

Le langage procédural en SQL sous PostGres

PL/pgSQL
(**P**rocedural **L**anguage/**p**ost**g**reSQL **S**tructured **Q**uery **L**anguage)

A decorative graphic on the left side of the slide, consisting of a vertical column of blue and yellow hexagons of various sizes, some of which are slightly offset or overlapping.

Avertissement

- Cette partie du cours n'est qu'un survol du langage PL/pgSQL, utile pour écrire des procédures stockées simples
- Elle laisse de côté de nombreuses fonctionnalités du langage

Introduction au PL/pgSQL

■ PL/pgSQL

- (*Procedural Language/PostgreSQL Structured Query Language*) est un langage procédural géré par PostgreSQL. Ce langage est très similaire au PL/SQL d'Oracle.

■ Pourquoi PL/pgSQL ?

- SQL est un langage non procédural
- Les traitements complexes sont parfois très difficile à écrire si on ne peut utiliser des variables et les structures de programmation comme les boucles et les alternatives
- On ressent vite le besoin d'un langage procédural pour lier plusieurs requêtes SQL avec des variables et dans les structures de programmation habituelles

Introduction

■ Principales caractéristiques de PL/pgSQL

- PL langage de programmation côté serveur
- inspiré du PL/SQL d'Oracle
- Extension de SQL : des requêtes SQL cohabitent avec les structures de contrôle habituelles de la programmation structurée (blocs, alternatives, boucles)
- Des variables permettent l'échange d'information entre les requêtes SQL et le reste du programme

A decorative vertical pattern on the left side of the slide, consisting of various colored hexagons (blue, yellow, black) and dots arranged in a somewhat abstract, upward-pointing shape.

Introduction

■ Utilisation de PL/pgSQL

- PL/pgSQL peut être utilisé pour l'écriture des procédures stockées et des triggers
- Il convient aussi pour écrire des fonctions utilisateurs qui peuvent être utilisées dans les requêtes SQL (en plus des fonctions prédéfinies)

Structure d'un programme

- PL/pgSQL est un langage structuré en blocs. Le texte complet de la définition d'une fonction doit être un bloc. Un bloc est défini comme :

```
<<label>> As $$  
DECLARE déclarations de variables  
BEGIN  
    instructions ;  
END; $$ LANGUAGE plpgsql;
```

- Chaque déclaration et chaque expression au sein du bloc est terminée par un point-virgule
- les \$\$ permettent de délimiter la définition de la fonction (*sans cela le moteur SQL considèrera le point virgule au sein du bloc comme une fin de requête SQL et générera une erreur*)

Structure d'un programme

■ Exemple

■ définition de la fonction

```
CREATE OR REPLACE FUNCTION moySalaire() RETURNS NUMERIC AS $$  
    DECLARE  
        moyenne numeric;  
    BEGIN  
        SELECT AVG(id_formation) INTO moyenne FROM assurerpar;  
        RETURN moyenne ;  
    END; $$  
LANGUAGE plpgsql;
```

■ *Exemple d'appel de la fonction*

```
SELECT moySalaire() ;
```

Structure d'un programme

■ Un bloc peut contenir d'autres blocs

```
CREATE or replace FUNCTION somefunc() RETURNS integer AS $$  
  DECLARE quantity integer := 30;  
  BEGIN  
    RAISE INFO 'Qte est %', quantity; -- affiche 30  
    quantity := 50;  
    -- Crée un sous-bloc  
    DECLARE  
      quantity integer := 80;  
    BEGIN  
      RAISE INFO 'Qte est %', quantity; -- affiche 80  
    END;  
    RAISE INFO 'Quantity here is %', quantity; -- affiche 50  
    RETURN quantity;  
  END; $$  
LANGUAGE plpgsql;
```

Pour en savoir plus sur les erreurs et messages avec RAISE
<http://docs.postgresqlfr.org/9.5/plpgsql-errors-and-messages.html>

Les commentaires

Les commentaires comme dans tout langage sont très utiles voir obligatoire!!!!

-- *un commentaire sur une seule ligne*

/* *un commentaire sur plusieurs
lignes* */

A decorative vertical pattern on the left side of the slide, composed of various colored hexagons (blue, yellow, black, white) arranged in a cluster.

Les variables

■ Règles d'écriture

■ Identificateurs PostGres :

- 60 caractères au plus ;
- doit commencer par une lettre ;
- peut contenir lettres, chiffres, _ , \$ et #
- insensible à la casse
- Portée habituelle des langages à blocs
- Doivent être déclarées avant d'être utilisées



Les variables

■ Types des variables

- Les types habituels de Postgres : integer, varchar, boolean.....
- Types composites adaptés à la copie du type d'une colonne ou d'une ligne d'une table SQL :
%TYPE, %ROWTYPE

■ La syntaxe générale d'une déclaration de variable est :

name [CONSTANT] *type* [NOT NULL] [{ DEFAULT | :=} *expression*];

- La clause DEFAULT, spécifie la valeur initiale si pas indiquée, initialisée à la valeur SQL NULL.
- L'option CONSTANT empêche l'assignation de la variable.

