4	musicos.conciertos.Codigo...	26	100.00 Using index
	1	PRIMARY	discos	ref	ref	PRIMARY,R_Discos_Grupo1_idx_idx_discos...	R_Discos_Grupo1_idx	4	musicos.grupos_Tocan_Co...	4	1.01 Using where
	1	PRIMARY	canciones	ref	ref	R_Canciones_Discos1_idx_idx_canciones_D...	R_Canciones_Discos1_idx	4	musicos.discos.Codigo_disco	13	50.00 Using where
	1	PRIMARY	entradas	ref	ref	R_Entradas_Conciertos_idx_idx_entradas_P...	R_Entradas_Conciertos_idx	4	musicos.conciertos.Codigo...	141	50.00 Using where
	1	PRIMARY	musicos	ref	ref	R_Musicos_Grupo1_idx	R_Musicos_Grupo1_idx	4	musicos.grupos_Tocan_Co...	5	100.00 Using where
	1	PRIMARY	<derived2>	ref	ref	<auto_key1>	<auto_key1>	4	musicos.grupos_Tocan_Co...	16	33.33 Using where
	2	DERIVED	musicos	index	index	R_Musicos_Grupo1_idx	R_Musicos_Grupo1_idx	4	516192	100.00	Using index

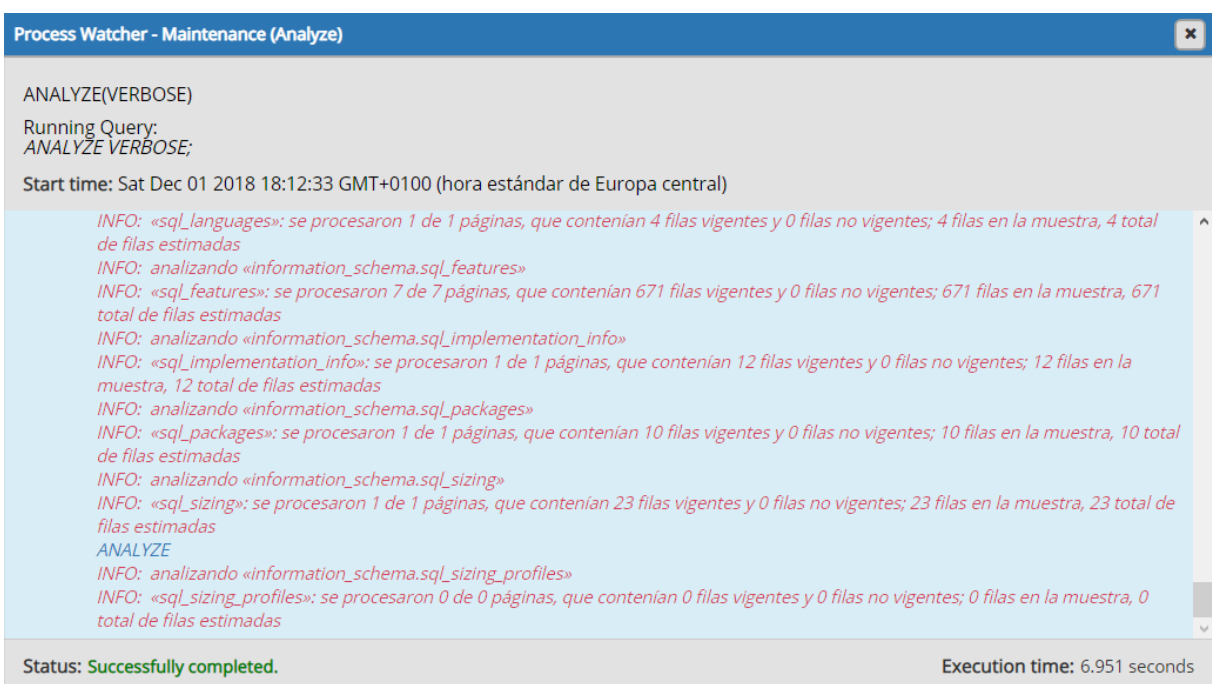
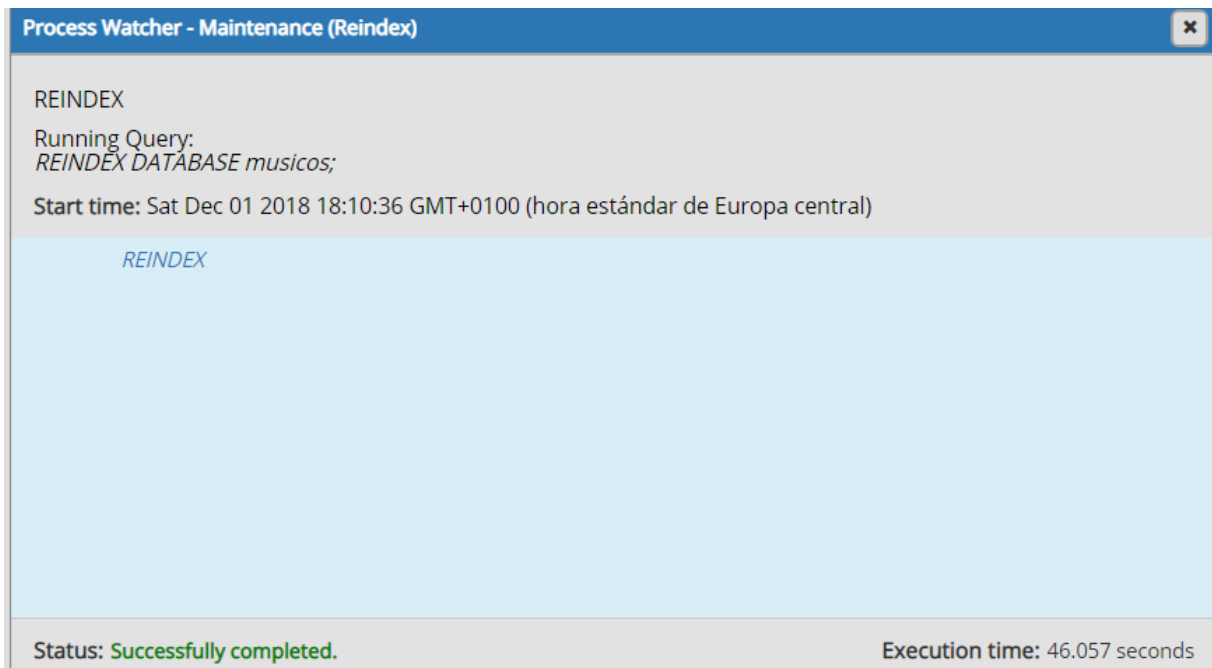
- Haciendo una comparación general con PostgreSQL se puede observar la diferencia de tiempo de borrado que hay entre ambos, a pesar de que MySQL tiene hasta 3GB de memoria disponibles y PostgreSQL 2GB. En estos tiempos se puede observar cómo, aunque tenga pocas entradas la tabla, su tiempo de borrado es muy superior al de PostgreSQL siendo esta diferencia mucho mayor cuando hay muchas entradas en la tabla como ocurre con entradas que en PostgreSQL el tiempo es de apenas 40 segundos y en MySQL es de 1417 segundos.
