

**SHAKE** THE FUTURE



# Bases de Données

**PL/SQL**

JY Martin

# Plan

- 1 Le contexte
- 2 Elements de syntaxe
- 3 Procédures et fonctions
- 4 PostgreSQL
- 5 Conclusion

# SQL

- SQL est un langage non procédural
- Les traitements complexes sont parfois difficiles à écrire.
  - Pas de variables
  - Pas de passage d'information d'une requête à l'autre
  - Pas de boucles, de tests

-> Nécessité d'un langage procédural ?

Objectif : Traduire le modèle conceptuel en modèle relationnel

## Pour quoi faire ?

- Ecriture de **procédures stockées** et de **triggers** (Certains SGBD comme Oracle acceptent aussi le langage JAVA)
- Ecriture de **fonctions utilisateur** pouvant être utilisées dans les requêtes SQL (en plus des fonctions prédéfinies)
- Implémenter certains types de SGBD (SIG)

# Un nouveau langage ?

## PL :Programming Language

- PL/SQL : Extension de SQL  
Faire cohabiter des requêtes SQL et des structures de contrôle habituelles de la programmation structurée (blocs, alternatives, boucles)
- Un programme est constitué de procédures et de fonctions
- Des variables permettent l'échange d'information entre les requêtes SQL et le reste du programme

# PL/SQL

PL/SQL est un langage propriétaire d'Oracle

PostgreSQL utilise un langage très proche : plpgsql

NB : Tous les langages L4G des différents SGBDs se ressemblent

# Plan

- 1 Le contexte
- 2 Elements de syntaxe
- 3 Procédures et fonctions
- 4 PostgreSQL
- 5 Conclusion

## Bloc d'instructions

3 types de blocs d'instructions :

- procédures anonymes
- procédures nommées
- fonctions nommées

Un bloc peut contenir d'autres blocs



## Structure d'un bloc

```
DECLARE
– définitions de variables
BEGIN
– Les instructions à exécuter
    EXCEPTION
    – La récupération des erreurs
END;
```

Les mots clés **BEGIN** et **END** sont obligatoires. **DECLARE** n'est utilisé que s'il y a des variables à déclarer.

# Les instructions

- Se terminent par ;
- Affectations
  - Instruction : variable := valeur
  - Résultat d'une requête : SELECT ... INTO variable
- Instructions de contrôle  
Boucles, tests, ...
- Appels de fonctions, de procédures

## Les variables

- Identificateurs :
  - 30 caractères au plus
  - commence par une lettre
  - peut contenir lettres, chiffres, \_, \$ et #
- Pas sensible à la casse
- Portée habituelle des langages à blocs
- Doivent être déclarées avant d'être utilisées.
- Sont typées.

## Les types de variables

- Les types utilisés par le SGBD support :  
integer, varchar,...
- Les types composites pour la récupération des colonnes et des lignes des tables SQL :  
%TYPE, %ROWTYPE
- Le type référence : REF

## Déclaration d'un variable

- Dans le bloc DECLARE
- Forme :  
Identificateur [CONSTANT] type [ := valeur ] ;
- **Déclaration multiples interdites** un par un

Exemples :

```
age integer ;  
nom varchar(20);  
dateNaissance date ;  
ok boolean := true ;  
i, j integer ; -- Ceci est par contre incorrect
```

## Les types composites : %TYPE

**%TYPE** donne le type de l'élément qui le précède

Exemple :

```
nom_emp employees.nom%TYPE ;
```

nom\_emp est du même type que la colonne nom de la table employees

## Les types composites : %ROWTYPE

**%ROWTYPE** désigne une ligne dans une table

Exemple :

```
emp employees%ROWTYPE ;
```

emp désigne une ligne de la table employees.

## Exemple d'utilisation des types composites

```
DECLARE
    emp employees%ROWTYPE ;
    nom employees.nom%TYPE ;
BEGIN
    SELECT * FROM employees WHERE matricule='1234' INTO emp ;
    nom := emp.nom ;
    emp.matricule := '6543' ;
    emp.service := 'COMMUNICATION' ;
    ...
    INSERT INTO employees VALUES emp ;
END ;
```

valeur \* est nommé comme « emp »



## Autres types : RECORD

- RECORD permet de définir des ensembles de colonnes ne correspondant pas forcément à une table existante.  
Analogue au struct du langage C

Exemple :

```
DECLARE
TYPE employe IS RECORD (
    matricule integer,
    nom varchar(30));
data employe;
BEGIN
    data.matricule := 50;
END;
```

## Autres types : TABLE 1/3

- TABLE Permet de définir un tableau dont le nombre de lignes est non connu à l'avance. Une table.
- Chaque ligne du tableau contient des données dont le type est précisé à la définition du tableau.

