Titulación: **Grado en Ingeniería Informática, Ingeniería en Sistemas de Información e InfoAde**

Curso: **2024-2025. 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: Sergio Caballero Ortego

DNI: 03216645A

ALUMNO 2:

Nombre y Apellidos: Laura Herranz Vázquez

DNI: 03226822Z

Fecha: _____

Profesor Responsable: Óscar

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 10 de marzo, Semana 17 de marzo, Semana 24 de marzo, semana 31 de marzo y 7 de abril.
Entrega de práctica:	Día 20 de abril, domingo a las 23:59. Aula Virtual
Documento a entregar:	Este mismo fichero con las respuestas a las cuestiones planteadas, el programa que genera los datos de carga de la base de datos, los ficheros de log de PostgreSQL de esta práctica. No se piden los ficheros de los datos en bruto 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.

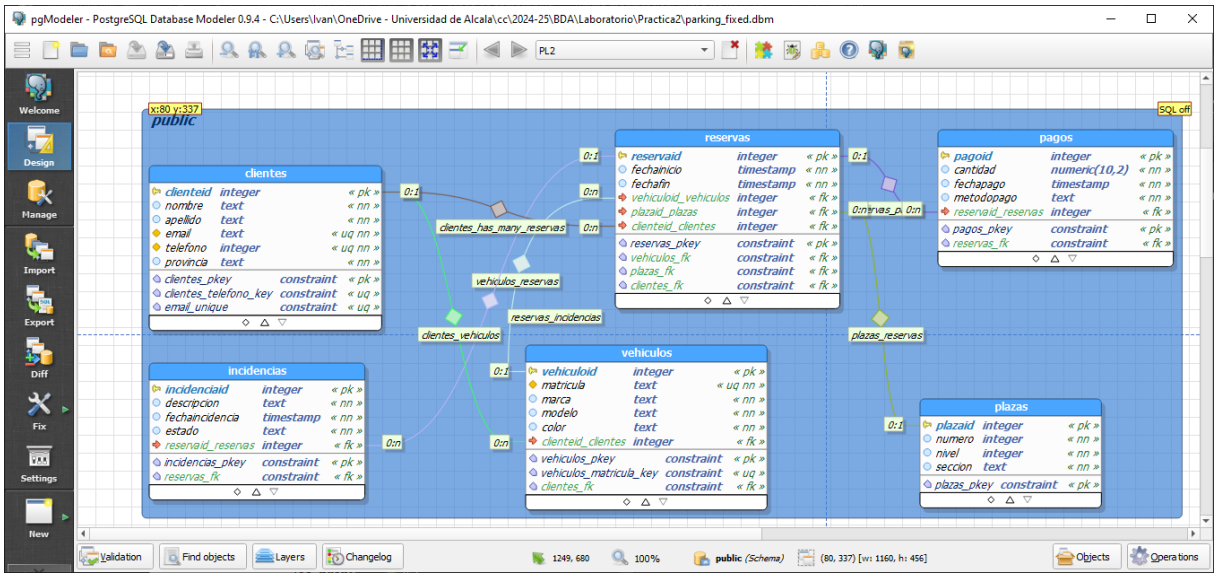
Introducción

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 (17). 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 **TELPARK**. Básicamente la base de datos guarda las reservas de los clientes que se realizan en el parking junto con su método de pago, los vehículos y las incidencias que se pueden producir. Se suministra el modelo relacional construido en **pgmodeler.**



Los datos referidos al año 2024 que hay que generar deben de ser los siguientes:

- Existen 3.000.000 clientes registrados en la APP de Telpark. Las provincias deben de ser asignada aleatoriamente de entre todas las provincias españolas.
- Existen 5.000.000 vehículos dados de alta. Se deben asignar los vehículos a los propietarios de manera aleatoria. Existen 500 marcas de automóviles diferentes donde cada una tiene entre 5 y 20 modelos, con colores entre 5 y 15 (el negro debe ser uno de ellos). La asignación de los valores de cada campo se debe de hacer de manera aleatoria.
- Existen 200.000 plazas de parking de la empresa Telpark. La planta debe ser un número aleatorio entre -10 y 0, y la sección un valor aleatorio comprendido entre las letras A-F.
- Se han producido 40.000.000 de reservas de plazas de parking por la APP. La fecha de inicio y final debe tener el día, mes, y hora generados de manera aleatorio, con una duración de estancia comprendida entre 1 día y 10 días.
- Cada reserva se ha pagado por un método de pago que puede ser efectivo, tarjeta de crédito, tarjeta de débito, PayPal, Bizzum o transferencia. La cantidad de la reserva debe ser el número de horas multiplicado por 3€. La fecha de pago debe ser anterior al comienzo de la estancia en el Parking.
- Se han generado 4.000.000 de incidencias generadas aleatoriamente de entre todas las reservas, donde el estado puede ser nueva, abierta, en proceso, resuelta y cerrada. Todas las asignaciones se hacen de manera aleatoria.

