INGENIERÍA UNIVERSIDAD DE LOS ANDES MÉRIDA VENEZUELA | DEPARTAMENTO DE COMPUTACIÓN ESCUELA DE SISTEMAS

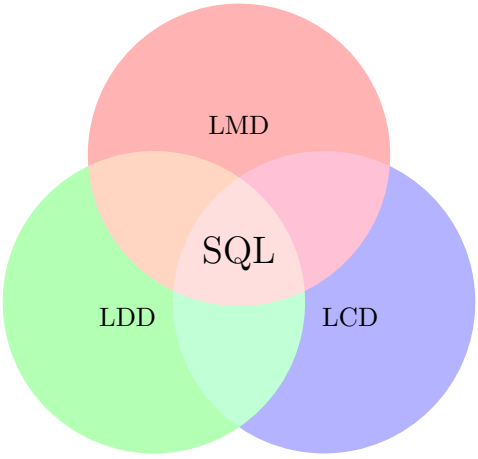
Laboratorio de Base de Datos
Práctica Nro. 7, Server Programming

Prof. Solazver Solé
Preps. Victor Albornoz, Yenifer Ramirez
Semestre B-2018

1. SQL

SQL por sus siglas en inglés (**Structured Query Languaje**) [1] es un lenguaje de dominio específico, usado en la programación y diseñado para administrar los datos persistentes en los sistemas de gestión de bases de datos relacionales (**SGBDR**).

Originalmente se basó en el **álgebra relacional** y en **cálculo relacional de tuplas**, SQL es tanto un lenguaje de manipulación de datos (**LMD**), como un lenguaje de definición de datos (**LDD**) y un lenguaje de control de datos (**LCD**).



[1] Lenguaje estructurado de consultas

2. Procedimientos almacenados

- Los lenguajes permitidos en Postgres para escribir funciones definidas por el usuario son generalmente llamados lenguajes procedurales "Procedural Languajes".
- Postgres soporta diferentes tipos de lenguajes procedurales como:
  - PL/pgSQL ->Everything we want!
  - PL/Tcl ->Codigo C.
  - PL/Perl ->Util manejo de strings.
  - PL/Python ->Util manejo de strings.
- Un procedimiento almacenado se puede definir como un programa, procedimiento o funcion, el cual esta almacenado en la base de datos y listo para ser usado.
- Los procedimientos almacenados van de la mano con los disparadores "**Triggers**", ofrecen estructuras de control, permiten realizar cómputos complejos desde el servidor y son fáciles de implementar.
- En general, cada parámetro debe tener un tipo de parámetro; uno de los tipos de datos de SQL. También debe tener un modo de parámetro, que es IN, OUT, o INOUT. Estos modos se corresponden con los parámetros cuyos valores son de sólo entrada, sólo salida, o de entrada y salida.
- Si el procedimiento (o función) se escribe en un lenguaje de programación de propósito general, es típico especificar el lenguaje.
- Ventajas:
  - Reduce el numero de peticiones del clientes al servidor Round trips".
  - Los resultados intermedios, no relevantes para el cliente no son transferidos al cliente.
  - Reduce el numero de consultas por parte del cliente.

La forma general para declarar un procedimiento almacenado es la siguiente:

```
1 CREATE PROCEDURE <nombre del procedimiento> ([IN | OUT |
    INOUT]<lista de parametros>)
2 [DECLARE] <lista de variables y tipos> AS $$
3 BEGIN
4   <cuerpo del procedimiento>
5 END;
6 $$ [LANGUAJE]  <nombre del lenguaje de programacion>
```

La forma general para declarar una funcion es la siguiente:

```
CREATE FUNCTION <nombre de la funcion> ([IN | OUT | INOUT]<
    parametros>)
RETURNS <tipo de devolucion>
<declaraciones locales>  AS
BEGIN
  <cuerpo de la funcion> ;
  RETURN;
END;
$$ [LANGUAJE]  <nombre del lenguaje de programacion>
```

