Elementos de Programación en SQL Extendido

Este apunte intenta resumir elementos básicos de programación en bases de datos relacionales, con ejemplos prácticos en PL/pgSQL para PostgreSQL y Transact-SQL para SQL Server. Dado el enfoque introductorio, no se cubren exhaustivamente todos los aspectos y características específicas de programación en estos entornos. Para una cobertura completa y actualizada, se recomienda consultar la documentación oficial de cada producto.

Los temas abordados son:

- Declaración de variables
- Manejo de cursores
- Utilización de batches
- Estructuras de control
- Uso de Secuencias
- Manejo de transacciones

Declaración de variables

La declaración de variables es una parte fundamental en los procedimientos almacenados y funciones. Los procedimientos pueden contener lógica compleja, y las variables ayudan a manejar y manipular los datos de manera eficiente.

PostgreSQL
La declaración y el uso de variables se hace dentro de bloques PL/pgSQL.

```
CREATE OR REPLACE FUNCTION
persona.obtener_informacion_provincia(codigo_provincia INT)
RETURNS TABLE(id BIGINT, descripcion VARCHAR)
LANGUAGE plpgsql AS $$
DECLARE
    v_id BIGINT;
    v_descripcion VARCHAR(50);
BEGIN
    SELECT id, descripcion INTO v_id, v_descripcion
    FROM persona.provincia
    WHERE codigo = codigo_provincia;

    RETURN QUERY SELECT v_id, v_descripcion;
END;
$$;
```

SQL Server
Las variables se declaran con la palabra clave DECLARE y pueden utilizarse para almacenar valores intermedios en los procedimientos.

```
CREATE PROCEDURE persona.obtener_informacion_provincia
    @codigo_provincia INT
AS
BEGIN
    DECLARE @v_id BIGINT;
    DECLARE @v_descripcion VARCHAR(50);

    SELECT @v_id = id, @v_descripcion = descripcion
    FROM persona.provincia
    WHERE codigo = @codigo_provincia;

    SELECT @v_id AS id, @v_descripcion AS descripcion;
END;
```

Manejo de cursores

Los cursores permiten manipular filas de datos de una consulta de forma individual. Son útiles para operaciones complejas en las que cada fila necesita ser procesada secuencialmente, y no pueden abordarse adecuadamente con una simple operación de conjunto.

Ejemplo en PostgreSQL

```
CREATE OR REPLACE FUNCTION venta.procesar_facturas()
LANGUAGE plpgsql AS $$
DECLARE
    factura_cursor CURSOR FOR SELECT id, total FROM venta.factura
      WHERE fecha >= CURRENT_DATE - INTERVAL '1 year';
    v_factura_id BIGINT;
    v_total NUMERIC;
BEGIN
    OPEN factura_cursor;

    LOOP
        FETCH factura_cursor INTO v_factura_id, v_total;
        EXIT WHEN NOT FOUND;

        RAISE NOTICE 'Factura ID: %, Total: %', v_factura_id, v_total;
    END LOOP;

    CLOSE factura_cursor;
END;
$$;
```

Condición: WHERE fecha >= CURRENT_DATE - INTERVAL '1 year'

- El filtrado se aplica sobre la columna fecha.
- PostgreSQL usa *current_date* para obtener la fecha actual sin incluir la hora (por ejemplo, si hoy es 2024-10-25, entonces current_date devuelve exactamente 2024-10-25).
- **interval '1 year'** representa un período de 1 año. Al restar este interval de current_date, se obtiene la fecha correspondiente a hace un año desde el día actual.
- En este caso, si hoy es 2024-10-25, *current_date - interval '1 year'* sería 2023-10-25.
- La consulta traerá todas las filas en las que fecha sea mayor o igual a 2023-10-25, es decir, todas las facturas generadas desde el 25 de octubre de 2023 hasta hoy.