Actividades y Cuestiones

Cuestión 0: Configurar el fichero de Error Reporting and Logging de PostgreSQL para que aparezcan recogidas las sentencias SQL DDL (Lenguaje de Definición de Datos) + DML (Lenguaje de Manipulación de Datos) generadas en dicho fichero. No se pide activar todas las sentencias. No activar la duración de la consulta. También se debe de configurar el log para que en el comienzo de la línea de registro de la información del log ("line prefix") aparezca el DNI de los alumnos que realizan la

práctica (ambos), el nombre del host con su puerto, y la fecha y hora de la operación que se ha realizado. Se ha de configurar también el servidor para que no use el procesamiento paralelo de consultas.

En la ubicación del ordenador donde están los datos de postgresSQL hay una carpeta "data" y dentro de esta está el archivo postgresql.conf. Ahí deberemos de modificar el apartado log_statement para que sea igual a 'mod'. Esto hace que solo registre las sentencias DML y DDL, las pedidas en el enunciado.
Para que no aparezca la duración de las consultas se establece log_duration como 'off'.
Para modificar la salida de los logs con un texto personalizado, se añade en log_line_prefix ambos dni junto a los datos que ya aparecían por defecto. Para que aparezca el nombre del Host ponemos %h, y para que aparezca el puerto añadimos 5432, que es el puerto que usamos.

Para desactivar el procesamiento paralelo de consultas deberemos buscar max_parallel_workers_per_gather en postgresql.conf y darle de valor 0.

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

Sí, PostgreSQL cuenta con un recolector de estadísticas que recopila información sobre los datos en cada campo de las tablas.

Estas incluyen el número total de tuplas en cada tabla y detalles específicos sobre cada columna, como su cardinalidad (cantidad de valores distintos), una lista de los valores más frecuentes junto con el número de veces que aparecen y un histograma que divide los valores en intervalos con cantidades similares de datos.

Además, PostgreSQL mantiene una correlación estadística entre el orden físico y lógico de los valores en una columna. Esto permite estimar el costo de acceder a los datos mediante índices, lo que es crucial para optimizar consultas.

Para que estas estadísticas sean precisas y útiles, el Administrador de Bases de Datos (ABD) debe asegurarse de que se actualicen periódicamente ejecutando el comando ANALYZE, que recalcula estos valores en función de los cambios en los datos.

Cuestión 2: Crear una nueva base de datos llamada **investigar** 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)
- contratos(codigo_contrato tipo numeric PRIMARY KEY, nombre tipo text, entidad tipo text, coste tipo numeric)
- investigadores_contratos(codigo_investigador tipo numeric que sea FOREIGN KEY del campo codigo_investigador de la tabla investigadores con restricciones de tipo RESTRICT en sus operaciones, codigo_contrato tipo numeric que sea FOREIGN KEY del campo codigo_contrato de la tabla contratos con restricciones de tipo RESTRICT en sus operaciones, horas de tipo numeric. La PRIMARY KEY debe ser compuesta de codigo_investigador y codigo_contrato.

Se pide:

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

Creamos la base de datos con CREATE DATABASE investigar;
Usamos \c investigar para conectarnos a dicha base de datos.

Definimos las tablas investigadores, contratos e investigadores_contratos con sus respectivas claves primarias y foráneas, se aplican restricciones de FOREIGN KEY con RESTRICT, lo que impide la eliminación o modificación de registros si están referenciados en otras tablas.
Antes de cargar los datos, se ejecuta el siguiente comando en la consola de psql para activar la medición de tiempos: \timing on
Después de ejecutar los comandos COPY, psql mostrará el tiempo que ha tardado cada carga.

Cargamos la información de los ficheros en dichas tablas
\COPY contratos FROM ./DATOS/datos_contratos WITH (FORMAT csv, HEADER, DELIMITER E',', NULL 'NULL', ENCODING 'UTF-8');
COPY 199999
Time: 1528,263 ms (00:01,528)

\COPY investigadores FROM ./DATOS/datos_investigadores.csv WITH (FORMAT csv, HEADER, DELIMITER E',', NULL 'NULL', ENCODING 'UTF-8');
COPY 2999999
Time: 19700,192 ms (00:19,700)

\COPY investigadores_contratos FROM ./DATOS/datos_investigadores_contratos WITH (FORMAT csv, HEADER, DELIMITER E',', NULL 'NULL', ENCODING 'UTF-8');
COPY 9000000
Time: 896084,898 ms (14:56,085)

Desactivamos el temporizador con \timing off

Cuestión 3: Mostrar las estadísticas obtenidas en este momento para cada tabla. ¿Qué se almacena? ¿Son correctas? Si no son correctas, ¿cómo se pueden actualizar?

Hacemos esta consulta para la tabla investigadores: SELECT relname, reltuples, relpages FROM pg_class WHERE relname = 'investigadores'; para que nos devuelva el número de tuplas y el numero de bloques. Nos devuelve que contiene 3 millones de tuplas, que ocupan 28040 bloques.

