Bases de Datos Avanzadas
Año 2024

GUIA PRACTICA – BD ACTIVAS

Posibles Escenarios de Soluciones.

Utilizaremos como motor de BD PostGreSQL.

1) Realizar un Stored Procedure/Función denominado finiciales que reciba como parámetro un varchar y devuelva las iniciales. Ejemplo si se aplica a "Maradona Diego Armando" debe devolver "MDA".

```
CREATE OR REPLACE FUNCTION finiciales(nombre VARCHAR)
RETURNS VARCHAR AS $$
DECLARE
   iniciales VARCHAR := '';
BEGIN
   -- Dividir el nombre en palabras utilizando un espacio como delimitador
   FOR word IN SELECT regexp_split_to_table(nombre, ' ') LOOP
      -- Agregar la primera letra de cada palabra a las iniciales
      iniciales := iniciales || upper(SUBSTRING(word FROM 1 FOR 1));
   END LOOP;

   RETURN iniciales;
END;
$$ LANGUAGE plpgsql;
```

Esta función, llamada finiciales, recibe un parámetro nombre de tipo VARCHAR y devuelve las iniciales como una cadena de texto.

```
SELECT finiciales('Maradona Diego Armando');
```

2) Realizar un Procedimiento que reciba un Cuit y devuelva un atributo que determine si es válido o no.

```
werror = "".
IF LENGTH(cuit_a_validar) <> 11
THEN werror = "Cuit con Menos de 11 Digitos".
ELSE DO: wcontrol = 0.
FOR wi = 1 TO 10 DO
BEGIN
wcontrol = wcontrol +
INTEGER(SUBSTRING(cuit_a_validar,wi,1)) *
INTEGER(SUBSTRING("5432765432",wi,1)).
END.
```

```
IF (( 11 - (wcontrol MODULO 11)) MODULO 11 ) <>
INTEGER(SUBSTRING(cuit_a_validar,11,1))
THEN werror = "Digito Invalido".
END.
IF werror = “” THEN msg “El cuit es valido”
```

En PostgreSQL, realiza la validación de un CUIT (Clave Única de Identificación Tributaria) y devuelve un mensaje indicando si es válido o no:

```
CREATE OR REPLACE PROCEDURE validar_cuit(cuit_a_validar VARCHAR)
LANGUAGE plpgsql
AS $$
DECLARE
   werror VARCHAR := '';
   wcontrol INT := 0;
   wi INT;
BEGIN
   IF LENGTH(cuit_a_validar) <> 11 THEN
      werror := 'Cuit con Menos de 11 Digitos';
   ELSE
      FOR wi IN 1..10 LOOP
         wcontrol := wcontrol + CAST(SUBSTRING(cuit_a_validar, wi, 1) AS INT) *
                  CAST(SUBSTRING('5432765432' FROM wi FOR 1) AS INT);
      END LOOP;

      IF ((11 - (wcontrol % 11)) % 11) <> CAST(SUBSTRING(cuit_a_validar, 11, 1) AS INT) THEN
         werror := 'Digito Invalido';
      END IF;
   END IF;

   IF werror = '' THEN
      RAISE NOTICE 'El CUIT es válido';
   ELSE
      RAISE NOTICE 'Error: %', werror;
   END IF;
END;
$$;
```

Para utilizarlo, pasar el CUIT como argumento:

```
CALL validar_cuit('20369459145');
```

3) Agregue una nueva tabla denominada DEPARTAMENTOS que contendrá los departamentos de la empresa con los siguientes datos:

ID_DPTO.	NOMBRE	ID_DPTO_PADRE
01	GERENCIA GRAL.	NULL.
10	GERENCIA ADMINISTRATIVA	01

15	GERENCIA COMERCIAL	01
30	TESORERIA	10
40	CONTADURIA	10
50	VENTAS	15
60	AREA NUEVOS PRODUCTOS	15
80	VENTAS NACIONALES	50
90	VENTAS INTERNACIONALES	50