Ejemplo en SQL Server

```
CREATE PROCEDURE venta.procesar_facturas
AS
BEGIN
    DECLARE factura_cursor CURSOR FOR
        SELECT id, total
        FROM venta.factura
        WHERE fecha >= DATEADD(YEAR, -1, GETDATE());

    DECLARE @v_factura_id BIGINT;
    DECLARE @v_total DECIMAL(38, 2);

    OPEN factura_cursor;

    FETCH NEXT FROM factura_cursor INTO @v_factura_id, @v_total;
    WHILE @@FETCH_STATUS = 0
    BEGIN
        PRINT 'Factura ID: ' + CAST(@v_factura_id AS VARCHAR) + ', Total: ' +
             CAST(@v_total AS VARCHAR);

        FETCH NEXT FROM factura_cursor INTO @v_factura_id, @v_total;
    END;

    CLOSE factura_cursor;
    DEALLOCATE factura_cursor;
END;
```

Condición: WHERE fecha >= DATEADD(YEAR, -1, GETDATE())

- Esta cláusula WHERE aplica un filtro sobre la columna fecha.
- *GETDATE()*: Esta función en SQL Server devuelve la fecha y hora actuales del sistema. A diferencia de current_date en PostgreSQL, GETDATE() incluye tanto la fecha como la hora.
- *DATEADD*: Esta función permite sumar o restar intervalos de tiempo a una fecha en SQL Server.
    - El primer argumento (YEAR) especifica que se trabajará con intervalos de años.
    - El segundo argumento (-1) indica que se restará 1 año a la fecha actual.
    - El tercer argumento (GETDATE()) es la fecha de referencia, en este caso la fecha actual.
- En resumen, DATEADD(YEAR, -1, GETDATE()) calcula la fecha y hora de un año atrás desde el momento actual.

Recomendaciones sobre el Uso de Cursores

Usar cursores en bases de datos trae beneficios en contextos específicos:

- Procesamiento de datos fila por fila: útiles cuando se necesita realizar operaciones secuenciales o lógicas en cada fila de una consulta. Por ejemplo, para ejecutar cálculos complejos o actualizaciones en cada registro de un conjunto grande de datos.
- Operaciones que no se pueden realizar en conjunto: Cuando el procesamiento en bloque no es adecuado, los cursores permiten aplicar condiciones o transformaciones únicas en cada fila de manera controlada.
- Iteración controlada en entornos críticos: Si una operación en varias filas depende de resultados individuales, los cursores facilitan la lógica y la consistencia.

  Sin embargo, como los cursores pueden afectar el rendimiento al realizar un procesamiento fila a fila, es recomendable usarlos con moderación y evaluar si una operación de conjunto es viable antes de elegir un cursor. Para operaciones masivas, usar instrucciones UPDATE, DELETE o INSERT en bloque es generalmente más rápido y eficiente.

Comparación con ejemplos

Ejemplo 1: Eliminación con un Simple DELETE

Se usa una única instrucción DELETE con una subconsulta para encontrar todos los productos que no tienen referencias en la tabla venta.factura_detalle (nunca fueron vendidos).

```
DELETE FROM producto.producto
WHERE id NOT IN (
    SELECT DISTINCT id_producto
    FROM venta.factura_detalle
);
```

- La subconsulta SELECT DISTINCT id_producto FROM venta.factura_detalle obtiene todos los id_producto que están referenciados en la tabla venta.factura_detalle.
- La cláusula WHERE id NOT IN (...) garantiza que solo se eliminarán aquellos productos cuyo id no aparece en la lista de id_producto de factura_detalle.
- Al ejecutarse en una sola operación, esta comando es eficiente y optimizado para la eliminación en conjunto.

Ejemplo 2: Eliminación con un Cursor

Realización de la misma operación usando un cursor.

```
DO $$
DECLARE
    product_cursor CURSOR FOR
        SELECT id FROM producto.producto
        WHERE id NOT IN (
            SELECT DISTINCT id_producto
            FROM venta.factura_detalle
        );
    v_id BIGINT;
BEGIN
    OPEN product_cursor;
    LOOP
        FETCH product_cursor INTO v_id;
        EXIT WHEN NOT FOUND;

        DELETE FROM producto.producto WHERE id = v_id;
    END LOOP;

    CLOSE product_cursor;
END;
$$;
```

- El cursor product_cursor selecciona todos los productos que no están referenciados en factura_detalle.
- El bucle LOOP obtiene el id de cada producto que cumple la condición.
- Para cada id, ejecuta un DELETE FROM producto.producto WHERE id = v_id.
- Se repite hasta que todos los registros no referenciados han sido eliminados.

