

Titulación: Grado en Ingeniería Informática y Sistemas de Información
Curso: 2019-2020. Convocatoria Ordinaria de Junio
Asignatura: Bases de Datos Avanzadas – Laboratorio

Practica 2: Carga Masiva de Datos, Procesamiento y Optimización de Consultas

ALUMNO 1:

Nombre y Apellidos: Adina Murg

DNI:

ALUMNO 2:

Nombre y Apellidos: Victoria Lorena Ordenes Orbegozo

DNI:

Fecha: 21/04/2020

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 calificará la asignatura como Suspenso – Cero.

Plazos

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

Entrega de práctica: Semana 14 de Abril (Martes). Aula Virtual

Documento a entregar: Este mismo fichero con las respuestas a las cuestiones planteadas y el programa que genera los datos de carga de la base de datos. No se pide el script de carga de los datos de la base de datos. Se entregará en un ZIP comprimido llamado: **DNI 'sdelosAlumnos_PECL2.zip**

AMBOS ALUMNOS DEBEN ENTREGAR EL FICHERO EN LA PLATAFORMA.

Introducción

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

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

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

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

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

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

Actividades y Cuestiones

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

1. Recolector de estadísticas

Sí, el servidor de Postgres posee un recolector de estadísticas, que es un subsistema que admite la recopilación de informes de información de actividad del servidor.

El recopilador puede contar el acceso a tablas e índices tanto en términos de bloque de disco como de filas individuales. También rastrea el número total de filas en cada tabla, y los últimos tiempos de vacío y análisis para cada tabla. También puede contar llamadas a funciones definidas por el usuario y el tiempo total empleado en cada una.

2. Qué tipos de estadísticas recolecta y dónde se guardan

Los tipos de estadísticas que se recolectan son estadísticas dinámicas y recopiladas.

Vistas de Estadísticas Dinámicas

<code>pg_stat_activity</code>	Una fila por muestra de información relacionada con la actividad actual de ese proceso, como el estado y la consulta actual.
<code>pg_stat_replication</code>	Una fila por proceso de remitente WAL, que muestra estadísticas sobre la replicación al servidor en espera conectado de ese remitente.
<code>pg_stat_wal_receiver</code>	Solo una fila, que muestra estadísticas sobre el receptor WAL del servidor conectado de ese receptor.
<code>pg_stat_subscription</code>	Al menos una fila por suscripción, que muestra información sobre los trabajadores de suscripción.
<code>pg_stat_ssl</code>	Una fila por conexión (regular y replicación), que muestra información sobre SSL utilizada en esta conexión.
<code>pg_stat_gssapi</code>	Una fila por conexión (regular y replicación), que muestra información sobre la autenticación y el cifrado GSSAPI utilizados en esta conexión.
<code>pg_stat_progress_create_index</code>	Una fila para cada back-end que se ejecuta CREATE INDEX o REINDEX, que muestra el progreso actual.
<code>pg_stat_progress_vacuum</code>	Una fila para cada backend (incluidos los procesos de trabajo de vacío automático) en ejecución VACUUM, que muestra el progreso actual.
<code>pg_stat_progress_cluster</code>	Una fila para cada back-end que se ejecuta CLUSTER o VACUUM FULL, que muestra el progreso actual.

Vista de Estadísticas Recopiladas

pg_stat_archiver	Solo una fila, que muestra estadísticas sobre la actividad del proceso del archivador WAL.
pg_stat_bgwriter	Solo una fila, que muestra estadísticas sobre la actividad del proceso del escritor en segundo plano.
pg_stat_database	Una fila por base de datos, que muestra estadísticas de toda la base de datos.
pg_stat_database_conflicts	Una fila por base de datos, que muestra estadísticas de toda la base de datos sobre cancelaciones de consultas debido a conflictos con la recuperación en servidores en espera.
pg_stat_all_tables	Una fila para cada tabla en la base de datos actual, que muestra estadísticas sobre los accesos a esa tabla específica.
pg_stat_sys_tables	Igual que pg_stat_all_tables, excepto que solo se muestran las tablas del sistema.
pg_stat_user_tables	Igual que pg_stat_all_tables, excepto que solo se muestran las tablas de usuario.
pg_stat_xact_all_tables	Similar a pg_stat_all_tables, pero cuenta las acciones realizadas hasta ahora dentro de la transacción actual (que aún <i>no</i> se incluyen en pg_stat_all_tables las vistas relacionadas). Las columnas para los números de filas vivas y muertas y acciones de vacío y análisis no están presentes en esta vista.
pg_stat_xact_sys_tables	Igual que pg_stat_xact_all_tables, excepto que solo se muestran las tablas del sistema.
pg_stat_xact_user_tables	Igual que pg_stat_xact_all_tables, excepto que solo se muestran las tablas de usuario.
pg_stat_all_indexes	Una fila para cada índice en la base de datos actual, que muestra estadísticas sobre los accesos a ese índice específico.
pg_stat_sys_indexes	Igual que pg_stat_all_indexes, excepto que solo se muestran los índices en las tablas del sistema.
pg_stat_user_indexes	Igual que pg_stat_all_indexes, excepto que solo se muestran los índices en las tablas de usuario.
pg_statio_all_tables	Una fila para cada tabla en la base de datos actual, que muestra estadísticas sobre E / S en esa tabla específica.
pg_statio_sys_tables	Igual que pg_statio_all_tables, excepto que solo se muestran las tablas del sistema.
pg_statio_user_tables	Igual que pg_statio_all_tables, excepto que solo se muestran las tablas de usuario.

<code>pg_statio_all_indexes</code>	Una fila para cada índice en la base de datos actual, que muestra estadísticas sobre E / S en ese índice específico.
<code>pg_statio_sys_indexes</code>	Igual que <code>pg_statio_all_indexes</code> , excepto que solo se muestran los índices en las tablas del sistema.
<code>pg_statio_user_indexes</code>	Igual que <code>pg_statio_all_indexes</code> , excepto que solo se muestran los índices en las tablas de usuario.
<code>pg_statio_all_sequences</code>	Una fila para cada secuencia en la base de datos actual, que muestra estadísticas sobre E / S en esa secuencia específica.
<code>pg_statio_sys_sequences</code>	Igual que <code>pg_statio_all_sequences</code> , excepto que solo se muestran las secuencias del sistema. (Actualmente, no se definen secuencias del sistema, por lo que esta vista siempre está vacía).
<code>pg_statio_user_sequences</code>	Igual que <code>pg_statio_all_sequences</code> , excepto que solo se muestran las secuencias de usuario.
<code>pg_stat_user_functions</code>	Una fila para cada función rastreada, que muestra estadísticas sobre ejecuciones de esa función.
<code>pg_stat_xact_user_functions</code>	Similar a <code>pg_stat_user_functions</code> , pero solo cuenta las llamadas durante la transacción actual (que aún <i>no</i> están incluidas <code>pg_stat_user_functions</code>).

El contenido de ambas tabla ha sido extraído del siguiente [enlace](#), que nos redirige a la página oficial de Postgres.

Como también mencionan en el enlace de la página oficial que utilizamos estos archivos se almacenan en el directorio nombrado por el parámetro `stats_temp_directory`, `pg_stat_tmp` de forma predeterminada.

Cuestión 2: Modifique el log de errores para que queden guardadas todas las operaciones que se realizan sobre cualquier base de datos. Indique los pasos realizados.

Por defecto los logs en Postgres solo apuntan errores y avisos de problemas. Pasaremos a modificar esto para que pueda guardar información también de delete, update, insert, truncate, copy from, etc.

1. DesdeC://>PostgreSQL>12(versión)>data> seleccionaremos postgresql.conf que es un archivo que pasaremos a modificar con un editor de texto.
2. Modificamos `log_statement = 'none'` por `'all'`. Lo modificamos por `all` para que registre todas las instrucciones. También borraremos el `'#'` para que deje de ser un comentario.

```

# e.g. '<u$$$d> '
#log_lock_waits = off          # log lock waits >= deadlock_timeout
log_statement = 'all'
# none, ddl, mod, all
#log_replication_commands = off
#log_temp_files = -1          # log temporary files equal or larger
                              # than the specified size in kilobytes;
                              # -1 disables, 0 logs all temp files
log_timezone = 'Europe/Brussels'

```

Nota: el parámetro **log_statement** se encarga de controlar que instrucciones SQL se registran. Podemos seleccionar none (ninguno), ddl (solo registra create, alter y drop), mod (registra todas las declaraciones ddl además de insert, update, delete, etc.) o all (registra todas las instrucciones). Por defecto el valor asignado es el de none.

3. Pasaremos al when to log:

log_min_messages: controla el nivel de la información de los mensajes que recibimos. Con debug5 nos proporcionará más mensajes y con el máximo de detalles. Por defecto el valor es de warning, que sólo nos informa de warnings y problemas.

log_min_error_statement: controla qué declaraciones que causen error se registrarán en el log. Por defecto el valor es error, que solo registra los mensajes de log, fatal errors y los considerados modos panic.

log_min_duration_statement: establece el límite de ejecución en milisegundos para registrar las instrucciones. Por defecto el valor es de -1, que modificaremos por 0 de forma que guarde en el log todas las declaraciones independientemente de sus tiempos. Todas las instrucciones SQL que se ejecuten durante más tiempo del establecido en este parámetro se registran.

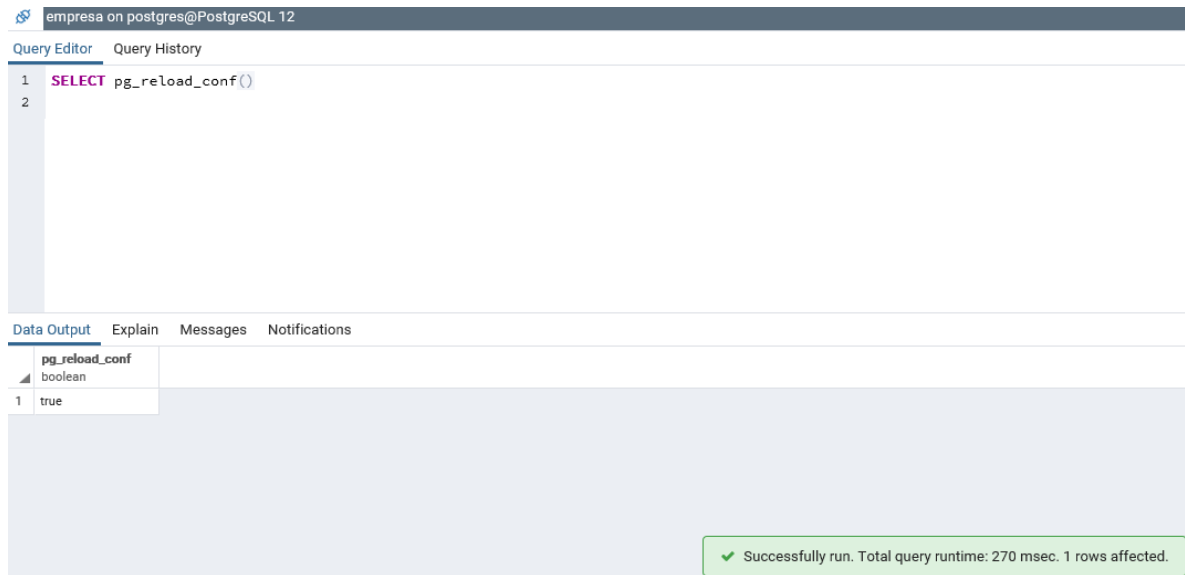
```

461 # - When to Log -
462 #
463 #log_min_messages = warning      # values in order of decreasing detail:
464 #                               # debug5
465 #                               # debug4
466 #                               # debug3
467 #                               # debug2
468 #                               # debug1
469 #                               # info
470 #                               # notice
471 #                               # warning
472 #                               # error
473 #                               # log
474 #                               # fatal
475 #                               # panic
476
477 #log_min_error_statement = error  # values in order of decreasing detail:
478 #                               # debug5
479 #                               # debug4
480 #                               # debug3
481 #                               # debug2
482 #                               # debug1
483 #                               # info
484 #                               # notice
485 #                               # warning
486 #                               # error
487 #                               # log
488 #                               # fatal
489 #                               # panic (effectively off)
490
491 log_min_duration_statement = 0
492 # -1 is disabled, 0 logs all statements
493 #   and their durations, > 0 logs only
494 #   statements running at least this number
495 #   of milliseconds
496
497 #log_transaction_sample_rate = 0.0 # Fraction of transactions whose statements
498 #   are logged regardless of their duration. 1.0 logs all

```

De todos estos parámetros modificaremos sólo **log_min_duration_statement** por el valor 0 como hemos mencionado, obviamente volviendo a eliminar el '#'.

4. Por último, siguiendo las instrucciones de postgresql.conf guardamos y hacemos un `SELECT pg_reload_conf()` para que nuestra base de datos guarde las modificaciones.



The screenshot shows a PostgreSQL query editor interface. At the top, the connection is labeled 'empresa on postgres@PostgreSQL 12'. Below the connection bar, there are tabs for 'Query Editor' and 'Query History'. The 'Query Editor' tab is active, showing a single query: `SELECT pg_reload_conf()`. Below the query editor, there are tabs for 'Data Output', 'Explain', 'Messages', and 'Notifications'. The 'Data Output' tab is active, displaying the result of the query. The result is a single row with the value 'true'. At the bottom right of the interface, a green status bar indicates: 'Successfully run. Total query runtime: 270 msec. 1 rows affected.'

Nota: la información se ha extraído del manual, apartado [aquí](#).

Por último, mostramos una comparación de un log anterior a las modificaciones y uno después para confirmar que hemos realizado correctamente las modificaciones.

ANTES:


```
2020-04-04 18:20:37.832 CEST [9116] ERROR: error de sintaxis en o cerca de Â«errorÂ» en carÃcter 1
2020-04-04 18:20:37.832 CEST [9116] SENTENCIA: error log

2020-04-04 18:21:06.550 CEST [9116] ERROR: no existe la columna Â«errorÂ» en carÃcter 8
2020-04-04 18:21:06.550 CEST [9116] SENTENCIA: select error log;

2020-04-04 18:21:13.516 CEST [9116] ERROR: error de sintaxis en o cerca de Â«errorÂ» en carÃcter 10
2020-04-04 18:21:13.516 CEST [9116] SENTENCIA: select * error log;

2020-04-04 18:21:16.722 CEST [9116] ERROR: error de sintaxis en o cerca de Â«errorÂ» en carÃcter 10
2020-04-04 18:21:16.722 CEST [9116] SENTENCIA: select * error.log;

2020-04-04 18:21:23.662 CEST [9116] ERROR: error de sintaxis en o cerca de Â«logÂ» en carÃcter 10
2020-04-04 18:21:23.662 CEST [9116] SENTENCIA: select * log;

2020-04-04 18:49:22.145 CEST [8804] ERROR: no existe la relaciÃ³n Â«mitablajhdxdbÂ» en carÃcter 15
2020-04-04 18:49:22.145 CEST [8804] SENTENCIA: select * from mitablajhdxdb;

2020-04-04 18:51:47.541 CEST [14060] ERROR: no existe la relaciÃ³n Â«aaaaaÂ» en carÃcter 15
2020-04-04 18:51:47.541 CEST [14060] SENTENCIA: select * from aaaaa;
2020-04-04 18:56:20.534 CEST [14060] ERROR: no existe la relaciÃ³n Â«mitablamitablaÂ» en carÃcter 15
2020-04-04 18:56:20.534 CEST [14060] SENTENCIA: select * from mitablamitabla;
2020-04-04 19:12:31.114 CEST [13224] ERROR: error de sintaxis en o cerca de Â«mariaÂ» en carÃcter 8
2020-04-04 19:12:31.114 CEST [13224] SENTENCIA: insert maria in nombre;
2020-04-04 19:14:01.905 CEST [13224] ERROR: el valor null para la columna Â«id_clienteÂ» viola la restricciÃ³n not null
2020-04-04 19:14:01.905 CEST [13224] DETALLE: La fila que falla contiene (null, Pepito, null, null, null).
2020-04-04 19:14:01.905 CEST [13224] SENTENCIA: INSERT INTO mitabla (nombre) VALUES ('Pepito');
2020-04-04 19:29:38.515 CEST [13224] ERROR: error de sintaxis en o cerca de Â«sudoÂ» en carÃcter 1
2020-04-04 19:29:38.515 CEST [13224] SENTENCIA: sudo service postgresql restart
```

