SQL Dinámico

¿Qué es el SQL Dinámico?

SQL dinámico es una técnica que permite construir y ejecutar sentencias SQL en tiempo de ejecución, generadas a partir de valores variables. A diferencia del SQL estático, donde las consultas están completamente definidas antes de ser ejecutadas, el SQL dinámico utiliza variables y cadenas para construir las consultas en función de parámetros que pueden cambiar en cada ejecución.

Se emplea para adaptar las consultas en escenarios donde las condiciones de filtrado, selección de columnas o nombres de tablas pueden variar. En general, SQL dinámico es útil en sistemas que requieren flexibilidad para construir consultas complejas y condicionales, siendo ampliamente utilizado en procedimientos almacenados, funciones y scripts de bases de datos.

Principales Fortalezas del SQL Dinámico

1. **Flexibilidad**: Permite construir consultas SQL a medida, adaptándose a parámetros de usuario o requisitos específicos en tiempo de ejecución.
2. **Automatización y escalabilidad**: Es ideal para automatizar operaciones que involucran muchas tablas, columnas o filtros variables, simplificando procesos que, de otro modo, requerirían numerosas consultas SQL estáticas.
3. **Reducción de código repetitivo**: Al centralizar lógica común en consultas dinámicas, se reduce la cantidad de código redundante y se facilita el mantenimiento.
4. **Generación de reportes complejos**: Permite generar reportes en los que las columnas y los datos de salida pueden cambiar dinámicamente en función de parámetros de tiempo o de otras condiciones de negocio.

Aplicaciones del SQL dinámico

1. **Reportes personalizados**: En escenarios de generación de reportes, donde los usuarios pueden elegir diferentes parámetros (por ejemplo, fechas, columnas, o tipos de agregación), el SQL dinámico permite generar la consulta en función de estas selecciones. Esto es especialmente útil en la transposición de datos, como en el caso de columnas dinámicas para reportes de ventas por mes o año. Veremos un ejemplo más adelante en este apunte.
2. **Consultas condicionales**: Cuando una consulta debe modificarse en función de múltiples condiciones o filtros opcionales, el SQL dinámico ayuda a crear la consulta de manera flexible, incluyendo solo las partes necesarias.
3. **Manejo de tablas y esquemas variables**: En aplicaciones con múltiples clientes o bases de datos con esquemas dinámicos, el SQL dinámico permite construir y ejecutar consultas en distintas tablas o bases de datos según el contexto.
4. **Automatización de tareas de administración**: Es muy útil en tareas de administración, como la generación automática de índices, la carga masiva de datos, la actualización de datos en múltiples tablas o la creación de columnas y tablas basadas en configuraciones externas.

Riesgos de Seguridad al Usar SQL Dinámico

1. **Inyección de SQL**: Si se concatenan valores de entrada de usuario directamente en una consulta SQL dinámica, un atacante puede inyectar código SQL malicioso. Por ejemplo, un usuario podría introducir un valor especialmente diseñado para modificar la consulta y ejecutar comandos no autorizados.
2. **Escalada de Privilegios**: Las consultas dinámicas pueden abrir oportunidades para que usuarios no autorizados accedan a datos o realicen operaciones que normalmente no se les permitirían si la consulta estática estuviera predefinida y controlada.
3. **Dificultad en la Auditoría y Detección de Vulnerabilidades**: Las consultas dinámicas pueden complicar el rastreo y auditoría de actividades en la base de datos, ya que no están completamente definidas en el código fuente y pueden variar según las entradas de los usuarios.