Repetimos esto para la tabla contrato, con la misma consulta, nos devuelve que hay 200000 tuplas que ocupan un total de 1800 bloques.

Por último, hacemos la consulta con la tabla investigadores_contratos. nos devuelve que contiene 9000196 tuplas y que ocupa 57328 bloques.

Los datos que nos han devuelto las tablas parecen correctos, viendo la cantidad de filas que contienen los csv. En el caso de que nos hubieran devuelto datos incorrectos, habriamos ejecutado ANALYZE más el nombre de cada una de las tablas. Lo que hace esto es actualizar los datos de las estadísticas de las tablas.

Cuestión 4: Aplicar el comando EXPLAIN a una consulta que obtenga la información de los investigadores con salario de menos de 24.000 euros. ¿Son correctos los resultados del comando EXPLAIN? ¿Por qué? Comparar con lo que se obtendría con lo visto en teoría obteniendo las estadísticas de las tablas con postgres.

Actualizamos las estadísticas con ANALYZE investigadores;
Hacemos la consulta: EXPLAIN SELECT * FROM investigadores WHERE salario < 24000;

Los resultados del EXPLAIN son correctos porque este comando es un optimizador que calcula el mejor camino para ejecutar la consulta, este ha sido utilizar la búsqueda secuencial ya que no existen índices.

Primero obtenemos la longitud de cada tupla de la tabla investigador: Lr = L(codigo_investigador) + L (nombre) + L(apellidos) + L(salario) = 8 + 16 + 19 + 8 = 51 + 4 bytes de puntero = 55.

Bútil = 8192 - información de control = 8192 - 24 = 8168.

fr = 8168 / 55 = 148,55 = 148
br = 3.000.000/148 = 20270 bloques.

El coste de la búsqueda secuencial será de 20270 bloques

Busqueda binaria no podemos usarla al no estar ordenada la tabla

No existe ningún índice por lo que tampoco podremos hacer este tipo de búsqueda.

Por lo tanto el comando EXPLAIN ha usado el tipo de busqueda correcto, ya que es el único que puede utilizar.

Cuestión 5: Aplicar el comando EXPLAIN a una consulta que obtenga el número de contratos en los cuales el investigador trabaja 8 horas y tiene un salario de 20.000 euros. ¿Son correctos los resultados del comando EXPLAIN? ¿Por qué? Comparar con lo que se obtendría con lo visto en teoría obteniendo las estadísticas de las tablas con postgres.

Actualizamos las estadísticas con ANALYZE investigadores; y ANALYZE investigadores_contratos;
Lanzamos la consulta: EXPLAIN SELECT COUNT(*) FROM investigadores i JOIN investigadores_contratos ic ON i.codigo_investigador = ic.codigo_investigador WHERE ic.horas = 8 AND i.salario = 20000;
Los resultados del EXPLAIN son correctos , es el que ha considerado mejor camino para ejecutar la consulta. Este ha sido primero, realizar una búsqueda secuencial en la tabla investigadores para filtrar con salario = 20000, segundo, se emplea la búsqueda por índice primario de investigadores_contratos y aplicar el filtro de horas = 8. Luego se hace el JOIN con bucle anidado y se agrupa para hacer el count. Ahora calcularemos el mejor camino según la teoría.

RAMA INVESTIGADORES: incialmente tiene 3.000.000 de tuplas, pero con la seleccion de salario = 20000 quedan solo 13, según postgres.

RAMA INVESTIGADORES_CONTRATOS: inicialmente 9.000.000 de tuplas, postgres nos dice que con la selección se reduce a 374412 tuplas.
Ambas tablas no tienen índices ni están ordenadas, por lo que habría que usar obligatoriamente la búsqueda secuencial
JOIN: llegarán 13 tuplas por un lado y 374412 por el otro, devolverá 13 tuplas. Como no hay índices, ni están ordenadas ni nos dan función Hash, solo podríamos aplicar el bucle anidado por bloques.
Para contar las tablas usaríamos la ordenación por mezcla externa.
La única diferencia es que postgres usa el índice por clave primaria en investigadores_contratos, un índice creado automaticamente por el programa sobre la tabla.

Cuestión 6: Aplicar el comando EXPLAIN a una consulta que obtenga la información del nombre de los contratos y entidad que tienen un coste mayor de 10000 y menor que 15000, y tienen investigadores de salario mayor de 16000 euros y trabajan 5 horas en ellos. ¿Son correctos los resultados del comando EXPLAIN? ¿Por qué? Comparar con lo que se obtendría con lo visto en teoría obteniendo las estadísticas de las tablas con postgres.

Hacemos la consulta: EXPLAIN SELECT c.nombre, c.entidad FROM contratos c JOIN investigadores_contratos ic ON c.codigo_contrato = ic.codigo_contrato JOIN investigadores i ON ic.codigo_investigador = i.codigo_investigador WHERE c.coste > 10000 AND c.coste < 15000 AND i.salario > 16000 AND ic.horas = 5;

Los resultados del EXPLAIN son correctos, hace un bucle anidado para unir investigadores (la lee usando su índice primario) con, a su vez, un Hash JOIN entre contratos (lectura secuencial) e investigadores contratos(lectura secuencial).