Exemple :

```
TYPE monTypeTableau IS TABLE OF personne%ROWTYPE  
TYPE monTypeTableau IS TABLE OF INTEGER INDEX BY BINARY INTEGER
```

## Autres types : TABLE 2/3

Des fonctions spécifiques sont utilisées sur les variables TABLE :

**Attention: juste dans la tableau virtuelle ! pas physique**

- EXIST(x)
- PRIOR(x)
- NEXT(x)
- DELETE(x)
- COUNT
- FIRST
- LAST
- DELETE

## Autres types : TABLE 3/3

```
DECLARE
    TYPE pilotesProspectes IS TABLE OF pilote%ROWTYPE
        INDEX BY BINARY INTEGER ;
    tabPilotes pilotesProspectes ;
    tmpIndex BINARY INTEGER ;
BEGIN
    ...
    tmpIndex := tabPilotes.FIRST ;
    tabPilotes(4).Age := 37 ;
    tabPilotes(4).Salaire = 42000 ;
    tabPilote.DELETE(5) ;
    ...
END ;
```

## Donner une valeur à une variable

- Affectation :=
- Directive INTO dans un SELECT  
Attention, le SELECT ne doit remonter qu'une seule ligne

Exemples :

```
dateNaissance := '2004-10-04';  
SELECT nom FROM employes WHERE matricule='509' INTO nom_emp;
```

## Extraction multiple de données

```
SELECT expr1, expr2,... INTO var1, var2,...
```

le nombre de colonnes du résultat et les variables qui reçoivent les données doivent correspondre

Attention : SELECT ne doit remonter qu'une seule ligne

## Conflits de noms

Si une variable porte le même nom qu'une colonne d'une table, c'est la colonne qui l'emporte

```
DECLARE
    nom varchar(30) := 'DUPOND' ;
BEGIN
    DELETE FROM employes WHERE nom = nom ;
END ;
```

Suggestion : ne pas donner de nom de colonne à une variable !

## Les structures de contrôle : tests

```
IF condition THEN  
    instructions ;  
END IF ;
```

```
IF condition THEN  
    instructions1 ;  
ELSE  
    instructions2 ;  
END IF ;
```



## Les structures de contrôle : tests

```
IF condition1 THEN
    instructions1 ;
ELSEIF condition2 THEN
    instructions2 ;
ELSEIF ...
    ...
ELSE
    instructionsN ;
END IF ;
```

## Les structures de contrôle : tests

```
CASE expression  
    WHEN condition1 instructions1 ;  
    WHEN condition2 instructions2 ;  
    ...  
  
    ELSE instructionsN ;  
END CASE ;
```

attention : expression est de type simple et ne peut être composé de plusieurs informations (comme un RECORD)

## Les structures de contrôle : boucles

```
WHILE condition LOOP  
    instructions ;  
END LOOP ;
```

```
LOOP  
    instructions ;  
EXIT [WHEN condition];  
    instructions ;  
END LOOP ;
```

## Les structures de contrôle : boucles

```
FOR compteur IN [REVERSE] inf..sup LOOP  
    instructions ;  
END LOOP ;
```

Exemple :

```
FOR i IN 1..100 LOOP  
    somme := somme + i ;  
END LOOP ;
```

## Commentaires

- sur une ligne : double tiret

-- pour un commentaire sur une ligne

- Sur plusieurs lignes /\* ... \*/

/\* Pour les autres  
commentaires \*/

## Modification des données de la base

INSERT, UPDATE et DELETE peuvent être employées comme instructions.

Selon le paramétrage de la base, les COMMIT / ROLLBACK doivent être ou non explicites.

## Modification des données de la base

```
DECLARE
    v_emp employes%ROWTYPE ;
    v_nom employes.nom%TYPE ;
BEGIN
    v_nom := 'Dupond' ;
    INSERT INTO employes (matricule, nom) VALUES (600, v_nom) ;
    v_emp.matricule := 610 ;
    v_emp.nom := 'Durand' ;
    INSERT INTO emp (matricule, nom) VALUES(v_emp.matricule, v_emp.nom) ;
    COMMIT ;
END ;
```

## Notion de curseurs

Comment parcourir les lignes résultats d'une requête ?

```
SELECT * FROM Personne WHERE ID <3
```

Personne	ID	Nom	Prénom
	1	EVERT	John
	2	SENTU	Jack



## Notion de curseurs (= iterator de JAVA)

- A chaque requête SQL exécutée est associé un **curseur** permettant d'avoir des informations sur le résultat de la requête
- Un curseur peut être **implicite** ou **explicite**.
  - un curseur implicite est créé lorsqu'une requête est exécutée
  - Un curseur explicite permet de parcourir les lignes du résultat d'une requête SELECT

## Attributs de tous les curseurs

implicites ou explicites

- %ROWCOUNT : nombre de lignes traitées par le curseur
- %FOUND : vrai si au moins une ligne a été traitée par la requête ou le dernier fetch
- %NOTFOUND : vrai si aucune ligne n'a été traitée par la requête ou le dernier fetch

explicites uniquement

- %ISOPEN : vrai si le curseur est "ouvert", c'est-à-dire que la requête a été exécutée.