Recomendaciones

1. **Usar parámetros en lugar de concatenación directa**: Siempre que sea posible, se recomienda utilizar parámetros en lugar de concatenar valores de entrada directamente en la consulta SQL.
2. **Validar y limitar las entradas**: Antes de usar datos de entrada en SQL dinámico, verifica que cumplan con los requisitos esperados. Por ejemplo, si se espera un número, validar que el valor sea numérido; si se espera un nombre de columna, verificar con una lista de columnas permitidas.
3. **Evitar SQL dinámico**: SQL dinámico debe utilizarse solo cuando realmente sea necesario. En muchos casos, las consultas pueden estructurarse de otra manera para evitar la necesidad de SQL dinámico, utilizando alternativas como vistas, procedimientos almacenados o funciones predefinidas.
4. **Controlar los permisos de usuario**: Chequear que los usuarios que ejecutan consultas dinámicas tengan solo los permisos mínimos necesarios. Si un procedimiento que usa SQL dinámico puede ser ejecutado por usuarios con privilegios limitados, chequear que no puedan realizar acciones no deseadas en la base de datos.
5. **Usar procedimientos y funciones**: Encapsular la lógica de SQL dinámico en procedimientos o funciones almacenadas, donde se controlan estrictamente las entradas, permite restringir el acceso y reducir la posibilidad de errores o uso indebido.
6. **Registrar y auditar las consultas dinámicas**: mantener auditoría de las consultas dinámicas ejecutadas. Esto permite detectar patrones de uso inusual o actividades sospechosas en la base de datos.

Ejemplo de código seguro vs. inseguro

Código Inseguro (Posibilidad de alteración del SQL por injección de código)

```
-- Supongamos que @nombreTabla viene de la entrada del usuario:
DECLARE @nombreTabla VARCHAR(50) = 'venta.factura; DROP TABLE
venta.factura_detalle; --';
DECLARE @query VARCHAR(1000);
SET @query = 'SELECT * FROM ' + @nombreTabla;
EXEC(@query);  -- Esto podría ejecutar un SQL malicioso si @nombreTabla es
manipulado
```

Código Seguro (Uso de Parámetros)

```
-- En SQL Server, utiliza una lista de tablas permitidas:
DECLARE @nombreTabla NVARCHAR(50) = 'venta.factura';
IF @nombreTabla NOT IN ('venta.factura', 'venta.factura_detalle')
    THROW 50001, 'Tabla no permitida', 1;

DECLARE @query NVARCHAR(1000) = N'SELECT * FROM ' + QUOTENAME(@nombreTabla);
EXEC sp_executesql @query;
```

El SQL dinámico es una herramienta poderosa y flexible, pero su uso conlleva riesgos de seguridad que deben gestionarse con cuidado.

Este apunte presenta, mediante ejemplos, diversas situaciones en las que el uso de SQL dinámico puede ser útil y efectivo. No pretende ser una guía exhaustiva, sino un punto de partida para inspirar su aplicación en otras áreas y problemas.

EJEMPLOS DE USO

SQL Dinámico Básico

SQL Server

En SQL Server, el SQL dinámico se ejecuta mediante el comando EXEC. A continuación, se presenta un ejemplo de cómo ejecutar una consulta en la que el nombre de la tabla y la columna se definen en tiempo de ejecución.

```
DECLARE @cadena VARCHAR(100)
DECLARE @nomTabla VARCHAR(30) = 'venta.factura'
DECLARE @nomColumna VARCHAR(60) = 'id_cliente'
SET @cadena = 'SELECT ' + @nomColumna + ' FROM ' + @nomTabla
EXEC (@cadena)
```

PostgreSQL

En PostgreSQL, el SQL dinámico se ejecuta con EXECUTE. A continuación, se muestra un ejemplo de una función que realiza una consulta dinámica basada en el nombre de la tabla y columna, definidos en tiempo de ejecución.

```
CREATE OR REPLACE FUNCTION sql_dinamico(nom_tabla VARCHAR, nom_columna VARCHAR)
RETURNS SETOF BIGINT AS
$$
DECLARE
    v_cadena VARCHAR;
BEGIN
    v_cadena := 'SELECT ' || nom_columna || ' FROM ' || nom_tabla;
    RAISE NOTICE 'Ejecutando: %', v_cadena;
    RETURN QUERY EXECUTE v_cadena;
END;
$$
LANGUAGE plpgsql;

Uso:

SELECT * FROM sql_dinamico('venta.factura', 'id_cliente');
```

Transposición de filas a columnas

Por transposición de datos se entiende a el proceso de convertir información dispuesta en filas a columnas. En este ejemplo, generaremos una columna para cada mes y sumaremos la cantidad vendida de cada producto en cada mes. Esto no es *SQL Dinámico*, pero expone como introducción a otro caso más complejo.

En estos ejemplos se sumariza la cantidad vendida por cada producto por cada mes.

