

**Sede San Carlos Prof. Efrén Jiménez**

**Curso: ISW-413**

**Aplicación de base de datos**

# Taller de postgresql

# Funciones

Antes de iniciar

CREATE TABLE clientes

(

id SERIAL PRIMARY KEY,

cc VARCHAR(10) NOT NULL,

nombre VARCHAR(50) NOT NULL

);

CREATE TABLE cuentas

(

id SERIAL PRIMARY KEY,

saldo REAL DEFAULT 0,

cli\_id INT NOT NULL REFERENCES CLIENTES(ID),

fecha\_sobregiro DATE,

intereses REAL DEFAULT 0

);

INSERT INTO clientes (cc, nombre) VALUES ('11111','CLIENTE 1');

INSERT INTO clientes (cc, nombre) VALUES ('22222','CLIENTE 2');

INSERT INTO clientes (cc, nombre) VALUES ('33333','CLIENTE 3');

INSERT INTO cuentas (saldo, cli\_id) VALUES (5000,1);

INSERT INTO cuentas (saldo, cli\_id) VALUES ( -2000,2);

INSERT INTO cuentas (saldo, cli\_id) VALUES (3000,3);

## Primer ejemplo

La estructura general de un bloque PL/PgSQL es la siguiente:

[ <<label>> ]

[ DECLARE

declarations ]

BEGIN

statements

END [ label ];

1. Por ejemplo, la siguiente función retorna su argumento multiplicado por 3:

CREATE OR REPLACE function mult\_3(x int)

returns INT as $$

begin

RETURN x\*3;

end; $$

LANGUAGE plpgsql;

1. La función puede ser utilizada en una cláusula SELECT:

select mult\_3(4);

mult\_3

--------

12

(1 row)

1. Las variables, en caso de ser necesarias, deben ser declaradas en la cláusula DECLARE:

CREATE OR REPLACE FUNCTION concat\_nombre (nombre VARCHAR, apellido VARCHAR)

RETURNS VARCHAR AS $$

DECLARE

temp VARCHAR;

BEGIN

temp:= nombre || ' ' || apellido;

RETURN INITCAP(temp);

END;

$$

LANGUAGE plpgsql;

1. Un ejemplo de uso:

select concat\_nombre('carlos','olarte');

concat\_nombre

---------------

Carlos Olarte

1. Ejemplo (Inserción de Datos en una Tabla)

Esta función inserta un registro en la tabla CLIENTES:

CREATE FUNCTION ins\_clientes (VARCHAR, VARCHAR)

RETURNS VOID AS $$

INSERT INTO clientes (cc, nombre) VALUES ($1, $2);

$$

LANGUAGE SQL;

1. Un ejemplo de uso:

SELECT ins\_clientes ('44444','CLIENTE 4');

Note que en este caso, los argumentos de la función no tienen nombre y se refieren a ellos como $1 y $2 dentro del cuerpo de la función. Además, como solo lanzamos sentencias SQL, el lenguaje de la función es SQL.

1. Ejemplo (Transfiriendo Dinero entre dos Cuentas)

/\* Función que transfiere dinero de una cuenta a otra \*/

CREATE FUNCTION transfer (cta1 cuentas.id%type, cta2 cuentas.id%type, monto cuentas.saldo%type)

RETURNS cuentas.saldo%type AS $$

DECLARE

nuevo\_saldo cuentas.saldo%type;

BEGIN

UPDATE cuentas SET saldo = saldo - monto WHERE id=CTA1;

UPDATE cuentas SET saldo = saldo + monto WHERE id=CTA2;

SELECT saldo INTO nuevo\_saldo FROM cuentas WHERE id=CTA2;

RETURN nuevo\_saldo;

END

$$

LANGUAGE PLPGSQL;

1. Ejemplo de uso:

postgres=# select transfer (3,2,1000);

transfer

----------

-1000

(1 row)

1. En este ejemplo hemos introducido algunas características interesantes del lenguaje:

* Los tipos de las variables se pueden declarar de acuerdo con los tipos de datos en las tablas como en CTA1 CUENTAS.ID%TYPE (es decir, CT1 es una variable del tipo del campo ID en las tablas CUENTAS).
* Hemos lanzado sentencias del DML como parte del procedimiento (UPDATE)
* Utilizamos SELECT … INTO VAR. Cuando un SELECT retorna una UNICA fila, esta se puede almacenar en una variable local.

Es posible realizar algunas validaciones. Por ejemplo, se debe verificar que las dos cuentas existan y que el monto de la segunda sea suficiente para realizar la transferencia.

