EXTENSIÓ PROCEDIMENTAL. EL LLENGUATGE PL/PGSQL

SQL HOSTATJAT

- Treballar en la base de dades de forma interactiva no és la millor manera.
- El millor és programar les sentències SQL mitjançant un llenguatge de programació.
- Les sentències SQL que treballen de forma hostatjada utilitzen aquestes sentències en un programa escrit en un llenguatge de programació.

LLENGUATGE PL/PGSQL

- El llenguatge PL/pgSQL (Procedural Language Extension) és un llenguatge procedimental dissenyat per treballar amb bases de dades, propi del SGBD PostgreSql.
- Té totes les característiques pròpies dels llenguatges de tercera generació: variables, estructura modular (procediments i funcions), estructures de control (alternatives, iteratives, etc.), control d'excepcions, etc.
- Els programes creats amb PL/pgSQL es poden emmagatzemar a la base de dades com qualsevol altre objecte.
- PL/pgSQL suporta les sentències DDL (llenguatge de definició de dades) i DML (llenguatge de manipulació de dades), aportant al llenguatge SQL elements propis dels llenguatges procedimentals de tercera generació (estructures de control, ...)
- Avantatges del llenguatge PL/pgSQL:
    - Facilitar la distribució del treball.
    - La instal·lació i manteniment del programa.
    - Reducció del cost de treball.

BLOCS PL/PGSQL

Quan es treballa amb PL/pgSQL, es fa amb blocs PL/pgSQL, que són un conjunt de declaracions, instruccions i mecanismes per gestionar errors i excepcions.

Tipus de blocs:

- Anònim
- Procediment
- Funció

BLOCS PL/PGSQL ANÒNIMS

Un bloc anònim no s'emmagatzema al servidor. És com un script, podem desar-lo en un arxiu i recuperar-lo i executar-lo quan ens convingui.

Estructura bàsica d'un bloc anònim:

```
[ <<label>> ]
[ declare
    declarations ]
begin
    statements;
        ...
end [ label ];
```

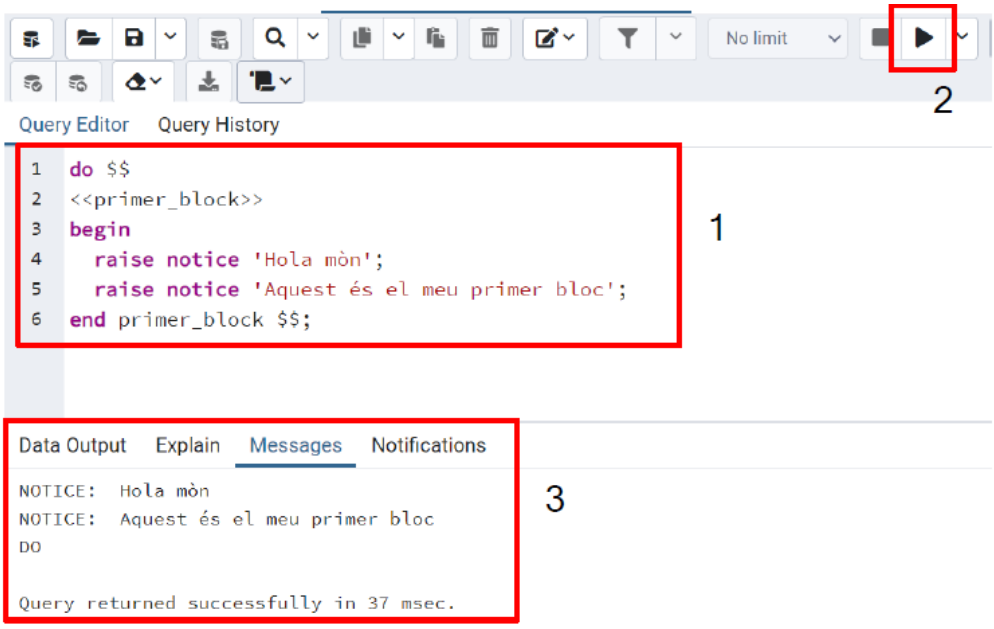
- **DECLARE**: Indica la zona de declaracions, és a dir, on es declaren els objectes locals necessaris (variables, constants, etc). És opcional.
- **BEGIN/END**: Zona on hi ha el conjunt d'instruccions que s'han d'executar.
- **<<label>>**: Opcionalment podem assignar una etiqueta al bloc.

