Curso de SQL Avanzado - Sesión 2

Scotiabank | **Belatrix**

Instructor: Alan Badillo Salas

Contenido

```
Módulo 2: Gestión de Transacciones y Concurrencia

1. Control de transacciones
2. Bloqueo y concurrencia efectiva
3. Manejo de conflictos.
```

Temas

201. Control de transacciones
202. Bloqueo y concurrencia efectiva
203. Manejo de conflictos

201. Control de transacciones

El control de transacciones en SQL, especialmente en un entorno de base de datos Oracle, es una parte fundamental del manejo de bases de datos que permite a los usuarios gestionar cambios en los datos de manera eficiente y segura. Las transacciones son secuencias de operaciones de base de datos que se tratan como una unidad única de trabajo. Si todas las operaciones en la transacción se completan con éxito, la transacción se confirma y todos los cambios de datos se hacen permanentes en la base de datos. Si alguna de las operaciones falla, la transacción se puede revertir, deshaciendo todos los cambios que se hayan hecho durante la transacción.

Aquí te dejo los conceptos clave del control de transacciones en Oracle:

1. Iniciar una Transacción

Una transacción comienza implícitamente cuando se ejecuta el primer comando SQL (como `INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE`, etc.) después de conectar a la base de datos o después de que la última transacción haya sido finalizada con un `COMMIT` o `ROLLBACK`.

2. COMMIT

El comando `COMMIT` se utiliza para finalizar una transacción, haciendo todos los cambios permanentes en la base de datos. Una vez que se emite un `COMMIT`, no puedes revertir los cambios realizados por la transacción.

3. ROLLBACK

El comando `ROLLBACK` se utiliza para deshacer todas las operaciones realizadas en la transacción actual, volviendo al estado en que se encontraba la base de datos antes de iniciar la transacción. Puedes emitir un `ROLLBACK` en caso de que algo salga mal en tu secuencia de operaciones.

4. SAVEPOINT

Los savepoints son puntos intermedios dentro de una transacción que puedes definir para marcar un estado específico de la base de datos. Si necesitas revertir parte de una transacción, puedes hacer un `ROLLBACK` a un `SAVEPOINT` específico, sin necesidad de deshacer toda la transacción. Esto ofrece una mayor flexibilidad en el manejo de transacciones.

5. SET TRANSACTION

Este comando se utiliza para establecer ciertas características de la transacción actual, como el nivel de aislamiento, que determina cómo la transacción debe ser aislada de los efectos de otras transacciones concurrentes. Los niveles de aislamiento incluyen `READ COMMITTED`, `SERIALIZABLE`, entre otros, y afectan cómo y cuándo los cambios hechos por una transacción son visibles para otras transacciones.

6. Autocommit

En algunos entornos de desarrollo, la opción de autocommit puede estar habilitada, lo que significa que cada instrucción SQL se trata como una transacción individual y se comete automáticamente. Sin embargo, en la mayoría de los entornos de producción, especialmente en aplicaciones críticas, se prefiere tener un control explícito sobre las transacciones para garantizar la integridad de los datos.

El control de transacciones es crucial para asegurar la consistencia de los datos, especialmente en aplicaciones que requieren alta disponibilidad y en entornos donde se

realizan muchas operaciones concurrentes. Comprender y utilizar adecuadamente los comandos de control de transacciones en Oracle te permitirá gestionar tus datos de forma más efectiva, asegurando la integridad y la consistencia de tu base de datos.

Ejemplo de una transacción con guardado, reversión y confirmación

A continuación se muestra un ejemplo práctico que involucra el uso de transacciones, `COMMIT`, `SAVEPOINT`, y `ROLLBACK` en un entorno de base de datos Oracle. Este ejemplo simula una secuencia de operaciones de base de datos en una situación hipotética en la que se actualizan registros de empleados y departamentos.