## Curseurs implicites

Se nomme SQL (quelle que soit la requête)

Exemple :

```
DECLARE
    nb_lignes integer ;
BEGIN
    DELETE FROM emp WHERE dept = 44 ;
    nb_lignes := SQL%ROWCOUNT ;
    ...
END ;
```

nb\_lignes = nombre de lignes effacées.

## Curseurs explicites

- Ces curseurs sont utilisés pour traiter les requêtes SELECT qui retournent potentiellement plusieurs lignes.
- Ils doivent être déclarés explicitement
- Ils utilisent les instructions

- obligatoire** ● **OPEN** (lance la requête et se positionne avant la 1e ligne),
- FETCH (se positionne sur la suivante, puis prend la ligne)
- obligatoire** ● **CLOSE** (termine l'utilisation du curseur).

## Exemple – curseur explicite

```
DECLARE
    salaires CURSOR IS SELECT sal FROM employees WHERE dept = 10 ;
    salaire numeric(8, 2) ;
    total numeric(10, 2) := 0 ;

BEGIN
    OPEN salaires ;
    LOOP
        FETCH salaires INTO salaire ;
        EXIT WHEN salaires%NOTFOUND ;
        IF salaire IS NOT NULL THEN
            total := total + salaire ;
        END IF ;
    END LOOP ;
    CLOSE salaires ; -- Ne pas oublier
    DBMS_OUTPUT.put_line(total) ; -- écriture
END ;
```

## Exemple – curseur explicite

```
DECLARE
    c CURSOR IS SELECT matr, nom, sal FROM employes ;
    employe c%ROWTYPE ;
BEGIN
    OPEN c ;
    FETCH c INTO employe ;
    IF employe.sal IS NULL THEN ...
    CLOSE c ;
END ;
```

## Boucle FOR pour un curseur

La boucle FOR permet de simplifier l'utilisation d'un curseur

- Elle assure l'ouverture du curseur avant la boucle (OPEN)
- Elle assure la fermeture du curseur après la boucle (CLOSE)
- Elle assure l'acquisition des lignes les unes après les autres à chaque itération (FETCH)
- Elle déclare implicitement une variable de type ROW permettant le parcours des lignes

## Exemple – curseur explicite

```
DECLARE
    c CURSOR IS
        SELECT dept, nom FROM employes WHERE dept = 10 ;
BEGIN
    FOR employe IN c LOOP
        dbms_output.put_line(employe.nom) ;
    END LOOP ;
END ;
```



## Curseur paramétré

Un **curseur paramétré** permet d'introduire un paramètre variable dans un curseur.

Il est impératif de fermer un curseur paramétré avant de le ré-ouvrir avec un autre paramètre.

## Exemple – curseur explicite paramétré

```
DECLARE
    c CURSOR (p_dept integer) IS
        SELECT dept, nom FROM emp WHERE dept = p_dept;
BEGIN
    FOR employe in c(10) LOOP
        dbms_output.put_line(employe.nom);
    END LOOP;
    FOR employe in c(20) LOOP
        dbms_output.put_line(employe.nom);
    END LOOP;
END;
```

## Modification des informations liées au curseur

Par défaut, lors de la consultation des informations liées à un curseur, il est impossible de modifier les données.

Pour faire des modifications lors de la déclaration du curseur, ajouter une clause **FOR UPDATE**.

Effectuer la modification (INSERT, UPDATE, DELETE) en utilisant la clause **\*WHERE CURRENT OF ...**

## Modification des informations liées au curseur

Lors de l'ajout d'une clause FOR UPDATE

- 2 formes
  - CURSOR ... FOR UPDATE
  - CURSOR ... FOR UPDATE OF col1, col2, ...
- La modification des informations de la ligne, ou des colonnes autres que celles spécifiées est impossible.
- Les autres transactions portant sur les données de la ligne ne peuvent être effectuées tant que le curseur n'a pas quitté cette ligne

## Modification des informations liées au curseur - Exemple

```
DECLARE
    c CURSOR IS
        SELECT matricule, nom, sal
        FROM employes
        WHERE dept = 10
        FOR UPDATE OF employes.sal ;

BEGIN
    ...
    IF salaire IS NOT NULL THEN
        total := total + salaire ;
    ELSE-- met 0 à la place de null
        UPDATE employes SET sal = 0 WHERE CURRENT OF c ;
    END IF ;
    ...
END ;
```

## Gestion des exceptions

Une exception est une erreur qui survient durant une exécution

On distingue

- Les exceptions définies par le système
- Les exceptions définies par l'utilisateur

DECLARE

– définitions de variables

BEGIN

– Les instructions à exécuter

EXCEPTION

– La récupération des erreurs

END ;

## Gestion des exceptions

Une exception provoque

- L'exécution des instructions prévues dans le bloc EXCEPTION si elle y est mentionnée (exception capturée)
- L'arrêt de l'exécution du bloc si elle n'y est pas mentionnée

Une exception non capturée remonte à la procédure appelante, et de procédure en procédure jusqu'à ce qu'elle soit capturée ou qu'on ne puisse plus remonter à l'appelant.