Exemple:

```
do $$
<<primer_bloc>>
Begin
raise notice 'Hola mon';
raise notice 'Aquest és el meu primer bloc';
End primer_bloc $$
```

- **do**: Aquesta instrucció no forma part del bloc anònim, sinó que s'utilitza per executar-lo.
- **$$**: Els blocs anònims s'escriuen entre cometes simples igual que qualsevol literal en SQL, el problema és que dins hi pot haver molts caràcters com ' o que requereixen l'ús de caràcters d'escapament. PostgreSQL mitjançant l'opció $$ permet escriure qualsevol literal sense utilitzar caràcters d'escapament.
- **RAISE NOTICE**: Ens permet escriure dades en la zona de missatges per a l'usuari.

Execució d'un bloc anònim en el pgAdmin4:



1. Escrivim el codi en l'editor. No oblidar-se de començar per do $$ i acabar amb $$.
2. Cliquem el botó Play per executar el codi.
3. Si tot ha anat bé podem veure el resultat en la pestanya 'Messages'.

VARIABLES I CONSTANTS

- Variables: S'utilitzen entre altres coses per emmagatzemar temporalment una dada i manipular valors emmagatzemats anteriorment.

Les variables PL/pgSQL s'han de declarar dins la secció corresponent i sempre abans de la seva utilització, és a dir, dins la secció DECLARE.

```
variable_name data_type [NOT NULL] [:= expression];
```

- variable_name: Identificador de la variable.
- data_type: Tipus de dada de la variable.
- NOT NULL: Per forçar que la variable tingui un valor. En aquest cas l'haurem d'inicialitzar amb l'operador d'assignació (:=)
- Constants: S'utilitzen per inicialitzar valors que es mantindran invariables en el nostre programa.

```
constant_name CONSTANT data_type := expression;
```

Exemple:

```
do $$
declare
  vat constant numeric := 0.1;
  net_price numeric 20.5;
begin
  raise notice 'Selling price is %', net_price * (1+vat);
end $$
```

- Clàusula **TYPE**: Permet declarar una variable del mateix tipus que una altra, o que una columna d'una taula.

```
column_variable table_name.field_name%TYPE

Exemple:

    declare
       data_lloguer rental.rental_date%TYPE;
       data_revisió data_lloguer%TYPE;
```

- Clàusula **ROWTYPE**: Permet crear una variable de registre on els seus camps es corresponen amb les columnes d'una taula o vista de la base de dades.

```
row_variable table_name%ROWTYPE

Exemple:

    declare
       lloguer rental%ROWTYPE;

Us:
   ...
   raise notice 'Data lloguer %', lloguer.rental_date;
```

Exemple:

```
do $$
declare
  selected_actor actor%rowtype;
begin
 -- select actor with id 10
select *
from actor
into selected_actor
where actor_id=10;
-- show the name of actor
raise notice 'The actor name is % %', selected_actor.first_name,
selected_actor.last_name;
end $$
```

CLASSIFICACIÓ DE LES VARIABLES PL/PGSQL

Dos tipus de variables:

- **Escalar**: Contenen un sol valor. Són els tipus més comuns que podem trobar amb PL/pgSQL, com ara VARCHAR, NUMERIC, DATE, CHAR, BOOL, etc.
- **Composta**: S'utilitzen per definir i manipular grups de camps dins de blocs PL/pgSQL. Són les taules PL/pgSQL i registres PL/pgSQL.

Els blocs poden contenir altres blocs dins seu, per tant, s'estableix una jerarquia de bloc pare i bloc fill. Això afecta a la visibilitat de les variables. Les variables definides en el bloc pare són visibles en els blocs fills, per contra, les variables definides en els blocs fills es destrueixen en finalitzar aquest bloc i, per tant, no tenen visibilitat en els blocs pare.

```
do $$
declare
   var1 numeric;
Begin
   var1 := 10;
   raise notice 'El valor de la variable 1 es %', var1;
   declare
      var2 numeric := var1;
   begin
      raise notice 'El valor de la variable 2 es %', var2;
   end;
   var2 := var1; /*ERROR: var2 no es coneix en aquest
                   àmbit ja que és local en el bloc fill
                   i ja ha acabat */
end; $$
```

OPERADORS

Es poden utilitzar tots els operadors de SQL.