SQL Server

```
SELECT p.descripcion AS Producto,
    SUM(CASE WHEN MONTH(f.fecha) = 1 THEN fd.cantidad ELSE 0 END) AS Ene,
    SUM(CASE WHEN MONTH(f.fecha) = 2 THEN fd.cantidad ELSE 0 END) AS Feb,
    SUM(CASE WHEN MONTH(f.fecha) = 3 THEN fd.cantidad ELSE 0 END) AS Mar,
    SUM(CASE WHEN MONTH(f.fecha) = 4 THEN fd.cantidad ELSE 0 END) AS Abr,
    SUM(CASE WHEN MONTH(f.fecha) = 5 THEN fd.cantidad ELSE 0 END) AS May,
    SUM(CASE WHEN MONTH(f.fecha) = 6 THEN fd.cantidad ELSE 0 END) AS Jun,
    SUM(CASE WHEN MONTH(f.fecha) = 7 THEN fd.cantidad ELSE 0 END) AS Jul,
    SUM(CASE WHEN MONTH(f.fecha) = 8 THEN fd.cantidad ELSE 0 END) AS Ago,
    SUM(CASE WHEN MONTH(f.fecha) = 9 THEN fd.cantidad ELSE 0 END) AS Sep,
    SUM(CASE WHEN MONTH(f.fecha) = 10 THEN fd.cantidad ELSE 0 END) AS Oct,
    SUM(CASE WHEN MONTH(f.fecha) = 11 THEN fd.cantidad ELSE 0 END) AS Nov,
    SUM(CASE WHEN MONTH(f.fecha) = 12 THEN fd.cantidad ELSE 0 END) AS Dic
FROM venta.factura f
INNER JOIN venta.factura_detalle fd ON f.id = fd.id_factura
INNER JOIN producto.producto p ON fd.id_producto = p.id
GROUP BY p.descripcion
ORDER BY p.descripcion;
```

PostgreSQL

```
SELECT p.descripcion AS producto,
    SUM(CASE WHEN EXTRACT(MONTH FROM f.fecha) = 1 THEN fd.cantidad ELSE 0 END) AS ene,
    SUM(CASE WHEN EXTRACT(MONTH FROM f.fecha) = 2 THEN fd.cantidad ELSE 0 END) AS feb,
    SUM(CASE WHEN EXTRACT(MONTH FROM f.fecha) = 3 THEN fd.cantidad ELSE 0 END) AS mar,
    SUM(CASE WHEN EXTRACT(MONTH FROM f.fecha) = 4 THEN fd.cantidad ELSE 0 END) AS abr,
    SUM(CASE WHEN EXTRACT(MONTH FROM f.fecha) = 5 THEN fd.cantidad ELSE 0 END) AS may,
    SUM(CASE WHEN EXTRACT(MONTH FROM f.fecha) = 6 THEN fd.cantidad ELSE 0 END) AS jun,
    SUM(CASE WHEN EXTRACT(MONTH FROM f.fecha) = 7 THEN fd.cantidad ELSE 0 END) AS jul,
    SUM(CASE WHEN EXTRACT(MONTH FROM f.fecha) = 8 THEN fd.cantidad ELSE 0 END) AS ago,
    SUM(CASE WHEN EXTRACT(MONTH FROM f.fecha) = 9 THEN fd.cantidad ELSE 0 END) AS sep,
    SUM(CASE WHEN EXTRACT(MONTH FROM f.fecha) = 10 THEN fd.cantidad ELSE 0 END) AS oct,
    SUM(CASE WHEN EXTRACT(MONTH FROM f.fecha) = 11 THEN fd.cantidad ELSE 0 END) AS nov,
    SUM(CASE WHEN EXTRACT(MONTH FROM f.fecha) = 12 THEN fd.cantidad ELSE 0 END) AS dic
FROM venta.factura f
INNER JOIN venta.factura_detalle fd ON f.id = fd.id_factura
INNER JOIN producto.producto p ON fd.id_producto = p.id
GROUP BY p.descripcion
ORDER BY p.descripcion;
```

Estos scripts son de utilidad siempre que conozcamos la cantidad de columnas de la salida. En estos casos doce columnas, una por cada mes.