Genere un Stored Procedure que reciba como parámetro un ID_DPTO y genere una fila por cada departamento dependiente de este con las columnas IID_DPTO, NOMBRE y DEPENDE

```
-- Crear la tabla DEPARTAMENTOS
CREATE TABLE departamentos (
   id smallint PRIMARY KEY,
   nombre varchar(40),
   id_dpto_padre smallint
);

-- Insertar datos en la tabla DEPARTAMENTOS
INSERT INTO departamentos (id, nombre, id_dpto_padre) VALUES
(01, 'GERENCIA GRAL.', NULL),
(10, 'GERENCIA ADMINISTRATIVA', 01),
(15, 'GERENCIA COMERCIAL', 01),
(30, 'TESORERIA', 10),
(40, 'CONTADURIA', 10),
(50, 'VENTAS', 15),
(60, 'AREA NUEVOS PRODUCTOS', 15),
(80, 'VENTAS NACIONALES', 50),
(90, 'VENTAS INTERNACIONALES', 50);

-- Crear el procedimiento almacenado
CREATE OR REPLACE FUNCTION obtener_departamentos_dependientes(id_dpto_param smallint)
RETURNS TABLE(iid_dpto smallint, nombre varchar(40), depende smallint) AS
$$
BEGIN
   RETURN QUERY WITH RECURSIVE DepartamentosDependientes AS (
      SELECT id, nombre, id_dpto_padre
      FROM departamentos
      WHERE id = id_dpto_param
      UNION ALL
      SELECT d.id, d.nombre, d.id_dpto_padre
      FROM departamentos d
      INNER JOIN DepartamentosDependientes dd ON d.id_dpto_padre = dd.id
   )
   SELECT * FROM DepartamentosDependientes;
END;
$$
LANGUAGE plpgsql;
```

Este procedimiento almacenado obtener_departamentos_dependientes acepta un parámetro id_dpto_param que representa el ID del departamento para obtener los departamentos dependientes.
Utilizamos una consulta recursiva para recorrer la jerarquía de departamentos y obtener todos los departamentos dependientes del departamento especificado. Luego, devuelve una tabla con los ID, nombres y el ID del departamento del cual dependen los departamentos encontrados.

4) Realizar un Stored Procedure que reciba como parámetros una tabla, un atributo y devuelva a que tipos de datos base pertenece y si es numérico el dominio del tipo base independientemente de las restricciones impuestas.

Recibe como parámetros una tabla y un atributo, y devuelve los tipos de datos base a los que pertenece ese atributo, así como si el dominio del tipo base es numérico:

```
CREATE OR REPLACE PROCEDURE verificar_tipo_dato(
   IN tabla_name VARCHAR,
   IN attr_name VARCHAR
)
LANGUAGE plpgsql
AS $$
DECLARE
   data_type VARCHAR;
   is_numeric BOOLEAN := false;
BEGIN
   -- Obtenemos el tipo de datos base del atributo
   SELECT data_type INTO data_type
   FROM information_schema.columns
   WHERE table_name = tabla_name AND column_name = attr_name;

   -- Verificamos si el dominio del tipo base es numérico
   IF data_type = 'integer' OR data_type = 'bigint' OR data_type = 'smallint' OR
      data_type = 'numeric' OR data_type = 'real' OR data_type = 'double precision' THEN
       is_numeric := true;
   END IF;

   -- Mostramos los resultados
   RAISE NOTICE 'El atributo % de la tabla % es de tipo base %.', attr_name, tabla_name,
data_type;
   RAISE NOTICE '¿El dominio del tipo base es numérico? %', is_numeric;
END;
$$;
```

Para ejecutarlo:

```
CALL verificar_tipo_dato('mi_tabla', 'mi_atributo');
```

5) Genere un Stored Procedure que retorne una fila por cada tabla con 2 atributos, nombre de tabla y cantidad de filas.
El Procedimiento debe ser totalmente dinámico y no depender de su modificación para realizar el agregado o eliminación de tablas nuevas o borradas.

Genera dinámicamente una fila por cada tabla en la base de datos, con dos atributos: el nombre de la tabla y la cantidad de filas que contiene:

```
CREATE OR REPLACE PROCEDURE contar_filas_tablas()
LANGUAGE plpgsql
AS $$
DECLARE
   tabla_record RECORD;
   tabla_name VARCHAR;
   fila_count INT;
BEGIN
   -- Creamos una tabla temporal para almacenar los resultados
   CREATE TEMP TABLE temp_table_counts (table_name VARCHAR, row_count INT);

   -- Obtenemos todas las tablas en el esquema público
   FOR tabla_record IN SELECT table_name FROM information_schema.tables WHERE
table_schema = 'public' AND table_type = 'BASE TABLE' LOOP
      tabla_name := tabla_record.table_name;

      -- Contamos las filas de cada tabla
      EXECUTE format('SELECT COUNT(*) FROM %I', tabla_name) INTO fila_count;

      -- Insertamos el nombre de la tabla y la cantidad de filas en la tabla temporal
      INSERT INTO temp_table_counts VALUES (tabla_name, fila_count);
   END LOOP;

   -- Devolvemos los resultados
   SELECT * FROM temp_table_counts;

   -- Eliminamos la tabla temporal
   DROP TABLE IF EXISTS temp_table_counts;
END;
$$;
```

Este procedimiento crea dinámicamente una tabla temporal llamada temp_table_counts donde almacena el nombre de la tabla y la cantidad de filas para cada tabla en el esquema público. Luego, devuelve todos los registros de esa tabla temporal, que contienen el nombre de la tabla y la cantidad de filas.

```
CALL contar_filas_tablas();
```

Devolverá una fila por cada tabla en el esquema público con dos atributos: el nombre de la tabla y la cantidad de filas.