Les variables

■ Déclaration d'une variable

Doit être défini dans la partie DECLARE de la fonction

□ Exemples simples:

```
age integer;  
nom varchar(30);  
dateNaissance date;  
ok boolean := true;
```

□ Exemples avec affectation de valeur par défaut :

```
quantity integer DEFAULT 32 ;  
url varchar := 'http://mysite.com' ;  
user_id CONSTANT integer := 10 ;
```

**!! Déclarations multiples
interdites : i, j integer;**

Les variables

■ Déclaration %TYPE

On peut déclarer qu'une variable est du même type qu'une colonne d'une table ou d'une vue :

■ Syntaxe :

name tablename.columnname%TYPE;

■ Exemple :

NomEmp emp.name%TYPE;

*La variable *NomEmp* sera du même type que la colonne *name* de la table *emp**

Les variables

■ Déclaration %ROWTYPE

Une variable de type composite est appelée variable ligne (ou variable row-type) elle peut contenir une ligne entière d'une table.

■ Syntaxe générale :

name table_name%ROWTYPE;

■ Exemple :

employe emp%ROWTYPE;

■ *la variable employe est une structure contenant le nom et le type des colonnes de la table emp*

■ Les colonnes de la ligne (row) sont accessibles en utilisant la notation pointée, par exemple :

employe.name

Les variables

■ Type RECORD

■ Syntaxe :

nom **RECORD;**

- Les variables record sont similaires aux variables de type ligne, mais n'ont pas de structure prédéfinie. Elles empruntent la structure effective de type ligne de la ligne à laquelle elles sont assignées durant une commande SELECT ou FOR. La sous-structure d'une variable record peut changer à chaque fois qu'on l'assigne. Une conséquence de cela est que jusqu'à ce qu'elle ait été assignée, elle n'a pas de sous-structure, et toutes les tentatives pour accéder à un de ses champs entraîneront une erreur d'exécution.

Les variables

■ Alias de paramètres de fonctions

- Chaque paramètre anonyme peut être identifié par `$n` où `n` est le numéro d'ordre du paramètre.
- Avant la version 8.0 il fallait explicitement déclarer l'alias : `nom ALIAS FOR $n`;

■ Syntaxe :

nom ALIAS FOR *\$n*; *n*: le nième paramètre

■ Exemple :

```
CREATE or replace FUNCTION prixTTC(prixHT real, real) RETURNS real AS $$  
DECLARE  
    taux ALIAS FOR $2; -- alias sur le 2ième paramètre de la fonction  
BEGIN  
    RETURN prixHT * (1+taux);  
END;  
$$ LANGUAGE plpgsql;
```


L'affectation

■ Affecter une valeur à une variable

- L'assignation d'une valeur à une variable ou à un champ row/record est écrite ainsi

identifiant := *expression*;

- Exemple :

dateNaissance := '2004-10-10';

L'affectation

■ Affecter une valeur à une variable

□ `SELECT source INTO destination`

Le résultat d'une commande SELECT manipulant plusieurs colonnes (**mais une seule ligne**) peut être assignée à une variable de type record ou ligne, ou une liste de valeurs scalaires.

```
SELECT col1, col2 INTO varcible1, varvarcible2  
FROM ...;
```

Exemple :

```
DECLARE  
  v_adr emp.empadr%TYPE;  
  v_prenom emp.empre%TYPE;  
BEGIN  
  SELECT emadr,empre INTO v_adr, v_prenom FROM emp  
  WHERE empname like p_myname;
```

!! Attention un SELECT doit toujours avoir une variable qui reçoit le résultat

L'affectation

- Le résultat d'une commande SELECT manipulant une seule colonne (**mais une seule ligne**) peut être assignée à une variable scalaire.

identifiant := (la requête);

Exemple :

```
v_prenom := (SELECT emprenom  
              FROM emp  
              WHERE id_emp=10);
```



L'affectation

- Le résultat d'une requête utilisé dans un test conditionnel :

if (la requête de comparaison) opérateur opérande

- Exemple :

```
IF (SELECT nom_formateur FROM formateur  
      WHERE id_formateur=p_id) = 'LEGRAND'  
THEN  
      .....  
END IF;
```

L'affectation

- ATTENTION :
UN SELECT DOIT TOUJOURS AVOIR UNE VARIABLE QUI REÇOIT LE RÉSULTAT

!!! A ne pas faire !!!!

```
CREATE or replace FUNCTION name_form(p_id int) RETURNS void AS $$  
  DECLARE  
    v_form formateur%ROWTYPE;  
  BEGIN  
    SELECT * FROM formateur  
      WHERE id_formateur=p_id ;  
    .....;  
  END; $$ LANGUAGE plpgsql;
```

Cette requête ne sert à rien : aucune variable ne reçoit le résultat et elle n'est pas utilisée dans un test logique... Elle génère une erreur

Erreur SQL :

ERREUR: la requête n'a pas de destination pour les données résultantes



L'affectation

■ Utilisation de PERFORM

- Exemple : pour savoir si une requête retourne une ligne

```
CREATE or replace FUNCTION name_form(p_id int) RETURNS void AS $$  
DECLARE  
BEGIN
```

```
    PERFORM * FROM formateur  
    WHERE id_formateur = p_id;
```

```
    if FOUND then -- FOUND retourne VRAI si la requête précédente retourne au moins une ligne  
        raise exception 'il existe';  
    else  
        raise exception 'il n'existe pas';  
    end if;  
END; $$ LANGUAGE plpgsql;
```

Les variables

■ Un exemple d'utilisation des types composites :

-- définition de la fonction

```
CREATE or replace FUNCTION name_form(p_id int) RETURNS text AS $$  
  DECLARE  
    form formateur%ROWTYPE;  
  BEGIN  
    SELECT * INTO form  
      FROM formateur  
      WHERE id_formateur=p_id ;  
    RETURN form.nom_formateur||' '|| form.prenom_formateur;  
  END; $$ LANGUAGE plpgsql;
```

-- appel de la fonction

```
select name_form(1);
```

Exécuter des commandes

■ EXECUTE chaîne-commande

chaîne-commande est une expression manipulant une chaîne (de type text) contenant la commande à exécuter.