Generación dinámica de columnas

Ahora, si necesitamos generar una salida donde las columnas de mes son variables en función de un rango (por ejemplo desde marzo-2022 hasta junio-2023), deberemos usar SQL dinámico.

SQL Server

Modalidad batch:

```
DECLARE @fechaInicio DATE = '2022-03-01';
DECLARE @fechaFin DATE = '2023-06-01';

DECLARE @fecha DATE;
DECLARE @col VARCHAR(30);
DECLARE @sqldin VARCHAR(300);

-- tabla temporal si no existe
IF OBJECT_ID('tempdb..#ventas_por_producto_mes') IS NOT NULL DROP TABLE
#ventas_por_producto_mes;

CREATE TABLE #ventas_por_producto_mes (
    id_producto BIGINT
);

-- Insertar productos únicos en el rango de fechas
INSERT INTO #ventas_por_producto_mes (id_producto)
SELECT DISTINCT p.id
FROM venta.factura f
JOIN venta.factura_detalle fd ON f.id = fd.id_factura
JOIN producto.producto p ON fd.id_producto = p.id
WHERE f.fecha BETWEEN @fechaInicio AND @fechaFin;

-- Agregar columnas dinámicas (meses) a la tabla temporal
SET @fecha = @fechaInicio;

WHILE (@fecha <= @fechaFin)
BEGIN
    SET @col = RIGHT(CONVERT(VARCHAR(7), @fecha, 120), 7);  -- Formato
YYYY-MM
    SET @fecha = DATEADD(MONTH, 1, @fecha);

    -- Agregar la columna si no existe
    SET @sqldin = 'ALTER TABLE #ventas_por_producto_mes ADD [' + @col +
'] NUMERIC(38, 2) NULL';
    EXEC(@sqldin);
END;

-- Declarar variables para el cursor
DECLARE @sqldin2 VARCHAR(300);
DECLARE @cant NUMERIC(38,2);
DECLARE @id_producto BIGINT;
DECLARE @mes VARCHAR(7);

-- Declarar el cursor para recorrer los datos de ventas por mes y
producto
DECLARE cur CURSOR FOR
SELECT p.id AS id_producto,
       RIGHT(CONVERT(VARCHAR(7), f.fecha, 120), 7) AS mes,  -- Formato
'YYYY-MM'
```

```
       SUM(fd.cantidad) AS cantidad
FROM venta.factura f
JOIN venta.factura_detalle fd ON f.id = fd.id_factura
JOIN producto.producto p ON fd.id_producto = p.id
WHERE f.fecha BETWEEN @fechaInicio AND @fechaFin
GROUP BY p.id, RIGHT(CONVERT(VARCHAR(7), f.fecha, 120), 7)
ORDER BY p.id, RIGHT(CONVERT(VARCHAR(7), f.fecha, 120), 7);

-- Abrir el cursor y procesar cada fila
OPEN cur;
FETCH NEXT FROM cur INTO @id_producto, @mes, @cant;

WHILE @@FETCH_STATUS = 0
BEGIN
    SET @sqldin2 = 'UPDATE #ventas_por_producto_mes SET [' + @mes + '] =
' + CONVERT(VARCHAR, @cant)
                   + ' WHERE id_producto = ' + CONVERT(VARCHAR,
@id_producto);
    EXEC(@sqldin2);

    FETCH NEXT FROM cur INTO @id_producto, @mes, @cant;
END;

-- Cerrar y liberar el cursor
CLOSE cur;
DEALLOCATE cur;

               -- Select con join para obtener la descripcion del producto
SELECT p.descripcion, v.*
FROM #ventas_por_producto_mes v
JOIN producto.producto p ON v.id_producto = p.id;
```

Paso a paso:

1. Declaración de variables de fechas y asignación del rango de consulta;

2. Declaración de variables que se utilizarán para construir columnas de mes dinámicas en la tabla temporal, y para generar y ejecutar sentencias SQL dinámicas.

3. Creación de tabla temporal

4. Inserción de productos únicos:

insert into #ventas_por_producto_mes (id_producto) SELECT DISTINCT p.id ...

5. Generación dinámica de columnas de mes a través de un *while*. En cada iteración, se construye un nombre de columna en formato YYYY-MM y se hace un “alter table” para agregarlo a la tabla temporaria.