Conclusiones

- **Eficiencia**: El primer método con DELETE realiza una única operación en bloque, lo que permite al motor de la base de datos optimizar el acceso y manipulación de datos.
- **Performance**: En el segundo método, el cursor ejecuta una sentencia DELETE para cada registro no referenciado, lo que resulta en múltiples operaciones de entrada/salida y bloqueo de registros, afectando significativamente el rendimiento en una tabla grande.
- **Sobrecarga**: El cursor incrementa la carga en el servidor al realizar múltiples comandos en vez de una sola operación en bloque.

Utilización de batches

En PostgreSQL, los bloques DO permiten ejecutar bloques de código PL/pgSQL (el lenguaje de procedimientos de PostgreSQL) de una forma similar a los bloques de Transact-SQL en SQL Server.

Bloques DO en PostgreSQL

Un bloque DO es un contenedor que permite ejecutar código PL/pgSQL en línea sin crear una función almacenada. Esto es útil cuando se necesita ejecutar un conjunto de instrucciones o lógica procedural de una vez sin la intención de reutilizar el código.

Sintaxis básica del bloque DO:

```
DO
$$
BEGIN
    -- Sentencias PL/pgSQL
END;
$$
```

- **DO**: Es la palabra clave que indica el inicio de un bloque anónimo de PL/pgSQL en PostgreSQL.
- **$$**: Es el delimitador de la función.
- **BEGIN ... END;**: El bloque en sí, donde se puede codificar cualquier lógica procedural como condicionales (IF), bucles (LOOP, FOR), declaraciones de control (RAISE, RETURN), y operaciones sobre tablas.

Ejemplo

```
DO
$$
BEGIN
    UPDATE ventas
    SET total = total * 1.1
    WHERE fecha >= current_date - interval '1 month';

    IF (SELECT COUNT(*) FROM ventas WHERE total > 1000) > 10 THEN
        RAISE NOTICE 'Más de 10 ventas superiores a 1000';
    END IF;
END;
$$
```

En este caso, el bloque DO ejecuta un conjunto de instrucciones:

1. Actualiza los totales de la tabla ventas.
2. Realiza una verificación condicional con IF y utiliza RAISE NOTICE para mostrar un mensaje si se cumple la condición.

Bloques de Transact-SQL

En SQL Server, el equivalente funcional es más directo porque se puede ejecutar cualquier conjunto de instrucciones Transact-SQL sin necesidad de un bloque específico. Puedes usar BEGIN...END sin delimitadores adicionales:

```
BEGIN
    UPDATE ventas
    SET total = total * 1.1
    WHERE fecha >= DATEADD(MONTH, -1, GETDATE());

    IF (SELECT COUNT(*) FROM ventas WHERE total > 1000) > 10
        PRINT 'Más de 10 ventas superiores a 1000';
END;
```

En SQL Server, el conjunto de instrucciones puede ejecutarse libremente y pueden incluir procedimientos almacenados, transacciones, condicionales, y declaraciones sin la necesidad de un contenedor especial como DO.

Otros ejemplos

PostgreSQL

```
DO $$
BEGIN
    INSERT INTO persona.provincia (id, version, codigo, descripcion)
    VALUES (persona.persona_sequence.nextval, 1, 999, 'Provincia Test');
    INSERT INTO persona.localidad (id, version, id_provincia, codigo,
descripcion, codigo_postal)
    VALUES (persona.persona_sequence.nextval, 1, 999, 999, 'Localidad Test',
12345);
    RAISE NOTICE 'Batch ejecutado con éxito.';
END;
$$;
```

SQL Server

```
BEGIN
    INSERT INTO persona.provincia (id, version, codigo, descripcion)
    VALUES (NEXT VALUE FOR persona.persona_sequence, 1, 999, 'Provincia Test');
    INSERT INTO persona.localidad (id, version, id_provincia, codigo,
descripcion, codigo_postal)
    VALUES (NEXT VALUE FOR persona.persona_sequence, 1, 999, 999, 'Localidad
Test', 12345);
    PRINT 'Batch ejecutado con éxito.';
END;
```

Estructuras de control

IF-THEN-ELSE

En **PostgreSQL** y **SQL Server**, el condicional IF-THEN-ELSE se usa en procedimientos almacenados o funciones. Permite ejecutar un bloque de código solo si se cumple una condición específica.