## Exceptions prédéfinie :

- NO\_DATA\_FOUND
- TOO\_MANY\_ROWS
- VALUE\_ERROR (erreur arithmétique)
- ZERO\_DIVIDE
- ...



# Gestion des exceptions

```
BEGIN
    ...
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            ...
        WHEN TOO_MANY_ROWS THEN
            ...
        WHEN OTHERS THEN – optionnel
            ...
END;
```

## Exceptions utilisateur

- Doivent être déclarées avec un type EXCEPTION
- Sont levées (générées) avec l'instruction RAISE

```
DECLARE
    salaire numeric(8,2);
    salaire_trop_bas EXCEPTION;
BEGIN
    SELECT sal FROM employes WHERE matricule = 50 INTO salaire;
    IF salaire < 300 THEN
        RAISE salaire_trop_bas;
    END IF;
    -- suite du bloc
    EXCEPTION
        WHEN salaire_trop_bas THEN ...;
        WHEN OTHERS THEN
            dbms_output.put_line(SQLERRM);
END;
```

# Plan

- 1 Le contexte
- 2 Elements de syntaxe
- 3 Procédures et fonctions**
- 4 PostgreSQL
- 5 Conclusion

# Procédures et fonctions

- Un bloc anonyme PL/SQL est un bloc DECLARE... BEGIN ... END
- Un bloc anonyme peut être transformé en procédure ou fonction pour être réutilisé

## Mise en œuvre d'une procédure

```
CREATE OR REPLACE  
PROCEDURE procedure_name(<liste_params >) IS  
– déclaration des variables  
BEGIN  
– code de la procédure  
END;
```

- liste\_params est la liste des déclarations (typées) des paramètres
- Pour chaque paramètre, on indique son nom, le mode (IN=entrée, OUT=sortie, IN OUT=entrée-sortie) et son type
- Ne comporte pas de section DECLARE (remplacé par ce qui se trouve entre IS et BEGIN)

## Exemple de mise en œuvre d'une procédure

```
CREATE OR REPLACE  
PROCEDURE supprimeAnciens(dateRef IN DATE) IS  
BEGIN  
    DELETE FROM acces WHERE Acces_Date < dateRef ;  
END ;
```

## Mise en œuvre d'une fonction

```
CREATE OR REPLACE  
FUNCTION function_name(<liste_params >)  
RETURN <type_retour >IS  
– déclaration des variables  
BEGIN  
    – code de la fonction  
    RETURN valeur ;  
END ;
```

- liste\_params est la liste des déclarations (typées) des paramètres
- type\_retour est le type de la valeur de retour de la fonction
- Pour chaque paramètre, son nom, le mode (IN) et son type.
- Ne comporte pas de section DECLARE (remplacé par ce qui se trouve entre IS et BEGIN)

## Exemple de mise en œuvre d'une fonction

```
CREATE OR REPLACE  
FUNCTION nombreAnciens(dateRef IN DATE)  
RETURN INTEGER IS  
    cpt INTEGER ;  
BEGIN  
    SELECT COUNT(*) FROM acces WHERE Acces_Date < dateRef INTO cpt ;  
    RETURN cpt ;  
END ;
```



## Appel des procédures et fonctions

- Les procédures et fonctions peuvent être utilisées dans d'autres procédures/fonctions ou dans des blocs PL/SQL anonymes
- Les fonctions peuvent être utilisées dans les requêtes SQL

Mécanisme d'appel :

- Appel d'une procédure :  
utiliser le nom de la procédure et indiquer entre parenthèses la liste des arguments
- Appel d'une fonction :  
utiliser le nom de la fonction et indiquer entre parenthèses la liste des arguments

## Exemple d'appel de fonction

```
CREATE OR REPLACE FUNCTION nombreAnciens(dateRef IN DATE)
RETURN INTEGER IS
    cpt INTEGER ; BEGIN
    SELECT COUNT(*) FROM Acces WHERE Acces_Date < dateRef INTO cpt ;
    RETURN cpt ;
END ;
```

```
SELECT Versions_Date FROM Versions WHERE nombreAnciens(Versions_Date) < 10
```