- Dejando los tiempos de borrado a un lado tenemos que la eficiencia de ambas consultas tras el borrado es similar, aunque en MySQL se pierde la capacidad de usar los índices que se crearon antes.

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?

- Vacuum: Limpia los registros que han sido borrados y libera el espacio que éstos ocupan. Optaremos por usar esta herramienta, ya que se han borrado muchos datos en la cuestión anterior, pero no se ha liberado el espacio que estos ocupaban. Existe también la posibilidad de realizar VACUUM FULL, pero decidimos no realizarlo porque el esfuerzo para obtener un extra de memoria no merece la pena.
- Analyze: Esta herramienta nos permitirá actualizar las estadísticas de nuestro planificador. También optaremos por usarla, ya que se han borrado muchos datos y el planificador necesita tener las nuevas estadísticas para trabajar más eficientemente.
- Reindex: Con esta herramienta reconstruiremos los índices, ya que al borrar tantos datos nuestras tablas se han reducido en tamaño y los índices tienen que actualizarse para trabajar de forma óptima.
- Cluster: Esta herramienta ordena físicamente todas las tuplas, pero solo se puede hacer sobre un campo y lleva mucho tiempo. Además, tiene un acceso exclusivo, por lo que no se pueden realizar operaciones en la base de datos mientras se utiliza. Por estas razones, hemos decidido no usarlo.