En cuanto a los JOIN, primero se hará el JOIN entre contratos e investigadores_contratos, lo que nos devolverá unas 33,373 tuplas. Esto se debe a que los filtros en estas tablas eliminan muchas tuplas antes de realizar el siguiente paso. Si se realizara primero el JOIN entre investigadores e investigadores_contratos, el resultado sería mucho mayor, de 376,050 tuplas. No tenemos indices, ni función de asociación ni están ordenadeas las tablas, por lo que en ambos JOIN solo podremos hacer un bucle anidado por bloques.

En ambos casos hacemos los JOIN en el mismo orden, solo que postgres considera usar un Hash para el segundo JOIN. Lo mismo pasa con las tablas, solo investigadores es leída con índice, el resto con secuenciales según postgres, mientras que para nosotros solo sería posible utilizar la secuencial.

Cuestión 7: Generar los datos solicitados al comienzo de la práctica para la base de datos **TELPARK** creando un programa para tal fin que deberá de estar en un único fichero y comentado. Pegar el código del fichero en el cuadro de texto que se adjunta a continuación.

El codigo completo no cabe, pero pondremos a modo de ejemplo como se crea el csv para clientes:

```
def guardar_csv(nombre, datos, campos):
    with open(nombre, mode='w', newline='', encoding='utf-8') as file:
        writer = csv.DictWriter(file, fieldnames=campos)
        writer.writeheader()
        writer.writerows(datos)
    print(f"{nombre} generado correctamente.")

for i in range(3_000_000):
    while (telefono := f"6{random.randint(10000000, 99999999)}") in telefonos_generados:
        pass
    telefonos_generados.add(telefono)

    clientes.append({
        "clienteid": i+1, "nombre": random.choice(nombres), "apellido": random.choice(apellidos),
"telefono": telefono, "email": f"cliente{i+1}@mail.com", "provincia": random.choice(provincias)
    })

guardar_csv("clientes.csv", clientes, ["clienteid", "nombre", "apellido","telefono", "email", "provincia"])
```

Cuestión 8: Realizar la carga masiva de los datos generados en la cuestión 7 en la base de datos **TELPARK**. Indicar el proceso seguido y el orden de carga de las tablas, explicando el porqué de dicho orden; y asegurando la consistencia e integridad de los datos cargados. Comparar los tiempos en las tablas implicadas y explicar a qué es debida la diferencia.

Tabla	Tiempo (seg)
Clientes	31
Plazas	0,26
Vehiculos	104
Reservas	1565
Pagos	249
Incidencias	212

Para cargar los datos hemos usado el comando \COPY de postgres, indicando para cada tabla la dirección de archivo del csv, indicando el formato, la existencia de header, el separador (en este caso la coma) y el formato de codificación UTF-8

Para asegurar la integridad de los datos hemos cargado las tablas en el siguiente orden: primero clientes y luego plazas, ya que estas dos tablas no tienen ninguna foreign key, luego se podrán crear primero. Después de estas dos hemos creado vehiculos, que solo tiene como clave foránea el id del cliente, y como este ya ha sido creado, no dará ningún tipo de error. Luego hemos creado reservas, que tiene de clave foránea vehiculoid, cliendeid y plazaid. Después de esta hemos cargado incidencias ya que su clave foránea es el id de la reserva, y por último se carga incidencias, cuya clave foránea también es el id de la reserva.

La que menos tarda es plazas, al ser muy pocos datos y sin restricciones. Clientes también son relativamente pocos datos, pero tarda bastante más que plazas debido a las restricciones UNIQUE de telefono y email. la tabla Vehiculos tarda aún más en cargarse ya que aparte de la restricción UNIQUE en matricula, tiene una clave foránea con la id de los clientes, por lo que por cada entrada debe verififcar que el id del cliente exista en Clientes.

Por eso reservas tarda en cargarse tanto tiempo, debido a sus 40.000.000 de entradas y a las tres claves foráneas con id coche, id cliente e id plaza.

pagos tarda algo menos ya que aunque también tiene 40.000.000 de registros, solo tiene que comprobar 1 clave foránea, el id_reserva

Por último la tabla incidencias, que al igual que la anterior no tiene restricciones y su única clave foránea es el id de reserva, además tiene solo 4.000.000 de registros.

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. Es obligatorio y queda prohibido cambiar la integridad referencial de la base de datos.

Cuestión 9: Realizar una consulta SQL que muestre el “porcentaje de clientes que cumplen que son de la comunidad de Andalucía realizando una reserva durante los meses de verano, pagando una cantidad de más de 150 €, estacionando sus vehículos por debajo de la planta -4, y que no hayan tenido incidencias cerradas con sus vehículos de color negro”.

Obtener el plan de ejecución con el resultado del comando EXPLAIN. 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.