Supongamos que tienes dos tablas en tu base de datos: `empleados` y `departamentos`. Quieres realizar las siguientes operaciones como parte de una única transacción:

1. Actualizar el salario de un empleado.
2. Asignar a un empleado a un nuevo departamento.
3. Después de realizar algunas operaciones, decides que quieres revertir el cambio en el departamento, pero mantener la actualización del salario.

A continuación, se muestra cómo podrías escribir esta secuencia de operaciones usando SQL en Oracle:

```
-- Inicia la transacción automáticamente con la primera operación
BEGIN
    -- Primera operación: Actualizar el salario del empleado con ID 1
    UPDATE empleados
    SET salario = salario + 1000
    WHERE empleado_id = 101;

    -- Crea un SAVEPOINT antes de la siguiente operación
    SAVEPOINT antes_de_actualizar_departamento;

    -- Segunda operación: Asignar al empleado con ID 101 al departame
    UPDATE empleados
    SET departamento_id = 5
    WHERE empleado_id = 101;

    -- Imagina que aquí sucede algo que te hace querer revertir el ca
    -- pero quieres mantener la actualización del salario.

    -- Revierte al SAVEPOINT creado previamente
    ROLLBACK TO SAVEPOINT antes_de_actualizar_departamento;

    -- La transacción aún está activa aquí, y el cambio de salario se

    -- Confirmar la transacción
    COMMIT;
END;
```

Este bloque de código realiza lo siguiente:

- **Inicia una transacción**: La primera operación `UPDATE` inicia la transacción automáticamente.
- **Crea un `SAVEPOINT`**: Antes de realizar la segunda operación, se crea un punto de guardado llamado `antes_de_actualizar_departamento`. Esto permite revertir a este estado específico sin afectar las operaciones previas.
- **Revertir a un `SAVEPOINT`**: Utiliza `ROLLBACK TO SAVEPOINT` para deshacer los cambios realizados después del `SAVEPOINT` específico, en este caso, revertir la asignación del empleado al nuevo departamento pero manteniendo el aumento de salario.
- **Confirmar la transacción**: Al final, se utiliza `COMMIT` para hacer permanentes los cambios que deseas mantener, en este caso, solo el aumento de salario.

Este ejemplo muestra cómo las transacciones, junto con `COMMIT`, `SAVEPOINT`, y `ROLLBACK`, ofrecen control y flexibilidad sobre el manejo de las operaciones de base de datos, permitiéndote asegurar la integridad y consistencia de los datos.

202. Bloqueo y concurrencia efectiva

El manejo de bloqueo y concurrencia efectiva son conceptos fundamentales en sistemas de gestión de bases de datos como Oracle, esenciales para mantener la integridad de los datos y el rendimiento del sistema en entornos con múltiples usuarios o aplicaciones accediendo y modificando los datos simultáneamente. Estos conceptos están estrechamente relacionados con el control de transacciones, pero se centran en cómo las transacciones interactúan y se aíslan entre sí.

Bloqueo

El bloqueo es un mecanismo que los sistemas de bases de datos utilizan para controlar el acceso a los datos durante las transacciones. Su propósito principal es asegurar la integridad de los datos evitando que múltiples transacciones modifiquen el mismo dato al mismo tiempo, lo que podría resultar en inconsistencias o corrupción de datos.

Tipos de Bloqueos

- **Bloqueos de Lectura (Shared Locks):** Permiten a una transacción leer datos mientras evitan que otros lo modifiquen durante la lectura. Varias transacciones pueden tener bloqueos de lectura sobre el mismo dato al mismo tiempo.
- **Bloqueos de Escritura (Exclusive Locks):** Se utilizan cuando una transacción quiere modificar datos. Este tipo de bloqueo previene que otras transacciones lean o escriban en los datos afectados hasta que el bloqueo sea liberado.

Concurrencia

La concurrencia se refiere a la capacidad de múltiples transacciones para acceder y modificar la base de datos al mismo tiempo de manera eficiente, sin interferir entre sí y manteniendo la integridad de los datos. La gestión efectiva de la concurrencia asegura que el sistema de base de datos sea escalable y pueda manejar cargas de trabajo altas sin degradar el rendimiento o comprometer la integridad de los datos.