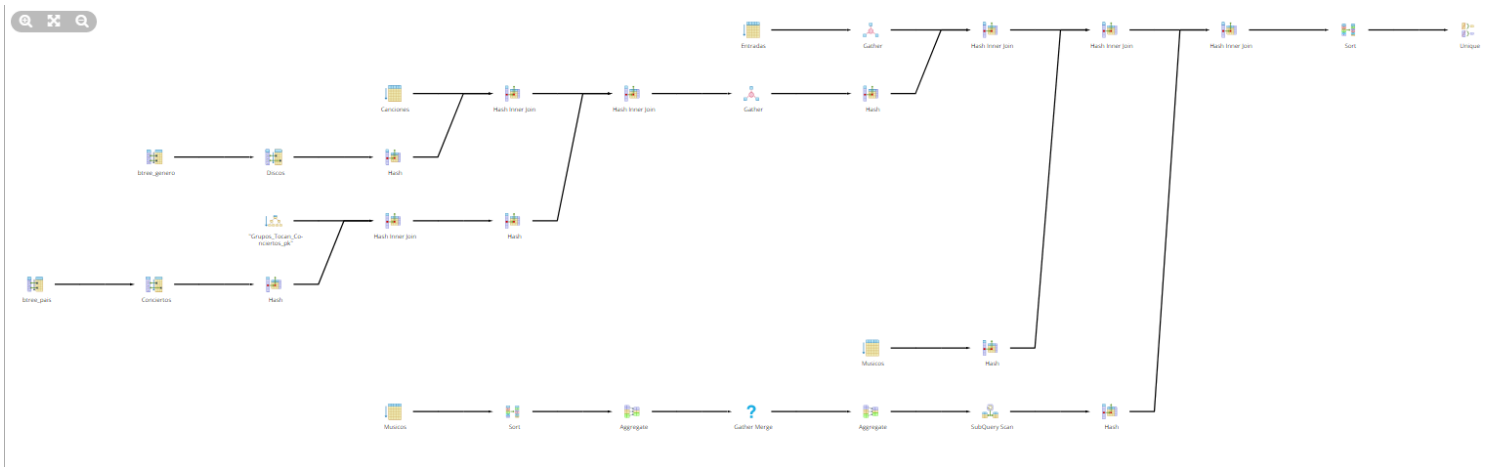
Cuestión 20: Usando PostgreSQL, lleve a cabo las operaciones propuestas en la cuestión anterior y ejecute el plan de ejecución de la misma consulta. Dibujar un diagrama con el nuevo resultado del comando EXPLAIN en forma de árbol de álgebra relacional. Compare los resultados del plan de ejecución con los de los apartados anteriores. Coméntelos.

- Para aplicar las técnicas de mantenimiento de la cuestión anterior en PostgreSQL seleccionaremos nuestra base de datos e iremos a Tools>Maintenance y ahí nos permitirá aplicarlas con varios parámetros que podremos seleccionar. En todos los utilizados aplicaremos el parámetro VERBOSE para que se muestren las operaciones que realiza PostgreSQL:





- Tras aplicar las técnicas de mantenimiento, ejecutaremos de nuevo la consulta de cuestión 13 para obtener el árbol del álgebra relacional:



- Como podemos ver, el árbol es el mismo que en la cuestión 17, ya que la consulta es la misma y no se han introducido mecanismos de optimización. Lo que si podemos ver al aplicar el comando EXPLAIN es que el coste de la consulta es menor que en la 17, ya que hemos liberado muchos registros y la búsqueda es más ágil.

Cuestión 21: Repita la cuestión 20 con MySQL. Compare los resultados con los obtenidos para PostgreSQL.