Consulta SQL creada, tiempo de ejecución y resultado obtenido:

La consulta es la siguiente:

```
SELECT  COUNT(DISTINCT c.clienteid) * 100.0 / 3000000 AS Porcentaje
FROM clientes c JOIN reservas r ON c.clienteid = r.clienteid_clientes
JOIN plazas p ON p.plazaid = r.plazaid_plazas JOIN  pagos pa ON pa.reservaid_reservas = r.reservaid
JOIN vehiculos v ON v.vehiculoid = r.vehiculoid_vehiculos LEFT JOIN  incidencias i ON
i.reservaid_reservas = r.reservaid AND i.estado = 'Cerrada'
WHERE
   c.provincia IN ('Almería', 'Cádiz', 'Córdoba', 'Granada', 'Huelva', 'Jaén', 'Málaga', 'Sevilla') AND
EXTRACT(MONTH FROM r.fechainicio) IN (6, 7, 8)  AND pa.cantidad > 150 AND p.nivel < -4  AND
v.color ILIKE 'negro' AND i.incidenciaid IS NULL;
```

El tiempo de ejecución de la consulta es de 43 segundos y devuelve un porcentaje de 0,32713333....

Resultado comando EXPLAIN

Primero se hace una lectura secuencial en la tabla incidencias, donde se filtra por el estado cerrada. Este resultado se encauza a través de un LEFT JOIN posterior con la tabla reservas, de forma que se mantendrán todas las reservas, incluso aquellas que no tienen incidencia cerrada.

A continuación, se realiza otra lectura secuencial sobre la tabla reservas, donde se aplica un filtro por los meses de verano. Este resultado se pasa a un LEFT JOIN mediante un bulce anidado con la tabla incidencias, donde se compara el reservaid de las reservas con el de las incidencias cerradas. Este paso se materializa, asegurando que se utilicen los resultados intermedios de las reservas al realizar la comparación.

Luego, se ejecuta una lectura secuencial sobre la tabla pagos con el filtro cantidad > 150. Los resultados de este escaneo se encauzan a través de un Hash Join con las reservas de antes, donde se construye una tabla hash para realizar la comparación eficiente entre los reservaid de pagos y reservas.

Después, se realiza una búsqueda por índice sobre la tabla clientes usando el índice primario. A esto además se le aplica el filtro de las provincias. Los resultados se encauzan para unirse con las tablas de reservas y pagos en el siguiente paso.

Seguidamente, se hace otra búsqueda por índice sobre la tabla plazas, utilizando el índice primario para buscar las plazas asociadas a cada reserva. El filtro de nivel < -4 se aplica durante esta lectura y los resultados se encauzan.

En la etapa siguiente, se realiza otra lectura por índice sobre la tabla vehiculos, aplicando un filtro por color de vehículo (color ILIKE 'negro'). Luego se combina con un bucle anidado con la tabla anterior.

Finalmente, después de que todas las tablas se hayan combinado, se ordenan los resultados por cliente_id. Luego, se realiza una agregación para contar el número de clientes distintos y calcular el porcentaje correspondiente a la consulta, multiplicando por 100 y dividiendo entre 3.000.000

Comentarios y explicaciones.

Primero calculamos el numero de tuplas que pasarán de cada tabla por las diferentes selecciones que se hacen, para ver el orden óptimo para hacer los JOIN de tal manera que se quiten la mayor cantidad de tuplas cuanto antes:
CLIENTES: Nrc = 8 * (Nr / V(provincia)) = 8 * (3.000.000/50) = 480.000 tuplas
RESERVAS Nrc = 3 * (Nr / V(mes)) = 3 * (40.000.000 / 12) = 9.999.999 tuplas
PAGO: consultamos a la base de datos, nos devuelve que hay 12.222.701 tuplas
PLAZA: Nrc = 6 * (Nr / V(nivel)) = 6 * (200.000 / 11) = 109.090 tuplas
VEHICULOS: Nrc = Nr / V(color) = 5.000.000 / 10 = 500.000 tuplas
INCIDENCIAS: Nrc = Nr / V(Estado) = 4.000.000 / 5 = 800.000 tuplas
De aqui, buscando la manera mas óptima de hacer las uniones hemos obtenido el siguiente orden para los JOIN: ((((Clientes JOIN Reservas) JOIN Plazas) JOIN Pagos) JOIN Vehiculos) LEFT JOIN Incidencias).Mientras que para el EXPLAIN el orden más optimo sería el siguiente: ((((Reservas LEFT JOIN Incidencias) JOIN Pagos) JOIN Clientes) JOIN Plazas) JOIN Vehiculos)
Esta diferencia se puede deber a que el EXPLAIN vea estadísticas internas o propiedades de las tablas que hagan las consultas más ótimas de una manera que de otra.

Cuestión 10: Usando PostgreSQL, y a raíz de los resultados de la cuestión anterior, ¿qué acciones realizaría para mejorar el rendimiento de la consulta sin cambiar el código SQL y por qué? Obtener la información pedida de la cuestión 9 y explicar los resultados. Obtener el plan de ejecución con el resultado del comando EXPLAIN. Comentar los resultados obtenidos y comparar con la cuestión anterior.