AHORA:

```
2020-04-05 13:30:51.168 CEST [10180] LOG: duraciÃ³n: 43.670 ms
2020-04-05 13:30:52.510 CEST [10180] LOG: sentencia: /*pga4dash*/
SELECT 'session_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
  (SELECT count(*) FROM pg_stat_activity WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Total",
  (SELECT count(*) FROM pg_stat_activity WHERE state = 'active' AND datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Active",
  (SELECT count(*) FROM pg_stat_activity WHERE state = 'idle' AND datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Idle"
) t
UNION ALL
SELECT 'tps_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
  (SELECT sum(xact_commit) + sum(xact_rollback) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Transactions",
  (SELECT sum(xact_commit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Commits",
  (SELECT sum(xact_rollback) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Rollbacks"
) t
UNION ALL
SELECT 'ti_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
  (SELECT sum(tup_inserted) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Inserts",
  (SELECT sum(tup_updated) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Updates",
  (SELECT sum(tup_deleted) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Deletes"
) t
UNION ALL
SELECT 'to_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
  (SELECT sum(tup_fetched) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Fetched",
  (SELECT sum(tup_returned) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Returned"
) t
UNION ALL
SELECT 'bio_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
  (SELECT sum(blks_read) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Reads",
  (SELECT sum(blks_hit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Hits"
) t
2020-04-05 13:30:52.620 CEST [10180] LOG: duraciÃ³n: 110.765 ms
```

También destacaremos la diferencia del peso, dado que por general nuestros logs que contienen sólo errores oscilan como máximo a 10kb y el último ya con las modificaciones ha llegado hasta los 6.550kb.

```

(SELECT sum(tup_fetched) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Fetched",
(SELECT sum(tup_returned) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Returned"
) t
UNION ALL
SELECT 'bio_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
(SELECT sum(blks_read) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Reads",
(SELECT sum(blks_hit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Hits"
) t
2020-04-05 13:34:31.054 CEST [10180] LOG:  duraciÃ³n: 29.303 ms
2020-04-05 13:34:31.533 CEST [6060] LOG:  se recibió SIGHUP, volviendo a cargar archivos de configuración
2020-04-05 13:34:31.534 CEST [6060] LOG:  parámetro «log_min_duration_statement» eliminado del archivo de configuración, volviendo al valor por omisión
2020-04-05 13:34:31.535 CEST [6060] LOG:  parámetro «log_statement» eliminado del archivo de configuración, volviendo al valor por omisión
2020-04-05 13:34:41.610 CEST [13856] ERROR:  no existe la columna «lkmjlk» en carÃ¡cter 8
2020-04-05 13:34:41.610 CEST [13856] SENTENCIA:  SELECT lkmjlk

```

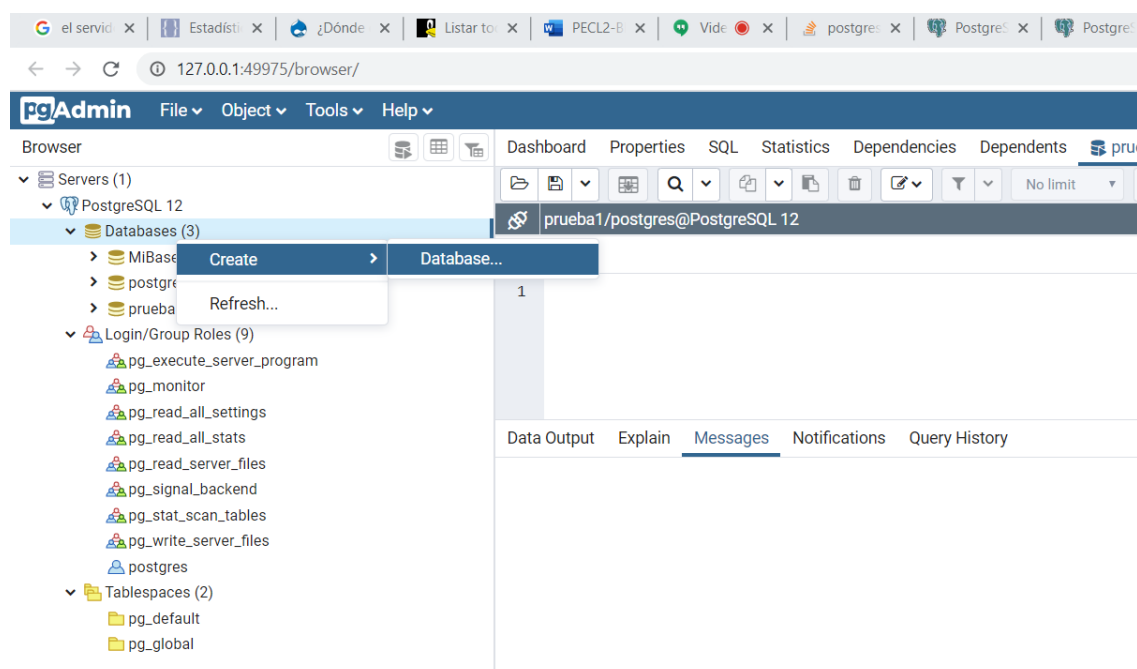
Cuestión 3: Crear una nueva base de datos llamada empresa y que tenga las siguientes tablas con los siguientes campos y características:

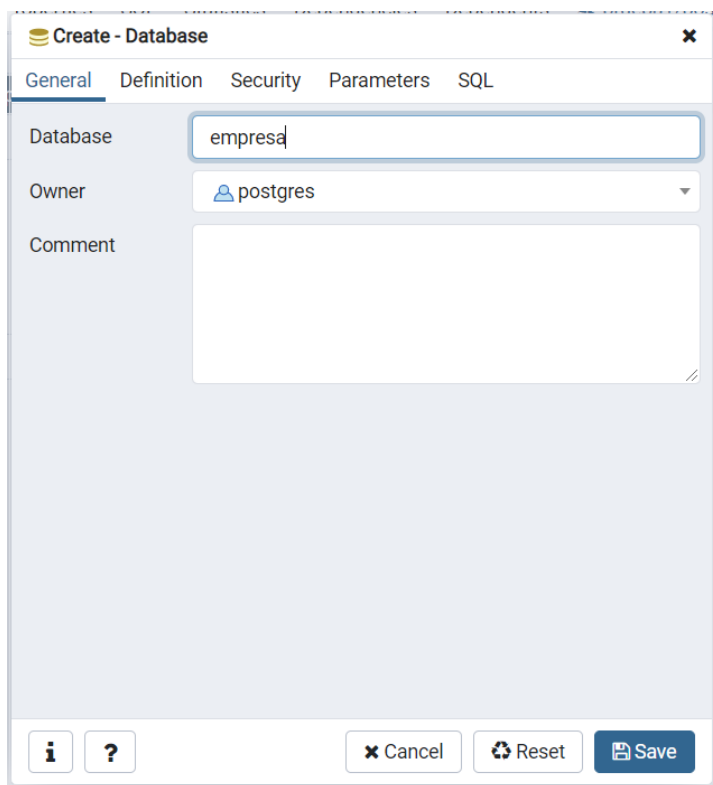
- **empleados(numero_empleado tipo numeric PRIMARY KEY, nombre tipo text, apellidos tipo text, salario tipo numeric)**
- **proyectos(numero_proyecto tipo numeric PRIMARY KEY, nombre tipo text, localización tipo text, coste tipo numeric)**
- **trabaja_proyectos(numero_empleado tipo numeric que sea FOREIGN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

Se pide:

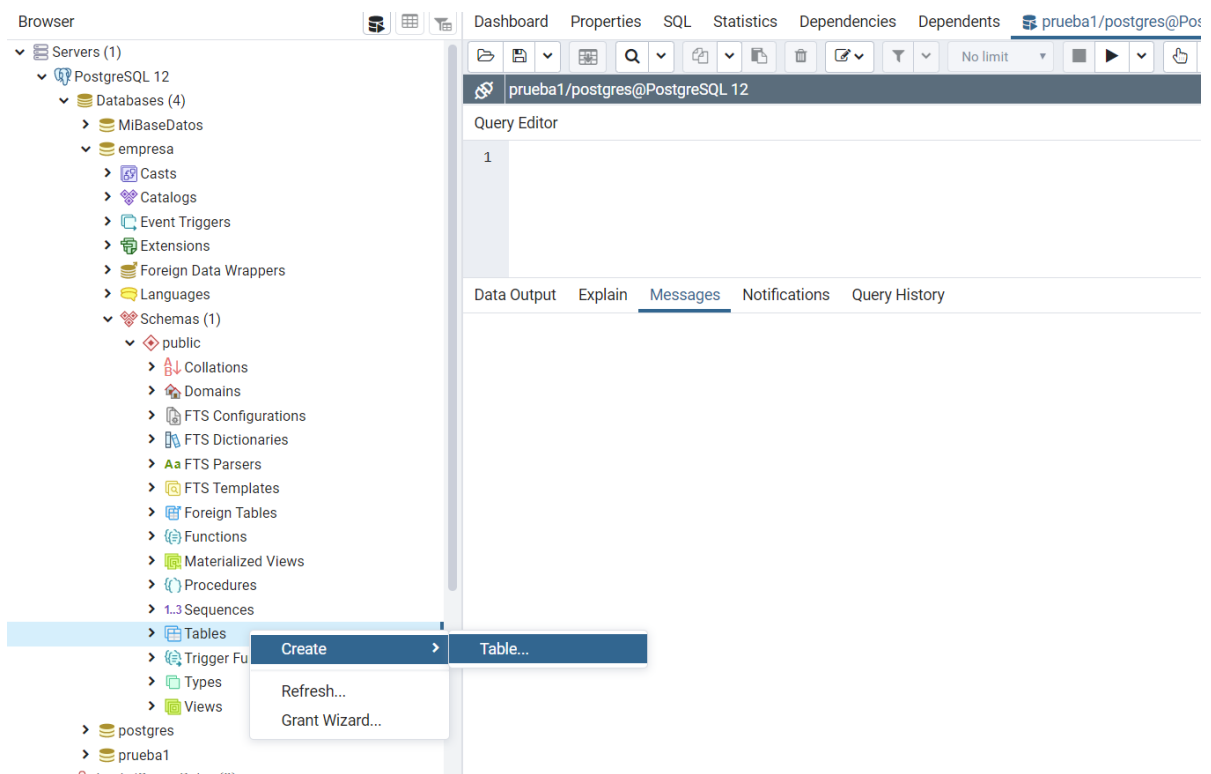
1 Indicar el proceso seguido para generar esta base de datos.

Creamos la base de datos

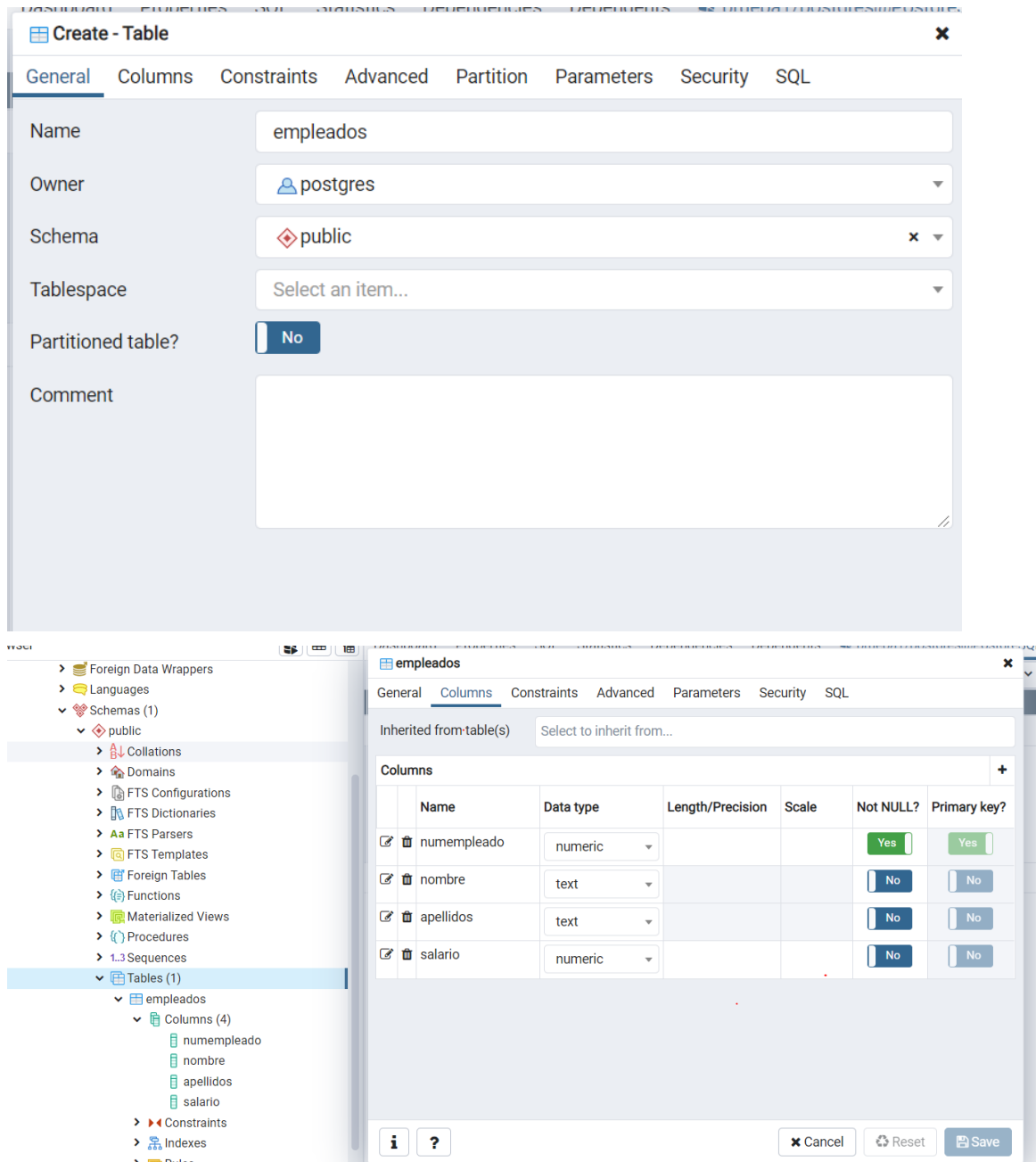




Una vez creada vamos la base de datos que está vacía, procedemos a crear las tablas que la forman:

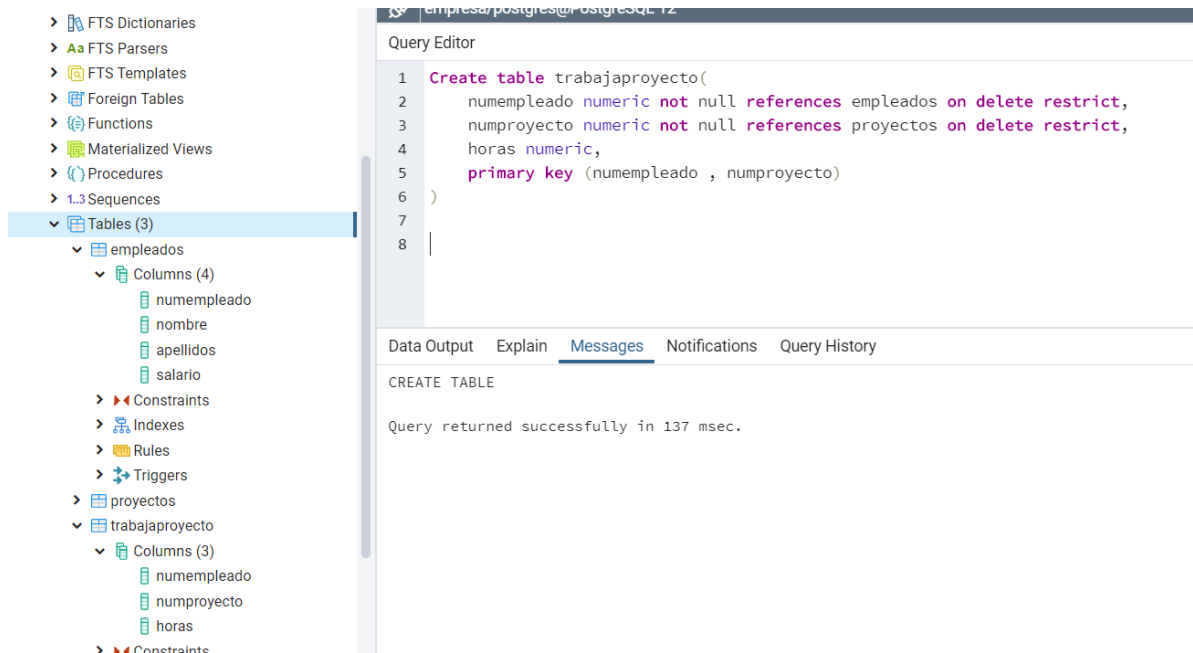


Empezamos con la tabla empleados y creamos las columnas con sus atributos de acuerdo al enunciado en la pestaña Columns, todo lo hemos hecho a través de pgAdmin.