/\* Versión2: Validación de los datos de entrada \*/

CREATE FUNCTION transfer2 (cta1 cuentas.id%type, cta2 cuentas.id%type, monto cuentas.saldo%type)

RETURNS cuentas.saldo%type AS $$

DECLARE

nuevo\_saldo cuentas.saldo%type;

saldo\_disponible cuentas.saldo%type;

test\_exists int;

BEGIN

SELECT COUNT(\*) INTO test\_exists FROM cuentas WHERE id=cta1;

IF test\_exists <> 1 THEN

RAISE EXCEPTION 'La cuenta de origen % no existe', cta1;

ELSE

SELECT COUNT(\*) INTO test\_exists FROM cuentas WHERE id=cta2;

IF test\_exists <> 1 THEN

RAISE EXCEPTION 'La cuenta destino % no existe', cta2;

ELSE

SELECT saldo INTO saldo\_disponible FROM cuentas WHERE id = cta1;

IF saldo\_disponible < monto THEN

RAISE EXCEPTION 'No hay fondos suficientes para la transferencia';

ELSE

/\* realizando la transferencia \*/

UPDATE cuentas SET saldo = saldo - monto WHERE id=cta1;

UPDATE cuentas SET saldo = saldo + monto WHERE id=cta2;

/\* Consultando y retornando el nuevo saldo de CTA2 \*/

SELECT saldo INTO nuevo\_saldo FROM cuentas WHERE id=cta2;

RETURN nuevo\_saldo;

END IF;

END IF;

END IF;

END

$$

LANGUAGE PLPGSQL;

1. Ejemplo (Actualizando los intereses)

Antes de presentar el ejemplo, vamos a adicionar algunas filas a la tabla cuentas:

INSERT INTO cuentas (saldo, cli\_id, fecha\_sobregiro) VALUES (-3000,3,'2010-01-01'), (-7000,2,'2010-03-01');

1. La siguiente función actualiza los intereses de mora en las cuentas que están sobregiradas. Los intereses que se cobran son proporcionales al saldo de la cuenta y se utiliza el porcentaje que se pasa como parámetro.

CREATE FUNCTION cobrar\_intereses (porcentaje REAL) RETURNS VOID AS $$

BEGIN

UPDATE cuentas

SET intereses = saldo \* porcentaje \*

EXTRACT ('DAYS'FROM (NOW()-fecha\_sobregiro))

WHERE fecha\_sobregiro IS NOT NULL;

END;

$$ LANGUAGE plpgsql;

1. Ejemplo de uso:

bdi00=> SELECT cobrar\_intereses (0.01);

1. Ejemplo (Utilizando Row Types)

Utilizando %ROWTYPE es posible declarar RECORDS del mismo tipo de una tabla. En el siguiente ejemplo, se crean RECORDS del tipo de las tablas CUENTAS y CLIENTES. La función simplemente imprime información relacionada con la cuenta que se pasa como parámetro.

CREATE FUNCTION inf\_cuenta (numcuenta INT) RETURNS VARCHAR AS $$

DECLARE

resultado VARCHAR;

cta\_row cuentas%ROWTYPE; --cta\_row es del mismo tipo de la tabla CUENTAS

cli\_row clientes%ROWTYPE;

BEGIN

-- Puede utilizar SELECT .. INTO cuando el resultado trae una sola fila

SELECT \* INTO cta\_row FROM cuentas WHERE id = numcuenta;

IF NOT FOUND THEN

RAISE EXCEPTION 'La cuenta % no existe. ', numcuenta;

ELSE

SELECT \* INTO cli\_row FROM clientes WHERE id = cta\_row.cli\_id;

resultado := ' La cuenta pertenece a ' || cli\_row.nombre ||

'. El saldo es ' || cta\_row.saldo;

RETURN resultado;

END IF;

END;

$$ LANGUAGE plpgsql;

1. Ejemplo de uso:

bdi00=> Select inf\_cuenta (1);

inf\_cuenta

----------------------------------------------------

La cuenta pertenece a CLIENTE 1. El saldo es 5000

(1 row)

Note el uso de IF NOT FOUND THEN … para verificar si el SELECT trajo o no algún resultado. Obviamente la función anterior se hubiera podido escribir de manera mucho más simple ejecutando una única consulta que traiga la información del cliente y de la cuenta:

CREATE FUNCTION inf\_cuenta2(numcuenta INT) RETURNS VARCHAR AS $$

DECLARE

saldo\_cta cuentas.saldo%type;

nombre\_cli VARCHAR;

BEGIN

SELECT cli.nombre, cta.saldo INTO nombre\_cli,saldo\_cta FROM cuentas cta

INNER JOIN clientes cli ON (cli.id = cta.cli\_id)

WHERE id = numcuenta;

IF NOT FOUND THEN

RAISE EXCEPTION 'La cuenta % no existe. ', numcuenta;

ELSE

RETURN ' La cuenta pertenece a ' || nombre\_cli ||

'. El saldo es ' || saldo\_cta;

RETURN resultado;

END IF;

END;

$$ LANGUAGE plpgsql;

En este caso, note que la cláusula SELECT … INTO puede asignar varias variables al tiempo.