Los procedimientos almacenados se ejecutan simplemente escribiendo su nombre en una instruccion SELECT. Si se trata de una funcion se llamara con la palabra CALL, de la siguiente forma:

```
CALL <nombre del procedimiento o funcion> (<lista de
    argumentos>) ;
```

2.1. Ejemplos de procedimientos almacenados

- Procedimiento sencillo con una consulta SELECT.

```
CREATE PROCEDURRE sp_obtener_clientes
AS $$
BEGIN
  SELECT * FROM clientes;
END;


CALL sp_obtener_clientes;
```

- PL/pgsql cuenta con la posibilidad de manejar estructuras de control, por ejemplo con la siguiente funcion que devuelve un valor segun el numero de empleados del departamento.

```
CREATE FUNCTION TamDpto(IN nodpto INTEGER)
RETURNS VARCHAR [7]
DECLARE NumEmps INTEGER ;
AS $$
BEGIN
SELECT COUNT(*) INTO NumEmps FROM EMPLEADO WHERE Dno =
    nodpto;

IF NumEmps > 100 THEN RETURN "MUY GRANDE"
ELSEIF NumEmps > 25 THEN RETURN "GRANDE"
ELSEIF NumEmps > 10 THEN RETURN "MEDIO"
ELSE RETURN "MINIMO"
END IF ;

END;
$$ LANGUAGE plpgsql;

CALL TamDpto(12);
```

- Procedimiento que determina el precio neto de un producto, es decir, precio bruto+iva.

```
CREATE FUNCTION PrecioIVA(precio_bruto real) RETURNS
    REAL AS $$
BEGIN
  RETURN precio_bruto + precio_bruto*0.12;
END;
$$ LANGUAGE plpgsql;

--select * from PrecioIVA (100); //120
```

- Podemos expresar la salida de la función en los parámetros, en este ejemplo se muestra una función que recibe el precio neto de un producto, el porcentaje de descuento y tiene como salida el precio con descuento.

```
CREATE FUNCTION precio_des
(precio real ,
porcentaje_desc real,
out preciofinal real) AS $$

BEGIN
  preciofinal = precio - precio * porcentaje_desc;
END;
$$ LANGUAGE plpgsql;
```

- Podemos utilizar el procedimiento anterior de la siguiente manera:

```
  select producto.nombre_producto,precio_bruto,
      PrecioIVA(producto.precio_bruto),preciofinal
  from producto,precio_des(PrecioIVA(producto.
      precio_bruto),0.15);
```

- Es posible consultar otras tablas para lograr retornar que necesite información de la base de datos.

```
CREATE FUNCTION precio_descuento_cliente(total real,
    cedula VARCHAR(80)) RETURNS real AS $$
DECLARE descuento_cliente real;
BEGIN
select cliente.descuento into descuento_cliente
from cliente
where cliente.cedula = precio_descuento_cliente.cedula;

return total - total * descuento_cliente;
END;
$$ LANGUAGE plpgsql;


--select * from precio_descuento_cliente(100,'1'); //98
```

- El siguiente ejemplo realiza una consulta y evalúa diferentes casos correspondientes a los productos que ha comprado un cliente.

```
CREATE FUNCTION compra_cliente(cedula VARCHAR(80))
    RETURNS TEXT AS $$
DECLARE num_compras integer;
DECLARE message text;
BEGIN
select COUNT(*) into num_compras
from cliente join compraproducto using (cedula) join
    producto using (id_producto)
where cliente.cedula = compra_cliente.cedula;

CASE
WHEN num_compras > 1 THEN
message:= 'Has comprado muy poco';

WHEN num_compras > 10 THEN
message:= 'Has comprado poco';

WHEN num_compras > 100 THEN
message:= 'Eres un buen cliente';

```

```
19 WHEN num_compras > 1000 THEN
20 message:= 'Conoces todos nuestros productos';
21
22 END CASE;
23 RETURN message;
24 END;
25 $$ LANGUAGE plpgsql;
26
27 --select * from compra_cliente('1');
```

3. Triggers:

- Los disparadores (Triggers) son funciones que son invocadas o ejecutadas cuando ocurre un evento especifico en la base de datos.
- Un evento que es la causa de una particular acción, proceso or situación.
- Un disparador es una acción definida en una tabla de nuestra base de datos y ejecutada automáticamente por una función definida por el usuario (**UDF**). Esta acción se activará, según la definamos, cuando realicemos un **INSERT**, un **UPDATE** ó un **DELETE** en la susodicha tabla.