Control de Concurrencia

Los sistemas de bases de datos implementan modelos de control de concurrencia para manejar cómo las transacciones concurrentes interactúan. Oracle utiliza principalmente el control de concurrencia basado en multiversión (MVCC, Multiversion Concurrency Control), que proporciona cada transacción con una "instantánea" de los datos en un punto específico en el tiempo, permitiendo lecturas consistentes sin bloquear y minimizando la necesidad de bloqueos de lectura.

Estrategias para Manejar la Concurrencia y el Bloqueo

1. **Niveles de Aislamiento de Transacciones:** SQL estándar define varios niveles de aislamiento que determinan el grado en que los efectos de una transacción son visibles para otras transacciones y en qué medida las transacciones están aisladas de los cambios de otras. Oracle soporta niveles como Read Committed y Serializable.
2. **Deadlocks:** Un deadlock ocurre cuando dos o más transacciones se bloquean mutuamente, esperando que la otra libere recursos. Oracle detecta automáticamente deadlocks y resuelve el problema cancelando una de las transacciones involucradas y devolviéndola con un error, permitiendo que las demás continúen.
3. **Optimización de Consultas:** Escribir consultas eficientes puede reducir la necesidad de bloqueos prolongados y mejorar la concurrencia. Esto incluye usar índices adecuados, evitar bloqueos de tabla completa cuando sea posible y diseñar esquemas de base de datos que faciliten la concurrencia.

En resumen, el manejo efectivo del bloqueo y la concurrencia en Oracle y otros sistemas de bases de datos es vital para asegurar que las transacciones se ejecuten de manera eficiente y segura, manteniendo la integridad de los datos y el rendimiento del sistema en entornos con múltiples usuarios y aplicaciones.

Ejemplo de un bloqueo en dos transacciones

A continuación se muestra| un ejemplo simplificado que ilustra cómo los bloqueos funcionan en un entorno de base de datos Oracle, utilizando el concepto de bloqueos exclusivos para modificar datos. Este ejemplo simulará dos transacciones que intentan actualizar el mismo registro en una tabla, lo que resulta en un bloqueo hasta que la primera transacción se completa.

Imagina que tienes una tabla llamada `cuentas`, la cual contiene columnas para `id_cuenta`, `nombre_usuario`, y `saldo`. Dos usuarios simultáneamente intentan actualizar el saldo de la misma cuenta.

Paso 1: Configuración de la Tabla

Primero, asegúrate de tener la tabla `cuentas`:

```
CREATE TABLE cuentas (
    id_cuenta INT PRIMARY KEY,
    nombre_usuario VARCHAR(50),
    saldo DECIMAL(10, 2)
);
```

Paso 2: Transacción 1 - Inicia y Adquiere un Bloqueo

Transacción 1 inicia para actualizar el saldo de la cuenta con `id_cuenta = 1`.

```
-- Transacción 1 inicia
BEGIN;
UPDATE cuentas
SET saldo = saldo + 500
WHERE id_cuenta = 1;
-- No hacemos COMMIT todavía, dejando la transacción abierta
```

En este punto, Transacción 1 tiene un bloqueo exclusivo sobre el registro con `id_cuenta = 1`, porque ha iniciado una operación de escritura (UPDATE) sobre ese registro.

Paso 3: Transacción 2 - Intenta Actualizar el Mismo Registro

Casi al mismo tiempo, otra transacción intenta realizar una operación de actualización sobre el mismo registro:

```
-- Transacción 2 inicia
BEGIN;
UPDATE cuentas
SET saldo = saldo - 200
WHERE id_cuenta = 1;
-- Esta transacción queda bloqueada, esperando que Transacción 1 haga
```

Transacción 2 no puede proceder inmediatamente porque Transacción 1 tiene un bloqueo exclusivo en el registro que intenta modificar. Transacción 2 debe esperar hasta que Transacción 1 complete su operación.