1. Cursores

Un cursor es una estructura que permite recuperar los datos de una consulta fila por fila.

Asuma una tabla con una serie de transacciones bancarias “pendientes”, i.e., transacciones que deben ser realizadas y afectar las cuentas de los clientes:

CREATE TABLE pendientes

(

id SERIAL PRIMARY KEY,

monto REAL NOT NULL,

cta\_id INT NOT NULL REFERENCES CUENTAS(id),

fecha DATE DEFAULT NOW(),

operacion CHAR(1),

realizada BOOL DEFAULT FALSE

);

INSERT INTO pendientes (monto, cta\_id, operacion) VALUES

(300,1,'D'), (400,2,'R'), (233,3,'D');

1. El siguiente procedimiento “recorre” la tabla pendientes y realiza la operación indicada en la tabla CUENTAS.

CREATE OR REPLACE FUNCTION upd\_pendientes() RETURNS VOID AS $$

DECLARE

-- Declaración del Cursor

-- Note la forma del Select ... FOR UPDATE!

cur\_pen cursor FOR SELECT \* FROM pendientes WHERE realizada=FALSE FOR UPDATE;

ACCION pendientes%rowtype;

BEGIN

-- Abrir el cursor

OPEN cur\_pen;

LOOP

-- Extraer una fila del cursor

FETCH cur\_pen INTO accion;

IF accion IS NULL THEN EXIT;

END IF;

IF accion.operacion = 'D' THEN

UPDATE cuentas SET saldo = saldo + accion.monto WHERE id = accion.cta\_id;

ELSE

UPDATE cuentas SET saldo = saldo - accion.monto WHERE id = accion.cta\_id;

END IF;

-- Actualizar una fila de acuerdo con la posición del cursor

UPDATE pendientes SET realizada=TRUE WHERE CURRENT OF cur\_pen;

END LOOP;

-- Cierre del cursor.

CLOSE cur\_pen;

END;

$$ LANGUAGE plpgsql;

1. La cláusula FOR UPDATE evita que otras transacciones estén modificando las filas seleccionadas.

Ejemplo de uso:

bdi00=> select \* from cuentas;

id | saldo | cli\_id | fecha\_sobregiro | intereses

----+-------+--------+-----------------+-----------

1 | 5000 | 1 | | 0

2 | -2000 | 2 | | 0

3 | 3000 | 3 | | 0

(3 rows)

bdi00=> select \* from pendientes;

id | monto | cta\_id | fecha | operacion | realizada

----+-------+--------+------------+-----------+-----------

1 | 300 | 1 | 2010-04-26 | D | f

2 | 400 | 2 | 2010-04-26 | R | f

3 | 233 | 3 | 2010-04-26 | D | f

(3 rows)

bdi00=> SELECT upd\_pendientes();

upd\_pendientes

bdi00=> select \* from cuentas;

id | saldo | cli\_id | fecha\_sobregiro | intereses

----+-------+--------+-----------------+-----------

1 | 5300 | 1 | | 0

2 | -2400 | 2 | | 0

3 | 3233 | 3 | | 0

bdi00=> select \* from pendientes;

id | monto | cta\_id | fecha | operacion | realizada

----+-------+--------+------------+-----------+-----------

1 | 300 | 1 | 2010-04-26 | D | t

2 | 400 | 2 | 2010-04-26 | R | t

3 | 233 | 3 | 2010-04-26 | D | t

(3 rows)

1. Otra forma de recorrer tablas

Otra forma de utilizar cursores “implícitamente” es utilizando FOR <VAR> IN <QUERY>. Por ejemplo, asuma que se tiene una tabla con la lista de usuarios que han efectuado mal uso de sus cuentas y deben ser desactivadas:

CREATE TABLE usuarios\_fraude(id SERIAL, cli\_id INT);

INSERT INTO usuarios\_fraude (cli\_id) VALUES (1), (3);

1. Ahora adicionamos un atributo en cuenta para saber si está activa o no:

ALTER TABLE cuentas ADD inactiva BOOL DEFAULT FALSE;