- Aritmètics: +, -, *, /
- Condicionals: <, >, <>, !=, =, >=, <=, NOT, AND, OR
- Concatenació: ||

INSTRUCCIONS CONDICIONALS

Una de les instruccions condicionals que ens ofereix PL/pgSQL com en la majoria de llenguatges de programació és la instrucció **IF**.

```
if condition_1 then
  statement_1;
[elsif condition_2 then
  statement_2
...
elsif condition_n then
  statement_n;
else
  else-statement;]
end if;
```

'Condition' serà qualsevol expressió que retorni un valor booleà (cert o fals)

> Exemple:
>
> ```
> do $$
> declare
>   v_film film%rowtype;
>   len_description varchar(100);
> begin
>
>   select * from film
>   into v_film
> ```

```
    where film_id = 100;

    if not found then
       raise notice 'Film not found';
    else
        if v_film.length >0 and v_film.length <= 50 then
           len_description := 'Short';
        elsif v_film.length > 50 and v_film.length < 120 then
           len_description := 'Medium';
        elsif v_film.length > 120 then
           len_description := 'Long';
        else
           len_description := 'N/A';
        end if;

         raise notice 'The % film is %.', v_film.title, len_description;
    end if;
  end $$
```

Una altra de les instruccions condicionals que ens ofereix PL/pgSQL és la instrucció **CASE**, que ja havíem utilitzat en el llenguatge SQL, en els dos formats que havíem vist.

```
case search-expression
     when expression_1 [, expression_2, ...] then
          when-statements
   [ ... ]
   [else
          else-
statements ] END
case;
```

- Search expression: Qualsevol expressió que volem avaluar amb Case.
- Expression_1, expression_2,... : Valors amb els que compararà utilitzant l'operador =

Exemple:

```
do $$
declare
    rate   film.rental_rate%type;
    price_segment varchar(50);
begin
    -- get the rental rate
    select rental_rate into rate
    from film
    where film_id = 100;

    -- assign the price segment
    if found then
       case rate
            when 0.99 then
               price_segment =  'Mass';
            when 2.99 then
```

```
                price_segment = 'Mainstream’;
             when 4.99 then
                price_segment = 'High End’;
             else
      price_segment = 'Unspecified’;
        end case;
        raise notice '%', price_segment;
    end if;
end; $$
```

```
case
when boolean-expression-1 then  statements
[ when boolean-expression-2 then
statements
... ]
[ else
statements ]
end case;
```

- boolean expression : Qualsevol expressió retorni una condició booleana (cert o fals).

Exemple:

```
do $$
declare
    total_payment numeric;
    service_level varchar(25) ;
begin
     select sum(amount) into total_payment
     from Payment
     where customer_id = 100;

     if found then
        case
           when total_payment > 200 then
               service_level = 'Platinum' ;
           when total_payment > 100 then
      service_level = 'Gold' ;
           else
               service_level = 'Silver' ;
        end case;
        raise notice 'Service Level: %', service_level;
     else
        raise notice 'Customer not found’;
     end if;
end; $$
```

INSTRUCCIONS ITERATIVES

Com en la majoria de llenguatges de programació, PL/pgSQL també ens ofereix diferents alternatives per a realitzar iteracions.

LOOP: És una estructura iterativa infinita, podem sortir mitjançant la comanda **EXIT**.

```
loop
statements;
[ exit when condition; ]
end loop;
```

Exemple: Sèrie de Fibbonacci

```
do $$
declare
   n integer:= 10;
   fib integer := 0;
   counter int> eger := 0 ;
   i integer := 0 ;
   j integer := 1 ;
begin
   if (n < 1) then
      fib := 0;
   end if;
   loop
     exit when counter = n ;
     counter := counter + 1 ;
     select j, i + j into i,    j ;
   end loop;
   fib := i;
   raise notice '%', fib;
end; $$
```

WHILE .. LOOP: Instrucció iterativa que avalua una condició inicial que s'ha de complir per continuar iterant.

```
while condition loop
   statements;
end loop;
```

- **condition**: S'avalua abans d'executar qualsevol instrucció.
- Dins la instrucció while haurem de canviar algunes variables perquè canviï la condició a avaluar.

Exemple: Mostrar els 5 primers nombres enters a partir de 0

```
do $$
declare
   counter integer := 0;
```

```
begin
   while counter < 5 loop
      raise notice 'Counter %', counter;
      counter := counter + 1;
   end loop;
end$$;
```

FOR .. LOOP: La instrucció FOR .. LOOP ens permet iterar un nombre definit de vegades.

```
for loop_counter in [ reverse ] from.. to
    [ by step ] loop
    statements
end loop;
```

- **loop_counter**: Variable de tipus enter que es genera en la pròpia sentència Loop i que només té visibilitat dins la pròpia sentència. Serveix com a comptador i per defecte s'incrementa de 1 en 1.
- **Reverse**: En comptes de sumar, resta per tant servirà per fer recorreguts de més a menys.
- **From .. To**: Rang de valors sobre el que s'executarà la sentència.
- **By step**: Per incrementar/decrementar amb un valor diferent de 1.