PostgreSQL

El siguiente ejemplo verifica si una localidad está asignada a una sucursal. Si no, la asigna por defecto.

```
DO $$
BEGIN
    IF NOT EXISTS (SELECT 1 FROM persona.sucursal WHERE id_localidad = 1) THEN
        INSERT INTO persona.sucursal (id, version, codigo, descripcion,
             domicilio, id_localidad)
             VALUES (nextval('persona.persona_sequence'), 1, 101, 'Sucursal
             Default', 'Domicilio Default', 1);
    ELSE
        RAISE NOTICE 'La localidad ya tiene una sucursal asignada.';
    END IF;
END $$;
```

SQL Server

En T-SQL la estructura IF-ELSE es similar:

```
IF NOT EXISTS (SELECT 1 FROM persona.sucursal WHERE id_localidad = 1)
BEGIN
    INSERT INTO persona.sucursal (id, version, codigo, descripcion, domicilio,
id_localidad)
    VALUES (NEXT VALUE FOR persona.persona_sequence, 1, 101, 'Sucursal Default',
'Domicilio Default', 1);
END
ELSE
BEGIN
    PRINT 'La localidad ya tiene una sucursal asignada.';
END;
```

Bucles FOR/WHILE

FOR en PostgreSQL

En PostgreSQL, el bucle FOR es muy útil para recorrer filas de una consulta. En SQL Server, se usa WHILE. El siguiente bloque revisa los productos en una subcategoría y actualiza el precio unitario si es menor a un valor específico.

```
DO $$
DECLARE
    producto RECORD;
BEGIN
    FOR producto IN
        SELECT id, precio_unitario FROM producto.producto WHERE id_subcategoria
= 1
    LOOP
        IF producto.precio_unitario < 100 THEN
            UPDATE producto.producto
            SET precio_unitario = 100
            WHERE id = producto.id;
        END IF;
    END LOOP;
END $$;
```

- Aquí se declara la variable `producto` como un tipo `RECORD`, que es una estructura que permite almacenar una fila completa de resultados (cada columna de una fila).
- `RECORD` es útil en casos como este, donde queremos trabajar con varias columnas de una tabla (en este caso `id` y `precio_unitario` de la tabla `producto`).
- Cada fila seleccionada se asigna a la variable `producto`, permitiendo el acceso a `producto.id` y `producto.precio_unitario` para cada iteración del bucle.

WHILE en SQL Server

En SQL Server, la iteración sobre registros generalmente se logra con cursores o bucles WHILE.

En este ejemplo se revisan y actualizan precios de una subcategoría.

```
DECLARE @id BIGINT, @precio_unitario DECIMAL(38, 2);

DECLARE cursor_producto CURSOR FOR
    SELECT id, precio_unitario FROM producto.producto WHERE id_subcategoria = 1;

OPEN cursor_producto;
FETCH NEXT FROM cursor_producto INTO @id, @precio_unitario;

WHILE @@FETCH_STATUS = 0
BEGIN
    IF @precio_unitario < 100
    BEGIN
        UPDATE producto.producto
        SET precio_unitario = 100
        WHERE id = @id;
    END;
    FETCH NEXT FROM cursor_producto INTO @id, @precio_unitario;
END;

CLOSE cursor_producto;
DEALLOCATE cursor_producto;
```

`WHILE` en PostgreSQL

Supongamos que queremos incrementar el precio de un producto en un 5% hasta que su precio alcance o supere un valor específico, por ejemplo, 500.

```
DO $$
DECLARE
    precio_actual NUMERIC(38,2);
    incremento NUMERIC(38,2) := 0.05;
BEGIN
    SELECT precio_unitario INTO precio_actual
      FROM producto.producto WHERE id = 1;
    WHILE precio_actual < 500 LOOP
        precio_actual := precio_actual * (1 + incremento);

        UPDATE producto.producto
        SET precio_unitario = precio_actual
        WHERE id = 1;
        -- Mostrar el precio actualizado en cada iteración
        RAISE NOTICE 'Nuevo precio: %', precio_actual;
    END LOOP;
END $$;
```