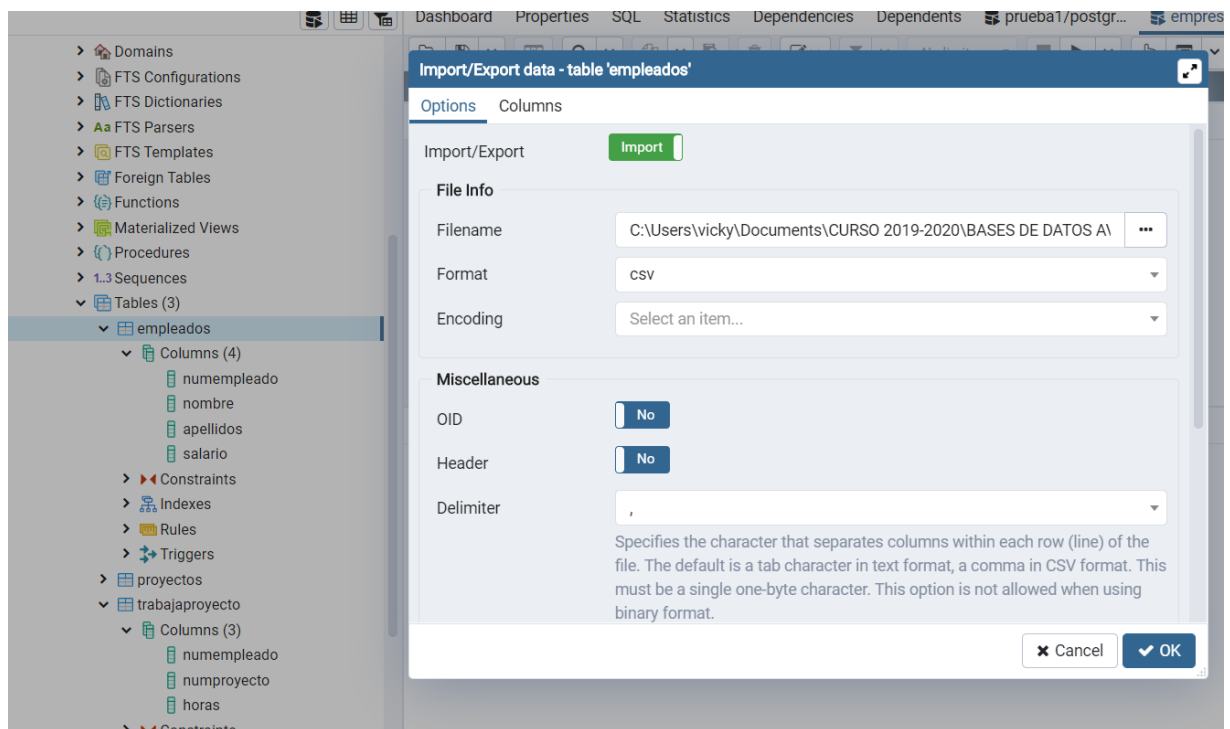
Hemos repetido el mismo procedimiento con la otra tabla proyecto, pero para la última hemos tenido que escribir el código de creación de la tabla con sus restricciones a través del pgAdmin:

La tabla trabajaproyectos es producto de una relación N:M entre empleados y proyectos. Esta tabla hace uso de RESTRICT, que según la [documentación](#) su función es la de evitar hacer un delete sobre dicha fila.



2 Cargar la información del fichero datos empleados.csv, datos proyectos.csv y datos trabaja proyectos.csv en dichas tablas de tal manera que sea lo más eficiente posible.

Primero con la tabla Empleados. Realizaremos una importación desde un csv dado que es la forma más rápida para cargar los datos.



Copying table data ✕

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

Sun Apr 05 2020 14:06:50 GMT+0200 (hora de verano de Europa central)

12.57 seconds More details... Stop Process

Successfully completed.

Ahora repetimos el mismo proceso con la tabla proyectos, el tiempo sería

Copying table data ✕

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

Sun Apr 05 2020 14:09:26 GMT+0200 (hora de verano de Europa central)

0.84 seconds More details... Stop Process

Successfully completed.

Este es el tiempo de carga de la tabla trabajaproyecto:

Process Watcher - Copying table data ✕

Copying table data 'public.trabajaproyecto' on database 'empresa' for the server 'PostgreSQL 12 (localhost:5432)'

Running command:

"C:\Program Files\PostgreSQL\12\bin\psql.exe" -command " \copy public.trabajaproyecto (numempleado, numproyecto, horas) FROM 'C:/Users/vicky/DOCUME~1/CURSO2~1/BASESD~1/LABORA~1/PECL2/DATOS_~3.CSV' DELIMITER ';' CSV QUOTE '\" ESCAPE '\""

Start time: Sun Apr 05 2020 14:10:54 GMT+0200 (hora de verano de Europa central) Stop Process

COPY 10000000
COPY 10000000

Successfully completed. Execution time: 636.92 seconds

3 Indicar los tiempos de carga.

Como se ha podido observar en las capturas:

- Tiempo de carga para tabla EMPLEADOS: 12.57 segundos (para 2.000.000 datos)
- Tiempo de carga para tabla PROYECTOS: 0.84 segundos (para 1.000.000 datos)
- Tiempo de carga para tabla TRABAJOPROYECTOS: 636.92 segundos (para 10.000.000 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?

Hacemos click sobre tablas y vamos a estadísticas:

Table name	Tuples inserted	Tuples updated	Tuples deleted	Tuples HOT updated	Live tuples	Dead tuples	Last vacuum	Last autovacuum	Last analyze	Last autoanalyze
empleados	4,000,000	0	2,000,000	0	2,000,000	0		2020-04-05 14:04:13.046266+02		2020-04-05 14:07:09.63133+02
proyectos	100,000	0	0	0	100,000	0				2020-04-05 14:10:04.033466+02
trabajaproyecto	10,000,000	0	0	0	9,999,864	0				2020-04-05 14:22:24.910274+02

Vacuum counter	Autovacuum counter	Analyze counter	Autoanalyze counter	Size
0	1	0	3	232 MB
0	0	0	1	9616 kB
0	0	0	1	886 MB

Las estadísticas almacenan datos como el número de tuplas insertadas, tuplas vivas, muertas, actualizadas, entre otros datos, para poder tener un control sobre todos nuestros registros y esto es importante porque, por ejemplo, podemos borrar (VACUUM) las tuplas muertas para liberar espacio y optimizar las consultas. También podemos observar sus pesos y detectar, por ejemplo, las tuplas más pesadas.

Las estadísticas son correctas en las dos primeras tablas, ya que coincide el número de tuplas insertadas con el número de tupla vivas, es decir son las tuplas que pueden ser leídas o modificadas. En cambio, no tenemos ninguna tupla muerta ya que no hemos realizado ninguna modificación que pudiera afectar a la disponibilidad de las

tuplas en un futuro. En la última tabla **trabajaproyecto** no coinciden las tuplas vivas con la insertadas, tenemos una cantidad inferior. Como son estadísticas es normal que algunos valores no coincidan, porque lo que realiza la base de datos son aproximaciones.

Se pueden actualizar los datos de las estadísticas hacemos click>refresh para refrescar las tablas y las estadísticas.

Nota: el número de tuplas insertadas en empleados son de 4.000.000 datos (posteriormente 2.000.000 borradas) dado que no se hizo captura de los segundos tardados en cargar y tuvimos de rehacer el proceso de cargar los datos.

Cuestión 5: Configurar PostgreSQL de tal manera que el coste mostrado por el comando EXPLAIN tenga en cuenta solamente las lecturas/escrituras de los bloques en el disco de valor 1.0 por cada bloque, independientemente del tipo de acceso a los bloques. Indicar el proceso seguido y la configuración final.

1. Igual que en la cuestión 2, procederemos a modificar el archivo postgresql.conf, borrando los '#' cuando sea requerido.

Las modificaciones las realizaremos sobre el apartado 'Query Tuning', sección que nos ayuda a optimizar el proceso de queryng de la base de datos. Dentro de este apartado modificaremos las constantes, cuyas explicaciones encontramos en esta [sección](#) del manual. Gracias a este otro [capítulo](#) descubrimos la fórmula que emplea pgAdmin para calcular el coste, que es de $(disk_pages_read * seq_page_cost) + (rows_scanned * cpu_tuple_cost)$. Nosotros solo vamos a querer tener en cuenta los bloques que se leen del disco y no el procesamiento de las tuplas en la CPU. Esto implica que, reduciremos a 0 el valor de las constantes `cpu_tuple_cost`, `cpu_index_tuple_cost` y `cpu_operator_cost`

Dado que no queremos que se tengan en cuenta. `Seq_page_cost` `random_page_cost` son dos variables relacionadas con la lectura de bloques por lo que serán las que modifiquemos por el valor 1.0 (aunque la primera variable ya venga por defecto con dicho valor y no haga falta modificarlo).

```
seq_page_cost = 1.0           # measured on an arbitrary scale
random_page_cost = 1.0       # same scale as above
cpu_tuple_cost = 0.00        # same scale as above
cpu_index_tuple_cost = 0.000 # same scale as above
cpu_operator_cost = 0.0000    # same scale as above
#parallel_tuple_cost = 0.1    # same scale as above
#parallel_setup_cost = 1000.0 # same scale as above
```

2. Actualizaremos en nuestro pgAdmin con la instrucción que nos proporciona postgresql.conf `SELECT pg_reload_conf()`

Dashboard Properties SQL Statistics Dependencies

empresa/postgres@PostgreSQL 12

Query Editor

```
1 SELECT pg_reload_conf()
```

Data Output Explain Messages Notifications Query I

pg_reload_conf	boolean
1	true

Cuestión 6: Aplicar el comando EXPLAIN a una consulta que obtenga la información de los empleados con salario de más de 96000 euros. ¿Son correctos los resultados del comando EXPLAIN? ¿Por qué? Comparar con lo que se obtendría con lo visto en teoría.

Query Editor Query History

```
1 explain select * from empleados where salario>96000
2 |
```

Data Output Explain Messages Notifications

QUERY PLAN	text
1	Seq Scan on empleados (cost=0.00..18673.00 rows=100128 width=41)
2	Filter: (salario > '96000'::numeric)

Πnumempleados, nombre, apellidos(σsalario>96.000)
Nr = 2.000.000 registros en la tabla Empleado
B = 8Kb = 8192 bytes por defecto

$L_r = 6 + 13 + 16 + 6 = 41$ bytes de longitud de registro

Dado que estamos haciendo una selección, ésta puede hacerse mediante una búsqueda lineal o binaria. En nuestro caso, el archivo no está ordenado según ninguno de nuestros atributos por lo que la búsqueda binaria queda descartada y sólo se podrá realizar la secuencial, de forma que tendrá que buscar en todos los registros. Además, buscaremos sobre un campo no clave por lo que el coste estimado será **Br**.

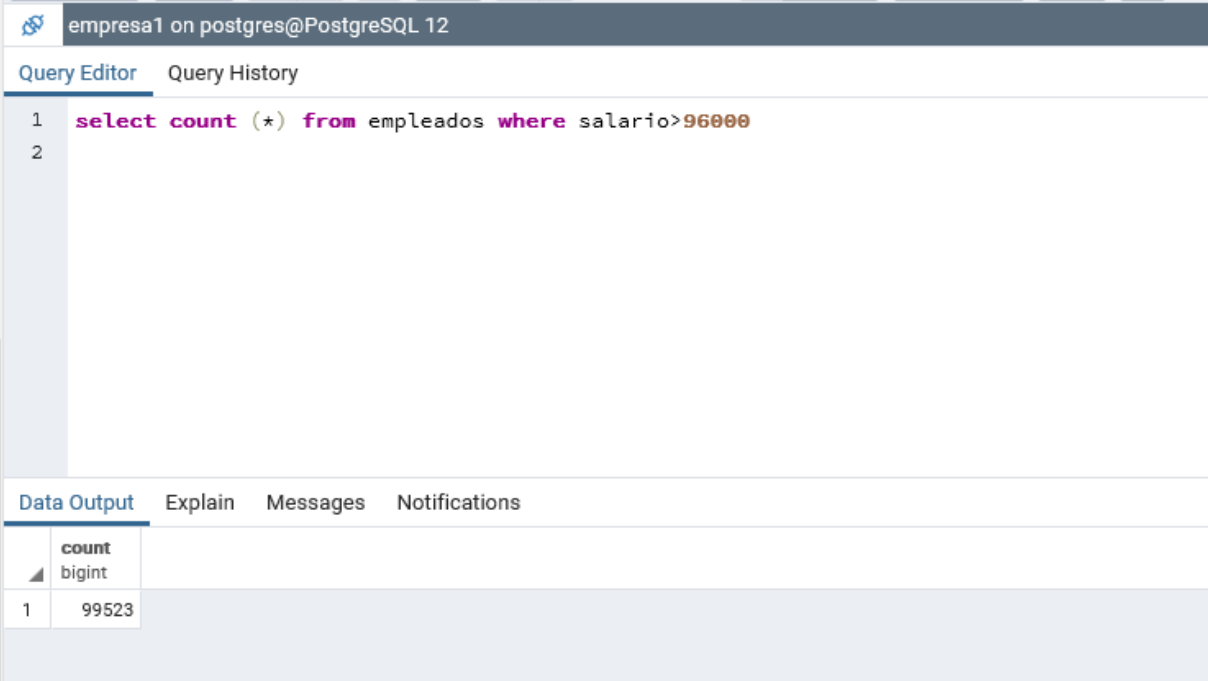
Curiosamente observamos que total de la longitud de nuestros registros coincide con el ancho nos devuelve cuando hacemos un explain sobre la tabla empleados como se aprecia en la imagen de arriba.

Estimación para el coste de la búsqueda secuencial:

Factor de bloque: $F_r = B/L_r = 8192/41 = 199$

$B_r = N_r/F_r = 2.000.000/199 = 10.051$ bloques/archivo (si las tuplas de r se almacenan juntas físicamente en un fichero)

La instrucción explain hace uso de la búsqueda secuencial, pero sólo hace estimaciones, por lo tanto no es exacto y el resultado se aleja de lo teórico. Por otro lado, podemos observar que el número de filas que nos devuelve explain es de 100.128 pero si hacemos un select count observamos que el resultado es distinto, de 99.523 filas.

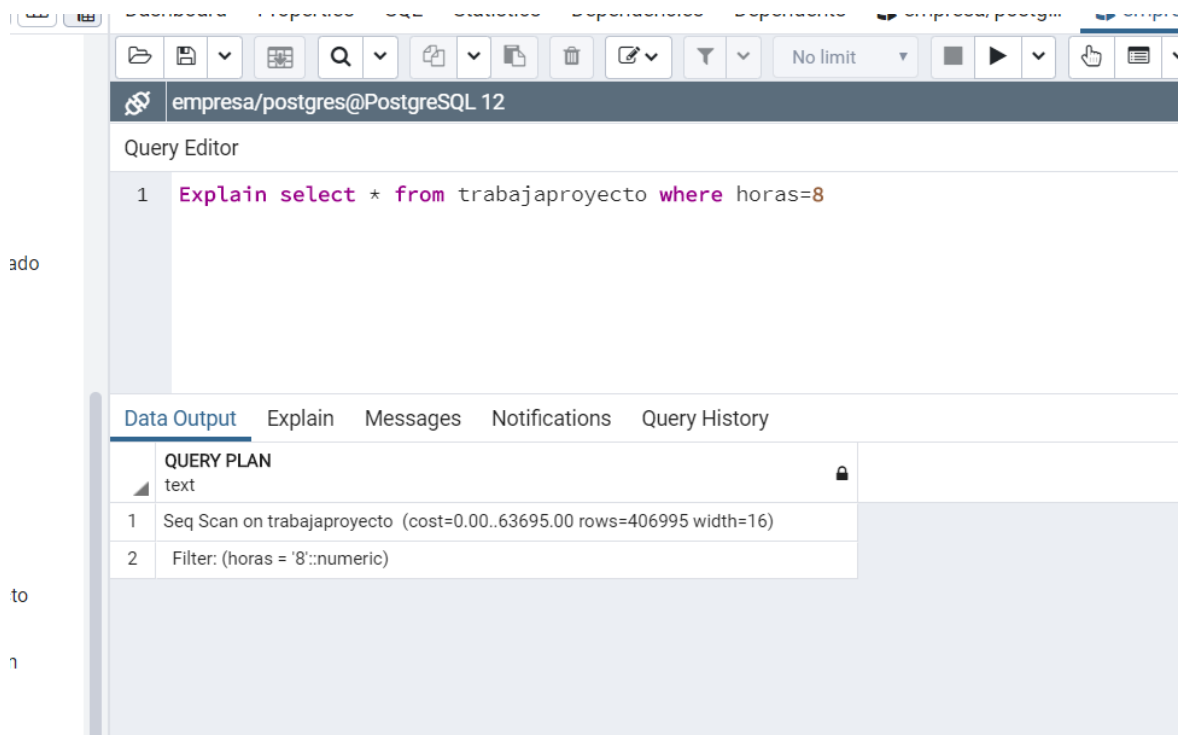


The screenshot shows a PostgreSQL query editor interface. The title bar indicates the connection is 'empresa1 on postgres@PostgreSQL 12'. The 'Query Editor' tab is active, displaying a SQL query: `select count (*) from empleados where salario > 96000`. Below the query editor, the 'Data Output' tab is active, showing the results of the query. The results are displayed in a table with two columns: 'count' and 'bigint'. The first row shows a count of 99523.

	count	bigint
1	99523	

Cuestión 7: Aplicar el comando EXPLAIN a una consulta que obtenga la información de los proyectos en los cuales el empleado trabaja 8 horas. ¿Son correctos los resultados del comando EXPLAIN? ¿Por qué? Comparar con lo que se obtendría con lo visto en teoría.