1. La siguiente función inhabilita las cuentas cuyo usuario se encuentra en la tabla de usuarios\_fraude

CREATE OR REPLACE FUNCTiON inactiva () RETURNS VOID AS $$

DECLARE

usr usuarios\_fraude%rowtype;

BEGIN

FOR USR IN SELECT \* FROM usuarios\_fraude LOOP

UPDATE cuentas SET inactiva = TRUE WHERE cli\_id = usr.cli\_id;

END LOOP;

END;

$$ LANGUAGE plpgsql;

1. Para este caso particular, un lector atento hubiera podido descubrir que el ejercicio se podía realizar con una sola sentencia del DML:

UPDATE cuentas SET inactiva = FALSE WHERE cli\_id IN (SELECT cli\_id FROM usuarios\_fraude);

# Triggers

Un trigger es una acción que se lanza cuando se inserta, elimina o actualiza una o varios registros de una tabla. En PL/pgSQL, los triggers son funciones sin argumentos en las cuales se crean las siguientes variables:

* NEW: Es un Record con los datos del registro que se está insertando (actualizando).
* OLD: Datos del registro que se está eliminando (o actualizando).
* TG\_OP: Operación que se está realizando, puede ser INSERT, UPDATE, DELETE, o TRUNCATE.

Un ejemplo de Auditoria

1. Suponga que cada que se modifica una cuenta, se debe registrar la acción en la siguiente tabla:

CREATE TABLE auditoria

(

operacion CHAR(1),

fechaupd TIMESTAMP,

usuario TEXT,

id INT,

saldo REAL,

cli\_id INT,

fecha\_sobregiro DATE,

intereses REAL,

inactiva BOOL

);

1. La siguiente función deja un rastro en la tabla AUDITORIA cada que se realiza una modificación en la tabla CUENTAS:

CREATE OR REPLACE FUNCTION process\_cue\_audit () RETURNS TRIGGER AS $$

BEGIN

IF (TG\_OP = 'DELETE') THEN

INSERT INTO auditoria SELECT 'D', now(), user, OLD.\*;

RETURN OLD;

ELSIF (TG\_OP = 'UPDATE') THEN

INSERT INTO auditoria SELECT 'U', now(), user, NEW.\*;

RETURN NEW;

ELSIF (TG\_OP = 'INSERT') THEN

INSERT INTO auditoria SELECT 'I', now(), user, NEW.\*;

RETURN NEW;

END IF;

END;

$$ LANGUAGE plpgsql;

1. Ahora se crea el trigger sobre la tabla CUENTAS que se activa cada que se inserta, actualiza o elimina una (o varias) fila(s) en CUENTAS:

CREATE TRIGGER cue\_audit

AFTER INSERT OR UPDATE OR DELETE ON cuentas

FOR EACH ROW EXECUTE PROCEDURE process\_cue\_audit();

Algunos ejemplos:

bdi00=> insert into cuentas (saldo,cli\_id) values (3500,2);

INSERT 0 1

bdi00=> select \* from auditoria;

operacion | fechaupd | usuario | id | saldo | cli\_id | fecha\_sobregiro | intereses

-----------+----------------------------+---------+----+-------+--------+-----------------+-----------

I | 2010-04-26 10:30:18.081372 | bdi00 | 6 | 3500 | 2 | | 0

(1 row)

bdi00=> update cuentas set fecha\_sobregiro = now () where id =1;

UPDATE 1

bdi00=> select \* from auditoria;

operacion | fechaupd | usuario | id | saldo | cli\_id | fecha\_sobregiro | intereses

-----------+----------------------------+---------+----+-------+--------+-----------------+-----------

I | 2010-04-26 10:30:18.081372 | bdi00 | 6 | 3500 | 2 | | 0

U | 2010-04-26 10:31:23.2547 | bdi00 | 1 | 4067 | 1 | 2010-04-26 | 0

(2 rows)

1. Un ejemplo de una Validación

Los triggers son muy útiles cuando se requieren hacer validaciones que que no se pueden realizar con las restricciones de integridad referencial (PK, FK, UK,CK). Por ejemplo, asume que un cliente por política del banco, no puede tener más de 3 cuentas activas. Cada que se inserta una nueva cuenta, o se actualiza el titular de la cuenta, se debe verificar que no se incumple esta propiedad. Primero implementamos la función:

CREATE OR REPLACE FUNCTION fun\_ck\_num\_clientes() RETURNS TRIGGER AS $$

DECLARE

NUM\_CUENTAS INT;

BEGIN

IF (TG\_OP = 'UPDATE' AND OLD.CLI\_ID <> NEW.CLI\_ID ) OR TG\_OP='INSERT' THEN

SELECT COUNT(\*) INTO NUM\_CUENTAS FROM CUENTAS

WHERE CLI\_ID = NEW.CLI\_ID;