Paso 4: Transacción 1 Completa

```
-- Transacción 1 hace COMMIT, liberando el bloqueo
COMMIT;
```

Una vez que Transacción 1 completa su operación y realiza un `COMMIT`, libera el bloqueo sobre el registro. Ahora, Transacción 2 puede proceder con su operación de actualización.

Paso 5: Transacción 2 Completa

Después de que Transacción 1 libera el bloqueo, Transacción 2 puede completar su operación de actualización y también hacer `COMMIT`:

```
-- Ahora Transacción 2 puede completar y hace COMMIT
COMMIT;
```

Este ejemplo demuestra cómo funciona el mecanismo de bloqueo en una situación de actualización de datos concurrente. Los bloqueos exclusivos aseguran que cada transacción que modifica datos tenga acceso exclusivo a esos datos durante su operación, evitando así conflictos y asegurando la integridad de los datos.

203. Manejo de conflictos

El manejo de conflictos en bases de datos, especialmente en sistemas que soportan transacciones concurrentes como Oracle, es crucial para mantener la integridad de los datos y el rendimiento del sistema. Los conflictos surgen cuando múltiples transacciones intentan acceder o modificar los mismos datos simultáneamente de manera incompatible, lo que puede llevar a problemas como la pérdida de actualizaciones, lecturas sucias, y condiciones de carrera.

Tipos de Conflictos

1. **Pérdida de Actualizaciones:** Ocurre cuando dos transacciones modifican el mismo dato de forma concurrente, y el trabajo de una transacción se sobrescribe por el de otra.
2. **Lecturas Sucias:** Sucede cuando una transacción lee cambios que otra transacción ha hecho pero aún no ha confirmado. Si la segunda transacción se revierte, la primera transacción habrá leído datos que nunca existieron oficialmente.
3. **Lecturas No Repetibles/Fantasmas:** Se da cuando una transacción lee el mismo dato dos veces y encuentra diferentes valores debido a las modificaciones de otras transacciones entre las dos lecturas.

Estrategias de Manejo de Conflictos

Para manejar estos conflictos, los sistemas de gestión de bases de datos implementan varias estrategias:

1. **Niveles de Aislamiento de Transacciones:** Los niveles de aislamiento permiten a los desarrolladores y administradores de bases de datos equilibrar la necesidad de concurrencia contra la probabilidad y aceptabilidad de ciertos tipos de conflictos. Niveles más altos de aislamiento reducen los conflictos pero a costa de la concurrencia y viceversa.
2. **Locking (Bloqueo):** Los sistemas de bases de datos utilizan bloqueos para controlar

el acceso a los datos. Los bloqueos pueden ser exclusivos o compartidos y ayudan a prevenir la corrupción de datos asegurando que solo una transacción pueda modificar los datos a la vez.

3. **Multiversion Concurrency Control (MVCC):** MVCC permite que diferentes transacciones vean diferentes versiones de los datos al mismo tiempo, lo que puede ayudar a evitar bloqueos y reducir conflictos entre transacciones concurrentes.
4. **Detección y Resolución de Deadlocks:** Los deadlocks ocurren cuando dos o más transacciones se bloquean mutuamente, esperando que la otra libere recursos. Los sistemas de bases de datos modernos pueden detectar automáticamente deadlocks y abortar una de las transacciones involucradas para romper el deadlock.
5. **Compensación de Transacciones:** En algunos casos, especialmente en sistemas distribuidos, se pueden diseñar transacciones de manera que si ocurre un conflicto que no se puede resolver de forma satisfactoria, las operaciones realizadas por una transacción se pueden "compensar" o deshacer con otra transacción.
6. **Retry Logic:** La lógica de reintento puede implementarse en la aplicación, donde si una transacción falla debido a un conflicto, la aplicación automáticamente reintenta la transacción después de un breve retraso, posiblemente después de ajustar los datos o la lógica para evitar el conflicto.