## Exemple d'appel de fonction

```
CREATE OR REPLACE  
FUNCTION nombreAnciens(dateRef IN DATE) IS  
    cpt INTEGER ; BEGIN  
    SELECT COUNT(*) FROM Acces WHERE Acces_Date < dateRef INTO cpt ;  
    RETURN cpt ;  
END ;
```

```
CREATE OR REPLACE PROCEDURE supprimeAnciens(dateRef IN DATE) IS  
BEGIN  
    DELETE FROM Versions WHERE nombreAnciens(Versions_Date) < 10 ;  
END ;
```

## Cas particulier des TRIGGERS et fonctions TRIGGER

- Un **TRIGGER** (déclencheur) est une action effectuée avant ou après l'exécution d'une instruction de modification (mise-à-jour, insertion, suppression) sur une table.
- Un TRIGGER s'appuie sur une **fonction de type TRIGGER**
- Un TRIGGER s'exécute
  - Soit sur toutes les lignes modifiées (trigger de **TABLE**)
  - Soit sur chacune des lignes modifiées (trigger de **LIGNE**)

# Les TRIGGERS

Le TRIGGER dépend :

- De la table à laquelle il est attaché (une seule)
- D'un ou plusieurs mécanismes de modification (INSERT / UPDATE / DELETE) qui le déclenchera
- D'un type de comportement
  - TRIGGER de TABLE (1 exécution par requête)
  - TRIGGER DE LIGNE (1 exécution par ligne modifiée par une requête)
- De son mode de déclenchement (AVANT ou APRES la modification)

Lorsqu'il se déclenche, il appelle une fonction TRIGGER.

# Les fonctions TRIGGER

- Elles retournent un objet de type TRIGGER
- Elles ne peuvent être appelées que depuis un TRIGGER
- Pour le reste, elles sont définies comme des fonctions

## Cas particulier des fonctions TRIGGER attachées à un TRIGGER de LIGNE

Une fonction TRIGGER attachée à un TRIGGER de LIGNE est appelée pour chaque ligne modifiée.

Il doit donc disposer de 2 informations :

- La ligne **AVANT** la modification (désignée par :OLD)
- la ligne **APRES** la modification (désignée par :NEW)

:NEW et :OLD sont objets de type RECORD correspondants à une ligne de la table.

- :OLD comporte une valeur pour UPDATE et DELETE
- :NEW comporte une valeur pour UPDATE et INSERT

## Cas particulier des fonctions TRIGGER attachées à un TRIGGER de LIGNE

Lorsque la fonction TRIGGER est appelée **AVANT** la modification, la valeur de retour désigne la valeur qui sera prise en compte pour la modification.

En conséquence :

- Si la ligne modifiée n'a aucune raison d'être altérée, il convient de retourner la valeur NEW
- Si la modification est altérée, modifiez NEW avant de le retourner.
- Si la modification doit être annulée parce qu'incorrecte, levez une exception



# Mise en œuvre des TRIGGERS et fonctions

## TRIGGER

### Fonction TRIGGER

```
CREATE OR REPLACE FUNCTION fonction_trigger() RETURN TRIGGER IS
    – Déclaration de variables
BEGIN
    – Code de la fonction
END;
```

### TRIGGER de TABLE

```
CREATE TRIGGER nom_trigger
    <BEFORE | AFTER >[<INSERT> [OR] [UPDATE] [OR] [DELETE] >
    ON uneTable
    [ FOR EACH ROW ] EXECUTE PROCEDURE fonction_trigger();
```

## Autres langages

Dans certains SGBD, il est possible de développer les triggers en JAVA, en C++

- **Avantage**
  - Bénéficier des spécificités du langage sélectionné
- **Inconvénient**
  - Pas toujours portable d'un serveur à l'autre
  - Sur peu de SGBD
  - Sécurité ?

# Plan

- 1 Le contexte
- 2 Elements de syntaxe
- 3 Procédures et fonctions
- 4 PostgreSQL**
- 5 Conclusion

# PLpgSQL

PLpgSQL est l'implémentation de PL/SQL version PostgreSQL.  
Elle ressemble à PL/SQL sans en être complètement identique.