Exemple 1: Mostrar els 5 primers nombres enters a partir de l'1

```
do $$
begin
   for counter in 1..5 loop
     raise notice 'counter: %', counter;
   end loop;
end; $$
```

Exemple 2: Mostrar els 5 primers nombres enters en ordre invers a partir del 5

```
do $$
begin
   for counter in reverse 5..1 loop
      raise notice 'counter: %', counter;
   end loop;
end; $$
```

Exemple 3: Mostrar els 3 primers nombres enters imparells a partir de l'1

```
do $$
begin
  for counter in 1..6 by 2 loop
    raise notice 'counter: %', counter;
```

```
    end loop;
  end; $$
```

MISSATGES A L'USUARI - INSTRUCCIÓ RAISE

Mitjançant la instrucció RAISE podem enviar missatges a l'usuari. Tenim diferents nivells de missatge i en funció de cada nivell (level) es reporten d'una manera o altre.

```
raise level format;
```

- **Level**: Disposem dels següents nivells: *debug*, *info*, *log*, *warning*, *notice* i *exception* (nivell per defecte)
- **Format**: Cadena de caràcters que representa el missatge a mostrar, utilitza el comodí % per substituir el valor dels paràmetres que li passarem, separats per ','. Hi ha d'haver el mateix nombre de % que paràmetres.

Exemple:

```
do $$
begin
  raise info 'information message %', now() ;
  raise log 'log message %', now();
  raise debug 'debug message %', now();
  raise warning 'warning message %', now();
  raise notice 'notice message %', now();
end $$;
```

Sortida:

```
info:  information message 2015-09-10 21:17:39.398+07
warning:  warning message 2015-09-10 21:17:39.398+07
notice:  notice message 2015-09-10 21:17:39.398+07
```

INSTRUCCIÓ RAISE EXCEPTION

- Quan volem enviar un missatge d'error treballarem amb el nivell Exception (per defecte).
- A més podem afegir informació addicional amb la comanda *using*.

```
using option = expression
```

- **Option**:
  - message : Missatge d'error mostrat a l'usuari.
  - hint : Informació resumida per clarificar l'error.
  - detail: Informació detallada de l'error.
  - errcode : Informació d'identificació de l'error. Es pot expressar per nom de l'error o codi d'error (SQLSTATE)

Exemple:

```
do $$
declare
  email varchar(255) := 'info@postgresqltutorial.com';
begin
  -- check email for duplicate
  -- ...
  -- report duplicate email
  raise exception 'duplicate email: %', email
                  using hint = 'check the email again';
end $$;
```

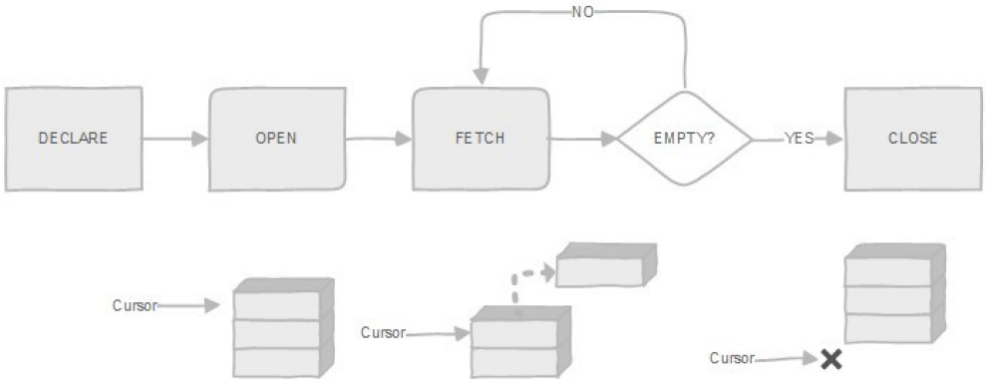
Missatges d'error predefinits de PostgreSql

CURSORS

Els cursors representen un element molt important en el llenguatge PL/pgSQL. Els cursors representen consultes SELECT sql que retornen més d'una fila i que permeten l'accés a cadascun dels registres. Això significa que el cursor sempre té un punter a una de les files del SELECT que representa el cursor.

Es processen amb quatre passes:

1. Declarar el cursor
2. Obrir el cursor. Després d'obrir el cursor, aquest apuntarà a la primera fila (si n'hi ha).
3. Avançar el cursor. La instrucció FETCH permet recórrer el cursor registre a registre fins que el punter arribi al final.
4. Tancar el cursor.



PAS 1. DECLARAR EL CURSOR

```
cursor_name [ [no] scroll ] cursor
[( name datatype, name data type, ...)] for query;
```

Exemple:

```
declare
    cur_films  cursor for
```

```
        select *
        from film;
    cur_films2 cursor (year integer) for
        select *
        from film
        where release_year = year;
```

PAS 2. OBRIR EL CURSOR

```
open cursor_variable[ (name:=value,name:=value,...)];
```

La seqüencia de passes que segueix en executar-se la comanda OPEN és la següent:

1. Reservar memòria suficient pel cursor.
2. Executar la sentència SELECT que defineix el cursor.
3. Situar el punter a la primera fila.

