Práctica 3. Conceptos Avanzados sobre Bases de Datos Relacionales

Versión 1.0 - 22/10/2025

Introducción

En esta práctica se profundiza en el uso de BBDD relacionales usando PostgreSQL. Mediante procedimientos almacenados, triggers, transacciones y bloqueos, el estudiante aprenderá a trasladar parte de la lógica de negocio hacia la propia BBDD, reduciendo así la dependencia del código externo y mejorando la consistencia de las operaciones. Además, se abordarán técnicas sencillas de optimización de consultas mediante el uso de índices y el análisis de planes de ejecución.

Objetivos docentes

Tras la realización de esta práctica, el estudiante debe ser capaz de:

- Entender y ser capaz de implementar procedimientos almacenados.
- Entender y manejar los triggers en una base de datos.
- Identificar posibles acciones que pueden mejorar el rendimiento de una consulta.
- Entender el funcionamiento de una transacción.
- Entender los mecanismos de bloqueos y deadlocks.

Base de datos

Para el manejo y gestión de datos se utilizará:

- PostgreSQL como gestor de base de datos. Al igual que en la práctica 2, se desplegará en un contenedor Docker mediante un fichero *docker-compose.yml*. Se debe mantener el usuario *alumnodb* con contraseña *1234* y la base de datos *si1*.
- SQLAlchemy (librería Python). Se empleará como interfaz de acceso a la base de datos PostgreSQL desde la API REST.

Tareas a realizar

Rediseño de la base de datos

A partir de la BBDD construida en la práctica 2, se busca construir la siguiente funcionalidad (si alguna de ellas ya la tienes implementada, menos tendrás que hacer en esta práctica):

- Queremos añadir una nueva funcionalidad para gestionar el stock de películas. Cada vez que se añada o elimine un producto del carrito, deberá actualizarse el stock, el precio final del carrito y/o cualquier otra acción necesaria mediante triggers. Si las acciones relacionadas con el carrito se gestionaban desde la API (por ejemplo, actualizar el precio final del carrito con cada insert/delete/update), ahora deberán ser gestionadas desde la propia BBDD mediante triggers.
- Queremos evitar problemas de sincronización con el usuario. Cada vez que se pague un carrito, queremos que su coste se descuente automáticamente del saldo del cliente mediante triggers. Si esto ya se gestionaba desde la API, ahora deberá ser gestionado desde la propia BBDD.
- Queremos almacenar el país de nacionalidad de los clientes.
- Queremos almacenar la fecha de pago de un pedido.
- Queremos almacenar un descuento porcentual que se aplique de forma específica a cada cliente. Cuando el cliente vaya a pagar el carrito, deberá aplicarse este descuento. Por defecto, el cliente tendrá un descuento de un 0%.
- Queremos modificar la implementación del cálculo de la valoración de una película. Para ello utilizaremos dos aproximaciones (si para cualquiera de ellos necesitas crear nuevos campos, hazlo). En la primera, se implementará un procedimiento almacenado quecalculará el valor promedio de las puntuaciones. En la segunda, un trigger actualizará dicho valor (que estará precalculado como atributo de la película) cada vez que se añada/modifique/borre una valoración por parte de un cliente.

Por tanto, se debe crear un nuevo script SQL:

- *actualiza.sql*: con todas las sentencias necesarias para satisfacer los requisitos que se han descrito anteriormente.

Al configurar el contenedor de la BBDD, ahora deberá llamar también a este nuevo script para que se ejecute automáticamente en el arranque junto a *schema.sql* y *populate.sql*.

Tester

Para evaluar este primer apartado, debéis crear un *cliente.py* que compruebe que se cumple con la funcionalidad descrita anteriormente. Además, se deben crear dos nuevas entradas a la API:

- */estadisticaVentas/<año>/<país>* que permita recuperar todas las ventas que se han hecho en un año dado como parámetro (por ejemplo 2020) para los clientes que pertenecen a un país determinado también dado como parámetro (por ejemplo, Perú).
- */clientesSinPedidos* que devuelva los clientes que no tienen pedidos.