- El mantenimiento en MySQL es muy distinto a PostgreSQL aquí solo puedes recuperar la memoria con OPTIMIZE TABLE no hay más opciones disponibles. También puedes hacer un ANALYZE para actualizar las estadísticas que se almacenan de las tablas. Aparte de eso hay otras funciones más de mantenimiento que se centran en la detección y corrección de errores.
- En nuestro caso únicamente aplicamos OPTIMIZE TABLE ya que era necesario tras haber borrado tantas tuplas de las tablas; aunque no siempre es recomendable usarlo. Al aplicar esta query, como se emplea INNODB para el almacenamiento de datos, la única opción que se le permitía para hacer la optimización era crear una nueva tabla donde se vuelcan los registros de la otra y borrar la anterior, para finalmente hacer ANALYZE obteniendo las nuevas estadísticas de la tabla. Este proceso se ve en el mensaje que devuelve tras el OPTIMIZE TABLE en la captura; aunque ponga que no soporta la optimización sí que lo hace, lo único que lo hace de esta forma que es distinta a como MyISAM lo hace.

The screenshot shows a MySQL SQL File editor window titled 'musicos' with a tab 'SQL File 3*'. The editor contains the command 'OPTIMIZE TABLE `entradas`'. Below the editor, the 'Result Grid' shows the output of the command:

Table	Op	Msg_type	Msg_text
musicos.entradas	optimize	note	Table does not support optimize, doing recreate + analyze instead
musicos.entradas	optimize	status	OK

Below the Result Grid, the 'Output' window shows the 'Action Output' for the command. It displays a list of actions performed on various tables, including 'canciones', 'musicos', 'discos', 'grupo', 'grupos_tocan_conciertos', 'conciertos', and 'entradas'. Each action is marked with a green checkmark and shows the time taken and the message returned.

#	Time	Action	Message	Duration / Fetch
1	22:59:56	OPTIMIZE TABLE `canciones`	2 row(s) returned	77.015 sec / 0.000 sec
2	23:03:11	OPTIMIZE TABLE `musicos`	2 row(s) returned	9.579 sec / 0.000 sec
3	23:06:30	OPTIMIZE TABLE `discos`	2 row(s) returned	21.578 sec / 0.000 sec
4	23:07:57	OPTIMIZE TABLE `grupo`	2 row(s) returned	1.641 sec / 0.000 sec
5	23:08:31	OPTIMIZE TABLE `grupos_tocan_conciertos`	2 row(s) returned	6.766 sec / 0.000 sec
6	23:09:08	OPTIMIZE TABLE `conciertos`	2 row(s) returned	2.594 sec / 0.000 sec
7	23:09:43	OPTIMIZE TABLE `entradas`	2 row(s) returned	128.282 sec / 0.000 sec

- Si volvemos a realizar la pregunta 13 obtenemos exactamente el mismo mensaje que en la pregunta del borrado, lo que indica que apenas se ha optimizado y los índices ya no son válidos para buscar los datos, en definitiva, que es como si no hubiéramos hecho nada tras el borrado.

SQL File 3"

```

1 EXPLAIN SELECT `musicos`.`Nombre`,`musicos`.`Grupo_Codigo_grupo`
2 FROM `musicos` INNER JOIN (SELECT `Grupo_Codigo_grupo`, COUNT(`Grupo_Codigo_grupo`) AS cuenta
3 FROM `musicos`
4 GROUP BY(`musicos`.`Grupo_Codigo_grupo`) ) AS miembros ON miembros.`Grupo_Codigo_grupo` = `musicos`.`Grupo_Codigo_grupo`
5 INNER JOIN `discos` ON `musicos`.`Grupo_Codigo_grupo` = `discos`.`Grupo_Codigo_grupo`
6 INNER JOIN `cantones` ON `discos`.`Codigo_disco` = `cantones`.`Discos_Codigo_disco`
7 INNER JOIN `grupos_Tocan_Conciertos` ON `discos`.`Grupo_Codigo_grupo` = `grupos_Tocan_Conciertos`.`Grupo_Codigo_grupo`
8 INNER JOIN `conciertos` ON `grupos_Tocan_Conciertos`.`Conciertos_Codigo_concierto` = `conciertos`.`Codigo_concierto`
9 INNER JOIN `entradas` ON `conciertos`.`Codigo_concierto` = `entradas`.`Conciertos_Codigo_concierto`
10 WHERE `Genero` = 'rock' AND `Duracion` > '3:00' AND `Precio` BETWEEN 20.00 AND 50.00 AND `conciertos`.`Pais` = 'España' AND (cuenta > 3)