Cette chaîne est littéralement donnée à manger au moteur SQL.

Exemple :

EXECUTE 'UPDATE tbl

SET **// quote_ident(colname) //** ' = ' quote_literal(p_name)

// 'WHERE ... ' ;

quote_ident : contient les variables contenant les identifiants de colonne et de table

quote_literal : contient les variables contenant les valeurs de type chaînes de caractères dans la commande construite

Elles effectuent les traitements appropriés pour renvoyer le texte entré, enfermé entre doubles ou simples guillemets respectivement, chaque caractère spécial correctement échappé

Exécuter des commandes

■ EXECUTE chaîne-commande ;

Exemple :

```
EXECUTE 'UPDATE tbl SET '      // quote_ident(colname)  
                                // ' = ' quote_literal(p_name)  
                                // 'WHERE ... ';
```

si la variable *colname* = nom_cli et *p_name* = 'Dupond'

La requête exécutée sera :

```
UPDATE tbl  
SET nom_cli= 'Dupond'  
WHERE ... ;
```

Structures de contrôle

■ Contrôles conditionnels :

PL/pgSQL a quatre formes de IF :

```
IF ... THEN ... END IF;
```

```
IF ... THEN ... ELSE ... END IF;
```

```
IF ... THEN ... ELSE IF ... END IF .. END IF;
```

```
IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF .. END IF;
```

■ Exemples :

```
IF parentid IS NULL OR parentid = ""  
    THEN RETURN vname ;  
    ELSE RETURN parentid || '/' || vname;  
END IF;
```

Structures de contrôle

■ Contrôles conditionnels :

Le CASE WHEN END CASE;

```
CASE WHEN ...  
  THEN .....  
  ELSE .....  
END CASE;
```

■ Exemples :

.....

```
CASE x WHEN 1, 2  
  THEN msg := 'un ou deux';  
  ELSE msg := 'autres valeurs que un ou deux';  
END CASE;  
Return msg;
```

.....

Structures de contrôle

■ Boucles Simples

LOOP : c'est une boucle inconditionnelle répétée indéfiniment jusqu'à ce qu'elle soit terminée par une instruction EXIT ou RETURN.

■ Syntaxe :

```
[ << label >> ] LOOP instructions END LOOP;
```

■ Exemple :

```
LOOP
```

```
... traitements
```

```
IF count > 0 THEN EXIT; -- sortie de boucle
```

```
END IF;
```

```
END LOOP;
```

```
LOOP -- Autre méthode pour sortir de la boucle
```

```
... traitements
```

```
EXIT WHEN count > 0;
```

```
END LOOP;
```

Il est obligatoire d'avoir une instruction EXIT pour sortir de la boucle

Structures de contrôle

■ Boucles Simples

■ Autre Exemple en utilisant un label:

```
CREATE or REPLACE function test(x int) returns text as $$  
DECLARE v_int int := x; y int :=0;  
BEGIN  
  <<loop1>>  
  LOOP  
    LOOP  
      EXIT loop1 when y > 4;  
      y := y+1;  
    END LOOP;  
    v_int := v_int+1;  
  END LOOP;  
  RETURN 'fini = ' || v_int;  
END; $$ LANGUAGE plpgsql;
```

L'appel à la fonction :

select test(1);

Renverra :

```
test  
fini = 1
```

Structures de contrôle

■ Boucles Simples

WHILE : L'instruction WHILE répète une séquence d'instructions aussi longtemps que l'expression conditionnelle est évaluée à vrai.

■ Syntaxe :

```
WHILE expression LOOP instructions END LOOP;
```

■ Exemple :

```
WHILE montant > 0 AND balance > 0  
LOOP  
    -- quelques traitements ici  
END LOOP;
```

```
-- Autre exemple  
WHILE NOT (prix = 0) LOOP  
    -- quelques traitements ici  
END LOOP;
```

Bien entendu comme dans n'importe quel autre langage vous devez faire évoluer les variables qui sont testées dans la condition pour espérer sortir de la boucle

Structures de contrôle

■ Boucles Simples :

FOR : Cette forme de FOR crée une boucle qui effectue une itération sur une plage de valeurs entières.

■ Syntaxe :

```
FOR nom IN [ REVERSE ] Debut.. fin  
  LOOP instruction END LOOP;
```

■ Exemple :

```
FOR i IN 1..10 LOOP -- progression croissante  
  RAISE NOTICE 'i is %', i;  
END LOOP;
```

-- *Autre exemple : itération décroissante*

```
FOR i IN REVERSE 10..1 LOOP -- progression décroissante  
  -- quelques traitements ici  
END LOOP;
```

Structures de contrôle

■ Boucler dans le résultat d'une requête

En utilisant un type de FOR différent, vous pouvez itérer au travers des résultats d'une requête et par là même manipuler ces données.