# Mise en œuvre des procédures et fonctions dans PostgreSQL

- PLpgSQL n'implémente pas les PROCEDURES. L'équivalent est obtenu par des fonctions qui ne renvoient "void".
- La syntaxe de création est légèrement différente
- Les types de données sont ceux de PostgreSQL
- Par défaut, tous les parametres sont en IN. Si vous avez besoin de parametres en sortie, il faut indiquer le mot clé OUT avant la définition du paramètre. Attention, dans ce cas la fonction ne peut pas retourner "void".
- On utilise NEW et OLD et pas :NEW et :OLD
- Le reste de la syntaxe est préservé

## Mise en œuvre des fonctions dans PostgreSQL - Syntaxe

Voici à quoi ressemble une requête de création de fonction :

```
CREATE OR REPLACE FUNCTION ma_fonction(c_id INTEGER) RETURNS INTEGER
language 'plpgsql' AS
$BODY$
DECLARE
    cpt INTEGER ;
BEGIN
    SELECT COUNT(Item.*) FROM Item INNER JOIN Category ON
(Item.Category_ID=Category.ID) WHERE Category.ID=c_id INTO cpt ;
    RETURN cpt ;
END ;
$BODY$ ;
```

## Mise en œuvre des fonctions dans PostgreSQL - Syntaxe

Et maintenant une procédure :

```
CREATE OR REPLACE FUNCTION ma_procedure1(c_id INTEGER) "void" INTEGER
language 'plpgsql' AS
$BODY$
DECLARE
    cpt INTEGER ;
BEGIN
    SELECT COUNT(Item.*) FROM Item INNER JOIN Category ON
(Item.Category_ID=Category.ID) WHERE Category.ID=c_id INTO cpt ;
END ;
$BODY$ ;
```

## Mise en œuvre des fonctions dans PostgreSQL - Syntaxe

Avec un paramètre de sortie :

```
CREATE OR REPLACE FUNCTION ma_procedure2(c_id INTEGER, OUT cpt
INTEGER) language 'plpgsql' AS
$BODY$
BEGIN
    SELECT COUNT(Item.*) FROM Item INNER JOIN Category ON
(Item.Category_ID=Category.ID) WHERE Category.ID=c_id INTO cpt;
END;
$BODY$;
```

Dans ce cas, le résultat d'un appel est un n-uplet constitué de toutes les variables OUT.



# Mise en œuvre des fonctions dans PostgreSQL

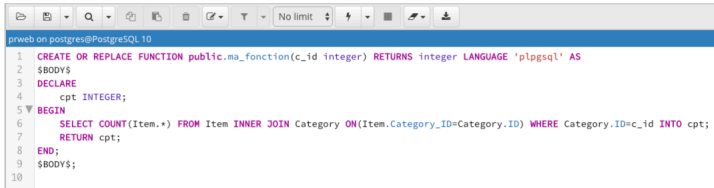
L'implémentation des fonctions peut se faire de plusieurs façons :

- Via un requêteur
- Via une interface d'administration

Nous allons le faire via PgAdmin 4.

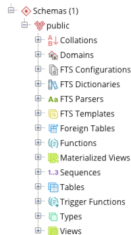
# Mise en œuvre des fonctions dans PostgreSQL - Requêteur

La mise en place via le requêteur ressemble à toute saisie de requête.



```
prweb on postgres@PostgreSQL 10
1 CREATE OR REPLACE FUNCTION public.ma_fonction(c_id integer) RETURNS integer LANGUAGE 'plpgsql' AS
2 $BODY$
3 DECLARE
4     cpt INTEGER;
5 BEGIN
6     SELECT COUNT(Item.*) FROM Item INNER JOIN Category ON (Item.Category_ID=Category.ID) WHERE Category.ID=c_id INTO cpt;
7     RETURN cpt;
8 END;
9 $BODY$;
10
```

# Mise en œuvre des fonctions dans PostgreSQL - Interface

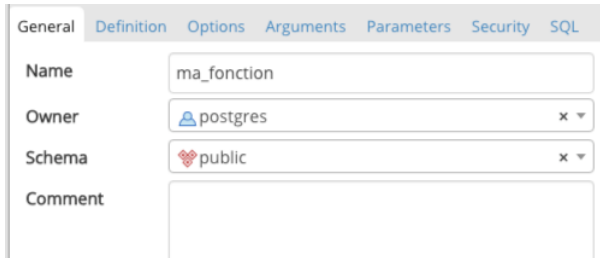


Faites un clic-droit sur "Functions" puis "Create" puis "Function"

Attention : Les fonctions TRIGGER sont implémentées de manière différente.

## Mise en œuvre des fonctions dans PostgreSQL - Interface

Dans le premier onglet, entrez le nom de la fonction. Vérifier son possesseur.



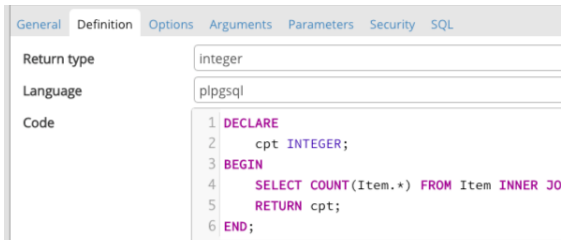
The screenshot shows a web-based interface for defining a PostgreSQL function. It features a horizontal tab bar with six tabs: 'General' (selected), 'Definition', 'Options', 'Arguments', 'Parameters', 'Security', and 'SQL'. Below the tabs, there are four input fields. The 'Name' field contains the text 'ma\_fonction'. The 'Owner' field shows a user icon followed by 'postgres' and a dropdown arrow. The 'Schema' field shows a database icon followed by 'public' and a dropdown arrow. The 'Comment' field is an empty text area.