```

id	select_type	table	part	type	possible_keys	key	key_ref	rows	filtered	Extra
1	PRIMARY	conciertos	ALL	PRIMARY	idx_conciertos_Pais	idx	idx	49948	0.00	Using where
1	PRIMARY	grupos_Tocan_Conciertos	ref	PRIMARY	fk_Grupos_Tocan_Conciertos_Con...	fk_Grupos_Tocan_Conciertos_Conciertos_...	4	musicos.conciertos.Codigo...	12	100.00 Using index
1	PRIMARY	discos	ref	PRIMARY	fk_Discos_Grupo1_idx	fk_Discos_Grupo1_idx	4	musicos.grupos_Tocan_Co...	4	1.01 Using where
1	PRIMARY	cantones	ref	PRIMARY	fk_Cantones_Discos1_idx	fk_Cantones_Discos1_idx	4	musicos.discos.Codigo_disco	11	50.00 Using where
1	PRIMARY	entradas	ref	PRIMARY	fk_Entradas_Conciertos_idx	fk_Entradas_Conciertos_idx	4	musicos.conciertos.Codigo...	242	50.00 Using where
1	PRIMARY	musicos	ref	PRIMARY	fk_Musicos_Grupo1_idx	fk_Musicos_Grupo1_idx	4	musicos.grupos_Tocan_Co...	4	100.00 Using index
1	PRIMARY	<derived2>	ref	PRIMARY	<auto_key1>	<auto_key1>	4	musicos.grupos_Tocan_Co...	16	33.33 Using where
2	DERIVED	musicos	index	PRIMARY	fk_Musicos_Grupo1_idx	fk_Musicos_Grupo1_idx	4	musicos.grupos_Tocan_Co...	498968	100.00 Using index

Result 8 x

Output

Action Output

#	Time	Action	Message	Duration / Fetch
7	23:09:43	OPTIMIZE TABLE 'entradas'	2 row(s) returned	128.282 sec / 0.000 sec
8	23:13:39	EXPLAIN SELECT `musicos`.`Nombre`,`musicos`.`Grupo_Codigo_grupo` FROM `musicos` INNER JOIN ...	8 row(s) returned	0.015 sec / 0.000 sec

- Comparando esto con PostgreSQL salta a la vista como apenas tenemos opciones de personalizar las consultas de mantenimiento a diferencia de PostgreSQL que ofrece cuatro tipos que a su vez pueden tener más opciones. Tampoco podemos reconstruir índices en MySQL como si se puede en PostgreSQL, lo cual significa que hay que borrar el índice y reconstruirlo entero o mantenerlo ineficientemente actualizando las estadísticas.
- Como conclusión se puede observar que PostgreSQL ofrece un sistema de mantenimiento mucho más eficiente, personalizable y completo que MySQL que ofrece unas funciones limitadas y básicas.

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.

- Tras analizar el log de operaciones de nuestra base de datos hemos llegado a la conclusión de que las consultas más utilizadas en nuestra práctica han sido del tipo COPY, DELETE, EXPLAIN, SELECT e INNER JOIN.
- En cuanto al número de consultas, si revisamos el log vemos que se han realizado muchas más consultas internas que las que hemos realizado nosotros, por lo que vemos que se han llevado a cabo un gran número de consultas. En la imagen podemos ver un ejemplo de sentencia que ejecuta la base de datos internamente:

```

2018-11-20 12:51:19.881 CET [16224] LOG:  sentencia: /*pga4dash*/
SELECT
  (SELECT sum(blks_read) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 49534)) AS "Reads",
  (SELECT sum(blks_hit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 49534)) AS "Hits"
2018-11-20 12:51:19.882 CET [16224] LOG:  duraci3n: 1.557 ms

```

- En cada consulta realizada podemos ver el tiempo de ejecución de esta, y sino está la duración de la consulta a continuación de esta, deberemos buscar en el log el número identificativo asociado a la consulta hasta encontrar su duración.

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>