Otra opción,

```
CREATE OR REPLACE FUNCTION obtener_cantidad_filas()
RETURNS TABLE (tabla_name TEXT, cantidad_filas BIGINT) AS
$$
DECLARE
   tabla RECORD;
   query TEXT;
BEGIN
   FOR tabla IN SELECT table_name FROM information_schema.tables WHERE table_schema =
'public' AND table_type = 'BASE TABLE' LOOP
      query := format('SELECT %L AS tabla_name, COUNT(*) AS cantidad_filas FROM %I.%I',
tabla.table_name, 'public', tabla.table_name);
      EXECUTE query INTO tabla_name, cantidad_filas;
      RETURN NEXT;
   END LOOP;
END;
$$
LANGUAGE plpgsql;


SELECT * FROM obtener_cantidad_filas();
```

6) Agregue el atributo cantidad_items a la tabla Facturas y genere un trigger que mantenga la redundancia respecto de los items que posee cada factura.

```
ALTER TABLE Facturas
ADD COLUMN cantidad_items INTEGER DEFAULT 0;
```

Creación del Trigger

```
CREATE OR REPLACE FUNCTION actualizar_cantidad_items()
RETURNS TRIGGER AS
$$
BEGIN
   IF TG_OP = 'INSERT' THEN
      UPDATE Facturas
      SET cantidad_items = cantidad_items + 1
      WHERE id_factura = NEW.id_factura;
   ELSIF TG_OP = 'DELETE' THEN
```

```
    UPDATE Facturas
    SET cantidad_items = cantidad_items - 1
    WHERE id_factura = OLD.id_factura;
  END IF;

  RETURN NULL;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER actualizacion_cantidad_items_trigger
AFTER INSERT OR DELETE ON Items
FOR EACH ROW
EXECUTE FUNCTION actualizar_cantidad_items();
```

Este trigger se activará después de que se inserte o elimine un registro en la tabla Items.

Dependiendo de la operación (inserción o eliminación), actualizará la columna cantidad_items en la tabla Facturas.

Con esto, la redundancia respecto a la cantidad de items en cada factura se mantendrá automáticamente actualizada en la columna cantidad_items de la tabla Factura

7) Genere un trigger que no permita ingresar Productos de Familia A y C simultáneamente en la misma factura.

```
CREATE OR REPLACE FUNCTION validar_familia_productos()
RETURNS TRIGGER AS
$$
DECLARE
  familia_producto TEXT;
BEGIN
  -- Obtener la familia del producto que se intenta insertar
  SELECT p.familia INTO familia_producto
  FROM Productos p
  WHERE p.codigo = NEW.producto_codigo;

  -- Verificar si la familia del producto es A o C
  IF familia_producto IN ('A', 'C') THEN
    -- Verificar si hay productos de otras familias en la misma factura
    IF EXISTS (
      SELECT 1
      FROM Facturas_Items fi
      JOIN Productos p ON fi.producto_codigo = p.codigo
      WHERE fi.factura_tipo = NEW.factura_tipo
      AND fi.factura_numero = NEW.factura_numero
```

```
        AND p.familia IN ('A', 'C')
        AND fi.producto_codigo <> NEW.producto_codigo
      ) THEN
        RAISE EXCEPTION 'No se pueden ingresar productos de Familia A y C simultáneamente en la misma factura';
      END IF;
  END IF;

  RETURN NEW;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER validar_familia_productos_trigger
BEFORE INSERT ON Facturas_Items
FOR EACH ROW
EXECUTE FUNCTION validar_familia_productos();
```

Este trigger se activará antes de que se inserte un registro en la tabla Facturas_Items.

Si encuentra algún producto de las familias A o C en la misma factura, lanza una excepción y aborta la operación de inserción.

8) Genere un trigger que convierta el nombre de clientes a mayúsculas.

Trigger BEFORE INSERT y BEFORE UPDATE.

```
CREATE OR REPLACE FUNCTION convertir_nombre_a_mayusculas()
RETURNS TRIGGER AS
$$
BEGIN
  NEW.nombre := UPPER(NEW.nombre); -- Convertir el nombre a mayúsculas
  RETURN NEW;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER convertir_nombre_a_mayusculas_trigger
BEFORE INSERT OR UPDATE ON Clientes
FOR EACH ROW
EXECUTE FUNCTION convertir_nombre_a_mayusculas();
```

Se activará antes de que se inserte o actualice un registro en la tabla Clientes. Convertirá el valor del campo nombre a mayúsculas utilizando la función UPPER () de PostgreSQL y asignará el valor convertido de nuevo a NEW.nombre. Luego, retornará el nuevo registro.