Consulta SQL creada, tiempo de ejecución y resultado obtenido:

Podríamos hacer que la consulta se haga algo más rápido ejecutando un ANALYZE antes de ejecutarla. esto actualizará los datos de las tablas, lo que ayudará a optimizar mejor la consulta

```
SELECT  COUNT(DISTINCT c.clienteid) * 100.0 / 3000000 AS Porcentaje
FROM clientes c JOIN reservas r ON c.clienteid = r.clienteid_clientes
JOIN plazas p ON p.plazaid = r.plazaid_plazas JOIN  pagos pa ON pa.reservaid_reservas = r.reservaid
JOIN vehiculos v ON v.vehiculoid = r.vehiculoid_vehiculos LEFT JOIN  incidencias i ON
i.reservaid_reservas = r.reservaid AND i.estado = 'Cerrada'
WHERE
   c.provincia IN ('Almería', 'Cádiz', 'Córdoba', 'Granada', 'Huelva', 'Jaén', 'Málaga', 'Sevilla') AND
EXTRACT(MONTH FROM r.fechainicio) IN (6, 7, 8)  AND pa.cantidad > 150 AND p.nivel < -4  AND
v.color ILIKE 'negro' AND i.incidenciaid IS NULL;
```

El ANALYZE ha tardado 4,2 segundos y la consulta 48 segundos, unos pocos segundos más de lo que tardaba antes. Se obtiene de resultado 0.3271333....

Resultado comando EXPLAIN

Comparando con los resultados del EXPLAIN Hecho antes, hemos visto que hace lo mismo: lee y selecciona en la tabla reservas y la une con incidencias, donde tambien lee y selecciona. El resultado lo une con la tabla de pagos ya filtrada, y el resultado de esto se une a su vez con clientes. Luego la salida de esto se une con plazas, y por último con vehiculos. Finalmente las ordena por clienteid y las agrupa para contarlas.

Todas las tablas las lee secuencialmente, menos clientes, en la que usa el índice primario (como en la cuestión 9). En cuanto a los JOIN, son todos bucles anidados menos el JOIN de reservas con incidencias y el JOIN de pagos. En este caso ha usado un HASH. Esto es diferente a como lo resuelve en la Cuestión 9, dónde solo usa HASH JOIN para pagos.

Comentarios y explicaciones.

Ejecutando el comando ANALYZE actualizamos todas las estadísticas de las tablas como pueden ser el número de filas, su tamaño, el número de valores distintos, etc.

Estos valores son los que utiliza el optimizador para elejir la manera óptima de ejecutar la consulta. Por esto la consulta puede experimentar ligeras modificaciones en los métodos usados y su orden de ejecución. En este caso en concreto el tiempo a aumentado un poco y solo ha cambiado la forma en la que se hace un JOIN. Esto nos puede decir que la consulta ya estaba lo sificientemente optimizada y/o que los datos de las tablas ya estaban actualizados.

Cuestión 11: Usando PostgreSQL, borre el 30% de los clientes almacenados de manera aleatoria y todos sus datos relacionados de la manera más eficiente posible, ¿cuál ha sido el proceso seguido? ¿Y el tiempo empleado en el borrado? Prohibido modificar la Integridad Referencial.

Primero hemos creado un índice para cada tabla:
- En reservas, hemos hecho un índioce por clienteid_clientes y otro para vehiculoid_vehiculos
- En pagos hemos hehco un índice por reservaid_reservas
- En incidencias hemos hecho otro índice, también ordenado por reservaid_reservas
- En vehiculos hemos hehco un índice por clienteid_clientes

La finalidad de estos índices es acelerar la busquedas a la hora de hacer el borrado y evitar los escaneos secuenciales continuos, algo que ha sido problemático para nosotros en las primeras versiones que hemos hecho de los borrados.
Una vez creados, hemos construido una tabla temporal en la que se elige aleatoriamente al 30% de clientes (900.000 tuplas). Para mantener la integridad referencial hemos hecho los DELETE en un orden epecífico:
Primero borramos todos los pagos de los clientes seleccionados, usando la tabla temporal creada anteriormente. Luego de esto borramos las incidencias, usando tambien esta tabla temporal. Luego eliminamos los pagos relacionados con reservas de los vehiculos del cliente, y lo mismo con incidencias, eliminamos las que esten relacionadas con reservas del vehiculo del cliente. Estos dos últimos DELETE los necesitamos hacer ya que si no los hicieramos, a la hora de borrar los vehiculos, nos daba problemas con la integridad referencial. Después de esto eliminamos las reservas de los clientes, seguido eliminamos las reservas de los vehiculos de los clientes por la misma razón que ya hemos explicado antes. Ahora sí, eliminamos los vehiculos de los clientes y por último, a los clientes.
Los tiempos han sido los siguientes:
- Seleccionar 900.000 clientes: 1,4 segundos
- Eliminar los pagos de reservas del cliente: 589 segundos
- Eliminar Incidencias de las reservas del cliente: 26 segundos
- Pagos de reservas del vehiculo del cliente: 418 segundos
- Incidencias de reservas del vehiculo del cliente: 23 segundos
- Reservas del cliente: 1148 segundos
- Reservas del vehiculo del cliente: 496 segundos
- Vehiculos del cliente: 139 segundos
- Clientes: 148 segundos

Este borrado ha tardado un total de 2988 segundos o 49,8 minutos.