Hemos interpretado que al pedir la información de los proyectos, se podría obtener de la tabla trabajaproyecto con el identificador del empleado, el número del proyecto y las horas. Por lo tanto hemos hecho el explain y la consulta sobre esa tabla.



$\Pi_{\text{horas, num_proyecto, num_empleado}}(\sigma_{\text{horas}=8})$

Nr = 10.000.000 registros en la tabla trabajaproyecto

B = 8Kb = 8192 bytes por defecto

Lr = 6 + 6 + 4 = 16 bytes de longitud de registro

Estimación para el coste de la búsqueda secuencial:

Factor de bloque: $Fr = B/Lr = 8192/16 = 512$

$Br = Nr/Fr = 10.000.000/512 = 19.532$ bloques/archivo (si las tuplas de r se almacenan juntas físicamente en un fichero)

De manera similar al ejercicio anterior, debemos hacer una búsqueda secuencial, dado que el campo no está ordenado el campo y es campo no clave.

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

```
Select empleados.numempleado, empleados.salario,
proyectos.numproyecto, proyectos.nombre, proyectos.coste,
trabajaproyecto.horas from empleados inner join trabajaproyecto on
```

```
empleados.numempleado=trabajaproyecto.numempleado inner join
proyectos on proyectos.numproyectos=trabajaproyecto.numproyecto
where (proyectos.coste>15000 AND salario=24000 AND salario=24000
and horas <=2)
```

empresa/postgres@PostgreSQL 12

Query Editor

```
1 Select empleados.numempleado , empleados.salario,proyectos.numproyecto, proyectos.nombre,proyectos.coste, trabajaproyecto.horas
2 from empleados inner join trabajaproyecto on empleados.numempleado=trabajaproyecto.numempleado
3 inner join proyectos on proyectos.numproyecto=trabajaproyecto.numproyecto
4 where (proyectos.coste>15000 AND salario=24000 AND horas<=2)
```

Data Output Explain Messages Notifications Query History

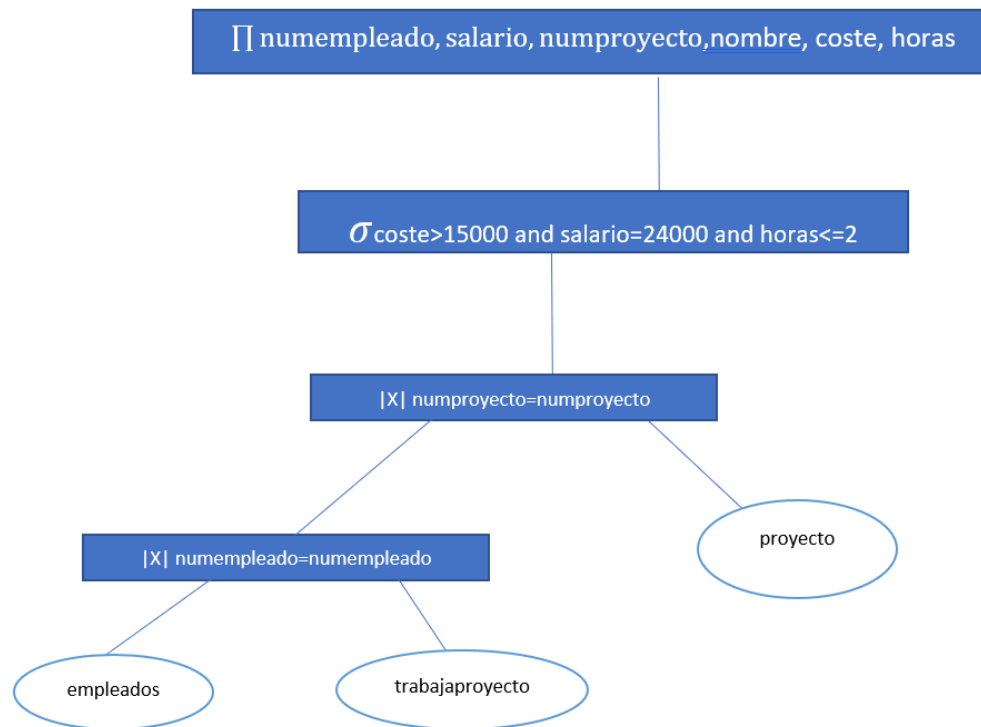
	numempleado numeric	salario numeric	numproyecto numeric	nombre text	coste numeric	horas numeric
1	14232	24000.00	49756	nombre497...	16081.00	2
2	14232	24000.00	50786	nombre507...	19450.00	1
3	58907	24000.00	76717	nombre767...	17812.00	2
4	659304	24000.00	63562	nombre635...	18032.00	1
5	1438601	24000.00	35474	nombre354...	16361.00	2
6	1438601	24000.00	37246	nombre372...	17795.00	2

```
1 Explain Select empleados.numempleado , empleados.salario,proyectos.numproyecto, proyectos.nombre,proyectos.coste, trabajaproyecto.horas
2 from empleados inner join trabajaproyecto on empleados.numempleado=trabajaproyecto.numempleado
3 inner join proyectos on proyectos.numproyecto=trabajaproyecto.numproyecto
4 where (proyectos.coste>15000 AND salario=24000 AND horas<=2)
```

Data Output Explain Messages Notifications Query History

	QUERY PLAN text
1	Nested Loop (cost=0.00..18820.01 rows=6 width=39)
2	-> Nested Loop (cost=0.00..18820.00 rows=13 width=22)
3	-> Seq Scan on empleados (cost=0.00..18673.00 rows=21 width=12)
4	Filter: (salario = '24000':numeric)
5	-> Index Scan using trabajaproyecto_pkey on trabajaproyecto (cost=0.00..7.00 rows=1 width=16)
6	Index Cond: (numempleado = empleados.numempleado)
7	Filter: (horas <= '2':numeric)
8	-> Index Scan using proyectos_pkey on proyectos (cost=0.00..0.00 rows=1 width=23)
9	Index Cond: (numproyecto = trabajaproyecto.numproyecto)
10	Filter: (coste > '15000':numeric)

Comparado con lo que realizaríamos en teoría:



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

Tabla	Tiempo sin integridad	Tiempo con integridad
Tienda (200.000)	1.65 segundos	1.76 segundos
Producto (1.000.000)	20.48 segundos	24.11 segundos
Tienda_Producto (20.000.000)	50.99 segundos	44.79 segundos
Trabajador (1.000.000)	9.47 segundos	6.63 segundos
Ticket (5.000.000)	46.99 segundos	23.54 segundos
Ticket_Producto (10.000.000)	26.42 segundos	17.58 segundos

Pasamos a crear la base de datos TIENDAS de manera similar a los anteriores ejercicios. Primero creamos las tablas con los atributos y posteriormente insertaremos los datos desde los .txt generados.

Create - Table

General **Columns** Constraints Advanced Partition Parameters Security SQL

Inherited from table(s)

Columns						
	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?
	codigobarras	character varying			<input type="checkbox"/> No	<input checked="" type="checkbox"/> Yes
	nombre	text			<input type="checkbox"/> No	<input type="checkbox"/> No
	tipo	character varying			<input type="checkbox"/> No	<input type="checkbox"/> No
	descripcion	text			<input type="checkbox"/> No	<input type="checkbox"/> No
	precio	integer			<input type="checkbox"/> No	<input type="checkbox"/> No

i ?

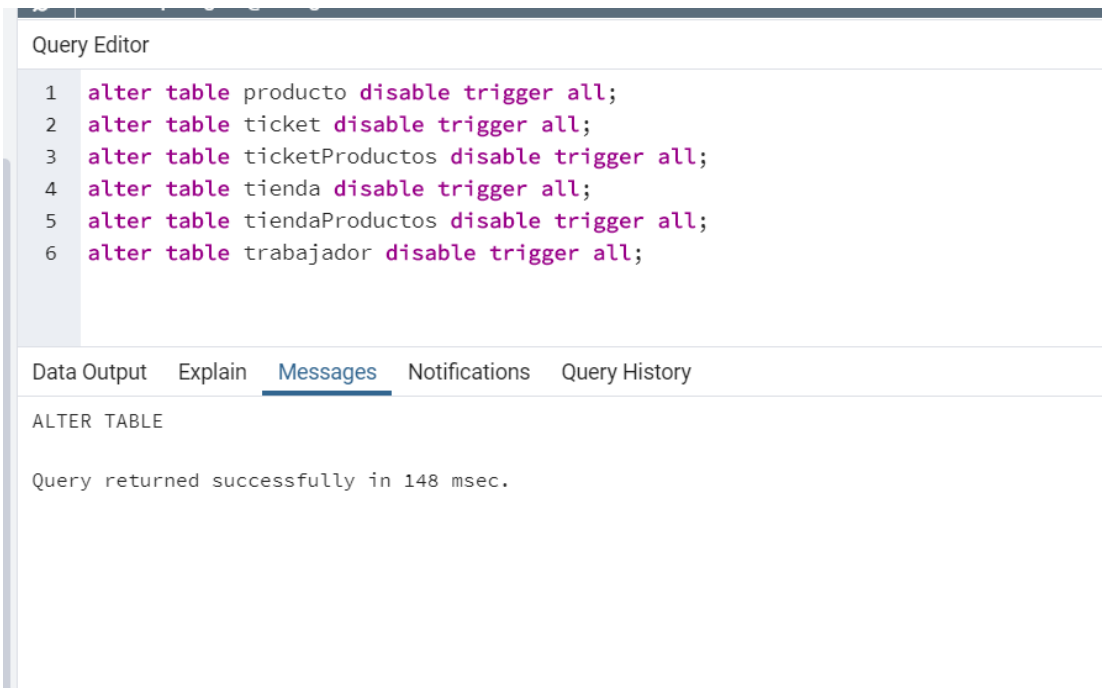
La integridad referencial nos permite, a la hora de cargar los datos, ir comprobando que hay una relación entre las tablas. Por ello, cuando tenemos la integridad activada, debemos ir insertando los datos en un orden adecuado para que las relaciones tengan sentido. El orden es:

1.Tienda > 2.Productos > 3.Tienda_Productos > 4.Trabajador > 5.Ticket > 6.Ticket_Producto

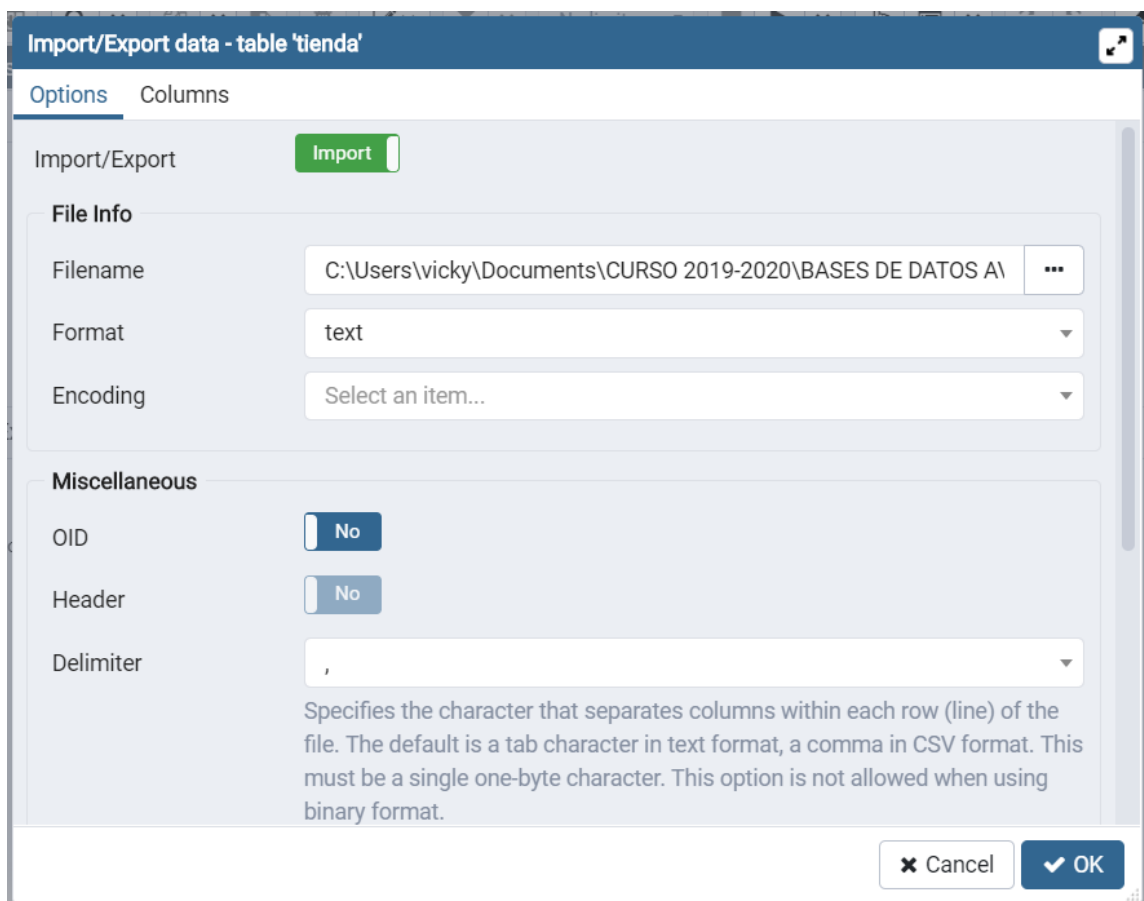
Este orden lo seguimos para mantener la relación PK-FK entre tablas. De no hacerlo así, la base de datos nos generaría un mensaje de error y no cargaría los datos. Se pueden intercambiar algunos ordenes, como por ejemplo cargar antes Productos que Tienda, ya que son independientes y no comparten atributos entre sí. Las tablas intermedias deben ir obligatoriamente detrás de las tablas con las PK correspondientes ya que hacen uso de sus PK.

A la hora de cargar los datos sin integridad, podemos seguir el orden que queramos ya que la base de datos no comprueba relaciones y permite tener más libertad.

Por defecto, Postgres tiene la integridad referencial activada. Para desactivarla, usaremos la instrucción: **ALTER TABLE tabla DISABLE TRIGGER ALL**




Cargaremos los archivos de los txt como lo hemos hecho en ejercicios y prácticas anteriores:






Sin integridad Referencial:

Tabla tienda

Copying table data 

Copying table data 'public.tienda' on database 'tiendas' and server (localhost:5432)

Wed Apr 15 2020 13:56:34 GMT+0200 (hora de verano de Europa central)

 1.65 seconds  More details...  Stop Process






 Successfully completed.

Tabla tiendaproducto

Copying table data 

Copying table data 'public.tiendaproductos' on database 'tiendas' and server (localhost:5432)

Wed Apr 15 2020 13:59:37 GMT+0200 (hora de verano de Europa central)

 50.99 seconds  More details...  Stop Process






 Successfully completed.

Tabla producto

Copying table data 

Copying table data 'public.producto' on database 'tiendas' and server (localhost:5432)

Wed Apr 15 2020 14:02:30 GMT+0200 (hora de verano de Europa central)

 20.48 seconds  More details...  Stop Process






 Successfully completed.

Tabla ticket

Copying table data 

Copying table data 'public.ticket' on database 'tiendas' and server (localhost:5432)

Wed Apr 15 2020 20:25:00 GMT+0200 (hora de verano de Europa central)

 46.99 seconds  More details...  Stop Process



 Successfully completed.

Tabla ticketProducto

Copying table data ✕

Copying table data 'public.ticketproductos' on database 'tiendas' and server (localhost:5432)

Wed Apr 15 2020 14:20:08 GMT+0200 (hora de verano de Europa central)

 26.42 seconds More details... Stop Process


 Successfully completed.