Si no hi ha registres per mostrar PostgreSQL no retorna cap error.

PAS 3. AVANÇAR EL CURSOR

```
fetch [ direction { from | in } ] cursor_variable
into target_variable;
```

- **Direction**: Indica la direcció d'avanç, pot ser: NEXT, LAST, PRIOR, FIRST, ABSOLUTE cont , RELATIVE cont , FORWARD, BACKWARD

Quan avancem el cursor primer ens retornarà les dades del registre o fila actual i seguidament avançarà fins al següent registre.

Podem moure el cursor però sense retornar les dades de la fila amb la sentència **MOVE** que admet els mateixos paràmetres que FETCH.

Exemple:

```
fetch cur_films into row_film;
fetch last from row_film into title, release_year;

move cur_films;
move last from cur_films;
move relative -1 from cur_films;
move forward 3 from cur_films;
```

PAS 4. TANCAR EL CURSOR.

```
close cursor_variable;
```

Quan es tanca el cursor s'allibera la memòria que ocupa i s'impedeix que s'utilitzi.

Exemple:

```
do $$
Declare
    titles text default ‘’;
    rec_film   record;
    cur_films cursor(p_year integer)
    for select title, release_year
        from film
        where release_year = p_year;
begin
   -- open the cursor
   open cur_films(p_year:=2006);

   loop
    -- fetch row into the film
      fetch cur_films into rec_film;
    -- exit when no more row to fetch
      exit when not found;

    -- build the output
      if rec_film.title like ‘%FUL%' then
         titles := titles || ',' || rec_film.title || ':' ||
rec_film.release_year;
      end if;
   end loop;

   -- close the cursor
   close cur_films;

   raise notice '%', titles;
end $$;
```

VARIABLES TIPUS RECORD

- Quan volem assignar un conjunt de dades d’una fila retornada per una consulta podem utilitzar el tipus record.
- Aquestes variables no tenen una estructura predeterminada sinó que vindrà donada per el resultat de la consulta on la utilitzem.
- Recordem que també tenim la possibilitat de declarar variables tipus %ROWTYPE.

Exemple: Us d'una variable tipus registre

```
do $$
declare
    rec record;
begin
    -- select the film
    select film_id, title, length
```

```
    into rec
    from film
    where film_id = 200;

    raise notice '% % %', rec.film_id, rec.title, rec.length;
end; $$
```

RECORREGUT DE CURSORS AMB LA INSTRUCCIÓ FOR

La instrucció FOR ens proporciona la forma més normal de recórrer totes les files d'un cursor. És un bucle FOR que s'encarrega de fer tres tasques:

- Obre el cursor (fa l'OPEN abans de començar el bucle).
- Recorre totes les files del cursor (a cada iteració es genera un FETCH implícit) i emmagatzema el contingut de cada fila en una variable de registre (que no cal declarar a la zona DECLARE).
- Tanca el cursor (quan acaba el FOR).

```
[ <<label>> ]
FOR recordvar IN bound_cursorvar
[ ( [ argument_name := ] argument_value [, ...] ) ] LOOP
    statements
END LOOP [ label ];
```

Exemple 1: Utilitzant una variable tipus cursor

```
do $$
declare
   cur_films cursor(p_year integer)
   for select title, release_year
       from film
       where release_year = p_year;
   titles text := '';
begin
   -- open and fetch cursor
   for rec_film in cur_films(p_year:=2006) loop
    -- build the output
      if rec_film.title like '%FUL%' then
         titles := titles || ',' || rec_film.title || ':' ||
rec_film.release_year;
      end if;
   end loop;
  raise notice '%', titles;
end $$;
```

Exemple 2: Escrivint la comanda SELECt dins del propi bucle.

```
do $$
declare
```

```
    titles text := '';
    rec_film record;
    p_year int := 2006;
begin
    -- open the cursor
    for rec_film in (select title, release_year from film
                     where release_year = p_year)  loop
     -- build the output
       if rec_film.title like '%FUL%' then
          titles := titles || ',' || rec_film.title || ':' ||
rec_film.release_year;
       end if;
    end loop;
    raise notice '%', titles;
end $$;
```

CURSORS PER ACTUALITZAR REGISTRES

A vegades un cursor ens podrà fer servei per actualitzar els seus registres. Una vegada el cursor està obert podem actualitzar o eliminar el registre apuntat utilitzant la clàusula **CURRENT OF**.

```
update table_name
set column = value, ...
where current of cursor_variable;

delete from table_name
where current of cursor_variable;
```