Buenas Prácticas

- **Diseñar con la Concurrencia en Mente:** Considerar la concurrencia y el potencial de conflictos desde el inicio del diseño de la base de datos y la aplicación puede ayudar a minimizar problemas más adelante.
- **Minimizar el Tiempo de Transacción:** Mantener las transacciones lo más cortas posible reduce la ventana durante la cual pueden ocurrir conflictos.
- **Monitorización y Ajuste:** Usar herramientas de monitorización para identificar y resolver cuellos de botella y ajustar la configuración de la base de datos y la aplicación según sea necesario para manejar la carga de trabajo concurrente de manera eficiente.

El manejo efectivo de conflictos es fundamental para el diseño y operación de cualquier sistema de base de datos que deba manejar cargas de trabajo concurrentes, asegurando que la base de datos permanezca consistente y performante bajo diversas condiciones de operación.

Vamos a ver cómo se pueden manejar los conflictos en transacciones con ejemplos prácticos, utilizando estrategias comunes como niveles de aislamiento de transacciones, bloqueo, y control de concurrencia basado en multiversión (MVCC). Estos ejemplos te ayudarán a entender cómo las bases de datos manejan situaciones potencialmente

problemáticas derivadas de la ejecución concurrente de transacciones.

Ejemplo 1: Niveles de Aislamiento de Transacciones

Supongamos que dos transacciones, T1 y T2, intentan leer y actualizar los mismos datos simultáneamente en una tabla `cuentas` que tiene columnas `id_cuenta` y `saldo`.

Situación sin control adecuado de aislamiento:

- T1 lee el saldo de la cuenta con `id_cuenta = 1`.
- T2 actualiza el saldo de la misma cuenta, incrementándolo en 100, y hace `COMMIT` de sus cambios.
- T1 lee nuevamente el saldo de la cuenta con `id_cuenta = 1` esperando obtener el mismo valor que antes, pero ahora obtiene un valor diferente debido a la actualización de T2.

Solución con Nivel de Aislamiento Serializable:

Para evitar este problema, se puede establecer el nivel de aislamiento de la transacción T1 a serializable. En este nivel, si T1 intenta leer nuevamente los datos después de que T2 ha realizado cambios, T1 recibiría un error indicando que no puede proceder debido a un conflicto de serialización, o vería los datos como estaban antes de cualquier cambio realizado por T2 (dependiendo de cómo el sistema gestione el nivel serializable).

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;
-- Lecturas y operaciones de T1
COMMIT;
```

Ejemplo 2: Bloqueo Pessimista

Imagina que T1 quiere actualizar el saldo de una cuenta y asegurarse de que nadie más pueda modificar esa cuenta hasta que T1 termine.

Implementación de Bloqueo Pessimista:

- T1 inicia y selecciona el saldo de la cuenta con `id_cuenta = 1` para actualización, lo que coloca un bloqueo exclusivo en ese registro.