Cuestión 12: Ejecute la consulta que pide los datos de la cuestión 9 de nuevo. Obtener el plan de ejecución con el resultado del comando EXPLAIN. Comparar con los resultados anteriores.

Consulta SQL creada, tiempo de ejecución y resultado obtenido:

La consulta es la siguiente:

```
SELECT  COUNT(DISTINCT c.clienteid) * 100.0 / 3000000 AS Porcentaje
FROM clientes c JOIN reservas r ON c.clienteid = r.clienteid_clientes
JOIN plazas p ON p.plazaid = r.plazaid_plazas JOIN  pagos pa ON pa.reservaid_reservas = r.reservaid
JOIN vehiculos v ON v.vehiculoid = r.vehiculoid_vehiculos LEFT JOIN  incidencias i ON
i.reservaid_reservas = r.reservaid AND i.estado = 'Cerrada'
WHERE
   c.provincia IN ('Almería', 'Cádiz', 'Córdoba', 'Granada', 'Huelva', 'Jaén', 'Málaga', 'Sevilla') AND
EXTRACT(MONTH FROM r.fechainicio) IN (6, 7, 8)  AND pa.cantidad > 150 AND p.nivel < -4  AND
v.color ILIKE 'negro' AND i.incidenciaid IS NULL;
```

El tiempo de ejecución de la consulta es de 36 minutos y 55 segundos y devuelve un porcentaje de 0,1607666....

Resultado comando EXPLAIN

Primero se hace una lectura secuencial en la tabla incidencias, donde se filtra por el estado cerrada. Este resultado se encauza a través de un LEFT JOIN posterior con la tabla reservas, de forma que se mantendrán todas las reservas, incluso aquellas que no tienen incidencia cerrada.

A continuación, se realiza otra lectura secuencial sobre la tabla reservas, donde se aplica un filtro por los meses de verano. Este resultado se pasa a un LEFT JOIN mediante un bulce anidado con la tabla incidencias, donde se compara el reservaid de las reservas con el de las incidencias cerradas. Este paso se materializa, asegurando que se utilicen los resultados intermedios de las reservas al realizar la comparación.

Tras esto se hace una búsqueda por índice en cliente, filtrando además solo las provincias de andalucía. Esto se une a la tabla anterior con un JOIN de bucle anidado

Seguidamente, se hace una búsqueda secuencial sobre la tabla plazas. El filtro de nivel < -4 se aplica durante esta lectura y los resultados se encauzan.

Luego se utiliza el índice de reservaid_reserva de la tabla pagos para filtrarla según el costo y esto se une mediante in JOIN de bulce anidado a la tabla resultante anterior.

Y por último, se realiza otra lectura por índice sobre la tabla vehiculos, aplicando un filtro por color de vehículo (color ILIKE 'negro'). Luego se combina con un bucle anidado con la tabla anterior.

Finalmente, después de que todas las tablas se hayan combinado, se ordenan los resultados por cliente_id. Luego, se realiza una agregación para contar el número de clientes distintos y calcular el porcentaje correspondiente a la consulta, multiplicando por 100 y dividiendo entre 3.000.000

Comentarios y explicaciones.

El proceso seguido por el planifiador es muy similar, pero hay algunas diferencias en el orden en el que hace las cosas y los métodos utilizados, además del tiempo de la consulta, que es mucho mayor.

El camino ideado es igual en ambas tablas hasta que se llega al tercer JOIN. En la consulta original este JOIN es entre pagos y la tabla obtenida de los JOIN anteriores, mientras que aqui la tabla pagos se sustituye por clientes. Cuanndo en la original se hace el JOIN de clientes, en la nueva se hace el de pagos. El resto de JOIN están igual. Además mientras que en la original se usaba un HASH para el JOIN de pagos y reservas, aqui lo hace todo con bucles anidados.
Los metodos de lectura en las tablas son iguales en su mayoría. Solo cambia la lectura de pagos, que se usa el índice por reservaid_reservas creado para el borrado anterior , y la lectura de plazas, que en la cuestión 9 usaba el índice primario y aqui usa lectura secuencial. Ambas acaban ordenando y agrupando la tabla resultante.

Cuestión 13: ¿Qué optimización/mejoras de la BD propondría para mejorar los resultados de dicho plan sin modificar el código SQL de la consulta? ¿Por qué? No implementarlo ahora.