3.1. Tipos de Triggers:

- Los disparadores pueden ser clasificados de dos maneras:

  Cuando son lanzados.

  Donde son lanzados.

3.1.1. Disparadores según el momento de ser lanzados

- **BEFORE TRIGGER**: Un disparador puede ser especificado para ser lanzado antes de que se intente una operación en una fila o después que la operación sea completada. Este tipo de disparadores es lanzado antes de que las restricciones sean verificadas y antes de que operaciones como **insert**, **update**, **delete** sean completadas.
- **AFTER TRIGGER**: Si el disparador es lanzando después del evento, todos los cambios, incluidos el efecto de otros disparadores, son visibles para este tipo de disparador.
- **INSTEAD OF TRIGGER**: Si el disparador(Trigger) se ejecuta antes o en lugar del evento, el disparador puede omitir la operación para la actual fila o cambiar la fila cuando es insertada. Esto significa que el disparador no puede ejecutar el evento original y hacer otra tarea. –VERIFICAR

3.1.2. Disparadores según donde son lanzados

- **FOR EACH ROW**: Un disparador que es configurado para cada fila (for each row) es llamado una vez por cada fila que la operación modifica. Por ejemplo: Una operación DELETE que afecte 10 filas sobre una tabla, costara la ejecución de cualquier disparador que reaccione al evento (DELETE) que este sobre esa tabla 10 veces, una por cada fila afectada.
- **FOR EACH STATEMENT**: A diferencia del disparador "for each row", un disparador que es configurado para cada sentencia (for each statement) solo se ejecuta una vez para cualquier operación. Por ejemplo una operación 'X' que afecte 0 filas sobre una tabla que reaccione a un evento 'x', ejecutara una vez el disparador.

3.1.3. Tipos de disparadores:

WHEN	EVENT	ROW-LEVEL	STATEMENT-LEVEL
BEFORE	insert/update/delete	tablas	tablas y vistas
BEFORE	trucate	-	tablas
AFTER	insert/update/delete	tablas	tablas y vistas
AFTER	trucate	-	tablas
INSTEAD OF	insert/update/delete/trucate	vistas	-
INSTEAD OF	trucate	-	-

3.2. Notas sobre Disparadores:

- Las acciones de los disparadores deben estar definidas a través de procedimientos almacenados e instaladas antes de definir el disparador.
- En postgres los disparadores solo pueden ejecutar procedimientos almacenados.
- No podemos crear un disparador para un evento "**SELECT**".
- Es posible crear multiples disparadores en tablas o vistas y si el orden de ejecución sera en orden alfabético según el nombre.
- Un procedimiento que se vaya a utilizar por un disparador no puede tener argumentos y tiene que devolver tipo "trigger".
- Un mismo procedimiento almacenado se puede utilizar por múltiples disparadores en diferentes tablas.

3.3. Como Crear Disparadores:

Al momento de crear un disparador se debe tomar en cuenta el modelo que se ha utilizado para especificar las reglas de las bases de datos activa, conocido como evento-condicion-accion, o modelo ECA. Una regla en el modelo ECA tiene tres componentes:

- El evento son las modificaciones a la base de datos que "lanzan.[el] disparador, o que ejecutan la accion.
- Una condicion occional para ejecutar la accion.
- La accion representa lo que hace el disparador. la accion es nonnalmente una secuencia de sentencias SQL, pero tambien puede ser una transaccion de base de datos o un programa externo que se ejecutara automaticamente.