Uso de Secuencias

Las **secuencias** son objetos en bases de datos relacionales que permiten generar números únicos y consecutivos. Se utilizan principalmente para crear valores automáticos en columnas clave, como los identificadores (id) de las filas, evitando duplicados y facilitando el manejo de datos. El uso de secuencias es una práctica común para gestionar claves primarias o cualquier campo que requiera un valor incremental.

Tanto en **PostgreSQL** como en **SQL Server**, las secuencias permiten el control de cómo se generan los números y ofrecen flexibilidad para adaptarse a distintos requerimientos.

PostgreSQL

Las secuencias son objetos independientes de las tablas, aunque es común vincularlas a columnas mediante la función nextval. Se permite configurar el valor inicial, el incremento y otros parámetros.

Creación y Uso de Secuencias

Crear una Secuencia:

```
CREATE SEQUENCE persona.persona_sequence
START WITH 1  -- Valor inicial
INCREMENT BY 1;  -- Incremento
```

Uso de Secuencias en una Inserción: Se usa la función `nextval` para generar un valor nuevo cada vez que se necesite:

```
INSERT INTO persona.provincia (id, version, codigo, descripcion)
VALUES (nextval('persona.persona_sequence'), 1, 101, 'Provincia Ejemplo');
```

Funciones de Secuencia:

- nextval('secuencia'): Obtiene el siguiente valor y lo incrementa.
- currval('secuencia'): Obtiene el valor actual de la secuencia (sin incrementarla).
- setval('secuencia', valor): Establece un nuevo valor para la secuencia.

Secuencias en SQL Server

Las secuencias se manejan con el mismo propósito, aunque el objeto de secuencia tiene su propia sintaxis de creación y uso. SQL Server introdujo las secuencias en versiones más recientes (a partir de SQL Server 2012).

Creación y Uso de Secuencias en SQL Server

Crear una Secuencia:

```
CREATE SEQUENCE persona_sequence
START WITH 1  -- Valor inicial
INCREMENT BY 1;  -- Incremento
```

Uso de Secuencias en una Inserción: Se usa la función `NEXT VALUE FOR` para obtener el siguiente valor en una secuencia.

```
INSERT INTO persona.provincia (id, version, codigo, descripcion)
VALUES (NEXT VALUE FOR persona_sequence, 1, 101, 'Provincia Ejemplo');
```

Funciones de Secuencia:

- NEXT VALUE FOR secuencia: Similar a nextval en PostgreSQL, obtiene el siguiente valor de la secuencia.
- ALTER SEQUENCE secuencia RESTART WITH valor: Reinicia la secuencia en el valor especificado.

Las secuencias en SQL Server y PostgreSQL son herramientas poderosas para el manejo de claves primarias y otros valores únicos o consecutivos. Proveen flexibilidad en el control de valores y aseguran la integridad de los datos en aplicaciones donde los identificadores deben ser únicos y sin colisiones.

Tipos de datos autoincrementales

Tanto PostgreSQL como SQL Server ofrecen columnas autoincrementales para generar automáticamente valores únicos y consecutivos, principalmente usadas para las claves primarias. Estas columnas se definen de manera similar a un tipo de dato `INTEGER`, y "detrás de escena" se crea y maneja una secuencia de forma implícita.