■ Syntaxe :

```
FOR record_ou_ligne IN requête  
LOOP  
    instructions  
END LOOP;
```

-- requête doit être une commande SELECT

-- La requête est exécutée une seule fois

-- Attention pas de point virgule(;) après la requête

Structures de contrôle

■ Boucler dans le résultat d'une requête

■ Exemple :

```
CREATE or replace FUNCTION copy_form(p_id int, p_temp varchar(20)) RETURNS void AS $$  
DECLARE use_form formateur%ROWTYPE;  
BEGIN  
    EXECUTE 'CREATE TABLE ' || quote_ident(p_temp)  
            || '(id_f integer, nom varchar(50));'  
  
    FOR use_form IN SELECT * FROM formateur where id_formateur < p_id  
    LOOP  
        EXECUTE 'INSERT INTO ' || quote_ident(p_temp) || '  
VALUES( ' || use_form.id_formateur || ',  
        || quote_literal(use_form.nom_formateur) || ');'  
  
    END LOOP;  
END; $$ LANGUAGE plpgsql;
```



Structures de contrôle

- Boucler dans les résultats d'une requête :

```
FOR record_ou_ligne IN EXECUTE expression_texte  
LOOP  
    instructions  
END LOOP;
```

Ceci est identique à la forme précédente, à ceci près que l'expression SELECT source est spécifiée comme une expression chaîne, évaluée et exécutée à chaque entrée dans la boucle FOR.

Les arguments d'une fonction

■ Introduction

- Vous pouvez utiliser tous les types classiques pour définir un argument
- Les arguments d'une fonction SQL peuvent être référencés dans le corps de la fonction en utilisant soit les noms soit les numéros.
- Si un argument est de type composite, la notation à point, `nom_argument.nom_champ` ou `$1.nom_champ` peut être utilisé pour accéder aux attributs de l'argument.

Les arguments d'une fonction

■ Exemple

```
CREATE FUNCTION add_em(x integer, y integer) RETURNS integer AS $$  
Begin  
    Return x + y;  
End; $$ LANGUAGE SQL;
```

-- Ou avec le N° d'ordre de déclaration

```
CREATE FUNCTION add_em( integer, integer) RETURNS integer AS $$  
Begin  
    Return $1 + $2;  
End; $$ LANGUAGE SQL;
```

Les arguments d'une fonction

■ Fonctions avec des valeurs par défaut pour les arguments

- Les fonctions peuvent être déclarées avec des valeurs par défaut pour certains des paramètres en entrée ou pour tous.
- Les valeurs par défaut sont insérées quand la fonction est appelée avec moins d'arguments que à priori nécessaires.
- Comme les arguments peuvent seulement être omis à partir de la fin de la liste des arguments, tous les paramètres après un paramètre disposant d'une valeur par défaut disposeront eux-aussi d'une valeur par défaut.

Les arguments d'une fonction

■ Fonctions avec des valeurs par défaut pour les arguments

■ Par exemple :

```
CREATE FUNCTION somme(x numeric, y numeric default 2, z numeric default 2) RETURNS  
numeric AS $$  
BEGIN  
    RETURN x + y + z ;  
END; $$ LANGUAGE plpgsql;
```

Les appels suivants retournent :

SELECT somme(3,5,6);

somme
14

SELECT somme(3,5);

somme
10

SELECT somme(3);

somme
7

Retour d'une fonction

- Une fonction peut retourner une valeur, une table ou le résultat d'une requête ;
 - RETURN *expression*;
 - RETURN NEXT *expression* ;
 - RETURN QUERY *requête* ;

Retour d'une fonction (une seule ligne)

- RETURN *expression* (avec RETURNS type);

- Syntaxe générale

```
CREATE OR REPLACE FUNCTION nom_fonct() RETURNS type AS $$  
DECLARE .....  
BEGIN  
    .....  
    RETURN expression;  
END; $$ LANGUAGE plpgsql;
```

- La fonction ne peut retourner plus d'une ligne !
- RETURN accompagné d'une expression termine la fonction et renvoie le valeur d'expression à l'appelant
- Le type de retour de la fonction peut être :
 - Un scalaire : integer, varchar ...
 - Un type composite: un record, une table

Retour d'une fonction (une seule ligne)

- RETURN *expression* (avec RETURNS type);
 - Appel de la fonction qui retourne un scalaire :

- Syntaxe :

SELECT *nom_fonction*(paramètres ...);

- Exemple :

```
CREATE OR REPLACE FUNCTION mon_int() RETURNS integer AS $$  
DECLARE  
    v_prix integer DEFAULT 34;  
BEGIN  
    RETURN v_prix;  
END; $$ LANGUAGE plpgsql;
```

-- appel de la fonction

```
SELECT mon_int();
```

```
SELECT * FROM voiture WHERE prix_modele < mon_int();
```

Retour d'une fonction (une seule ligne)

- RETURN *expression* (avec RETURNS type);
 - Appel de la fonction qui retourne une table (la table dans le schéma de données) :

- Syntaxe :

```
SELECT * from nom_fonction(paramètres ...);
```

- Exemple :

```
CREATE or replace FUNCTION mon_film() RETURNS film AS $$  
DECLARE  
    v_table film%ROWTYPE;  
BEGIN  
    SELECT * INTO v_table FROM film WHERE idfilm=1;  
    RETURN v_table ;  
END; $$ LANGUAGE plpgsql;
```

-- appel de la fonction

```
SELECT * FROM mon_film() ;
```

Retour d'une fonction (une seule ligne)

- RETURN *expression* (avec RETURNS type);
 - Appel de la fonction qui retourne un record :

- Syntaxe :

```
SELECT * from nom_fonction(paramètres ...)  
as (nom1 type, nom2 type ...);
```