Primero haremos un vaccum para limpiar la base de datos de todas las tuplas muertas que han quedado del DELETE de la pregunta 11. Este comando lo usaremos para todas las tablas de la base de datos. Luego para hacer las lecturas de tablas y los JOIN más eficientes, podremos crear índices en cada una de las tablas. Crearíamos un índice en pagos por reservaid y cantidad, otro en clientes por provincia, uno en inincidencias por reservaid_reservas y estado, y por último en reservas por reservaid

Cuestión 14: Usando PostgreSQL, lleve a cabo las operaciones propuestas en la cuestión anterior y ejecute el plan de ejecución de la consulta que pide los datos de la cuestión 9. Obtener el plan de ejecución con el resultado del comando EXPLAIN. Compare los resultados del plan de ejecución con los de los apartados anteriores. Coméntelos.

Consulta SQL creada, tiempo de ejecución y resultado obtenido:

La consulta es la siguiente:

```
SELECT  COUNT(DISTINCT c.clienteid) * 100.0 / 3000000 AS Porcentaje
FROM clientes c JOIN reservas r ON c.clienteid = r.clienteid_clientes
JOIN plazas p ON p.plazaid = r.plazaid_plazas JOIN  pagos pa ON pa.reservaid_reservas = r.reservaid
JOIN vehiculos v ON v.vehiculoid = r.vehiculoid_vehiculos LEFT JOIN  incidencias i ON
i.reservaid_reservas = r.reservaid AND i.estado = 'Cerrada'
WHERE
   c.provincia IN ('Almería', 'Cádiz', 'Córdoba', 'Granada', 'Huelva', 'Jaén', 'Málaga', 'Sevilla') AND
EXTRACT(MONTH FROM r.fechainicio) IN (6, 7, 8)  AND pa.cantidad > 150 AND p.nivel < -4  AND
v.color ILIKE 'negro' AND i.incidenciaid IS NULL;
```

El tiempo de ejecución de la consulta es de 37 minutos y devuelve un porcentaje de 0,1607666....

Resultado comando EXPLAIN

En este caso, todos los JOIN son bucles anidados, no se usa ningún Hash como sí ocurria en preguntas anteriores.
Primero se hace una lectura secuencial en reservas filtrando por la fecha de inicio y se hace un LEFT JOIN con incidencias, que se lee usando el índice por reservaid_reservas y se filtra.

Luego se lee clientes usando el índice primario, se filtra y se une con un JOIN al resultado de la tabla anterior.

El resultado de esto se une con plazas, que se lee secuencialmente y se filtra.

Luego se lee pagos usando su índice organizado por reservaid y se une al resultado anterior

Por último se lee y filtra vehiculos usando su índice primario y se une al resultado anterior

Esta tabla resultante de unir todas las anteriores se ordena, se agrupa por cliente id y se cuenta. Luego el resultado se multiplica por 100 y se divide entre 3.000.000.

Comentarios y explicaciones.

El plan de ejecución en su mayoría se mantiene igual al de la cuestión 12. Los JOINS son del mismo tipo, y se hacen en el mismo orden, pero lo que si cambia ligeramente es la forma en la que se hacen las lecturas de cada tabla.

Incidencias originalmente se leía secuencialmente, ahora se utiliza el índice que la ordena por reservaid. Para clientes se sigue usando el índice primario para la lectura. La tabla plazas se lee secuencialmente, mientras que en pagos se usa el índice por reservas. Por último se lee la tabla vehiculo con su índice primario, se ordenan y se agrupan las tuplas de la tabla resultante de todos los JOIN encadenados, igual que en la cuestión 12.

Vemos que a pesar de las optimizaciones que hemos hecho de VACUUM el tiempo de consulta apenas ha cambiado. Ademas de que de los 4 índices que hemos creado para la optimización, solo se han usado 2, para los otros el optimizador a considerado más eficiente la lectura secuencial o el uso de índices primarios en su lugar.

Cuestión 15: 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.

Al escribir una consulta, psql sigue un proceso en difernetes pasos con el fin de obtener el camino más óptimo posible:

- Primero, el programa verifica que la sintaxis de la consulta sea valida, y que haga referencias a tablas y atributos de estas también válidos y existentes.

Una vez que se haya verificado que la consulta es válida, Postgres genera diferentes planes de ejecución. para determinar cual es el más eficiente, el optimizador utilizará las estadísticas internas de las tablas (número de elementos, cantidad de tuplas, la distribución de valores, si existen índices o no, etc) para realizar una estimación de los costes de cada uno de los distintos planes de ejecución diseñados. De todos estos planes de ejecución, elejirá el de menor coste. Aqui es donde tiene importancia la ejecución del comando ANALYZE. Este comando actualiza las estadisticas de las tablas a su estado actual ,algo esencial al realizar un plan de ejecución. Lógicamente los planes de ejecución que se diseñen serán mejores si las estadísticas con las que trabajan corresponden a los datos reales de las tablas.

Una vez elejido el plan más óptimo, el sistema ejecutará la consulta paso a paso, devolviendo al usuario los resultados.

Bibliografía

PostgreSQL (17)

- 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 68: How the Planner Uses Statistics.
- https://pgtune.leopard.in.ua/
- https://explain.dalibo.com/