Tabla Trabajador


*Como dato, tras errores descubrimos de la base de datos que hace uso de UTF-8 y por ello sólo acepta caracteres globales, por lo que tuvimos que eliminar los acentos y las 'ñ'.

Copying table data ✕

Copying table data 'public.trabajador' on database 'tiendas' and server (localhost:5432)

Wed Apr 15 2020 20:53:15 GMT+0200 (hora de verano de Europa central)

 9.47 seconds More details... Stop Process

 Successfully completed.

A continuación, borramos y volvemos a cargar los datos desde o sin modificar nada más dado que como hemos mencionado anteriormente por defecto Postgres usa la integridad referencial.

Con integridad Referencial

Tienda

Copying table data

✕

Copying table data 'public.tienda' on database 'tiendas' and server (localhost:5432)
Wed Apr 15 2020 21:17:12 GMT+0200 (hora de verano de Europa central)
🕒 1.76 seconds

📄 More details...

✕ Stop Process

✓

Successfully completed.

Productos

Copying table data

✕

Copying table data 'public.productos' on database 'tiendas' and server (localhost:5432)
Wed Apr 15 2020 21:18:24 GMT+0200 (hora de verano de Europa central)
🕒 24.11 seconds

📄 More details...

✕ Stop Process

✓

Successfully completed.

Tienda Productos

Copying table data

✕

Copying table data 'public.tiendaproductos' on database 'tiendas' and server (localhost:5432)
Wed Apr 15 2020 21:19:46 GMT+0200 (hora de verano de Europa central)
🕒 44.79 seconds

📄 More details...

✕ Stop Process

✓

Successfully completed.

Trabajador

Copying table data

✕

Copying table data 'public.trabajador' on database 'tiendas' and server (localhost:5432)
Wed Apr 15 2020 21:21:43 GMT+0200 (hora de verano de Europa central)

⌚

6.63 seconds

ℹ

More details...

✕

Stop Process

✓

Successfully completed.

Ticket

Copying table data

✕

Copying table data 'public.ticket' on database 'tiendas' and server (localhost:5432)
Wed Apr 15 2020 21:29:53 GMT+0200 (hora de verano de Europa central)

⌚

23.54 seconds

ℹ

More details...

✕

Stop Process

✓

Successfully completed.

Ticket Productos

Copying table data

✕

Copying table data 'public.ticketproductos' on database 'tiendas' and server (localhost:5432)
Wed Apr 15 2020 21:23:22 GMT+0200 (hora de verano de Europa central)

⌚

17.58 seconds

ℹ

More details...

✕

Stop Process

✓

Successfully completed.

Ejemplo de muestra de prueba de que los datos están bien cargados:

27

Query Editor

```

1 select codigobarras from producto where nombre='Platanos'
2 --select idtienda from tienda where ciudad='Madrid' and barrio='Malagueta'

```

Data Output Explain Messages Notifications Query History

	codigobarras	
1	9P	
2	11P	
3	15P	
4	50P	
5	63P	
6	104P	
7	165P	
8	185P	

La diferencia de tiempos es muy pequeña entre la carga de datos con integridad y sin integridad, siendo incluso menor la de la carga con integridad. Por razonamiento lógico la carga de datos con integridad debería ser mayor, dado que el hacer comprobaciones entre las relaciones de las tablas implicarían un tiempo extra. En nuestro caso los tiempos de carga son menores. Los tiempos de carga también vienen condicionados por la cantidad de registros y atributos que poseen, por eso la tabla que más tiempo tarda en cargar es Tienda_producto por tener 20.000.000 de registros.

Cuestión 10: Realizar una consulta SQL que muestre “el nombre y DNI de los trabajadores que hayan vendido algún trabajador que ganan entre 3000 y 5000 euros de salario en la Comunidad de Madrid en las cuales hay por lo menos un producto con un stock de menos de 100 unidades y que tiene un precio de más de 400 euros.”

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

1. Consulta SQL

<

Podemos ver que obtenemos resultados.

```
select trabajador.nombre, trabajador.dni from trabajador
inner join ticket on
trabajador.codigotrabajador=ticket.codigotrabajador
inner join ticketproductos on
ticket.numticket=ticketproductos.numticket
inner join producto on
ticketproductos.codigobarras=producto.codigobarras
inner join tiendaproductos on
producto.codigobarras=tiendaproductos.codigobarras
inner join tienda on tiendaproductos.idtienda=tienda.idtienda
where (salario>=3000 and salario<=5000) AND ((fecha>='01/09/2019'
and fecha<='31/12/2019') and importe>500) AND cantidad>4
AND precio>=400 AND stock<=100 AND ciudad='Madrid'
```

2. Obtenemos el plan de ejecución con comando explain en forma de árbol de álgebra relacional

Hacemos uso de la consulta anterior pero adicionalmente añadimos: explain (format json) select trabajador...

Hacemos uso del formato json, que nos va a proporcionar la misma información que un formato texto.

Y obtenemos lo siguiente:

tiendas on postgres@PostgreSQL 10

```

1 explain (format json) select trabajador.nombre, trabajador.dni from trabajador inner join ticket
2 on trabajador.codigotrabajador=ticket.codigotrabajador inner join ticketproductos on ticket.numticket=ticketproductos.numticket
3 inner join producto on ticketproductos.codigobarras=producto.codigobarras
4 inner join tiendaproductos on producto.codigobarras=tiendaproductos.codigobarras
5 inner join tienda on tiendaproductos.idtienda=tienda.idtienda
6 where (salario>=3000 and salario<=5000) AND ((fecha>='01/09/2019' and fecha<='31/12/2019') and importe>500) AND cantidad>4
7 AND precio>=400 AND stock<=100 AND ciudad='Madrid'

```

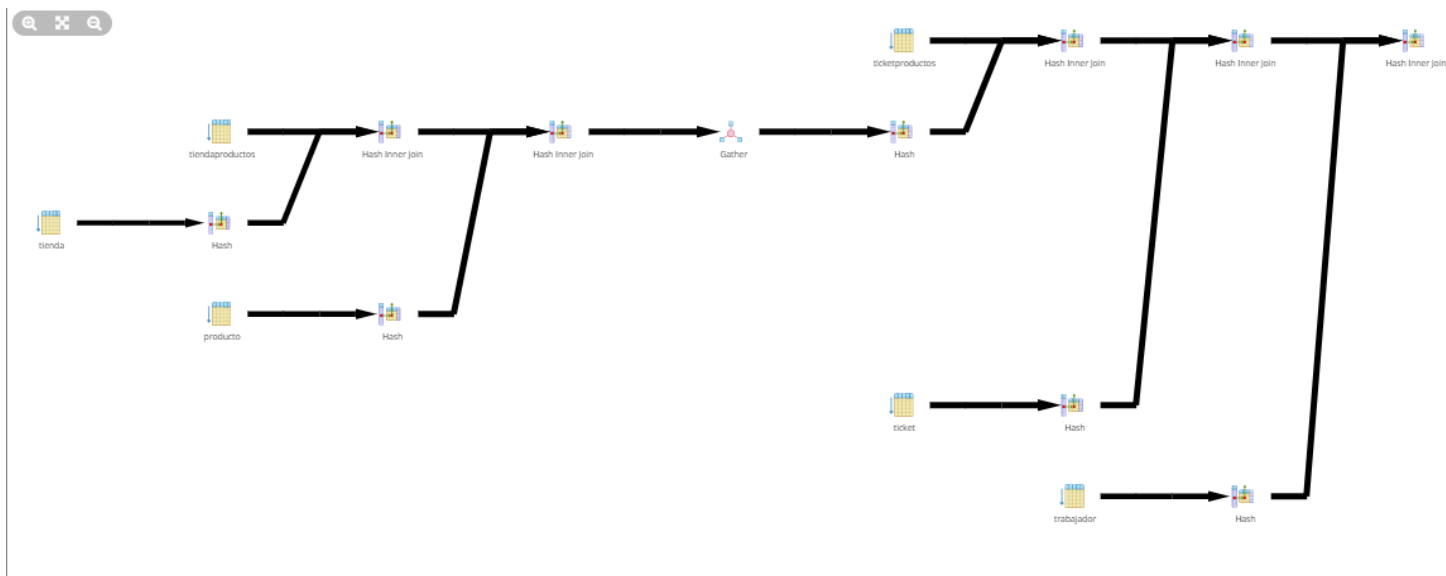
Data Output Explain Messages Notifications Query History

```

explain (format json) select trabajador.nombre, trabajador.dni
from trabajador
inner join ticket on
trabajador.codigotrabajador=ticket.codigotrabajador
inner join ticketproductos on
ticket.numticket=ticketproductos.numticket
inner join producto on
ticketproductos.codigobarras=producto.codigobarras
inner join tiendaproductos on
producto.codigobarras=tiendaproductos.codigobarras
inner join tienda on tiendaproductos.idtienda=tienda.idtienda
where (salario>=3000 and salario<=5000) AND ((fecha>='01/09/2019'
and fecha<='31/12/2019') and importe>500) AND cantidad>4
AND precio>=400 AND stock<=100 AND ciudad='Madrid'

```

Aquí realizamos una ampliación del resultado:



El resultado que obtenemos de sólo ejecutar `explain select`:

```
"Hash Join (cost=547732.99..913858.88 rows=1262260 width=17) (actual
time=30191.615..42530.095 rows=1189847 loops=1)"
" Hash Cond: (ticket.codigotrabajador = trabajador.codigotrabajador)"
" -> Hash Join (cost=513789.90..850651.23 rows=2522869 width=4) (actual
time=29478.683..40463.614 rows=2379611 loops=1)"
" Hash Cond: (ticketproductos.numticket = ticket.numticket)"
" -> Hash Join (cost=330822.53..618580.66 rows=3253026 width=4) (actual
time=22427.598..29966.529 rows=3070386 loops=1)"
" Hash Cond: ((ticketproductos.codigobarras)::text =
(producto.codigobarras)::text)"
" -> Seq Scan on ticketproductos (cost=0.00..179057.19 rows=5589431
width=11) (actual time=0.033..3764.516 rows=5555508 loops=1)"
" Filter: (cantidad > 4)"
" Rows Removed by Filter: 4444492"
" -> Hash (cost=321339.73..321339.73 rows=545504 width=14) (actual
time=22421.295..22421.296 rows=552330 loops=1)"
" Buckets: 131072 Batches: 16 Memory Usage: 2641kB"
" -> Gather (cost=37924.11..321339.73 rows=545504 width=14) (actual
time=2239.202..22068.481 rows=552330 loops=1)"
" Workers Planned: 2"
" Workers Launched: 2"
```

```

"          -> Hash Join (cost=36924.11..265789.33 rows=227293 width=14)
(actual time=2195.464..21759.588 rows=184110 loops=3)"
"          Hash Cond: ((tiendaproductos.codigobarras)::text =
(producto.codigobarras)::text)"
"          -> Hash Join (cost=4396.00..227031.37 rows=361467 width=7)
(actual time=67.292..18769.968 rows=291905 loops=3)"
"          Hash Cond: (tiendaproductos.idtienda = tienda.idtienda)"
"          -> Parallel Seq Scan on tiendaproductos
(cost=0.00..212276.52 rows=3946143 width=11) (actual time=1.934..17511.246
rows=3193033 loops=3)"
"          Filter: (stock <= 100)"
"          Rows Removed by Filter: 3473634"
"          -> Hash (cost=4167.00..4167.00 rows=18320 width=4)
(actual time=61.322..61.322 rows=18276 loops=3)"
"          Buckets: 32768 Batches: 1 Memory Usage: 899kB"
"          -> Seq Scan on tienda (cost=0.00..4167.00 rows=18320
width=4) (actual time=0.258..54.726 rows=18276 loops=3)"
"          Filter: (ciudad = 'Madrid'::text)"
"          Rows Removed by Filter: 181724"
"          -> Hash (cost=22211.00..22211.00 rows=628809 width=7)
(actual time=2119.508..2119.508 rows=630976 loops=3)"
"          Buckets: 131072 Batches: 16 Memory Usage: 2571kB"
"          -> Seq Scan on producto (cost=0.00..22211.00
rows=628809 width=7) (actual time=0.306..1815.249 rows=630976 loops=3)"
"          Filter: (precio >= 400)"
"          Rows Removed by Filter: 369024"
"          -> Hash (cost=119347.86..119347.86 rows=3877721 width=8) (actual
time=7022.020..7022.020 rows=3874009 loops=1)"
"          Buckets: 131072 Batches: 64 Memory Usage: 3387kB"
"          -> Seq Scan on ticket (cost=0.00..119347.86 rows=3877721 width=8)
(actual time=0.033..5852.909 rows=3874009 loops=1)"
"          Filter: ((fecha >= '01/09/2019'::text) AND (fecha <= '31/12/2019'::text)
AND (importe > 500))"
"          Rows Removed by Filter: 1125991"
"          -> Hash (cost=24757.00..24757.00 rows=500327 width=21) (actual
time=712.465..712.465 rows=499681 loops=1)"
"          Buckets: 65536 Batches: 8 Memory Usage: 3839kB"
"          -> Seq Scan on trabajador (cost=0.00..24757.00 rows=500327 width=21)
(actual time=2.109..509.990 rows=499681 loops=1)"

```

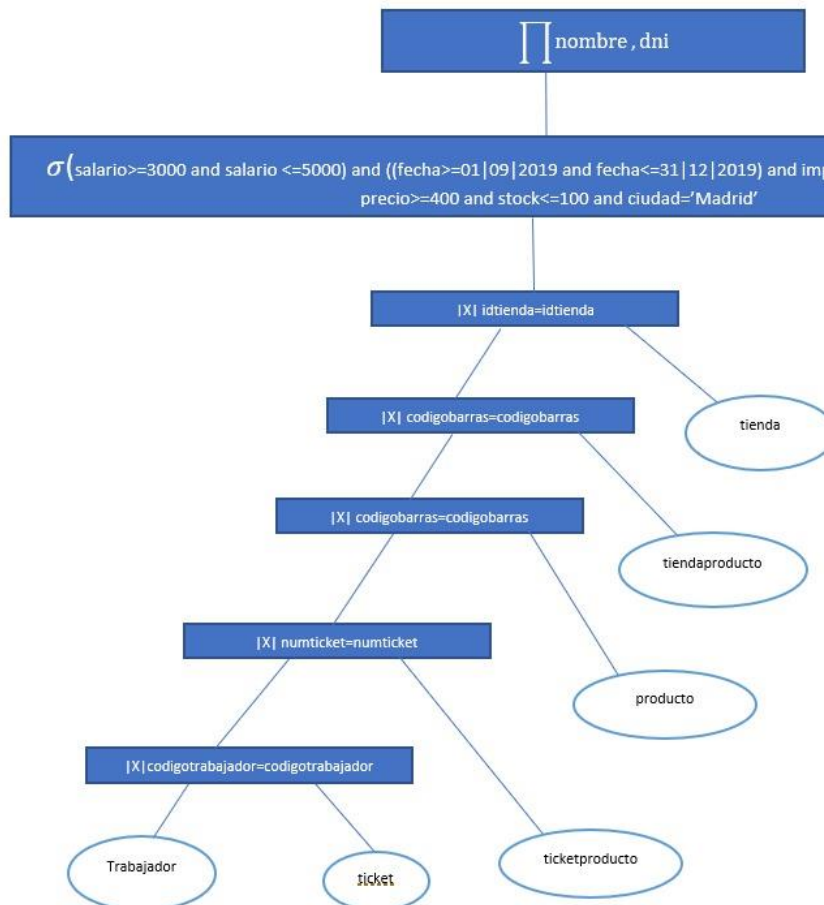

" Filter: ((salario >= 3000) AND (salario <= 5000))"

" Rows Removed by Filter: 500319"

"Planning time: 5.607 ms"

"Execution time: 42569.777 ms"