3.4. Extructura de un disparador:

```
1
2  <trigger> ::=   CREATE TRIGGER <nombre del trigger>
3        ( AFTER | BEFORE ) <eventos de activacion>  ON <nombre
             de la tabla>
4        [ FOR EACH ROW]
5        [ WHEN <condicion>]
6        <acciones del trigger> ;
7
8
9
10 <eventos de activacion>   ::= <evento trigger> {OR <evento
       trigger> }
11 <evento trigger>    ::= INSERT | DELETE | UPDATE [OF <nombre
       columna> { , <nombre columna> } ]
12 <accion trigger>    :: = <bloque PL/SQL>
```

```
1
2  CREATE TRIGGER check_update
3  BEFORE UPDATE ON accounts
4  FOR EACH ROW
5  EXECUTE PROCEDURE check_account_update();
6
7
8  TODO:
9  Hacer ejemplo de auditoria de bases de datos (After Trigger)
       (NEW,OLD,TG_OP)
```

3.5. Ejemplo de before trigger

- El siguiente disparador verfica el precio del producto antes de ser insertado en la tabla PRODUCTO.

```
CREATE FUNCTION check_product() RETURNS trigger AS
    $check_product$
BEGIN
  -- Verifica que el precio_bruto de un producto sea
      mayor a cero.
  IF NEW.precio_bruto < 0 THEN
  RAISE EXCEPTION '% No puede tener un precio_bruto
      negativo', NEW.nombre_producto;
  END IF;

  RETURN NEW;
END;
$check_product$ LANGUAGE plpgsql;

CREATE TRIGGER check_product BEFORE INSERT OR UPDATE ON
    PRODUCTO
FOR EACH ROW EXECUTE PROCEDURE check_product();

/*
insert into
producto(id_producto, nombre_producto ,cantidad_producto
    ,precio_bruto)
values  ('4','arroz',5,-20000);
*/
```

3.6. Ejemplo de after trigger

- El siguiente ejemplo registra en una tabla auditora todas las operaciones que son realizadas a la tabla PRODUCTO.

```
CREATE TABLE PRODUCTOAUDITORIA(
operacion          char(1)    NOT NULL,
fecha              timestamp  NOT NULL,
userid             text       NOT NULL,
nombre_producto    text       NOT NULL,
precio_bruto  real
);

CREATE OR REPLACE FUNCTION proceso_producto_auditoria()
    RETURNS TRIGGER AS $producto_auditoria$
BEGIN
  IF (TG_OP = 'DELETE') THEN
  INSERT INTO PRODUCTOAUDITORIA SELECT 'D', now(), user,
       OLD.nombre_producto,OLD.precio_bruto;
  RETURN OLD;

  ELSIF (TG_OP = 'UPDATE') THEN
  INSERT INTO PRODUCTOAUDITORIA SELECT 'U', now(), user,
       NEW.nombre_producto,NEW.precio_bruto;
  RETURN NEW;

  ELSIF (TG_OP = 'INSERT') THEN
  INSERT INTO PRODUCTOAUDITORIA SELECT 'I', now(), user,
       NEW.nombre_producto,NEW.precio_bruto;
  RETURN NEW;
  END IF;

  RETURN NULL; -- resultado ignorado
END;
$producto_auditoria$ LANGUAGE plpgsql;

CREATE TRIGGER auditor_productos
AFTER INSERT OR UPDATE OR DELETE ON PRODUCTO
FOR EACH ROW EXECUTE PROCEDURE
    proceso_producto_auditoria();
```

4. Transacciones SQL

Una transaccion es un conjunto de operaciones, que van a ser tratadas como una unica unidad atomica de trabajo que se completa en su totalidad o no se lleva a cabo en absoluto. Estas transacciones deben cumplir 4 propiedades fundamentales comúnmente conocidas como ACID (atomicidad, coherencia, aislamiento y durabilidad).