Field	Value
Name	ma_fonction
Owner	postgres
Schema	public
Comment	

## Mise en œuvre des fonctions dans PostgreSQL - Interface

Dans le deuxième onglet, entrez le type de retour de la fonction, le langage utilisé (plpgsql) et enfin le code.

Vous n'avez pas à entrer la partie création, ... seules les sections DECLARE, BEGIN, EXCEPTION y figurent.

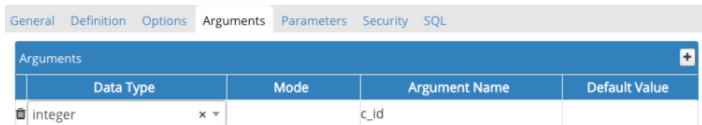


The screenshot shows the 'Definition' tab of a PostgreSQL function editor. The 'Return type' is set to 'integer', the 'Language' is set to 'plpgsql', and the 'Code' section contains the following SQL code:

```
1 DECLARE
2     cpt INTEGER;
3 BEGIN
4     SELECT COUNT(Item.*) FROM Item INNER JO
5     RETURN cpt;
6 END;
```

## Mise en œuvre des fonctions dans PostgreSQL - Interface

Dans le quatrième onglet, entrez les paramètres de la fonction. Utilisez le + à droite pour ajouter un paramètre.



The screenshot shows the PostgreSQL function editor interface. At the top, there are five tabs: 'General', 'Definition', 'Options', 'Arguments', and 'Parameters'. The 'Arguments' tab is currently selected. Below the tabs, there is a table with four columns: 'Data Type', 'Mode', 'Argument Name', and 'Default Value'. The first row of the table contains the following values: 'integer' (with a dropdown arrow), an empty 'Mode' cell, 'c\_id', and an empty 'Default Value' cell. A '+' button is located at the top right of the table area.

Data Type	Mode	Argument Name	Default Value
integer x ▾		c_id	

Vous pouvez sauvegarder votre travail. La fonction est alors créée.

# Les fonctions TRIGGER

Les fonctions TRIGGER ressemblent aux fonctions, mais elles retournent obligatoirement un objet de type TRIGGER.

La création de ces fonctions est donc très proche de la création des fonctions.

# Mise en œuvre des fonctions TRIGGER dans PostgreSQL

```
CREATE OR REPLACE FUNCTION ma_fct_trigger() RETURNS TRIGGER language 'plpgsql' AS
$BODY$
DECLARE
    cpt INTEGER ;
BEGIN
    SELECT COUNT(*) FROM Category WHERE Category.ID=NEW.Category_ID INTO
cpt ;
    IF (cpt == 0) THEN
        INSERT INTO Category(id, name) VALUES (NEW.Category_ID, 'dummy') ;
        COMMIT ;
    END IF ;
    RETURN NEW ;
END ;
$BODY$;
```



# Mise en œuvre des TRIGGERS dans PostgreSQL

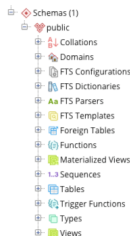
```
CREATE TRIGGER mon_trigger  
  BEFORE INSERT OR UPDATE  
  ON Item  
  FOR EACH ROW EXECUTE PROCEDURE ma_fct_trigger();
```

# Mise en œuvre des TRIGGERs et fonctions

## TRIGGER dans PostgreSQL - Requêteur

```
prweb on postgres@PostgreSQL 10
1 CREATE OR REPLACE FUNCTION ma_fct_trigger() RETURNS TRIGGER language 'plpgsql' AS
2 $BODY$
3 DECLARE
4     cpt INTEGER;
5 BEGIN
6     SELECT COUNT(*) FROM Category WHERE Category.ID=NEW.Category_ID INTO cpt;
7     IF (cpt == 0) THEN
8         INSERT INTO Category(id, name) VALUES (NEW.Category_ID, 'dummy');
9         COMMIT;
10    END IF;
11    RETURN NEW;
12 END;
13 $BODY$;
14
15 CREATE TRIGGER mon_trigger
16 BEFORE INSERT OR UPDATE
17 ON Item
18 FOR EACH ROW EXECUTE PROCEDURE ma_fct_trigger();
```

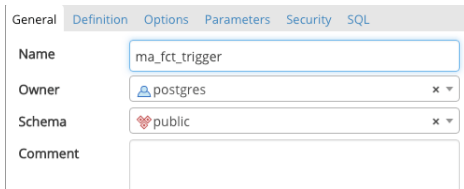
# Mise en œuvre des fonctions TRIGGER dans PostgreSQL - Interface



Faites un clic-droit sur "Trigger Functions" puis "Create" puis "Trigger Function"

## Mise en œuvre des fonctions TRIGGER dans PostgreSQL - Interface

Dans le premier onglet, entrez le nom de la fonction. Vérifier son possesseur.