Exemple:

```
update film
set release_year = p_year
where current of cur_films;
```

SUBPROGRAMES: STORED PROCEDURE I FUNCTION

STORED PROCEDURES

A diferència dels blocs anònims els stored procedure i function s'emmagatzemen al servidor i s'executen cada vegada que s'invoquen.
Els stored procedures NO retornen un valor.

```
create [or replace] procedure procedure_name(param_list)
   language plpgsql
  as
$$
declare
-- variable declaration
begin
 -- logic
end;
$$
```

La crida a un stored procedure es fa mitjançant la comanda **CALL**: *call stored_procedure_name(argument_list)*

> Exemple:
>
> ```
> create or replace procedure transfer(sender int, receiver int, amount dec)
> language plpgsql
> as $$
> begin
> --subtracting the amount from the sender's account
> update accounts
> set balance = balance - amount
> where id = sender;
> --adding the amount to the receiver's account
> update accounts
> set balance = balance + amount
> where id = receiver;
> commit;
> end;$$
> ```

FUNCTIONS

S'emmagatzema al servidor i s'executa cada vegada que s'invoca. Les funcions retornen un valor.

```
create [or replace] function function_name(param_list)
   returns return_type
   language plpgsql
  as
$$
declare
-- variable declaration
begin
 -- logic
end;
$$
```

- **RETURN**: amb aquesta sentència retornem la dada des de la funció. Una funció que no retorna res (void) seria com un stored procedure.

> Exemple:

```
create or replace function find_film_by_id(p_film_id int)
returns varchar
language plpgsql
as $$
declare
   film_title film.title%type;
begin
  -- find film title by id
  select title into film_title
  from film where film_id = p_film_id;

  return film_title;

end;$$
```

CRIDES A SUBPROGRAMES

Les funcions es poden cridar des d'una comanda SELECT.

```
Select get_film_count(40,90);

GET_FILM_COUNT
--------------
325
```

En el cas tant de procediments com de funcions podem cridar-los des d'un bloc anònim.

```
do $$
begin
   raise notice '%', get_film_count(40,90);
end $$;

NOTICE: 325
```

PAS DE PARÀMETRES

IN: . Són els paràmetres que en altres llenguatges s'anomenen *per valor*. El procediment rep una copia del valor o variable que s'utilitza com a paràmetre al cridar el procediment. Aquests paràmetres poden ser: valors literals (18 per exemple), variables (v_num per exemple) o expressions (com v_num+18). A aquests paràmetres se'ls hi pot assignar un valor per defecte. Si no especifiquem cap clàusula en el pas de paràmetres per defecte s'enten que és un paràmetre tipus IN.

Exemple:

```
create or replace function find_film_by_id(IN p_film_id int)
returns varchar
language plpgsql
```

```
as $$
declare
   film_title film.title%type;
begin
  -- find film title by id
  select title into film_title
  from film where film_id = p_film_id;

  return film_title;

end;$$
```

OUT: Relacionats amb el pas *per variable o referència* d'altres llenguatges. Només poden ser variables i no poden tenir un valor per defecte. Son variables que es poden utilitzar per enviar un valor al procediment i/o perquè el procediment hi guardi algun valor. És a dir, els paràmetres OUT són variables que s'envien al procediment de manera que si en el procediment canvia el seu valor, aquest valor es manté quan el procediment acaba.

Exemple:

```
create or replace function get_film_stat(
    out min_len int,
    out max_len int,
    out avg_len numeric)
language plpgsql
as $$
begin

  select min(length),
         max(length),
         avg(length)::numeric(5,1)
  into min_len, max_len, avg_len
  from film;

end;$$
```

INOUT: Són una combinació dels dos anteriors. Es tracta de variables on el valor es pot utilitzar pel procediment que, a més a més, pot guardar hi un valor. No se'ls hi pot assignar un valor per defecte.

Exemple:

```
create or replace function swap(
    inout x int,
    inout y int)
language plpgsql
as $$
begin
   select x,y into y,x;
end; $$;
```

EXCEPCIONS - BLOC EXCEPTION

S'anomena excepció a una acció que li succeeix a un programa que provoca que la seva execució acabi. Òbviament això causa que el programa acabi de forma anòmala.

Quan es produeix una excepció?

- Quan es produeixi un error detectat per Postgresql (per exemple si un SELECT no retorna dades es produeix l'error P0002 també anomenat NO_DATA_FOUND).
- Quan el programador les provoqui (comanda RAISE).

Per capturar les excepcions i fer-ne el tractament corresponent per evitar l'acabament inesperat del codi PL/pgSQL utilitzarem el bloc **EXCEPTION**.