IF NUM\_CUENTAS >= 3 THEN

RAISE EXCEPTION 'LIMITE DEL NUMERO DE CUENTAS ALCANZADO POR EL CLIENTE';

ELSE

RETURN NEW;

END IF;

ELSE

RETURN NEW;

END IF;

END;

$$ LANGUAGE plpgsql;

/\* Creacion del Trigger \*/

CREATE TRIGGER trg\_num\_clientes

AFTER INSERT OR UPDATE ON cuentas

FOR EACH ROW EXECUTE PROCEDURE fun\_ck\_num\_clientes();

1. Valores Calculados

Los triggers también pueden ser utilizados para mantener valores calculados. Por ejemplo, asumamos que en la tabla CLIENTES se desea mantener el acumulado en dinero que tiene el cliente en sus distintas cuentas. Para esto, modificamos la tabla:

ALTER TABLE clientes ADD COLUMN saldo REAL DEFAULT 0.0;

Como no habíamos implementado el trigger que se encargara de llevar este saldo, por la primera vez lo calculamos “manualmente”:

CREATE OR REPLACE FUNCTION update\_saldos() RETURNS VOID AS $$

DECLARE

cliente\_saldo REAL;

cliente\_id REAL;

BEGIN

FOR cliente\_id, cliente\_saldo IN SELECT cli\_id, SUM(saldo) FROM cuentas GROUP BY cli\_id LOOP

UPDATE clientes SET saldo=cliente\_saldo WHERE id = cliente\_id;

END LOOP;

RETURN;

END;$$

LANGUAGE plpgsql;

Ahora, por cada modificación del saldo en una cuenta, actualizamos el saldo en el cliente:

CREATE OR REPLACE FUNCTION set\_saldo\_cliente() RETURNS TRIGGER AS $$

BEGIN

IF(TG\_OP = 'INSERT') THEN

UPDATE clientes SET saldo = saldo + new.saldo WHERE id = new.cli\_id;

RETURN NEW;

ELSIF (TG\_OP = 'UPDATE') THEN

IF new.cli\_id = old.cli\_id THEN

UPDATE clientes SET saldo = saldo + new.saldo - old.saldo WHERE id = new.cli\_id;

RETURN NEW;

ELSE /\* LA CUENTA CAMBIO DE TITULAR \*/

UPDATE clientes SET saldo = saldo - old.saldo WHERE id = old.cli\_id;

UPDATE clientes SET saldo = saldo + new.saldo WHERE id = new.cli\_id;

RETURN NEW;

END IF;

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

/\* Creacion del Trigger \*/

CREATE TRIGGER trg\_saldo\_clientes

AFTER INSERT OR UPDATE ON cuentas

FOR EACH ROW EXECUTE PROCEDURE set\_saldo\_cliente();

Realice algunas operaciones sobre la tabla cuentas (cambiando el titular y/o el saldo) para verificar que la cuenta se lleva correctamente.

# Manejo de Errores

1. Es posible lanzar y capturar errores de la siguiente manera:

CREATE OR REPLACE FUNCTION test(int, int) RETURNS INT AS $$

DECLARE

X INT;

BEGIN

X := $1/$2;

RETURN X;

EXCEPTION

WHEN division\_by\_zero THEN

RAISE EXCEPTION 'Division por cero';

RETURN 0;

WHEN others THEN

RAISE NOTICE 'Error';

RETURN 0;

END;

$$ LANGUAGE plpgsql;

Ejemplo de uso:

bdi00=> select test(6,2);

test

------

3

(1 row)

bdi00=> select test(6,0);

NOTICE: Division por cero

test

------

0

(1 row)

# Tarea

Cuando un usuario quiere ser amigo de otro, debe enviar una invitación. Los estados de la invitación pueden ser “pendiente”, “aceptado”, “rechazado”.

1. Cree el script de tablas
2. Implemente un procedimiento que cada que se acepte una invitación, se adicione la la relación de amistad en la tabla amigos.
3. Por cada nuevo amigo se debe actualizar el número de amigos en el campo USUARIOS.NUM\_AMIGOS
4. La relación de amigos se asume que es simétrica, es decir, si A es amigo de B entonces B es amigo de A. Realice un trigger que evite la inserción de (A,B) en la tabla AMIGOS si la tuple (B,A) ya se encuentra. Además, se deben rechazar invitaciones de A, B si A,B ya son amigos (o B,A ya son amigos)
5. Adicione un trigger que rechace nuevas invitaciones de A a B si B ha rechazado previamente la invitación.