6. Cursor para calcular las ventas por producto y por mes. El cursor selecciona id_producto, mes (en formato YYYY-MM), y cantidad (sumando las cantidades vendidas) para cada producto y mes dentro del rango de fechas.

7. Dentro del cursor para cada registro, se construye y ejecuta una sentencia SQL dinámica UPDATE para actualizar la columna correspondiente al mes en #ventas_por_producto_mes con la cantidad vendida.

8. Se cierra el cursor.

9. *Select* para visualización de resultados.

A continuación, la misma funcionalidad se codifica en un procedimiento almacenado, de modo de que quede encapsulado.

```
CREATE PROCEDURE generar_reporte_ventas_por_producto_mes
    @fechaInicio DATE,
    @fechaFin DATE
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @fecha DATE;
    DECLARE @col VARCHAR(30);
    DECLARE @sqldin VARCHAR(300);

    -- Crear tabla temporal si no existe
    IF OBJECT_ID('tempdb..#ventas_por_producto_mes') IS NOT NULL DROP TABLE
#ventas_por_producto_mes;

    CREATE TABLE #ventas_por_producto_mes (
        id_producto BIGINT
    );

    -- Insertar productos únicos en el rango de fechas
    INSERT INTO #ventas_por_producto_mes (id_producto)
    SELECT DISTINCT p.id
    FROM venta.factura f
    JOIN venta.factura_detalle fd ON f.id = fd.id_factura
    JOIN producto.producto p ON fd.id_producto = p.id
    WHERE f.fecha BETWEEN @fechaInicio AND @fechaFin;

    -- Generar columnas dinámicas (meses) en la tabla temporal
    SET @fecha = @fechaInicio;

    WHILE (@fecha <= @fechaFin)
    BEGIN
        SET @col = RIGHT(CONVERT(VARCHAR(7), @fecha, 120), 7);  -- Formato
'YYYY-MM'
        SET @fecha = DATEADD(MONTH, 1, @fecha);

        -- Agregar la columna si no existe
        SET @sqldin = 'ALTER TABLE #ventas_por_producto_mes ADD [' + @col + ']
NUMERIC(38, 2) NULL';
        EXEC(@sqldin);
    END;

    -- Declarar variables para el cursor
    DECLARE @sqldin2 VARCHAR(300);
    DECLARE @cant NUMERIC(38,2);
    DECLARE @id_producto BIGINT;
    DECLARE @mes VARCHAR(7);
```

```
    -- Declarar el cursor para recorrer los datos de ventas por mes y producto
    DECLARE cur CURSOR FOR
    SELECT p.id AS id_producto,
           RIGHT(CONVERT(VARCHAR(7), f.fecha, 120), 7) AS mes,  -- Formato
'YYYY-MM'
           SUM(fd.cantidad) AS cantidad
    FROM venta.factura f
    JOIN venta.factura_detalle fd ON f.id = fd.id_factura
    JOIN producto.producto p ON fd.id_producto = p.id
    WHERE f.fecha BETWEEN @fechaInicio AND @fechaFin
    GROUP BY p.id, RIGHT(CONVERT(VARCHAR(7), f.fecha, 120), 7)
    ORDER BY p.id, RIGHT(CONVERT(VARCHAR(7), f.fecha, 120), 7);

    -- Abrir el cursor y procesar cada fila
    OPEN cur;
    FETCH NEXT FROM cur INTO @id_producto, @mes, @cant;

    WHILE @@FETCH_STATUS = 0
    BEGIN
        -- Construir y ejecutar la instrucción SQL dinámica para actualizar la
columna
        SET @sqldin2 = 'UPDATE #ventas_por_producto_mes SET [' + @mes + '] = ' +
CONVERT(VARCHAR, @cant)
                       + ' WHERE id_producto = ' + CONVERT(VARCHAR,
@id_producto);
        EXEC(@sqldin2);

        FETCH NEXT FROM cur INTO @id_producto, @mes, @cant;
    END;

    -- Cerrar y liberar el cursor
    CLOSE cur;
    DEALLOCATE cur;

    -- Devolver los resultados finales con JOIN para obtener la descripcion del
producto
    SELECT p.descripcion, v.*
    FROM #ventas_por_producto_mes v
    JOIN producto.producto p ON v.id_producto = p.id;
END;

USO:

EXEC generar_reporte_ventas_por_producto_mes '2022-03-01', '2023-06-01';
```