3. Árbol obtenido por nosotros



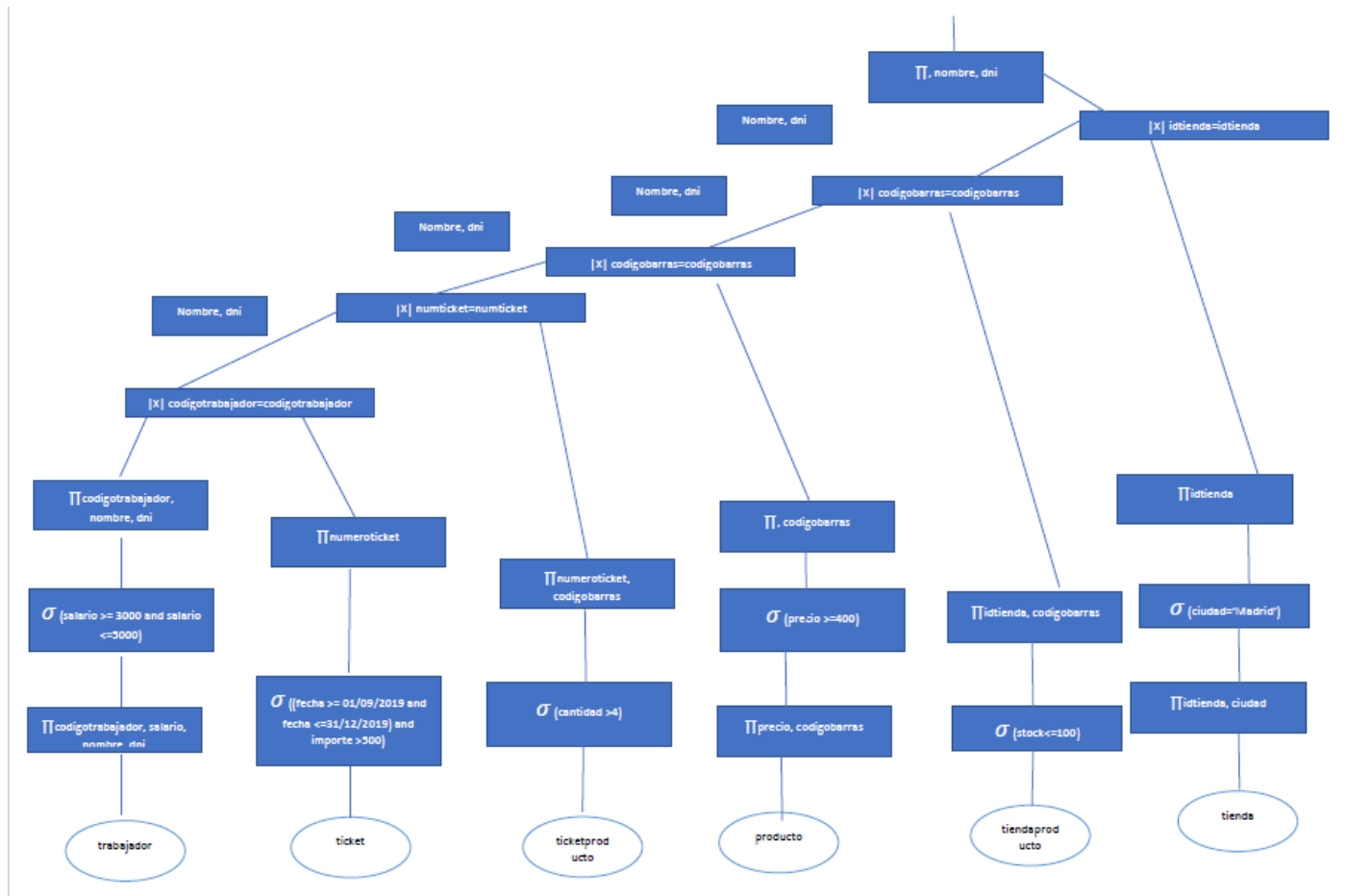
4. Comparación entre resultados:

Podemos ver que hay una diferencia entre nuestra manera de desarrollar el árbol frente a la manera en la cual lo hace Postgres. Podemos apreciar las diferencias con los join, no realiza la consulta en el mismo orden ni relacionando las tablas de la misma manera.

Cuestión 11: Usando PostgreSQL, y a raíz de los resultados de la cuestión anterior, ¿qué modificaciones realizaría para mejorar el rendimiento de la misma y por qué? Obtener la información pedida de la cuestión 10 y explicar los resultados. Obtener el plan de ejecución con el resultado del comando EXPLAIN en forma de árbol de algebra relacional. Comentar los resultados obtenidos y comparar con la cuestión anterior.

Para mejorar el rendimiento de la consulta realizaremos dos tareas:

1. Optimizar el árbol de la consulta



2. Hacer uso de los índices, ya que ayudan a acelerar las secuencias SQL

Crearemos un índice (tipo B-tree, ya que es el que emplea por defecto Postgres) sobre todos los campos sobre los que se hará una búsqueda, de esta forma podremos optimizar al máximo la consulta. Los campos serán: salario, fecha, importe, cantidad, precio, stock y ciudad. Estos campos los hemos obtenido de la optimización del árbol anteriormente planteada.

La consulta que emplearemos para crear los índices es la siguiente, que la encapsularemos en una transacción para asegurar que se crean todos los índices o ninguno.

```
BEGIN;
create index in_salario on trabajador(salario);
create index in_fecha on ticket(fecha);
create index in_importe on ticket(importe);
```

```

create index in_cantidad on ticketproductos(cantidad);
create index in_precio on producto(precio);
create index in_stock on tiendaproductos(stock);
create index in_ciudad on tienda (ciudad);
COMMIT;

```

tiendas on postgres@PostgreSQL 10

```

1 BEGIN;
2 create index in_salario on trabajador(salario);
3 create index in_fecha on ticket(fecha);
4 create index in_importe on ticket(importe);
5 create index in_cantidad on ticketproductos(cantidad);
6 create index in_precio on producto(precio);
7 create index in_stock on tiendaproductos(stock);
8 create index in_ciudad on tienda (ciudad);
9 COMMIT;

```

Data Output Explain Messages Notifications Query History

COMMIT

Query returned successfully in 2 min 55 secs.

Vemos que se ha ejecutado correctamente la transacción.

Usando el explain (format json) también aseguramos que se han producido modificaciones sobre nuestro árbol:

Data Output Explain Messages Notifications Query History



Si volvemos a ejecutar explain select sobre la misma consulta anterior:

tiendas on postgres@PostgreSQL 10

```

1 explain select trabajador.nombre, trabajador.dni from trabajador
2 inner join ticket on trabajador.codigotrabajador=ticket.codigotrabajador
3 inner join ticketproductos on ticket.numticket=ticketproductos.numticket
4 inner join producto on ticketproductos.codigobarras=producto.codigobarras
5 inner join tiendaproductos on producto.codigobarras=tiendaproductos.codigobarras
6 inner join tienda on tiendaproductos.idtienda=tienda.idtienda
7 where (salario>=3000 and salario<=5000) AND ((fecha>='01/09/2019' and fecha<='31/12/2019') and importe>500) AND cant
8 AND precio>=400 AND stock<=100 AND ciudad='Madrid'

```

Data Output Explain Messages Notifications Query History

QUERY PLAN
text
1 Hash Join (cost=545833.40..911952.59 rows=1262467 width=17)
2 Hash Cond: (ticket.codigotrabajador = trabajador.codigotrabajador)
3 -> Hash Join (cost=511888.12..848741.92 rows=2522804 width=4)
4 Hash Cond: (ticketproductos.numticket = ticket.numticket)
5 -> Hash Join (cost=328920.54..616673.34 rows=3252942 width=4)
6 Hash Cond: ((ticketproductos.codigobarras)::text = (producto.codigobarras)::text)
7 -> Seq Scan on ticketproductos (cost=0.00..179055.00 rows=5589333 width=11)
8 Filter: (cantidad > 4)
9 -> Hash (cost=319437.79..319437.79 rows=545500 width=14)

tiendas on postgres@PostgreSQL 10

```

1 explain select trabajador.nombre, trabajador.dni from trabajador
2 inner join ticket on trabajador.codigotrabajador=ticket.codigotrabajador
3 inner join ticketproductos on ticket.numticket=ticketproductos.numticket
4 inner join producto on ticketproductos.codigobarras=producto.codigobarras
5 inner join tiendaproductos on producto.codigobarras=tiendaproductos.codigobarras
6 inner join tienda on tiendaproductos.idtienda=tienda.idtienda
7 where (salario>=3000 and salario<=5000) AND ((fecha>='01/09/2019' and fecha<='31/12/2019') and importe>500) AND cantidad
8 AND precio>=400 AND stock<=100 AND ciudad='Madrid'

```

Data Output Explain Messages Notifications Query History

Successfully run. Total query runtime: 93 msec.
31 rows affected.

Vemos que el tiempo de la consulta se ha reducido como era de esperar, pasando de **42569.777 ms** a **93 ms**. Y también si observamos la misma consulta arriba el coste que nos aparece es **913858.88** y al hacer los índices vemos que se ha reducido ligeramente a **911952.59**.

Cuestión 12: Usando PostgreSQL, borre el 50% de las tiendas almacenadas de manera aleatoria y todos sus datos relacionados ¿Cuál ha sido el proceso seguido? ¿Y el tiempo empleado en el borrado? Ejecute la consulta de nuevo. Obtener el plan de ejecución con el resultado del comando EXPLAIN en forma de árbol de algebra relacional. Comparar con los resultados anteriores.

1.Borrar el 50% de las tiendas + Explicación del proceso seguido

Primero, para evitar problemas, desactivaremos la integridad referencial de todas las tablas, con alter table nombretabla disable trigger all;

```
1 alter table producto disable trigger all;
2 alter table ticket disable trigger all;
3 alter table ticketproductos disable trigger all;
4 alter table tienda disable trigger all;
5 alter table tiendaproductos disable trigger all;
6 alter table trabajador disable trigger all;
7
8
9
```

Data Output Explain Messages Notifications Query History

ALTER TABLE

Query returned successfully in 203 msec.

En segundo lugar, iremos borrando el 50% de las tablas. Empezaremos por la propia tabla Tienda, usando delete from. Borraremos los elementos de 0-100.000 para tener un control sobre dichos datos y borrarlos de las siguientes tablas (y asegurar que borramos los mismos datos)

```
1
2 delete from tienda where idtienda>=0 AND idtienda<100000
```

Data Output Explain Messages Notifications Query History

DELETE 100000

Query returned successfully in 458 msec.

A continuación, mostramos con un select count los registros que nos quedan:

The screenshot shows a database management tool interface. On the left is a sidebar with a tree view containing items like 'Configurations', 'Dictionaries', 'Parsers', 'Templates', 'Ignored Tables', 'Sessions', 'Serialized Views', 'Sessions', 'Sessions (6)', 'Producto', 'Ticket', 'Ticketproductos', 'Tienda', 'Tiendaproductos', 'Trabajador', and 'User Functions'. The main area displays a SQL query with two lines:

```
1 --delete from tienda where idtienda>=
2 select count(*) from tienda
```

Below the query, there are tabs: 'Data Output', 'Explain', 'Messages', 'Notifications', and 'Query History'. The 'Data Output' tab is active, showing a table with the following data:

	count
1	100000

Ahora hay que borrar las filas de las tablas que hagan referencia a los elementos que hemos borrado:

Empezamos por Tienda_Poductos borraremos los idtienda que estén entre 0 y 100.000 para seguir manteniendo el orden.

The screenshot shows the same database management tool interface. The SQL query area now contains:

```
1 delete from tiendaproductos where idtienda>=0 AND idtienda<100000
2
```

Below the query, the tabs are 'Data Output', 'Explain', 'Messages', 'Notifications', and 'Query History'. The 'Messages' tab is active, showing the following message:

```
DELETE 10004370
Query returned successfully in 1 min 36 secs.
```

Pasaremos a la tabla ticketproductos y borraremos los tickets pertenecientes a los trabajadores que tienen asociado ese idtienda en el rango de 0 a 100000. Esa comunicación lo haremos mediante la tabla ticket que nos permite enlazar numticket con la tabla donde queremos eliminar las filas y codigotrabajador para comunicar ticket con la tabla trabajador. Primero mostraremos una imagen de los registros que hay con esas características, después los registros borrados que están asociados a las tiendas que tengan un identificador dentro de ese rango.

1	select count(*) from ticketproductos inner join ticket
2	on ticket.numticket=ticketproductos.numticket
3	inner join trabajador on ticket.codigotrabajador=trabajador.codigotrabajador
4	where idtienda>=0 and idtienda<100000
5	
6	
7	
8	
9	
10	

Data Output	Explain	Messages	Notifications	Query History
-------------	---------	----------	---------------	---------------

	count
	bigint
1	1007038

```
delete from ticketproductos where numticket in (select numticket
from ticket where codigotrabajador in( select codigotrabajador from
trabajador where
ticket.codigotrabajador=trabajador.codigotrabajador AND
trabajador.idtienda>0 AND trabajador.idtienda<100000))
```

1	delete from ticketproductos
2	where numticket in (select numticket from ticket where codigotrabajador in(select codigotrabajador from trabajador
3	where ticket.codigotrabajador=trabajador.codigotrabajador AND trabajador.idtienda>0 AND trabajador.idtienda<100000))
4	
5	
6	
7	
8	
9	
10	
11	

Data Output	Explain	Messages	Notifications	Query History
-------------	---------	----------	---------------	---------------

DELETE 1007038
Query returned successfully in 5 min 16 secs.

Después repetiremos el mismo proceso con la tabla ticket y eliminaremos los registros que sean de alguno de los trabajadores pertenecientes a las tiendas que hemos borrado.

```
select count(*) from ticket
inner join trabajador on
ticket.codigotrabajador=trabajador.codigotrabajador
where(trabajador.idtienda>=0 AND trabajador.idtienda<100000)
```

1	<code>select count(*) from ticket</code>
2	<code>inner join trabajador on ticket.codigotrabajador=trabajador.codigotrabajador</code>
3	<code>where(trabajador.idtienda>=0 AND trabajador.idtienda<100000)</code>
4	
5	
6	
7	
8	

Data Output	Explain	Messages	Notifications	Query History
	count			
	bigint			
1	503194			

```
delete from ticket
where exists(select codigotrabajador from trabajador where
ticket.codigotrabajador=trabajador.codigotrabajador
and trabajador.idtienda>=0 AND trabajador.idtienda<100000)
```

1	<code>delete from ticket</code>
2	<code>where exists(select codigotrabajador from trabajador where ticket.codigotrabajador=trabajador.codigotrabajador</code>
3	<code>and trabajador.idtienda>=0 AND trabajador.idtienda<100000)</code>
4	
5	
6	
7	
8	

Data Output	Explain	Messages	Notifications	Query History
DELETE 503194				
Query returned successfully in 15 secs 309 msec.				

Por último, pasaremos a la tabla Trabajador y borraremos los trabajadores (de los 1.000.000 que teníamos anteriormente) que tengan asociados un idtienda de 0-100.000.

Hacemos un select count y vemos que obtenemos 100.474 trabajadores asociados a tiendas cuyos idtienda 0-100.000, que serán los que debemos borrar.

3	
4	<code>select count(*) from trabajador where idtienda>=0 and idtienda<=100000</code>

Data Output	Explain	Messages	Notifications	Query History
	count			
	bigint			
1	100474			

```
delete from trabajador
```



```
where (trabajador.idtienda>=0 AND trabajador.idtienda<100000)
```

```
1 delete from trabajador
2 where (trabajador.idtienda>=0 AND trabajador.idtienda<100000)
3
4
5
6
7
8
```

Data Output Explain Messages Notifications Query History

DELETE 100474

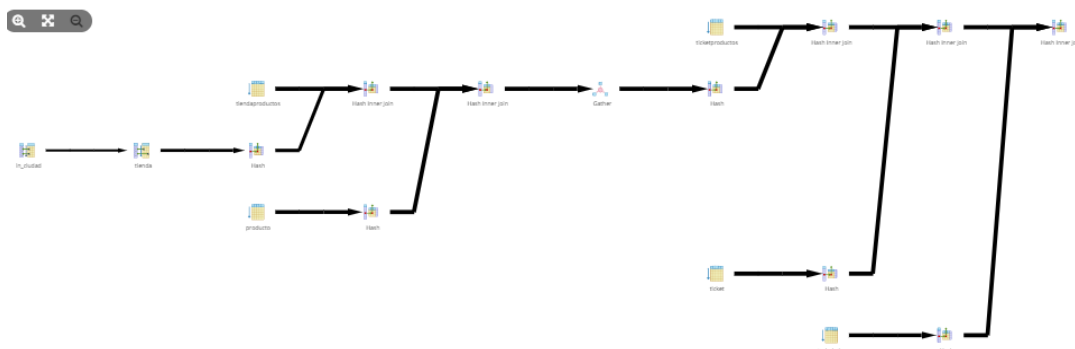
Query returned successfully in 3 secs 766 msec.

2. Tiempo empleado en el borrado

NOMBRE TABLA	TIEMPO BORRADO
tienda	554 msec
tienda_producto	1 min 16 sec
ticketproductos	5 mins 16 sec
ticket	15 secs 309 msec.
trabajador	3 secs 766 msec.

3. Volver a realizar la consulta con el comando EXPLAIN en forma de árbol (format json)

El tiempo a ejecutar esta consulta 536 msec.



4. Comparar resultados

Ejecutaremos un explain select sobre la consulta inicial y los costes en principio se debe de reducir ya que hemos eliminado el 50% de nuestros elementos en la base de

datos tienda como observamos en la siguiente imagen. Aunque hemos eliminado tuplas seguan manteniendo el espacio más adelante haremos la correcta eliminación de ese espacio.