En **PostgreSQL**, existen dos maneras de definir columnas autoincrementales: SERIAL: es un alias para crear una columna INTEGER que usa una secuencia automáticamente; y BIGSERIAL (similar al SERIAL, pero para un rango más grande: BIGINT).

```
CREATE TABLE producto.categoria (

    id SERIAL PRIMARY KEY,  -- Columna autoincremental

    version INTEGER NOT NULL,

    codigo INTEGER NOT NULL UNIQUE,

    descripcion VARCHAR(50) NOT NULL

);
```

En **SQL Server**, las columnas autoincrementales se definen usando la propiedad `IDENTITY`, que genera automáticamente un valor único y consecutivo.

```
CREATE TABLE producto_categoria (
    id INT IDENTITY(1,1) PRIMARY KEY,  -- Columna autoincremental
    version INT NOT NULL,
    codigo INT NOT NULL UNIQUE,
    descripcion NVARCHAR(50) NOT NULL
);

/*IDENTITY(1,1): Define la columna id como autoincremental, comenzando en 1 e
incrementándose en 1 cada vez que se inserta una nueva fila.*/
```

Manejo de transacciones

En el contexto de bases de datos relacionales, una **transacción** es una unidad de trabajo compuesta por una o varias sentencias de base de datos (generalmente INSERT, UPDATE, DELETE) que se ejecutan como una sola unidad lógica. Su objetivo principal es garantizar la **integridad** y **consistencia** de los datos, incluso en caso de errores o fallos.

Conceptos básicos

1. Deben cumplir con las propiedades **ACID**:
    - Atomicidad (*Atomicity*): La transacción debe ser "todo o nada". Si alguna operación falla, ninguna operación de la transacción debe aplicarse.
    - Consistencia (*Consistency*): Al finalizar la transacción, la base de datos debe estar en un estado válido, cumpliendo con todas las reglas de integridad.
    - Aislamiento (*Isolation*): Las operaciones de una transacción no deben interferir con otras transacciones concurrentes, lo cual asegura que los resultados sean consistentes.
    - Durabilidad (*Durability*): Una vez que una transacción se confirma (se hace COMMIT), sus cambios son permanentes, incluso en caso de fallos de sistema.
2. Ciclo de vida de una transacción:
    - Inicio de la transacción: Se inicia con un comando como BEGIN o BEGIN TRANSACTION.
    - Operaciones: Se realizan varias operaciones de consulta y/o modificación de datos.
    - Confirmación (COMMIT): Cuando todas las operaciones se completan con éxito, un COMMIT guarda los cambios.
    - Reversión (ROLLBACK): Si ocurre un error o si se desean deshacer los cambios, un ROLLBACK anula todas las operaciones realizadas en la transacción.
3. Puntos de guardado (SAVEPOINT): Los puntos de guardado o puntos de control permiten crear puntos intermedios en una transacción para realizar reversión parcial, útil en operaciones complejas sin necesidad de cancelar toda la transacción.
4. Control de concurrencia: Para asegurar el **aislamiento** en entornos con múltiples transacciones, los sistemas de bases de datos implementan mecanismos de bloqueo y control de concurrencia. Esto asegura que varias transacciones puedan ejecutarse al mismo tiempo sin afectar la consistencia de los datos.
5. Niveles de aislamiento: Estos definen cuánto se aíslan las transacciones entre sí para manejar problemas de concurrencia. Algunos niveles comunes son:
    - Read Uncommitted: Permite leer datos sin confirmar, riesgo de inconsistencias.
    - Read Committed: Solo permite leer datos confirmados, evitando lecturas sucias.
    - Repeatable Read: Garantiza que las lecturas dentro de una transacción sean consistentes.
    - Serializable: Ofrece el máximo aislamiento, evitando interferencias de otras transacciones.

Aunque las operaciones DML (INSERT, UPDATE, y DELETE) son las más comunes en una transacción, los motores de bases de datos relacionales también permiten incluir operaciones DDL (Data Definition Language) dentro de una transacción.

De todas formas, aunque es técnicamente posible incluir DDL en una transacción no siempre es la mejor práctica porque:

- **Genera bloqueos más grandes**: las operaciones DDL suelen provocar bloqueos amplios en las tablas y pueden impactar el rendimiento de la base de datos.
- **Impacto en concurrencia**: realizar cambios estructurales en la base de datos dentro de transacciones largas puede afectar negativamente a otras transacciones que dependen de esas estructuras.