Cada consulta deberá ejecutarse en un único execute(), no es válido lanzar varios execute() y luego combinar los resultados en python.

Optimización

En este apartado vamos a estudiar el impacto de un índice. Para ello, vamos a utilizar la sentencia EXPLAIN sobre la consulta anterior (*/estadisticaVentas/<año>/<país>)* para estudiar el plan de ejecución de la consulta.

Con la sentencia EXPLAIN, podríamos identificar cómo mejorar el rendimiento de la consulta (por ejemplo, mediante la creación de un índice). Compara el resultado de EXPLAIN antes y después de hacer los cambios necesarios. Prueba distintas estrategias y discute los resultados.

Para evaluarlo, debéis entregar un script SQL:

- *optimizacion.sql*: con todas las sentencias necesarias para optimizar la consulta anterior.

Transacciones

Cread una entrada a la API que defina una transacción para borrar los clientes de un país dado. La transacción deberá tener mecanismos de rollback. Se utilizarán sentencias SQL (vía execute()) para gestionar la transacción. Para poder realizar este ejercicio, se deberán desactivar (o eliminar) en la BBDD, si las hubiera, todas las restricciones ON DELETE CASCADE referentes al cliente y sus pedidos, pero manteniendo las *foreign keys* que aseguran la integridad. Así no se propagará de forma automática el borrado y tendremos que hacerlo de forma manual.

El objetivo de este apartado es que hagáis distintas versiones de *borraPais*:

- */borraPais/<país>* que funcione correctamente y borre a todos los clientes de un país, incluyendo todo lo relacionado con sus pedidos y/o historial.
- */borraPaisIncorrecto/<país>* que funcione mal por haber implementado el borrado de forma desordenada y de un error de *foreign key*. En este caso, deberá hacer ROLLBACK para dejar todo como estaba.
- */borraPaisIntermedio/<país>* que funcione mal, pero tenga un COMMIT intermedio antes de producirse el error para comprobar que los cambios realizados antes del COMMIT persisten tras el ROLLBACK.

Bloqueos y deadlocks

Vamos a alterar el mecanismo de pago del carrito. Modificad donde consideréis oportuno (triggers, procedures, etc.) para añadir un SLEEP durante el pago de un carrito. Mientras dura el SLEEP, comprobad que no se ha actualizado la BBDD y discutid por qué sucede esto.

Por otro lado, debéis provocar un deadlock. Para ello, mientras dura el SLEEP del pago, actualizad en otra sesión el descuento del cliente. Discutid cómo hacerlo y por qué se está produciendo. ¿Cómo podríais afrontar o evitar estos problemas?

Memoria

Entregad una memoria breve y explicativa con las discusiones pertinentes.

Planificación

Esta práctica está planteada para seguir la planificación:

Semana 1	Rediseño de la BBDD
Semana 2	Optimización
Semana 3	Transacciones y bloqueos
Semana 4	Repaso y dudas

Entrega

La fecha concreta y normas de entrega de las prácticas se encuentran en Moodle. Como mínimo se deben entregar los siguientes archivos:

- Rediseño de la base de datos: *schema.sql, populate.sql, actualiza.sql*
- Integración con Python: *user.py*, *api.py* y *cliente.py*
- Optimización: *optimizacion.sql*
- Ficheros de creación y configuración de los distintos contenedores
- Memoria

Bibliografía

[1] Documentación de PostgreSQL: https://www.postgresql.org/docs/10/index.html

[2] Acceso a bases de datos en Python usando SQLAlchemy: https://www.sqlalchemy.org/

[3] Información sobre transacciones: https://www.postgresql.org/docs/current/sql-commands.html

[4] Información sobre mecanismos de bloqueo:

https://www.postgresql.org/docs/current/mvcc.html

https://www.postgresql.org/docs/current/monitoring-locks.html

https://www.postgresql.org/docs/current/view-pg-locks.html