- Exemple :

```
CREATE or replace FUNCTION mon_film() RETURNS record AS $$  
DECLARE  
    v_ligne record;  
BEGIN  
    SELECT titre,annee, genre INTO v_ligne FROM film WHERE idfilm=1;  
    RETURN v_ligne ;  
END; $$ LANGUAGE plpgsql;
```

-- appel de la fonction

```
SELECT * FROM mon_film() AS (tit varchar, an date, gr varchar);
```

tit	an	gr
Crash	1996-06-01	Drame

Retour d'une fonction (plus d'une ligne)

- RETURN NEXT *expression* (avec RETURNS SETOF type);

```
CREATE OR REPLACE FUNCTION nom_fonct() RETURNS SETOF type AS $$  
DECLARE .....  
BEGIN  
    .....  
    RETURN NEXT expression;  
    .....  
    RETURN;  
END; $$ LANGUAGE plpgsql;
```

- Lorsqu'une fonction PL/pgSQL est déclarée pour renvoyer un *SETOF type* quelconque, la procédure à suivre est légèrement différente. Dans ce cas, les items individuels à renvoyer sont spécifiés dans les commandes RETURN NEXT, et ensuite une commande RETURN finale, sans arguments est utilisée pour indiquer que la fonction a terminé son exécution.
- RETURN NEXT peut renvoyer des types scalaires et des types composites de données;

Retour d'une fonction (plus d'une ligne)

- RETURN NEXT *expression* (avec RETURNS SETOF type);
 - Appel de la fonction qui retourne un scalaire (integer, varchar ...):

- Syntaxe :

SELECT *nom_fonction*(paramètres ...)
OU SELECT * FROM *nom_fonction*(paramètres ...)

- Exemple :

```
CREATE or replace FUNCTION mon_annee() RETURNS SETOF date AS $$  
DECLARE i date;  
BEGIN  
    FOR i IN SELECT annee FROM film  
    LOOP RETURN NEXT i; -- retourne la valeur courant de i  
    END LOOP;  
    RETURN;  
END; $$ LANGUAGE plpgsql;
```

-- appel de la fonction

SELECT * FROM mon_annee(); Ou SELECT mon_annee();

mon_annee
2002-08-24
1998-08-18
1994-08-18
1988-08-10

Retour d'une fonction (plus d'une ligne)

- RETURN NEXT *expression* (avec RETURNS SETOF type);
 - Appel de la fonction qui retourne un type composé (un record ou un %ROWTYPE) :

- Syntaxe :

```
SELECT * FROM nom_fonction() AS (nom1 type, .....)
```

- Exemple :

```
CREATE or replace FUNCTION mes_films() RETURNS SETOF record AS $$  
DECLARE var record;  
BEGIN  
    FOR var IN SELECT titre,annee, genre FROM film  
    LOOP RETURN NEXT var;  
    END LOOP;  
    RETURN;  
END; $$ LANGUAGE plpgsql;
```

-- appel de la fonction

```
SELECT * FROM mes_films() AS (tit varchar, an date, gr varchar);
```

tit	an	gr
Dogville	2002-06-24	Drame
Breaking the waves	1996-06-18	Drame
Pulp Fiction	1994-06-16	Policier

Retour d'une fonction (plus d'une ligne)

- RETURN NEXT *expression* (avec RETURNS SETOF type);
 - Appel de la fonction qui retourne un type composé (une table ou %ROWTYPE) :

- Syntaxe :

```
SELECT * FROM nom_fonction()
```

- Exemple :

```
CREATE or replace FUNCTION mes_films() RETURNS SETOF film AS $$  
DECLARE var film%ROWTYPE;  
BEGIN  
    FOR var IN SELECT * FROM film  
    LOOP RETURN NEXT var;  
    END LOOP;  
    RETURN;  
END; $$ LANGUAGE plpgsql;
```

-- appel de la fonction

```
SELECT * FROM mes_films();
```

num_film	num_ind	titre	genre	annee
5	13	Dogville	Drame	2002-08-24
4	13	Breaking the waves	Drame	1998-08-18

Retour d'une fonction (plus d'une ligne)

- RETURN QUERY *expression* (avec RETURNS SETOF type);

```
CREATE OR REPLACE FUNCTION nom_fonct() RETURNS SETOF type AS $$  
DECLARE
```

```
.....
```

```
BEGIN
```

```
.....
```

```
    RETURN QUERY requête;
```

```
.....
```

```
END; $$ LANGUAGE plpgsql;
```

- RETURN QUERY retourne le résultat d'une requête, il est obligatoirement associé à un *SETOF type*, le *type* dépend de ce qui est retourné par la requête :
 - un scalaire
 - un type composite
- RETURN QUERY n'a finalement peut d'intérêt puisqu'il retourne le résultat d'une requête

Retour d'une fonction (plus d'une ligne)

- RETURN QUERY *expression* (avec RETURNS SETOF type);
 - Appel de la fonction qui retourne un type composé (une table) :
 - Syntaxe :

```
SELECT * FROM nom_fonction();
```

- Exemple :

```
CREATE or replace FUNCTION mes_films() RETURNS SETOF film AS $$  
BEGIN  
    RETURN QUERY SELECT * FROM film;  
END; $$ LANGUAGE plpgsql;
```

-- *appel de la fonction*

```
SELECT * FROM mes_films();
```

ou

```
SELECT titre, genre FROM mes_film();
```

titre	genre
Dogville	Drame
Breaking the waves	Drame
Pulp Fiction	Policier

Retour d'une fonction (plus d'une ligne)