  ```
  SELECT saldo FROM cuentas WHERE id_cuenta = 1 FOR UPDATE;
  ```

- Mientras T1 tiene el bloqueo, cualquier otra transacción (T2) que intente leer o

modificar el mismo registro quedará bloqueada hasta que T1 haga COMMIT o ROLLBACK de sus cambios, liberando el bloqueo.

Ejemplo 3: MVCC y Lecturas Consistentes

Supongamos que tienes una transacción T1 que está leyendo una gran cantidad de registros de una tabla pedidos para generar un reporte. Al mismo tiempo, T2 actualiza algunos de los registros que T1 está leyendo.

Manejo de Conflictos con MVCC:

- T1 comienza su operación de lectura. Con MVCC, T1 ve una instantánea consistente de los datos en el momento en que comenzó, incluso si T2 realiza cambios en los datos que T1 está leyendo.
- T2 hace sus actualizaciones y COMMIT. Estos cambios no afectan la instantánea que T1 está utilizando para su lectura, garantizando que T1 tenga una vista consistente de los datos para su reporte.

Buenas Prácticas en el Manejo de Conflictos

1. **Evaluar y Seleccionar el Nivel de Aislamiento Apropiado:** Dependiendo de las necesidades de concurrencia y coherencia de datos de tu aplicación, selecciona un nivel de aislamiento que ofrezca el balance correcto.
2. **Utilizar Bloqueos con Precaución:** Los bloqueos son necesarios para mantener la integridad de los datos, pero un uso excesivo puede llevar a deadlocks y afectar el rendimiento. Utiliza bloqueos solo cuando sea estrictamente necesario.
3. **Aprovechar MVCC para Lecturas Consistentes:** MVCC es una potente característica para mantener lecturas consistentes sin bloquear escrituras, especialmente útil en entornos de lectura intensiva.

Estos ejemplos ilustran cómo diferentes estrategias de manejo de conflictos pueden ser aplicadas en situaciones específicas para mantener la integridad y el rendimiento de las bases de datos en entornos concurrentes.

Manejo de excepciones dentro de la transacción

Prevenir errores dentro de una transacción y asegurar que se realice un ROLLBACK en caso de error es una práctica crucial para mantener la integridad de los datos en cualquier sistema de gestión de bases de datos, incluido Oracle. La idea es capturar cualquier error que ocurra durante la ejecución de la transacción y, automáticamente, revertir todas las operaciones realizadas hasta ese punto. Esto se puede lograr mediante el manejo de excepciones en procedimientos almacenados o bloques PL/SQL, o mediante la lógica de aplicación.

A continuación, se muestra cómo manejar esto en un bloque PL/SQL en Oracle:

Uso de Bloques PL/SQL para Manejo de Excepciones

```
BEGIN
    -- Inicia la transacción
    -- Ejemplo: Actualizar el saldo de una cuenta
    UPDATE cuentas SET saldo = saldo - 100 WHERE id_cuenta = 123;

    -- Aquí podrías tener más operaciones SQL como parte de tu transac

    -- Si todo va bien hasta este punto, se hace COMMIT
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        -- Si ocurre algún error, revierte todas las operaciones
        ROLLBACK;

        -- Opcionalmente, registra el error o envía un mensaje de erro
        -- Por ejemplo, podrías usar DBMS_OUTPUT.PUT_LINE o registrar
        DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);

        -- Puedes optar por propagar el error después del rollback
        RAISE;
END;
```

En este ejemplo, cualquier operación dentro del bloque `BEGIN...END` forma parte de la transacción. Si se produce un error en cualquier punto dentro de este bloque, el flujo de control pasa al bloque `EXCEPTION`, donde se manejan los errores.

- **`ROLLBACK`**: Se ejecuta para revertir todas las operaciones realizadas en la transacción actual, asegurando que no se apliquen cambios parciales a la base de datos que podrían violar la integridad de los datos.
- **`WHEN OTHERS`**: Es un manejador de excepciones genérico que captura cualquier error. Para manejar errores específicos de manera diferente, puedes definir múltiples secciones `WHEN` para diferentes tipos de excepciones de Oracle.
- **`DBMS_OUTPUT.PUT_LINE`**: Se utiliza para imprimir el mensaje de error en la consola o interfaz de usuario, lo cual es útil para fines de depuración o registro.
- **`RAISE`**: Propaga el error capturado después de realizar el rollback para que pueda ser manejado o registrado por una capa superior, como la aplicación que llamó al procedimiento almacenado.

Consideraciones Adicionales

- **Manejo de Errores Específicos**: Puedes capturar y manejar errores específicos

utilizando sus identificadores de excepción en lugar de usar `WHEN OTHERS` para todas las excepciones. Esto te permite tener un control más granular sobre el manejo de errores y realizar acciones específicas basadas en el tipo de error.

- **Log de Errores**: Considera registrar los errores en una tabla dedicada a logs dentro de tu base de datos. Esto puede ser invaluable para el diagnóstico y análisis post-mortem de incidentes.
- **Lógica de Aplicación**: En algunos casos, especialmente cuando se trabaja con múltiples bases de datos o sistemas, es posible que desees manejar el `ROLLBACK` en la lógica de la aplicación. Asegúrate de que tu aplicación pueda manejar adecuadamente las excepciones y realizar los `ROLLBACK` necesarios cuando se interactúa con la base de datos.

Implementar un manejo robusto de errores en tus transacciones te ayudará a mantener la consistencia y fiabilidad de tu base de datos frente a condiciones inesperadas.

Ejemplo de propagación de excepciones con disparadores y procedimientos almacenados

Generar y manejar un error cuando se intenta colocar un saldo negativo en una cuenta implica establecer reglas de negocio dentro de tu base de datos o aplicación. Hay varias formas de implementar esta lógica en un entorno de base de datos Oracle, incluyendo el uso de restricciones en la base de datos, triggers (disparadores) y bloques PL/SQL para la validación de datos antes de permitir la actualización o inserción de registros.

Usando Triggers para Prevenir Saldos Negativos

Una forma efectiva de evitar saldos negativos es mediante el uso de un trigger que se ejecute antes de actualizar o insertar en la tabla de cuentas. Este trigger puede verificar si la operación resultará en un saldo negativo y, de ser así, generar un error para abortar la transacción.

Aquí te muestro cómo podrías hacerlo:

```
CREATE OR REPLACE TRIGGER evitar_saldo_negativo
BEFORE UPDATE OR INSERT ON cuentas
FOR EACH ROW
BEGIN
    IF :NEW.saldo < 0 THEN
        RAISE_APPLICATION_ERROR(-20001, 'El saldo de la cuenta no pued
    END IF;
END;
/
```

Este trigger se activa antes de cada operación de inserción o actualización en la tabla `cuentas`. Utiliza `RAISE_APPLICATION_ERROR` para generar un error con un mensaje personalizado si el nuevo valor de `saldo` es menor que cero. El código de error `-20001` es un número arbitrario que elegí para este ejemplo; puedes utilizar cualquier número en el rango de `-20000` a `-20999`, que Oracle reserva para errores definidos por el usuario.

Usando Bloques PL/SQL

Otra forma de manejar esta validación es dentro de un procedimiento almacenado o bloque PL/SQL que actualiza el saldo. Este método te da más flexibilidad para realizar validaciones complejas y manejar errores de manera más sofisticada.

```
CREATE OR REPLACE PROCEDURE actualizar_saldo(id_cuenta IN INT, monto I
BEGIN
    -- Verificar si el monto resulta en un saldo negativo
    IF monto < 0 THEN
        -- Selecciona el saldo actual para verificar
        SELECT saldo INTO v_saldo_actual FROM cuentas WHERE id_cuenta
        IF v_saldo_actual + monto < 0 THEN
            RAISE_APPLICATION_ERROR(-20002, 'La operación resultaría e
        END IF;
    END IF;

    -- Procede con la actualización si no hay errores
    UPDATE cuentas SET saldo = saldo + monto WHERE id_cuenta = id_cuer

    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        RAISE;
END actualizar_saldo;
/
```

En este procedimiento, primero se verifica si la aplicación del `monto` al `saldo` actual resultaría en un saldo negativo. Si es así, se genera un error utilizando `RAISE_APPLICATION_ERROR`, similar al trigger. La ventaja de este enfoque es que puedes incorporar lógica adicional antes de la actualización y manejar transacciones de forma más controlada dentro del procedimiento.

Consideraciones

- **Restricciones de Integridad:** Oracle también permite definir restricciones de integridad directamente en la definición de la tabla. Sin embargo, para el caso de evitar saldos negativos que dependen del valor previo y nuevo, un trigger o

procedimiento almacenado es más adecuado.

- **Manejo de Errores en Aplicaciones:** Es importante manejar adecuadamente los errores generados por `RAISE_APPLICATION_ERROR` en tu aplicación, asegurando que los usuarios reciban feedback claro sobre por qué una operación no se pudo completar.

Implementar una validación a nivel de base de datos como esta asegura que tu base de datos se mantenga consistente y que las reglas de negocio se apliquen de manera efectiva, incluso antes de que los datos sean modificados o insertados.