*Una **transacción** es una secuencia controlada de operaciones que cumple con propiedades ACID, asegurando que los datos se mantengan seguros, consistentes y aislados, incluso en escenarios de concurrencia y fallos.*

Ejemplos

PostgreSQL

```
BEGIN;

UPDATE persona.sucursal
SET descripcion = 'Sucursal Actualizada'
WHERE codigo = 123;

UPDATE persona.cliente
SET fecha_alta = CURRENT_DATE
WHERE codigo = 456;

COMMIT;
```

Si ocurre un error, se puede usar `ROLLBACK` para revertir.

```
BEGIN;

UPDATE persona.sucursal
SET descripcion = 'Sucursal Actualizada'
WHERE codigo = 123;

UPDATE persona.cliente
SET fecha_alta = CURRENT_DATE
WHERE codigo = 456;

ROLLBACK; -- Revierte los cambios en caso de error
```

SQL Server

```
BEGIN TRANSACTION;

UPDATE persona.sucursal
SET descripcion = 'Sucursal Actualizada'
WHERE codigo = 123;

UPDATE persona.cliente
SET fecha_alta = GETDATE()
WHERE codigo = 456;

COMMIT TRANSACTION;
```

Para revertir en caso de error:

```
BEGIN TRANSACTION;

BEGIN TRY
    UPDATE persona.sucursal
    SET descripcion = 'Sucursal Actualizada'
    WHERE codigo = 123;

    UPDATE persona.cliente
    SET fecha_alta = GETDATE()
    WHERE codigo = 456;

    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT 'Error en la transacción. Se han revertido los cambios.';
END CATCH;
```

En PostgreSQL, el manejo de errores en transacciones puede lograrse utilizando bloques BEGIN y EXCEPTION, similares a los bloques TRY y CATCH de SQL Server.

Ejemplo

```
DO $$
BEGIN
    -- Iniciar la transacción
    BEGIN;

    -- Actualizar la sucursal
    UPDATE persona.sucursal
    SET descripcion = 'Sucursal Actualizada'
    WHERE codigo = 123;

    -- Actualizar el cliente
    UPDATE persona.cliente
    SET fecha_alta = CURRENT_DATE
    WHERE codigo = 456;

    -- Confirmar la transacción si todo va bien
    COMMIT;

EXCEPTION
    WHEN OTHERS THEN
        -- En caso de error, revertir la transacción
        ROLLBACK;
        RAISE NOTICE 'Error en la transacción. Se han revertido los cambios.';
END $$;
```

En este ejemplo:

- DO $$ ... $$ es un bloque anónimo de código PL/pgSQL.
- Dentro de BEGIN...EXCEPTION...END se ejecutan las operaciones de la transacción.
- Si ocurre algún error, el bloque EXCEPTION realiza el ROLLBACK y muestra un mensaje indicando el error.

Los bloques BEGIN...EXCEPTION en PostgreSQ, tanto como los TRY y CATCH de SQL Server. se pueden usar para manejar diversos tipos de errores, desde fallas en conversiones de tipos hasta problemas al ejecutar consultas o transacciones. Esto permite hacer que los scripts sean más robustos, evitando fallos inesperados y proporcionando una lógica alternativa para cuando ocurren errores.

Verificación de locks en transacciones en curso

PostgreSQL

Para ver los locks que se están generando por una transacción que está en curso, se puede usar la vista del sistema *pg_locks*. Esta vista muestra todos los locks que están actualmente activos en el sistema.

```
SELECT
    pg_stat_activity.datname,
    pg_stat_activity.pid,
    pg_stat_activity.usename,
    pg_stat_activity.query,
    pg_locks.locktype,
    pg_locks.mode,
    pg_locks.granted,
    pg_locks.relation::regclass AS locked_table
FROM
    pg_locks
JOIN
    pg_stat_activity ON pg_locks.pid = pg_stat_activity.pid
WHERE
    pg_stat_activity.state = 'active';
```

Este comando realiza las siguientes acciones:

- Selecciona información del proceso (pid), usuario (usename), consulta (query), tipo de lock (locktype), modo del lock (mode), si fue otorgado (granted) y la tabla bloqueada (locked_table).
- pg_locks contiene la información de los locks y pg_stat_activity tiene información de las actividades activas.
- Filtramos por aquellas actividades que estén en estado 'active'.