- RETURN QUERY *expression* (avec RETURNS SETOF type);
 - Appel de la fonction qui retourne un type composé (un record) :

- Syntaxe :

```
SELECT * FROM nom_fonction() AS (nom1 type, .....);
```

- Exemple :

```
CREATE or replace FUNCTION mes_films() RETURNS SETOF record AS $$  
BEGIN  
    RETURN QUERY SELECT titre, annee, genre FROM film;  
END; $$ LANGUAGE plpgsql;
```

-- appel de la fonction

```
SELECT * FROM mes_films()  
AS (tit varchar, an date, gr varchar);
```

tit	an	gr
Dogville	2002-06-24	Drame
Breaking the waves	1996-06-18	Drame
Pulp Fiction	1994-06-16	Policier

Retour d'une fonction *(en utilisant des paramètre en sortie)*

■ Fonctions SQL avec des paramètres en sortie

- Une autre façon de décrire les résultats d'une fonction est de la définir avec des paramètres en sortie comme dans cet exemple :

```
CREATE FUNCTION somme(IN x numeric, IN y numeric , OUT somme numeric) AS $$  
BEGIN
```

```
    somme := x + y ;
```

```
END; $$ LANGUAGE plpgsql;
```

- **Attention** définir un paramètre comme paramètre de sortie, implique de ne pas mettre de RETURN dans la fonction et pas de RETURNS type avant le AS
- L'appel à la fonction sera ***select somme(4,7)*** par exemple
- Vous pouvez omettre le IN devant les paramètres d'entrée (par défaut en entrée)

Messages et erreurs

- L'instruction RAISE permet de renvoyer différent niveau de message

- Syntaxe :

RAISE niveau 'texte', [expr[,expr]]

niveau ∈ {DEBUG, LOG, INFO, NOTICE, WARNING, EXCEPTION}

texte est une chaîne où % est remplacé par l'expression (expr) correspondante.

- Exemple :

RAISE NOTICE 'taux négatif % et prix = %', taux, prix;

La génération d'une exception interrompt la transaction en cours

Gestion des erreurs

■ Récupération des erreurs

```
[ <<label>> ]  
[ DECLARE declarations ]  
BEGIN  
instructions  
EXCEPTION WHEN condition [ OR condition ... ]  
THEN instructions_gestion_erreurs  
[ WHEN condition [ OR condition ... ] THEN  
    instructions_gestion_erreurs ... ]  
END;
```

Voir les [Codes d'erreurs de PostgreSQL](#).

Gestion des erreurs

■ Exemple

```
CREATE OR REPLACE FUNCTION formateurs(p_diviseur int) RETURNS void AS $$
DECLARE
    x integer :=30;
    y integer;
BEGIN
    y := x / p_diviseur ;
    INSERT INTO formateur VALUES (3,'Morel','Paul',45);
    EXCEPTION
        WHEN division_by_zero THEN /* erreur possible sur la division */
            RAISE EXCEPTION 'Erreur division_by_zero';
        WHEN unique_violation THEN /* erreur possible sur l'INSERT */
            UPDATE formateur
                SET (nom_form, prenom_form, nbh)=('Morel','Paul',45)
                WHERE id_formateur=3;
    RETURN ;
END; $$ LANGUAGE plpgsql;
```

Gestion des erreurs

■ 1er appel

`Select formateurs(3) ;`

Mise à jour du formateur de la table formateur

■ 2ème appel

`Select formateurs(0) ;`

La division par zéro déclenche l'exception

Le INSERT INTO formateur ... ne sera pas exécuté

Erreur SQL :

ERREUR: Erreur division_by_zero

Dans l'instruction :

`Select formateurs(0) ;`

Les curseurs

■ Principe :

- Structure du langage qui permet de manipuler les résultats de requêtes ligne par ligne;
- Il s'agit de préparer des requêtes qui seront ensuite exécutées et dont on pourra parcourir le résultat dans une boucle ;
- Fonctionne comme une tête de lecture sur les résultats de requêtes.

Les curseurs

■ Déclaration d'un curseur

DECLARE

`curseur1` *refcursor*; -- à définir lors de l'ouverture

`curseur2` CURSOR FOR SELECT ...;

`curseur3` CURSOR(*key integer*) IS SELECT ... WHERE id=*key*;

- Ces variables sont toutes du type de données *refcursor*, mais la première peut être utilisées avec n'importe quelle requête, alors que la seconde a une requête complètement spécifiée qui lui est déjà liée, et la dernière est liée à une requête paramétrée. (*key* sera remplacée par un paramètre de valeur entière lors de l'ouverture du curseur.)
- La variable `curseur1` est dite non liée puisqu'elle n'est pas liée à une requête particulière, il faudra la définir lors de l'ouverture du curseur
- Les curseurs associés à une requête sont qualifiés de curseur lié (*bound_cursor*)

Les curseurs

■ Ouverture et fermeture d'un curseur

- Avant toute utilisation le curseur doit être ouvert

- Ouverture d'un curseur lié (*ouvre et exécute la requête associée*):

```
OPEN Curseur2 ;  
OPEN Curseur3(42);
```

- Ouverture d'un curseur NON lié :

- *OPEN FOR SELECT*

```
OPEN Curseur1 FOR SELECT * FROM foo WHERE key = p_ykey;
```

- *OPEN FOR EXECUTE*

```
OPEN Curseur1 FOR EXECUTE 'SELECT * FROM '||quote_ident($1);
```

- Après utilisation le curseur doit être fermé

```
CLOSE Curseur1 ;  
CLOSE Curseur2 ;  
CLOSE Curseur3 ;
```

Les curseurs

■ Utilisation d'un curseur

Une fois qu'un curseur a été ouvert, il peut être manipulé grâce à l'instruction suivante :

```
FETCH curseur1 INTO cible;
```

FETCH retourne la ligne courante du résultat du curseur dans une cible, qui peut être une variable ligne, une variable record ou une liste de simples variables séparées d'une virgule, exactement comme SELECT INTO. Comme pour SELECT INTO, la variable spéciale FOUND peut être vérifiée pour voir si une ligne a été obtenue ou pas.