Para fines de recuperación, el sistema debe hacer el seguimiento de cuándo se inicia, termina y confirma o aborta una transaccion. Por consiguiente, el gestor de recuperación hace un seguimiento de las siguientes operaciones:

- "**BEGIN TRANSACTION**". Marca el inicio de la ejecucion de una transaccion.
- "**END TRANSACTION**". Marca el final de la ejecucion de la transaccion. Sin embargo, en este punto puede ser necesario comprobar si los cambios introducidos por la transaccion pueden aplicarse de forma permanente a la base de datos (confirmados) o si la transaccion se ha cancelado porque viola la serializacion o por alguna otra razon.
- "**COMMIT TRANSACTION**". Señala una finalizacion satisfactoria de la transaccion, por lo que los cambios (actualizaciones) ejecutados por la transaccion se pueden enviar con seguridad a la base de datos y no se desharan.
- "**ROLLBACK (o ABORT)**". Señala que la transaccion no ha terminado satisfactoriamente, por lo que deben deshacerse los cambios o efectos que la transaccion pudiera haber aplicado a la base de datos.

4.1. Ejemplo de una transaccion en un procedimiento

```
1
2  CREATE PROCEDURE sp_agregar_venta (var_dni VARCHAR(10),
       var_idarticulo INTEGER, var_cantidad INTEGER)
3  DECLARE var_pu REAL; AS $$
4  BEGIN
5    BEGIN TRANSACTION
6      SELECT precio INTO var_pu FROM articulos WHERE
           idarticulos = var_idarticulo;
7
8      INSERT INTO ventas (dni, idarticulo, cantidad, importe)
           VALUES IN (var_dni, var_idarticulo, var_cantidad,
           var_pu*cantidad);
9
10     UPDATE articulos SET stock=stock-var_cantidad WHERE
           idarticulos = var_idarticulo;
11
12     IF <condicion de error>THEN
13       ROLLBACK
14     ELSE
15       COMMIT
16     END IF;
17
18   END TRANSACTION
19
20 END; $$
```

NOTA: En postgreSQL las transacciones son usadas principalmente en comandos de consola, por lo tanto para el manejo de errores (excepciones) en procedimientos almacenados se recomienda revisar el comando "**EXCEPTION**".

5. Esquema SQL

```
1
2  CREATE TABLE CLIENTE(
3  nombre       VARCHAR(50) NOT NULL,--
4  apellido     VARCHAR(50) NOT NULL,--
5  ciudad       VARCHAR(80) NOT NULL, --
6  cedula       VARCHAR (80),
7
8  CONSTRAINT PK_CLIENTE PRIMARY KEY (cedula));
9
10 CREATE TABLE SOCIO(
11 fecha_registro    TIMESTAMP DEFAULT CURRENT_DATE,
12 descuento   REAL DEFAULT 0.02,
13
14 CONSTRAINT PK_SOCIO PRIMARY KEY (cedula)
15
16 )inherits(CLIENTE);
17
18 CREATE TABLE CLIENTEREGULAR (
19
20 CONSTRAINT PK_CLIENTEREGULAR PRIMARY KEY (cedula)
21 )inherits(CLIENTE);
22
23 CREATE TABLE PRODUCTO (
24 id_producto   VARCHAR (80),
25 nombre_producto   VARCHAR(80) NOT NULL,
26 cantidad_producto INTEGER DEFAULT 0,
27 precio_bruto    REAL NOT NULL,
28
29 CONSTRAINT PK_PRODUCTO PRIMARY KEY (id_producto));
30
31 CREATE TABLE COMPRAPRODUCTO(
32 cedula       VARCHAR(80),
33 id_producto   VARCHAR(80),
34 cantidad     INTEGER DEFAULT 0,
35
36 CONSTRAINT PK_COMPRAPRODUCTO
37 PRIMARY KEY (cedula,id_producto),
38
39 CONSTRAINT FK_COMPRAPRODUCTOCLIENTE
40 FOREIGN KEY  (cedula)
41 REFERENCES CLIENTE (cedula) ON UPDATE CASCADE ON DELETE
       CASCADE,
42
43 CONSTRAINT FK_COMPRAPRODUCTO  FOREIGN KEY  (id_producto)
44 REFERENCES PRODUCTO (id_producto) ON UPDATE CASCADE ON
       DELETE CASCADE);
```