The screenshot shows a web-based interface for PostgreSQL with a tabbed menu at the top: General, Definition, Options, Parameters, Security, and SQL. The 'General' tab is selected. Below the tabs, there are four input fields: 'Name' with the value 'ma\_fct\_trigger', 'Owner' with a user icon and the value 'postgres', 'Schema' with a database icon and the value 'public', and 'Comment' which is empty. Each of the first three fields has a small 'x' and a dropdown arrow on the right side.

## Mise en œuvre des fonctions TRIGGER dans PostgreSQL - Interface

Dans le deuxième onglet, vérifiez le type de retour de la fonction (trigger), le langage utilisé (plpgsql) et entrez le code.  
Vous n'avez pas à entrer la partie création, ... seules les sections DECLARE, BEGIN, EXCEPTION y figurent.

The screenshot shows the PostgreSQL function editor interface with the 'Definition' tab selected. The 'Arguments' field is empty. The 'Return type' is set to 'trigger'. The 'Language' is set to 'plpgsql'. The 'Code' field contains the following SQL code:

```
1 DECLARE
2     cpt INTEGER;
3 BEGIN
4     SELECT COUNT(*) FROM Category;
5     IF (cpt > 0) THEN
```

# Mise en œuvre des TRIGGER dans PostgreSQL - Interface

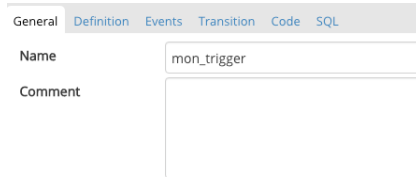
Un trigger est lié à une table.

Il faut donc vous rendre sur la table à la quelle sera attaché le TRIGGER.

Faites un clic droit sur la table concernée puis "Create" et "Trigger".

# Mise en œuvre des TRIGGER dans PostgreSQL - Interface

Sur le premier onglet, entrez un nom pour le trigger.



The screenshot shows a web-based interface for configuring a PostgreSQL trigger. At the top, there are five tabs: 'General', 'Definition', 'Events', 'Transition', and 'Code SQL'. The 'General' tab is currently selected. Below the tabs, there are two input fields. The first is labeled 'Name' and contains the text 'mon\_trigger'. The second is labeled 'Comment' and is currently empty.

## Mise en œuvre des TRIGGER dans PostgreSQL - Interface

Sur le deuxième onglet, indiquez s'il s'agit ou pas d'un trigger de ligne et quelle est la fonction TRIGGER associée.

General	Definition	Events	Transition	Code	SQL
Trigger enabled?	<input checked="" type="checkbox"/>	Yes			
Row trigger?	<input checked="" type="checkbox"/>	Yes			
Constraint trigger?	<input type="checkbox"/>	No			
Deferrable?	<input type="checkbox"/>	No			
Deferred?	<input type="checkbox"/>	No			
Trigger Function	<input type="text" value="public.ma_fct_trigger"/>				
Arguments	<input type="text"/>				



## Mise en œuvre des TRIGGER dans PostgreSQL - Interface

Sur le troisième onglet, indiquez si le TRIGGER doit être déclenché AVANT ou APRES la modification dans la table, et quelles sont les modifications qui vont le déclencher (INSERT, UPDATE, ...).

The screenshot shows the 'Events' tab of a PostgreSQL trigger configuration window. At the top, there are tabs for 'General', 'Definition', 'Events' (selected), 'Transition', 'Code', and 'SQL'. Below the tabs, the 'Fires' section has a dropdown menu set to 'BEFORE'. The 'Events' section contains four rows of event types with corresponding 'Yes' or 'No' toggle buttons:

Event Type	Yes	No
INSERT	<input checked="" type="checkbox"/>	<input type="checkbox"/>
UPDATE	<input checked="" type="checkbox"/>	<input type="checkbox"/>
DELETE	<input type="checkbox"/>	<input checked="" type="checkbox"/>
TRUNCATE	<input type="checkbox"/>	<input checked="" type="checkbox"/>

# Plan

- 1 Le contexte
- 2 Elements de syntaxe
- 3 Procédures et fonctions
- 4 PostgreSQL
- 5 Conclusion

## Conclusion

- Permet d'étendre les possibilités du langage SQL
- Permet de prendre en charge des actions sur la base de données sans nécessité de mettre en place un programme de mise-à-jour
- Les TRIGGERS permettent d'assurer que les actions sur la base seront exécutées au bon moment.