■ Exemple :

```
FETCH curseur1 INTO var_ligne;  
FETCH curseur2 INTO foo, bar, baz;
```

Il existe d'autres paramètres à l'instruction fetch

<http://docs.postgresqlfr.org/9.1/sql-fetch.html>

Les curseurs

■ Exemple d'utilisation d'un curseur

```
CREATE or replace FUNCTION copy_form curseur(p_id int) RETURNS void AS $$
DECLARE
    use_form formateur%ROWTYPE;
    -- déclaration d'un curseur paramétré
    curs1 CURSOR (key integer) for SELECT * FROM formateur WHERE id_formateur < key;
BEGIN
    EXECUTE 'drop TABLE temp';
    EXECUTE 'CREATE TABLE temp (id_f integer, nom varchar(50))';
    OPEN curs1 (p_id); -- ouverture du curseur paramétré
    LOOP -- parcours du curseur paramétré
        FETCH curs1 INTO use_form;
        EXIT WHEN NOT FOUND; -- sort de la boucle:le curseur ne retourne plus rien
        EXECUTE 'INSERT INTO temp VALUES('||use_form.id_formateur||','||
            quote_literal(use_form.nom_formateur)||')';
    END LOOP;
    Close curs1;
END; $$ LANGUAGE plpgsql;
```

Les triggers -Procédures déclenchées-

■ Définition

- Un trigger est déclenché avant ou après un INSERT, un UPDATE ou un DELETE.
- Un trigger utilise une fonction trigger, ce sont des fonctions particulières sans paramètre et de type trigger
- Lors de l'exécution d'un trigger, des variables spéciales sont créées automatiquement : NEW, OLD, ...
- Après la définition de la fonction, il faut encore définir le trigger lui-même avec CREATE TRIGGER ...

Attention il faut d'abord définir la fonction trigger avant le trigger

Les triggers -Procédures déclenchées-

■ Création d'un trigger

■ Syntaxe

```
CREATE TRIGGER nom { BEFORE | AFTER } { evenement [ OR ... ] }  
ON table [ FOR [ EACH ] { ROW | STATEMENT } ]  
EXECUTE PROCEDURE nomfonc ( arguments )
```

nom : Le nom du nouveau déclencheur.

BEFORE | AFTER : Détermine si la fonction est appelée avant ou après l'événement.

evenement : événement déclencheur (INSERT, UPDATE ou DELETE);

table : nom de la table à laquelle est rattaché le déclencheur.

FOR EACH ROW | FOR EACH STATEMENT : Précise si la procédure du déclencheur doit être lancée pour chaque ligne affectée par l'événement ou simplement pour chaque instruction SQL. FOR EACH STATEMENT est la valeur par défaut.

nomfonc : nom de la fonction utilisateur, déclarée sans argument et renvoyant le type trigger, exécutée à l'activation du déclencheur.

arguments : Une liste optionnelle d'arguments séparés par des virgules à fournir à la fonction lors de l'activation du déclencheur.

Les triggers -Procédures déclenchées-

■ Création d'une fonction trigger

□ Syntaxe :

```
CREATE FUNCTION nomfonc() RETURNS trigger AS $$  
BEGIN  
    ...  
    RETURN NEW;  
END; $$ LANGUAGE plpgsql;
```

- Une fonction trigger doit renvoyer soit NULL soit une valeur record ayant exactement la structure de la table pour laquelle le trigger a été lancé.

Les triggers -Procédures déclenchées-

■ Création d'une fonction trigger

- Quand une fonction PL/pgSQL est appelée en tant que trigger, plusieurs variables spéciales sont créées :

NEW : variable contenant la nouvelle ligne pour les opérations INSERT/UPDATE dans les triggers de niveau ligne (*FOR EACH ROW*). Cette variable est NULL dans un trigger de niveau instruction (*FOR EACH STATEMENT*)

OLD : variable contenant la ligne affectée pour les opérations UPDATE/DELETE dans les triggers de niveau ligne. Cette variable est NULL dans les triggers de niveau instruction.

TG_NAME: nom du trigger réellement lancé.

TG_WHEN : BEFORE soit AFTER, selon la définition du trigger.

TG_LEVEL : ROW soit STATEMENT, selon la définition du trigger.

TG_OP : INSERT, UPDATE ou DELETE, indiquant pour quelle opération le trigger a été lancé.

TG_RELID : identifiant de l'objet ayant déclenché le trigger.

TG_TABLE_NAME : nom de la table qui a déclenché le trigger.

TG_TABLE_SCHEMA : nom du schéma de la table qui a appelé le trigger.

TG_NARGS: le nombre d'arguments donnés à la procédure trigger dans l'instruction CREATE TRIGGER.