1	<code>EXPLAIN select trabajador.nombre, trabajador.dni from trabajador</code>
2	<code>inner join ticket on trabajador.codigotrabajador=ticket.codigotrabajador</code>
3	<code>inner join ticketproductos on ticket.numticket=ticketproductos.numticket</code>
4	<code>inner join producto on ticketproductos.codigobarras=producto.codigobarras</code>
5	<code>inner join tiendaproductos on producto.codigobarras=tiendaproductos.codigobarras</code>
6	<code>inner join tienda on tiendaproductos.idtienda=tienda.idtienda</code>
7	<code>where (salario>=3000 and salario<=5000) AND ((fecha>='01/09/2019' and fecha<='31/12/2019') and importe>500) AND cantidad>4</code>
8	<code>AND precio>=400 AND stock<=100 AND ciudad='Madrid'</code>
9	
10	

Data Output	Explain	Messages	Notifications	Query History
QUERY PLAN text				
1	Hash Join (cost=528193.38..860929.23 rows=1098912 width=17)			
2	Hash Cond: (ticket.codigotrabajador = trabajador.codigotrabajador)			
3	-> Hash Join (cost=496742.29..803760.69 rows=2213112 width=4)			
4	Hash Cond: (ticketproductos.numticket = ticket.numticket)			
5	-> Hash Join (cost=328920.54..592551.67 rows=2850003 width=4)			
6	Hash Cond: ((ticketproductos.codigobarras)::text = (producto.codigobarras)::text)			
7	-> Seq Scan on ticketproductos (cost=0.00..166518.65 rows=4978651 width=11)			
8	Filter: (cantidad > 4)			

Cuestión 13: ¿Qué técnicas de mantenimiento de la BD propondría para mejorar los resultados de dicho plan sin modificar el código de la consulta? ¿Por qué?

Información obtenida de [aquí](#).

Algunas instrucciones que podemos emplear para mejorar los resultados y optimizar las consultas son:

VACUUM: se puede emplear adicionalmente con ANALYZE. Sirve para la eliminación de tuplas muertas, que son tuplas que no podemos emplear y que nos consumen espacio, por lo que al eliminarlas tendríamos menores tiempos de búsqueda de datos y consultas ya que liberamos el espacio que ocupaban. Podemos usar también Full VACUUM que reclama mucho más espacio que un VACUUM.

ANALYZE: actualiza y recolecta estadísticas de la base de datos sobre el planificador para crear una ejecución de la consulta lo más eficiente posible. Elige el plan más apropiado y por ello aumenta el procesamiento de la consulta.

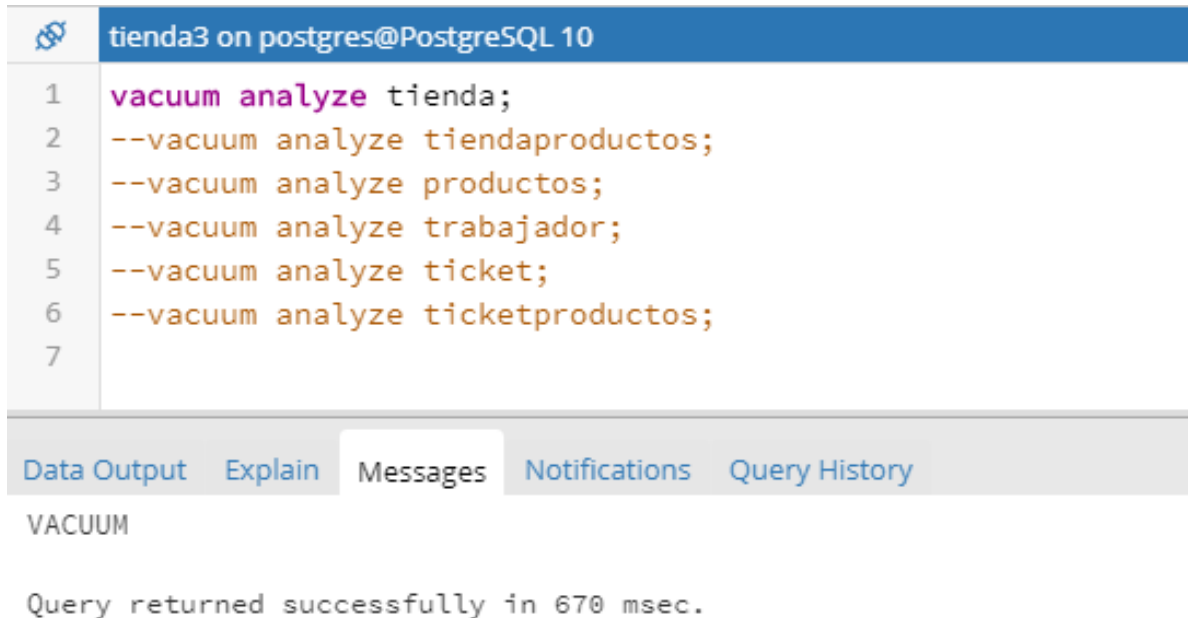
REINDEX: reconstruye uno o más índices y reemplaza las versiones anteriores de los índices. Algunos usos pueden ser para reconstruir índices corruptos, que no contienen datos válidos, entre otros. Ayuda a la optimización, por ejemplo, al borrar la mitad de los registros de la tabla tienda dado que es conveniente actualizar los índices, porque ya no deberá apuntar el índice a dichos registros.

Cuestión 14: Usando PostgreSQL, lleve a cabo las operaciones propuestas en la cuestión anterior y ejecute el plan de ejecución de la misma consulta. Obtener el plan de ejecución con el resultado del comando EXPLAIN en forma de árbol de algebra relacional. Compare los resultados del plan de ejecución con los de los apartados anteriores. Coméntelos.

1. Consulta con reindex, analyze y vacuum

Query de combinación de vacuum + analyze, que nos actualizará las estadísticas del planificador y determinará la forma más eficiente de ejecutar la consulta.

```
vacuum analyze tienda;
```



The screenshot shows a PostgreSQL query execution window titled "tienda3 on postgres@PostgreSQL 10". The query is as follows:

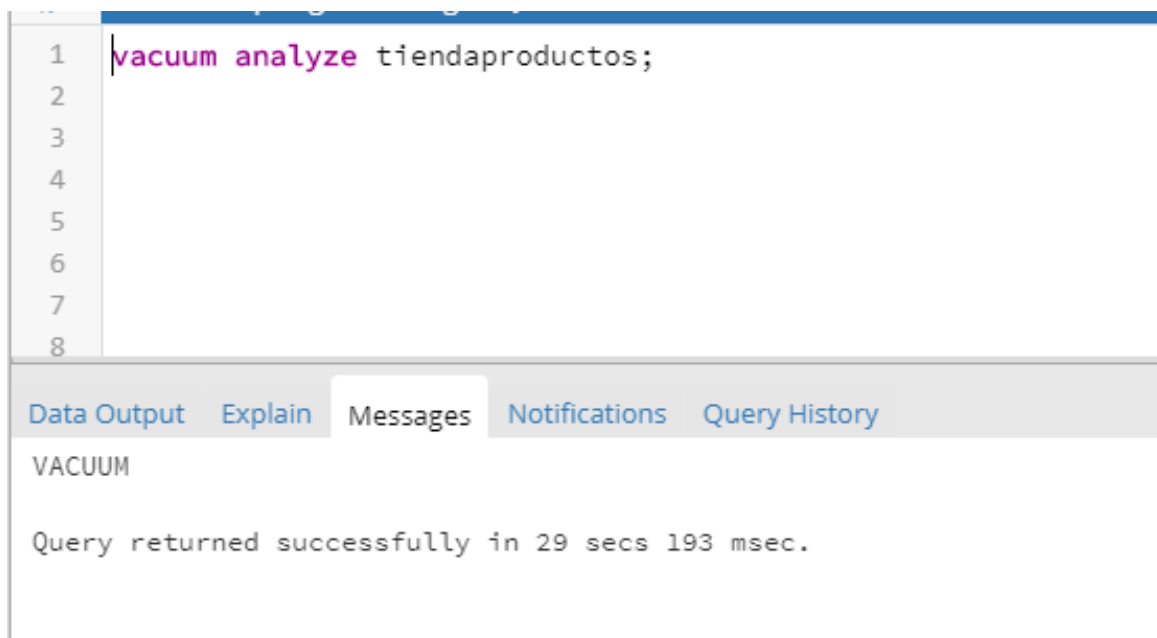
```
1 vacuum analyze tienda;
2 --vacuum analyze tiendaproductos;
3 --vacuum analyze productos;
4 --vacuum analyze trabajador;
5 --vacuum analyze ticket;
6 --vacuum analyze ticketproductos;
7
```

Below the query, there are tabs for "Data Output", "Explain", "Messages", "Notifications", and "Query History". The "Messages" tab is selected, showing the output:

VACUUM

Query returned successfully in 670 msec.

```
vacuum analyze tiendaproductos;
```



The screenshot shows a PostgreSQL query execution window. The query is as follows:

```
1 vacuum analyze tiendaproductos;
2
3
4
5
6
7
8
```

Below the query, there are tabs for "Data Output", "Explain", "Messages", "Notifications", and "Query History". The "Messages" tab is selected, showing the output:

VACUUM

Query returned successfully in 29 secs 193 msec.

```
vacuum analyze producto;
```

```
1 vacuum analyze producto;  
2  
3  
4  
5  
6  
7  
8
```

[Data Output](#) [Explain](#) [Messages](#) [Notifications](#) [Query History](#)

VACUUM

Query returned successfully in 4 secs 241 msec.

```
vacuum analyze trabajador;
```

```
1 vacuum analyze trabajador;  
2  
3  
4  
5  
6  
7  
8
```

[Data Output](#) [Explain](#) [Messages](#) [Notifications](#) [Query History](#)

VACUUM

Query returned successfully in 4 secs 306 msec.

```
vacuum analyze ticket;
```

```
1 vacuum analyze ticket;
2
3
4
5
6
7
8
```

Data Output Explain Messages Notifications Query History

VACUUM

Query returned successfully in 56 secs 721 msec.

```
vacuum analyze ticketproductos;
```

```
1 vacuum analyze ticketproductos;
2
```

Data Output Explain Messages Notifications Query History

VACUUM

Query returned successfully in 1 min 16 secs.

Query para refrescar los índices usando reindex:

```
BEGIN;
reindex index in_salario;
reindex index in_fecha;
reindex index in_importe;
reindex index in_cantidad;
reindex index in_precio;
```

```
reindex index in_stock;
reindex index in_ciudad;
COMMIT;
```

```
1 BEGIN;
2 reindex index in_salario;
3 reindex index in_fecha;
4 reindex index in_importe;
5 reindex index in_cantidad;
6 reindex index in_precio;
7 reindex index in_stock;
8 reindex index in_ciudad;
9 COMMIT;
10
```

Data Output Explain Messages Notifications Query History

COMMIT

Query returned successfully in 2 min 45 secs.

2. Plan de ejecución de la consulta anterior (árbol algebra relacional)

Volvemos a realizar la consulta de ejercicios anteriores:

```
explain (format json) select trabajador.nombre, trabajador.dni
from trabajador
inner join ticket on
trabajador.codigotrabajador=ticket.codigotrabajador
inner join ticketproductos on
ticket.numticket=ticketproductos.numticket
inner join producto on
ticketproductos.codigobarras=producto.codigobarras
inner join tiendaproductos on
producto.codigobarras=tiendaproductos.codigobarras
inner join tienda on tiendaproductos.idtienda=tienda.idtienda
where (salario>=3000 and salario<=5000) AND ((fecha>='01/09/2019'
and fecha<='31/12/2019') and importe>500) AND cantidad>4
AND precio>=400 AND stock<=100 AND ciudad='Madrid'
```

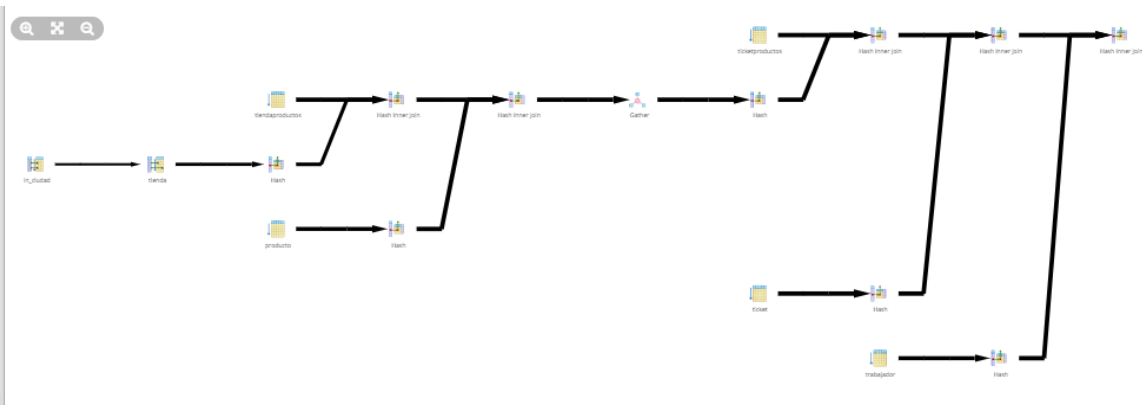
```

1 explain (format json) select trabajador.nombre, trabajador.dni from trabajador
2 inner join ticket on trabajador.codigotrabajador=ticket.codigotrabajador
3 inner join ticketproductos on ticket.numticket=ticketproductos.numticket
4 inner join producto on ticketproductos.codigobarras=producto.codigobarras
5 inner join tiendaproductos on producto.codigobarras=tiendaproductos.codigobarras
6 inner join tienda on tiendaproductos.idtienda=tienda.idtienda
7 where (salario>=3000 and salario<=5000) AND ((fecha>='01/09/2019' and fecha<='31/12/2019') and importe>500) AND cantidad>4
8 AND precio>=400 AND stock<=100 AND ciudad='Madrid'

```

Data Output Explain Messages Notifications Query History

Successfully run. Total query runtime: 136 msec.
1 rows affected.



3. Comparación y comentarios

INSTRUCCIÓN	TIEMPOS
reindex	2 min 45 secs
explain (format json)	136 mse

Vamos a hacer una comparativa entre las distintas instrucciones que hemos utilizado. Arriba tenemos un pequeño cuadro donde nos aparecen reindex y explain (format json). En reindex observamos que ha tardado bastante en comparación a los demás y eso se debe a que ha tenido que reconstruir los índices, actualizarlos ya que hemos borrado la mitad de nuestra base de datos y por lo tanto para que apunten a datos correctos. En explain obtendremos unas estadísticas y el tiempo que se ha reducido en comparación al ejercicio anterior ya que tenemos menor cantidad de datos. Por último, los vacuum analyze hemos tenido que ir uno por uno ejecutándolos lo más significativo en este apartado respecto a los tiempos, en el vacuum analyze ticket producto es el que más se ha demorado debido a la gran cantidad de tuplas muertas.

Cuestión 15: Usando PostgreSQL, analice el LOG de operaciones de la base de datos y muestre información de cuáles han sido las consultas más utilizadas en su práctica, el número de consultas, el tiempo medio de ejecución, y cualquier otro dato que considere importante.

Información obtenida de [aquí](#).

Analizamos el LOG de operaciones (C:\Program Files\PostgreSQL\12\data\log) que en ejercicios anteriores hemos usado. De manera similar a los ejercicios previos (modificando en el .conf: log_statement = 'all'), comprobamos que el peso de los logs es muy grande, oscilando tamaños como 6.749 kb. Observamos que las consultas más empleadas son UNION ALL y SELECT (las que realiza la base de datos, ya que nuestras consultas hacen uso de INNER JOIN, EXPLAIN, etc).