PostgreSQL

A continuación un ejemplo en PostgreSQL, donde la lógica se encapsula en una función. Los resultados de la función quedan registrados en una tabla que es sobre la que después se realiza la consulta.

```
CREATE OR REPLACE FUNCTION
venta.generar_reporte_ventas_por_producto_mes(fecha_inicio DATE, fecha_fin DATE)
RETURNS VOID
LANGUAGE plpgsql
AS $$
DECLARE
    fecha DATE;
    col_name VARCHAR;
    sql_alter TEXT;
    sql_update TEXT;
    id_producto BIGINT;
    mes VARCHAR(7);
    cantidad NUMERIC(38,2);
BEGIN
    -- Crear la tabla de reporte si no existe
    CREATE TABLE IF NOT EXISTS venta.reporte_ventas_por_producto_mes (
        id_producto BIGINT
    );
    -- Limpiar la tabla de reporte antes de llenarla de nuevo
    TRUNCATE TABLE venta.reporte_ventas_por_producto_mes;
    -- Insertar productos únicos en el rango de fechas
    INSERT INTO venta.reporte_ventas_por_producto_mes (id_producto)
    SELECT DISTINCT p.id
    FROM venta.factura f
    JOIN venta.factura_detalle fd ON f.id = fd.id_factura
    JOIN producto.producto p ON fd.id_producto = p.id
    WHERE f.fecha BETWEEN fecha_inicio AND fecha_fin;
    -- Generar columnas dinámicas (meses) en la tabla de reporte
    fecha := fecha_inicio;
    WHILE fecha <= fecha_fin LOOP
        col_name := TO_CHAR(fecha, 'YYYY-MM');
        sql_alter := FORMAT('ALTER TABLE venta.reporte_ventas_por_producto_mes
ADD COLUMN IF NOT EXISTS "%s" NUMERIC(38, 2);', col_name);
        EXECUTE sql_alter;
        fecha := fecha + INTERVAL '1 month';
    END LOOP;
    -- Calcular y actualizar las cantidades vendidas por cada producto y cada
mes
    FOR id_producto, mes, cantidad IN
        SELECT p.id AS id_producto,
               TO_CHAR(f.fecha, 'YYYY-MM') AS mes,
               SUM(fd.cantidad) AS cantidad
        FROM venta.factura f
        JOIN venta.factura_detalle fd ON f.id = fd.id_factura
        JOIN producto.producto p ON fd.id_producto = p.id
        WHERE f.fecha BETWEEN fecha_inicio AND fecha_fin
        GROUP BY p.id, TO_CHAR(f.fecha, 'YYYY-MM')
        ORDER BY p.id, TO_CHAR(f.fecha, 'YYYY-MM')
    LOOP
        -- Construir y ejecutar la instrucción SQL dinámica para actualizar la
columna del mes específico
```

```
        sql_update := FORMAT(
            'UPDATE venta.reporte_ventas_por_producto_mes SET "%s" = %L WHERE
id_producto = %L;',
            mes, cantidad, id_producto
        );
        EXECUTE sql_update;
    END LOOP;
END $$;

USO:
SELECT venta.generar_reporte_ventas_por_producto_mes('2022-09-01', '2023-03-
01');

-- Consulta los resultados con JOIN para obtener la descripcion del producto
SELECT p.descripcion, r.*
FROM venta.reporte_ventas_por_producto_mes r
JOIN producto.producto p ON r.id_producto = p.id;
```

Paso a paso:

La función generar_reporte_ventas_por_producto_mes se encarga de generar un **reporte de ventas por producto, desglosado por mes**, en un rango de fechas especificado por el usuario.