TG_ARGV[] Type de donnée text ; les arguments de l'instruction CREATE TRIGGER. L'index débute à 0.

Les triggers -Procédures déclenchées-

■ Création d'une fonction trigger

□ Les structures OLD et NEW:

- Les structures **OLD** et **NEW** sont générées automatiquement lors du déclenchement du trigger en suivant les règles suivantes :
 - Les triggers de niveau ligne (FOR EACH ROW) lancés BEFORE peuvent renvoyer NULL pour indiquer au gestionnaire de trigger de ne pas exécuter l'instruction SQL (INSERT/UPDATE/DELETE) qui a déclenché le trigger pour cette ligne.
 - Si une valeur non NULL est renvoyée alors l'instruction SQL se déroule avec cette valeur ligne. Renvoyer une valeur ligne différente de la valeur originale de NEW modifie la ligne qui sera insérée ou mise à jour (mais n'a pas d'effet sur le cas DELETE).
 - Pour modifier la ligne à stocker, il est possible de remplacer des valeurs seules directement dans NEW et de renvoyer NEW, ou de construire un nouveau record/ligne à renvoyer.
 - Les triggers de niveau instruction (FOR EACH STATEMENT) BEFORE ou AFTER ou un trigger de niveau ligne AFTER ignore toujours la valeur de retour ;

Les triggers -Procédures déclenchées-

■ Exemple

□ Définition de la fonction trigger

```
CREATE or replace FUNCTION emp_stamp() RETURNS trigger AS $$
BEGIN -- Verifie que nom_employe et salaire sont donnés
    IF NEW.nom_employe IS NULL
        THEN RAISE EXCEPTION ' l'employe doit avoir un nom';
    END IF;
    IF NEW.salaire IS NULL
        THEN RAISE EXCEPTION '% doit avoir un salaire', NEW.nom_employe;
    END IF;
    -- Qui travaille pour ce salaire ?
    IF NEW.salaire < 0 THEN
        RAISE EXCEPTION '% doit avoir un salaire positif', NEW.nom_employe;
    END IF; -- qui a changé le salaire et quand
    NEW.date_dermodif := current_timestamp;
    NEW.utilisateur_dermodif := current_user;
    RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

Colonne	Type
nom_employe	text
salaire	integer
date_dermodif	timestamp without time zone
utilisateur_dermodif	text

Les triggers -Procédures déclenchées-

■ Exemple

□ Définition du trigger

```
CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp FOR EACH  
ROW EXECUTE PROCEDURE emp_stamp();
```

□ Test du déclenchement du trigger

```
INSERT INTO emp VALUES('Carmignac', -4000, null, null);  
INSERT INTO emp VALUES('Carmignac', 4000, null, null);  
-- insertion réalisée avec succès
```

```
UPDATE emp SET salaire = -3000  
WHERE nom_employe LIKE 'Carmignac'; -- provoque une erreur
```

Erreur SQL :

ERROR: Carmignac doit avoir un salaire positif

Dans l'instruction :

```
insert into emp values('Carmignac',-4000,null,null);
```

Les triggers -Procédures déclenchées-

■ Autre exemple :

■ tracer les modifications sur une table

- Nous allons tracer dans une table `log_formation` les événements survenues sur la table `formation`, cad savoir si il a eu une insertion, modification ou suppression, a quelle date et heure et qui est l'auteur de l'événement

La table log :

```
CREATE TABLE log_formation (  
    id_formation int,  
    intitule_form TEXT,  
    nbheures int default '0',  
    niveau varchar(50) default NULL,  
    Even char(1),  
    date_even timestamp,  
    user_even text ) ;
```

Les triggers -Procédures déclenchées-

■ Autre exemple :

■ tracer les modifications sur une table

```
CREATE or replace FUNCTION log_formation() RETURNS trigger AS $$  
BEGIN -- Verifie que nom_employe et salary sont donnés  
IF (TG_OP = 'DELETE') THEN  
    INSERT INTO log_formation SELECT OLD.*, 'D', current_timestamp, user;  
    RETURN OLD;  
ELSIF (TG_OP = 'UPDATE') THEN  
    INSERT INTO log_formation SELECT NEW.*, 'U', current_timestamp, user;  
    RETURN NEW;  
ELSIF (TG_OP = 'INSERT') THEN  
    INSERT INTO log_formation SELECT NEW.*, 'I', current_timestamp, user;  
    RETURN NEW;  
END IF;  
    RETURN NULL;  
END; $$ LANGUAGE plpgsql;
```

Les triggers -Procédures déclenchées-

- Autre Exemple :
 - tracer les modifications sur une table

- Définition du trigger

```
CREATE TRIGGER trigger_log_formation AFTER INSERT OR DELETE OR UPDATE ON formation  
FOR EACH ROW EXECUTE PROCEDURE log_formation();
```

- Test du déclenchement du trigger

```
INSERT INTO formation
```

```
values (50,'Formation pl/pgSQL',45,'Confirme');
```

```
UPDATE formation SET nbheures=55 WHERE id_formation=50;
```

```
DELETE FROM formation WHERE id_formation=50;
```

id_formation	intitule_form	nbheures	niveau	even	date_even	user_even
50	Formation pl/pgSQL	45	Confirme I		2011-11-17 10:50:30.179075	carmignac.l
50	Formation pl/pgSQL	55	Confirme U		2011-11-17 10:50:30.179075	carmignac.l
50	Formation pl/pgSQL	55	Confirme D		2011-11-17 10:50:30.179075	carmignac.l