Si el programador no tracta una excepció aquesta serà gestionada per Postgresql.
Tipus d'excepcions:

- Excepcions preestablertes. Ja tenen assignat un nom d'excepció.
- Excepcions del postgresql sense definir. No tenen nom assignat però se'ls pot tractar mitjançant SQLSTATE i el seu codi d'excepció.
- Definides per l'usuari. Les ha de cridar el programador.

```
<<label>>
declare
begin
    statements;
exception
    when condition [or condition...] then
       handle_exception;
   [when condition [or condition...] then
       handle_exception;]
   [when others then
       handle_other_exceptions;
   ]
end;
```

Les possibles excepcions s'avaluen en l'ordre que les hem escrit.
WHEN OTHERS, per tractar qualsevol excepció no controlada prèviament.

Exemple:

```
do $$
declare
  x numeric :=0;
  y numeric :=3;
  res numeric;
begin
  res := y/x;
  raise notice '%', res;
exception
  when division_by_zero then
```

```
    raise notice 'No es pot dividir per zero.';
  when others then
    raise notice 'Error no esperat.';
end $$;
```

En el Postgresql totes les excepcions tenen un codi i un nom associat, podem però tractar les utilitzant només el codi amb la variable global **SQLSTATE**.

Exemple:

```
do $$
declare
  x numeric :=0;
  y numeric :=3;
  res numeric;
begin
  res := y/x;
  raise notice '%', res;
exception
  when SQLSTATE '22012' then
    raise notice 'No es pot dividir per zero.';
  when others then
    raise notice 'Error no esperat.';
end $$;
```

DISPARADORS - TRIGGERS

- Un trigger és una 'funció' invocada automàticament quan es produeix un determinat event associat a una taula.
- Aquest event pot ser un dels següents: INSERT, UPDATE, DELETE o TRUNCATE.
- Un trigger és una funció d'usuari especial associada a una taula.
- Per a crear triggers haurem de definir primer una funció tipus 'trigger' i a continuació associar aquesta funció a una taula.
- La diferència entre un trigger i una funció d'usuari és que el trigger s'executarà automàticament.

TIPUS DE DISPARADORS

Disparadors **d'instrucció**: El cos del trigger s'executa una única vegada per cada esdeveniment que fa que s'activi el trigger. Aquesta és la opció per defecte. El codi s'executa encara que la instrucció DML no afecti a cap fila.

Disparadors **de fila**: El codi s'executa una vegada per cada fila afectada per l'esdeveniment. Per exemple, si una instrucció UPDATE que activa un trigger i aquest UPDATE actualitza 10 files si el trigger és de fila llavors s'executa una vegada per cada fila.

Exemple: Us de trigger de fila

1. Crearem la funció de trigger.

```
CREATE FUNCTION trigger_function()
   RETURNS TRIGGER
   LANGUAGE PLPGSQL
AS $$
BEGIN
   -- trigger logic
END;
$$
```

```
CREATE OR REPLACE FUNCTION log_last_name_changes()
  RETURNS TRIGGER
  LANGUAGE PLPGSQL
  AS $$
BEGIN
  IF NEW.last_name <> OLD.last_name THEN
     INSERT INTO employee_audits(employee_id,last_name,changed_on)
     VALUES(OLD.id,OLD.last_name,now());
  END IF;
  RETURN NEW;
END; $$
```

Les funcions que retornen un trigger reben dades del propi entorn que les ha cridat mitjançant estructures de dades especials anomenades TriggerData que contenen un conjunt de variables locals. Per exemple, **OLD** i **NEW** representen els estats d'una fila de la taula abans i després de l'execució del trigger.

2. Crearem el trigger i l'associarem a la funció.

```
CREATE TRIGGER trigger_name
   {BEFORE | AFTER} { event }
   ON table_name
   [FOR [EACH] { ROW | STATEMENT }]
       EXECUTE PROCEDURE trigger_function
```

```
CREATE TRIGGER last_name_changes
  BEFORE UPDATE
  ON employees
  FOR EACH ROW
  EXECUTE PROCEDURE log_last_name_changes();
```

Exemple: Us de trigger d'instrucció Abans de crear el trigger crearem una nova taula anomenada 'auditoria_depts':

```
CREATE TABLE auditoria_depts (
   usuari    varchar(50),
   data      timestamp,
   operacio  char(1),
   primary key (usuari,data));
```