El número de consultas es que se reflejan en el log es superior al de las realizadas, ya que la base de datos registra mucha más información a nivel interno. Primero, se gestiona la base de datos según el OID y no con el nombre que nosotros hayamos establecido. Gestiona información como los 'Hits', que son los bloques que tenemos ya disponibles en la memoria caché, por lo que no ha sido necesario ir a buscarlos a disco y lo que implica que nos genera coste de lectura. También informa de los 'Reads', (bloques leídos en la base de datos), los 'Updates', 'Deletes' e 'Inserts' (número de filas insertadas, actualizadas e insertadas mediante consultas), 'Transactions', 'Commits'(transacciones aceptadas y ejecutadas) y 'Rollbacks' (transacciones deshechas)

```
SELECT 'session_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
  (SELECT count(*) FROM pg_stat_activity WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Total",
  (SELECT count(*) FROM pg_stat_activity WHERE state = 'active' AND datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Active",
  (SELECT count(*) FROM pg_stat_activity WHERE state = 'idle' AND datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Idle"
) t
UNION ALL
SELECT 'tps_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
  (SELECT sum(xact_commit) + sum(xact_rollback) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Transactions",
  (SELECT sum(xact_commit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Commits",
  (SELECT sum(xact_rollback) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Rollbacks"
) t
UNION ALL
SELECT 'ti_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
  (SELECT sum(tup_inserted) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Inserts",
  (SELECT sum(tup_updated) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Updates",
  (SELECT sum(tup_deleted) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Deletes"
) t
UNION ALL
SELECT 'to_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
  (SELECT sum(tup_fetched) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Fetched",
  (SELECT sum(tup_returned) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Returned"
) t
UNION ALL
SELECT 'bio_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
  (SELECT sum(blks_read) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Reads",
  (SELECT sum(blks_hit) FROM pg_stat_database WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Hits"
) t
```

También nos informa del tiempo que ha tardado en realizar las instrucciones y consultas. El error de escritura del texto 'duración' se debe a la codificación UTF-8 que no acepta tildes (´ó´)

) t

2020-04-05 12:46:48.972 CEST [10180] LOG: duraciÃ³n: 29.383 ms

2020-04-05 12:46:50.250 CEST [10180] LOG: sentencia: /*pga4dash*/

```
SELECT 'session_stats' AS chart_name, row_to_json(t) AS chart_data
FROM (SELECT
```

```
  (SELECT count(*) FROM pg_stat_activity WHERE datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Total",
  (SELECT count(*) FROM pg_stat_activity WHERE state = 'active' AND datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Active",
  (SELECT count(*) FROM pg_stat_activity WHERE state = 'idle' AND datname = (SELECT datname FROM pg_database WHERE oid = 24586)) AS "Idle"
```

Cuestión 16: A partir de lo visto y recopilado en toda la práctica. Describir y comentar cómo es el proceso de procesamiento y optimización que realiza PostgreSQL en las consultas del usuario.

1. La base de datos trata de almacenar todo lo posible en memoria (hits) antes que en disco (bloques leídos), para que el coste sea mucho menor y por tanto las consultas sean mucho más ágiles.
2. Tiene activada la integridad referencial, de forma que al cargar los datos la base de datos se asegura que hay relación entre lo insertado, por lo que al realizar las consultas ya no tiene que ir verificando las relaciones entre tablas.
3. Nos proporciona (que no realiza independientemente) diversas herramientas para optimizar a nuestro gusto la base de datos. Esto puede ser:
 - a) Modificando en el .conf en el apartado de Memory el atributo `shared_buffers`, para proporcionarle más memoria a la base de datos y que acumule más elementos en la memoria antes de enviar a disco. También se puede modificar para el mismo motivo elementos de la sección `QUERY TUNNING`, como reducir a 0 los valores de `cpu_tuple_cost`, `cpu_index_tuple_cost` y `cpu_operator_cost`.
 - b) Hacer un seguimiento y análisis de los logs o del recolector de estadísticas, por ejemplo, si modificamos en el .conf los campos de `log_min_duration_statement` para poder detectar las consultas más lentas y actuar sobre ellas.
 - c) Instrucciones de optimización como las anteriormente mencionadas, `reindex`, `vacuum` y `analyze`.
 - d) Opción de creación de índices que optimizan la consulta.
4. Postgres usa la fórmula de $(disk\ pages\ read * seq_page_cost) + (rows\ scanned * cpu_tuple_cost)$ para calcular los costes. Por lo tanto, podemos modificar el .conf para optimizar la consulta (como se comentó más arriba).
5. Postgres realiza búsquedas secuenciales (o binarias en caso de poder).
6. Antes de realizar la consulta busca los índices.
7. Realiza el mejor árbol posible (elige las mejores rutas). Podemos observar el resultado con `format json`.
8. Al realizar una consulta elige la ruta óptima (observable con `ANALYZE`)

Anexo SCRIPTS PYTHON

○ Producto

```
import random

#MÉTODO PARA CREAR EL ARCHIVO TXT CON LOS DATOS DE LA TABLA
PRODUCTO

def crear_archivo(registros_productos):
    f = open ('Producto.txt','w')
    for elemento in registros_productos:
        f.write('%s \n' % elemento)
    f.close()
```

```

#CREACION DE LA LISTA CON VALORES NO PK
nombre=["Pan de molde","Huevos","Cafe","Helado", "Arroz", "Leche",
"Zumo de naranja", "Platanos", "Lentejas", "Latas de sardinas",
"Harina", "Azucar", "Levadura","Chocolate","Galletas","Cereales",
"Mandarinas", "Fresas", "Zanahoria", "Puerros", "Aceitunas",
"Queso", "Maiz", "Aceite de oliva", "Patatas", "Yogur", "Oregano",
"Maicena", "Magdalenas", "Batido de chocolate", "Natillas",
"Mangos"]

tipo = ["1a calidad", "Fresco", "Novedad", "Ultima unidad", "Con
descuento", "En oferta", "Ecologico", "Calidad extra", "Natural",
"Dietetico"]

descripcion = ["Origen Grecia", "Origen Italia", "Origen
Marruecos", "Origen Portugal" , "Origen Polonia", "Origen
Andalucia", "Origen Valencia", "Origen Ecuador" ]

#VALORES QUE COMPLETAREMOS MÁS ADELANTE
codigo_barras=[]
registros_productos=[]

#TAMAÑO DE LAS LISTAS
tamaño_nombre=len(nombre)
tamaño_tipo=len(tipo)
tamaño_descripcion=len(descripcion)
maximo=1000000

print("CREAMOS LA LISTA CON LOS NUMEROS QUE SERÁN PK")
#CREAR EL NUMERO ALEATORIO SIN REPETICION
for j in range(maximo):
    codigo_barras.append(str(j)+'P')

tamaño_codigo=len(codigo_barras)

#CREAMOS LOS REGISTROS CON TODO LOS CAMPOS
for i in range(maximo):
    aleatorio_nombre=random.randrange(1,tamaño_nombre)
    aleatorio_tipo=random.randrange(1,tamaño_tipo)
    aleatorio_descripcion=random.randrange(1,tamaño_descripcion)
    precio=random.randrange(50,1000)

```

```

print(i)

#tengo que crear un registro para poder guardar datos de diferentes
tipos //PORQUE AQUÍ LOS NÚMEROS LOS ESTOY CONVIRTIENDO A STR Y NO
SÉ SI FUNCIONARÁ EN LA BASE DE DATOS

registros_productos.append(str(codigo_barras[i])+","+nombre[aleatorio_nombre]+","+tipo[aleatorio_tipo]+","+descripcion[aleatorio_descripcion]+","+str(precio))

crear_archivo(registros_productos)

```

○ **Ticket**

```

import random

#MÉTODO PARA CREAR EL ARCHIVO TXT CON LOS DATOS DE LA TABLA
PRODUCTO

def crear_archivo(registros_ticket):
    f = open ('Ticket.txt','w')
    for elemento in registros_ticket:
        f.write('%s \n' % elemento)
    f.close()

#Listas Vacias
registros_ticket=[]
codigo_trabajador=[]
codigo_ticket=[]
fecha=[]

maximo_trabajador=1000000
maximo_ticket=5000000

#CREAR LA FPK DE CODIGO_TRABAJADOR
for t in range(maximo_trabajador):
    codigo_trabajador.append((t))

tamaño_trabajador=len(codigo_trabajador)

#CREACION PK NUMERO DE TICKET

```

```

for s in range(maximo_ticket):
    codigo_ticket.append(s)

total_ticket=len(codigo_ticket)

#CREACIÓN DE FECHA
for i in range(100):
    dia = str(random.randrange(1, 31))
    mes = str(random.randrange(1, 12))
    año = '2019'
    fecha.append(dia + '/' + mes + '/' + año)

tamaño_fecha=len(fecha)

for i in range(maximo_ticket):
    importe=random.randrange(100,10000)
    numero_fecha=random.randrange(1,tamaño_fecha)
    trabajador=random.randrange(1,tamaño_trabajador)
    print(i)
    #ESTAMOS CREANDO OTRA LISTA QUE GUARDARA LOS DIFERENTES DATOS
    registros_ticket.append(str(codigo_ticket[i])+","+str(fecha[numero_
    fecha])+","+str(importe)+","+str(trabajador))

crear_archivo(registros_ticket)

```

○ **Ticket producto**

```

import random

#MÉTODO PARA CREAR EL ARCHIVO TXT CON LOS DATOS DE LA TABLA
Ticket_Productos

def crear_archivo(ticket_producto):
    f = open ('Ticket_Productos.txt','w')
    for elemento in ticket_producto:

```

```

f.write('%s \n' % elemento)
f.close()

#Listas Vacias
codigo_ticket=[]
codigo_barras=[]
ticket_producto=[]

maximo_producto=1000000
maximo_ticket=5000000
maximo=10000000

#CREAR LA FPK DE CODIGO_PRODUCTO
for j in range(maximo_producto):
    codigo_barras.append(str(j)+'P')

tamaño_codigo=len(codigo_barras)

#CREACION FPK NUMERO DE TICKET
for s in range(maximo_ticket):
    codigo_ticket.append(s)

total_ticket=len(codigo_ticket)

for i in range(maximo):
    aleatorio_ticket=random.randrange(1,total_ticket)
    cantidad=random.randrange(1,10)
    aleatorio_producto=random.randrange(1,tamaño_codigo)
    print(i)
    #ESTAMOS CREANDO OTRA LISTA QUE GUARDARA LOS DIFERENTES DATOS
    ticket_producto.append(str(codigo_ticket[aleatorio_ticket])+","+codigo_barras[aleatorio_producto]+" "+str(cantidad))

crear_archivo(ticket_producto)

```

○ **Tienda**

```
import random

#MÉTODO PARA CREAR EL ARCHIVO TXT CON LOS DATOS DE LA TABLA TIENDA
def crear_archivo(registros_tienda):
    f = open ('Tienda.txt','w')
    for elemento in registros_tienda:
        f.write('%s \n' % elemento)
    f.close()

#CREACION DE LA LISTA CON VALORES NO PK
nombre=["Pedro","Ana","Maria","Esther","Raul","Adrian","Roberto","Cristina","Alejandra","Marcos","Esteban","Adriana","Junior","Tomas","Javier","Fernando","Sandra","Arturo","Marta","Laura","Mario"]
ciudad=["Ciudad11","Sevilla","Madrid","Barcelona","Valencia","Granada","Toledo","Palma","Salamanca","Ceuta","Melilla","Avila"]
barrio=["Laurel","Centro","Malagueta","Vegueta","Principal","Gracia","Russafa","Albaicin"]
provincia=["Barcelona","Alicante","Malaga","Vizcaya"]

#CREACION DEL REGISTRO QUE TENGO QUE GUARDAR EN EL TXT Y TENDRA EL TOTAL DE REGISTROS, Y PK
registros_tienda=[]
numeros=[]

#TAMAÑO DE LAS LISTAS
tamaño_nombre=len(nombre)
tamaño_ciudad=len(ciudad)
tamaño_barrio=len(barrio)
tamaño_provincia=len(provincia)
maximo=200000

print("CREAMOS LA LISTA CON LOS NUMEROS QUE SERÁN PK")
#CREAR EL NUMERO ALEATORIO SIN REPETICION
```

```

for j in range(maximo):
    numeros.append(j)

tamaño_numero=len(numeros)
print("CREAMOS LOS REGISTROS CON TODOS LOS CAMPOS")
#CREAMOS LOS REGISTROS CON TODO LOS CAMPOS
for i in range(maximo):
    aleatorio_nombre=random.randrange(1,tamaño_nombre)
    aleatorio_ciudad=random.randrange(1,tamaño_ciudad)
    aleatorio_barrio=random.randrange(1,tamaño_barrio)
    aleatorio_provincia=random.randrange(1,tamaño_provincia)
    print(i)
    #tengo que crear un registro para poder guardar datos de diferentes
    tipos //PORQUE AQUÍ LOS NÚMEROS LOS ESTOY CONVIRTIENDO A STR Y NO
    SÉ SI FUNCIONARÁ EN LA BASE DE DATOS
    registros_tienda.append(str(numeros[i])+","+nombre[aleatorio_nombre
    ]+","+ciudad[aleatorio_ciudad]+","+barrio[aleatorio_barrio]+","+pro
    vincia[aleatorio_provincia])
    crear_archivo(registros_tienda)

```

- **Tienda producto**

```

import random

#MÉTODO PARA CREAR EL ARCHIVO TXT CON LOS DATOS DE LA TABLA
PRODUCTO
def crear_archivo(tienda_productos):
    f = open ('Tiendas_productos.txt',"w")
    for elemento in tienda_productos:
        f.write('%s \n' % elemento)
    f.close()

#LISTAS VACIAS
numeros=[]
codigo_barras=[]

```

```

tienda_productos=[]

maximo_tienda=200000 #Tienda
maximo_cod_bar=1000000 #Maximo para codigo de barras

print("CREAMOS LA LISTA CON LOS NUMEROS QUE SERÁN PK")
#CREAR LA FPK DE TIENDA
for j in range(maximo_tienda):
    numeros.append(j)

tamaño_numero=len(numeros)

#CREAR LA FPK DE CODIGO_BARRAS
for j in range(maximo_cod_bar):
    codigo_barras.append(str(j)+'P')

tamaño_codigo=len(codigo_barras)

#CREAMOS LOS REGISTROS CON TODO LOS CAMPOS
for i in range(20000000):
    tienda=random.randrange(1,tamaño_numero)
    codigo=random.randrange(1,tamaño_codigo)
    stock=random.randrange(10,200)
    print(i)
#LISTA CON DIFERENTES REGISTROS
tienda_productos.append(str(numeros[tienda])+", "+str(codigo_barras[
codigo])+", "+str(stock))

crear_archivo(tienda_productos)

```

○ **Trabajador**

```

import random

#MÉTODO PARA CREAR EL TXT CON LOS DATOS DE LA TABLA TRABAJADOR

```



```

def crear_archivo(registros_trabajador):
    f = open ('trabajador.txt','w')
    for elemento in registros_trabajador:
        f.write('%s \n' % elemento)
    f.close()

#LISTAS CON VALORES NO PK
nombre=["Pedro","Ana","Maria","Esther","Raul","Adrian","Roberto","Cristina","Alejandra","Marcos","Esteban","Adriana","Junior","Tomas","Javier","Fernando","Sandra","Arturo","Adina","Victoria","Marta","Laura","Mario"]
apellidos=["Casas","Apellido2","De la Mata","De Palma","Da Casa","Garcia","Gonzales","Roales","Perez","Sanchez","Gomez","Fernandez","Moreno","Moron","Blanco","Jimenez","Apellido3","Apellidos4"]
puesto=["Reponedor","Encargado","Cajero","Almacen","Responsable seccion","Degustacion","Otros" ]

#Listas Vacias
id_tienda=[]
registros_trabajador=[]
codigo_trabajador=[]

#LONGITUD DE LAS LISTAS INICIALES
tamaño_nombre=len(nombre)
tamaño_apellidos=len(apellidos)
tamaño_puesto=len(puesto)

#FPK DE ID_TIENDA
for j in range(1000000):
    id_tienda.append(j)
tamaño_tienda=len(id_tienda)
#CREAR LA PK DE CODIGO_TRABAJADOR
for t in range(1000000):
    codigo_trabajador.append((t))

```

```
#CREAMOS LOS REGISTROS CON TODO LOS CAMPOS
for i in range(1000000):
    aleatorio_nombre=random.randrange(1,tamaño_nombre)
    aleatorio_apellidos=random.randrange(1,tamaño_apellidos)
    aleatorio_puesto=random.randrange(1,tamaño_puesto)
    salario=random.randrange(1000,5000)
    dni=random.randrange(100000000,999999999)
    aleatorio_tienda=random.randrange(1,tamaño_tienda)
    print(i)
#ESTAMOS CREANDO OTRA LISTA QUE GUARDARA LOS DIFERENTES DATOS
registros_trabajador.append(str(codigo_trabajador[i])+","+str(dni)+
"+nombre[aleatorio_nombre]+","+apellidos[aleatorio_apellidos]+","+
+puesto[aleatorio_puesto]+","+str(salario)+","+str(id_tienda[aleato
rio_tienda]))

crear_archivo(registros_trabajador)
```

Bibliografía

PostgreSQL (12.x)

- Capítulo 14: Performance Tips.
- Capítulo 19: Server Configuration.
- Capítulo 15: Parallel Query.
- Capítulo 24: Routine Database Maintenance Tasks.
- Capítulo 50: Overview of PostgreSQL Internals.
- Capítulo 70: How the Planner Uses Statistics.