1. **Tabla de reporte permanente**: la función utiliza una tabla de reporte permanente llamada reporte_ventas_por_producto_mes, donde almacena los resultados. Esta tabla se limpia al inicio de cada ejecución.
2. **Inserción de productos**: se insertan en la tabla de reporte cada producto que tiene ventas dentro del rango de fechas (fecha_inicio y fecha_fin), garantizando que solo los productos relevantes sean incluidos en el reporte.
3. **Generación de columnas dinámicas para meses**: para cada mes dentro del rango, la función agrega una columna dinámica en la tabla de reporte, nombrada en formato YYYY-MM (por ejemplo, 2022-09, 2022-10, etc.). Esto permite que cada mes tenga su propia columna en el reporte.
4. **Actualización de cantidades vendidas**: se calcula la cantidad total vendida de cada producto en cada mes y se actualizan las columnas correspondientes en la tabla de reporte con estos valores.

Consulta de los resultados

Una vez ejecutada, la tabla reporte_ventas_por_producto_mes contiene el id_producto y las ventas mensuales en columnas de mes. Para ver los resultados con la descripción del producto, se realiza una consulta haciendo join con la tabla producto.

Resumen:

El enfoque presentado para ambos motores de bases de datos, permite una **estructura de reporte flexible y dinámica**, ideal para reportes que incluyen columnas de salida variables (en este caso, meses según el rango de fechas especificado).

Importación de datos desde archivos

A veces se necesita cierta “flexibilidad” para importar datos a la base de datos, y para eso es de utilidad utilizar SQL dinámico.

A continuación un ejemplo de función en PostgreSQL para importar marcas. La ubicación del archivo es variable; el separador también es un argumento de la función.

```
CREATE OR REPLACE FUNCTION importar_marcas_desde_csv(ruta_archivo TEXT,
    separador CHAR DEFAULT ',')
RETURNS TABLE (total_registros INT, marcas_nuevas INT) AS $$
DECLARE
    registro RECORD;
    marcas_insertadas INT := 0;
    sql_copy TEXT;
    max_codigo INT;
BEGIN
    -- Crear una tabla temporal con varias columnas de tipo TEXT
    CREATE TEMP TABLE temp_marcas_csv (col1 TEXT, col2 TEXT, col3 TEXT, col4
TEXT);
    -- Construir y ejecutar la sentencia COPY para cargar los datos en la tabla
temporal
    sql_copy := FORMAT(
        'COPY temp_marcas_csv FROM %L WITH (FORMAT csv, DELIMITER %L, HEADER)',
        ruta_archivo,
        separador
    );
    EXECUTE sql_copy;
    -- Crear otra tabla temporal con solo la columna 'marca' que necesitamos
    CREATE TEMP TABLE temp_marcas AS
    SELECT col4 AS marca FROM temp_marcas_csv;

    -- Obtener el valor máximo actual de 'codigo' en la tabla de marcas
    SELECT COALESCE(MAX(codigo), 0) INTO max_codigo FROM producto.marca;

    -- Recorrer cada registro e insertar en la tabla de marcas si no existe
    FOR registro IN SELECT DISTINCT marca FROM temp_marcas LOOP
```

```
        -- Verificar si la marca ya existe
        IF registro.marca IS NOT NULL AND NOT EXISTS (
            SELECT 1 FROM producto.marca WHERE descripcion = registro.marca
        ) THEN
            -- Insertar la nueva marca con un código correlativo y un id usando
la secuencia
            INSERT INTO producto.marca (id, version, codigo, descripcion)
            VALUES (nextval('producto.producto_sequence'),1,
                max_codigo + 1,registro.marca);
            marcas_insertadas := marcas_insertadas + 1;
            max_codigo := max_codigo + 1;
        END IF;
    END LOOP;
    -- Conteo de registros procesados y marcas nuevas insertadas
    RETURN QUERY SELECT
        CAST((SELECT COUNT(*) FROM temp_marcas) AS INT) AS total_registros,
        CAST(marcas_insertadas AS INT) AS marcas_nuevas;

    -- Limpiar tablas temporales
    DROP TABLE IF EXISTS temp_marcas_csv;
    DROP TABLE IF EXISTS temp_marcas;
END;
$$ LANGUAGE plpgsql;
```

Paso a paso:

- Se crea una tabla temporal (temp_marcas_csv) con cuatro columnas de tipo TEXT para cargar todas las columnas del archivo CSV, sin importar cuáles sean.
- Carga de Datos con SQL Dinámico: Se arma un COPY dinámico usando el separador de columnas especificado y carga los datos del archivo CSV en temp_marcas_csv.
- Filtrado de la Columna marca: se crea una segunda tabla temporal, temp_marcas, que solo selecciona la cuarta columna (col4), que se asume es la columna marca.
- Cálculo de código máximo sctual: se calcula el valor máximo actual de codigo en la tabla producto.marca para generar códigos correlativos para las nuevas marcas.
- Para cada marca en temp_marcas, se verifica si ya existe en producto.marca. Si no existe:
    - Se inserta con un id generado por la secuencia producto.producto_sequence.
    - Se le asigna un codigo correlativo y se incrementa el contador marcas_insertadas.
- Retorno de Resultados: se devuelve el número total de registros procesados y las nuevas marcas insertadas.

En resumen, una función de importación de datos que usa SQL dinámico es más adaptable, fácil de mantener y versátil frente a archivos con diferentes formatos, delimitadores y estructuras de datos. Esto permite procesar datos de manera eficiente y confiable en entornos donde los formatos de los archivos de entrada pueden variar considerablemente.

Tareas de administración de bases de datos

A continuación se presenta un ejemplo de función codificada en PostgreSQL para contar la cantidad de filas que tienen todas las tablas de una base de datos.

Es una función relativamente simple, cuya finalidad es ofrecer una vista rápida y detallada del estado y tamaño de las tablas en la base de datos, lo cual es importante para la gestión, optimización y monitoreo de los datos.

```
CREATE OR REPLACE FUNCTION contar_filas_tablas()
RETURNS TABLE (esquema_nombre TEXT, tabla_nombre TEXT, filas BIGINT) AS $$
DECLARE
    registro RECORD;
    sql_query TEXT;
BEGIN
    -- Recorrer todas las tablas de todos los esquemas de usuario en la base de
datos
    FOR registro IN
        SELECT n.nspname AS esquema_nombre, c.relname AS tabla_nombre
        FROM pg_class c
        JOIN pg_namespace n ON c.relnamespace = n.oid
        WHERE c.relkind = 'r'         -- Solo tablas (relkind = 'r')
          AND n.nspname NOT IN ('pg_catalog', 'information_schema') -- Excluir
esquemas del sistema
    LOOP
        -- Construir una consulta dinámica para contar filas en cada tabla de
cada esquema
        sql_query := FORMAT('SELECT %L AS esquema_nombre, %L AS tabla_nombre,
COUNT(*) AS filas FROM %I.%I',
                            registro.esquema_nombre, registro.tabla_nombre,
                            registro.esquema_nombre, registro.tabla_nombre);

        -- Ejecutar la consulta y devolver el nombre del esquema, nombre de la
tabla y el conteo de filas
        RETURN QUERY EXECUTE sql_query;
    END LOOP;
END;
$$ LANGUAGE plpgsql;

USO:

SELECT * FROM contar_filas_tablas() order by 1,2;
```

Paso a paso:

- La función contar_filas_tablas se declara con un retorno de tipo TABLE, con tres columnas: esquema_nombre, tabla_nombre y filas.
- La función recorre todas las tablas en la base de datos utilizando un bucle FOR.
- La consulta inicial selecciona el nombre de cada esquema (n.nspname como esquema_nombre) y el nombre de cada tabla (c.relname como tabla_nombre).
- Solo se incluyen las tablas de usuario (c.relkind = 'r') y se excluyen las tablas en los esquemas del sistema (pg_catalog y information_schema).

- Para cada tabla encontrada, se construye una consulta dinámica (sql_query) que cuenta las filas de esa tabla específica.
- La consulta se genera con FORMAT, que usa %L para insertar literales de texto (esquema_nombre y tabla_nombre) y %I para tratar esquema_nombre y tabla_nombre como identificadores seguros en la consulta COUNT(*).
- La función ejecuta la consulta dinámica para cada tabla utilizando RETURN QUERY EXECUTE.
- Cada ejecución de RETURN QUERY EXECUTE sql_query devuelve un resultado con el nombre del esquema, el nombre de la tabla, y el conteo de filas para esa tabla en particular.
- Una vez procesadas todas las tablas, la función termina, habiendo devuelto los nombres de esquema, nombres de tablas y conteo de filas para cada tabla en los esquemas de usuario de la base de datos.