A continuació crearem la funció de trigger.

```
CREATE OR REPLACE FUNCTION process_dept_audit()
RETURNS TRIGGER AS $dept_audit$
DECLARE
  var_operacio char(1);
BEGIN
      IF TG_OP = 'DELETE' THEN
       var_operacio := 'D';
      ELSIF TG_OP = 'INSERT' THEN
       var_operacio := 'I';
      ELSEIF TG_OP = 'UPDATE' THEN
       var_operacio := 'U';
      END IF;

      INSERT INTO auditoria_depts values (user, now(),
                  var_operacio);
      RETURN NULL;
END;
$dept_audit$ LANGUAGE plpgsql;
```

Seguidament creem el trigger i li associem la funció anterior.

```
CREATE TRIGGER control_depts
    BEFORE INSERT OR DELETE OR UPDATE ON employees
    FOR EACH STATEMENT EXECUTE PROCEDURE process_dept_audit();
```

Possible resultat:

```
UPDATE employees
  SET  department_id = 3
WHERE  employee_id = 120;

SELECT * FROM auditoria_depts;

USUARI    DATA                                OPERACIÓ
---------------------------------------
postres   2022-01-10 15:34:05 U
```

VALORS DE RETORN D'UN TRIGGER

Les funcions de TRIGGER sempre es declaren amb una “RETURNS TRIGGER", però el que realment es torna és:

- per als TRIGGERS de nivell d’instrucció, el valor NULL.
- per als TRIGGERS de nivell de fila, una fila de la taula en què es defineix el disparador El valor de retorn s'ignora per als triggers de nivell de fila AFTER , de manera que també podem tornar NULL en aquest cas. Això deixa el nivell de fila BEFORE dels TRIGGERS com l'únic cas interessant.

En el nivell de fila BEFORE dels activadors, el valor de retorn té el significat següent:

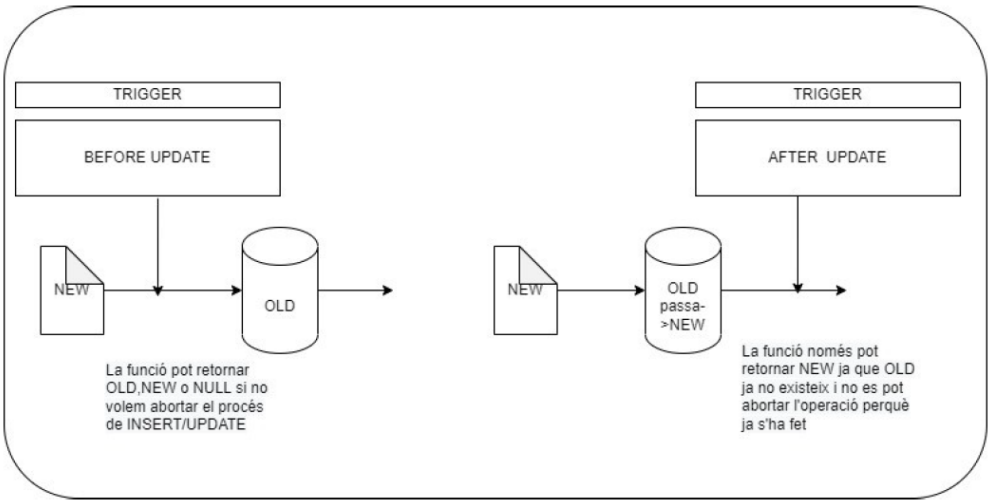
- si el disparador retorna NULL, l'operació d'activació s'aborta i la fila no es modificarà.
- per als activadors INSERT i UPDATE, la fila retornada és l'entrada per a la declaració DML que s'activa

Tinguem en compte també que podem tenir més d'un TRIGGER BEFORE a nivell de fila activat en una taula. En aquest cas, els TRIGGERS s'executen en l'ordre alfabètic del seu nom i el resultat de la funció TRIGGER anterior es converteix en l'entrada per a la següent funció TRIGGER.

NEW i OLD EN ELS TRIGGER DE FILA

Les variables especials NEW i OLD en una funció TRIGGER de nivell de fila contenen la versió nova(NEW) i antiga(OLD) de la fila. Es poden modificar i utilitzar en la declaració RETURN.
Tinguem en compte que NEW és NUL als TRIGGERS ON DELETE i OLD és NULL als TRIGGERS ON INSERT.

trigger invocation	NEW is set	OLD is set
ON INSERT	✓	
ON UPDATE	✓	✓
ON DELETE		✓



ADMINISTRACIÓ DE TRIGGERS

Esborrar triggers:

```
DROP TRIGGER [IF EXISTS] trigger_name
ON table_name [ CASCADE | RESTRICT ];
```

Desactivar un trigger:

```
ALTER TABLE table_name
DISABLE TRIGGER trigger_name | ALL
```

Activar un trigger:

```
ALTER TABLE table_name
ENABLE TRIGGER trigger_name | ALL
```