



Bases de données relationnelles

Fonctions

Maintien de l'intégrité

Déclencheurs : principes, déclenchement

<http://docs.postgresqlfr.org/8.2/sql-createtrigger.html>

Procédures stockées (PL/pgSQL)

<http://docs.postgresqlfr.org/8.2/plpgsql.html>

Livre de F. M Colonna (PostGres TP – eni edition)



Maintien de l'intégrité

- Plusieurs façons d'exprimer les contraintes d'intégrité d'une BD :
 - définition du domaine ou du type d'un attribut,
 - condition sur les valeurs des attributs d'un n -uplet (clause CHECK de SQL),
 - définition de la clé primaire et des clés étrangères d'une relation,
 - **Déclencheurs (trigger)**



Déclencheur

- Un **déclencheur** est une règle, dite **active**, de la forme:
«événement - condition -action»
 - L'action est déclenchée à la suite de l'événement, si la condition est vérifiée.
 - Une action peut être une vérification ou une mise à jour

Pour définir un déclencheur, il faut savoir écrire une fonction



Fonctions

- Tout utilisateur peut programmer une fonction s'il en a les droits
 - Solution simple pour regrouper un ensemble de traitements qui doivent être répétés sur la BD
 - Écrites en langage SQL
 - Suite de commandes SQL
 - Si plus complexe → PL/pgSQL (langage procédural)
 - Hébergées sur le serveur



Fonctions

- Suite de commandes SQL
 - séparées par des ;
 - retourne *NULL* si la dernière requête ne produit pas de résultat, sinon le résultat de la dernière requête
 - les paramètres ne peuvent être que des valeurs de données (pas de noms de tables)
- Sinon (en PL/pgSQL)
 - retourne *VOID* si la dernière instruction ne renvoie rien, sinon le résultat de la dernière instruction



Fonctions

- CREATE OR REPLACE FUNCTION
- DROP FUNCTION
 - (Si le type de retour de la fonction est modifiée)

```
CREATE OR REPLACE FUNCTION name ( [ <ftype> [, ...] ] )  
RETURNS <rtype>  
AS  
    <definition>  
LANGUAGE <langname>;
```

- Le corps de la fonction est une constante de type chaîne (guillemets dollar)
- \$i désigne le ième paramètre passé à la fonction

Exemple

CREATE OR REPLACE FUNCTION PrixSport(**varchar**)

RETURNS **float**

AS \$\$

SELECT tarifUnité FROM Sport

WHERE nomS=**\$1** ;

\$\$ LANGUAGE SQL;

*SELECT * from PrixSport("Tennis");*

tarifUnité

6

Afficher la liste des fonctions \df



Même exemple

```
CREATE OR REPLACE FUNCTION PrixSport(IN varchar, OUT "Prix" float)
AS $$
```

```
    SELECT tarifUnité FROM Sport
```

```
    WHERE nomS=$1 ;
```

```
$$ LANGUAGE SQL RETURNS NULL ON NULL INPUT ;
```

→ l'utilisation de OUT, affichage du résultat avec une colonne renommée

Prix

6 (1 row)



Précision

```
SELECT Mafonction(x,y);
```

```
SELECT * from Mafonction(x,y);
```

Mafonction	attribut1		attribut2		attribut3
(a1,b1,c1)	a1		a2		a3

Autre exemple

```
CREATE OR REPLACE FUNCTION mafct (OUT Sport)
```

```
AS $$
```

```
    SELECT * FROM Sport;
```

```
$$ LANGUAGE SQL;
```

→ Renvoie un seul résultat (ordre ?)

```
SELECT * from mafct();
```

nomS	unitéLoc	tarifUnité
------	----------	------------

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

Tennis	1 heure	6
--------	---------	---

Renvoyer un résultat ou un ensemble de résultats

CREATE OR REPLACE FUNCTION mafct (OUT SETOF Sport)

AS \$\$

SELECT * FROM Sport;

\$\$ LANGUAGE SQL;

*SELECT * from mafct();*

La table entière...

<i>nomS</i>	<i>UniteLoc</i>	<i>TarifUnite</i>
tennis	1 heure	6
VTT	1/2 journée	7,5
planche-voile	1 journée	11,5
canoë	2 heures	3,75
pétanques	2 heures	2



Renvoyer un résultat ou un ensemble de résultats

```
CREATE OR REPLACE FUNCTION f1 () RETURNS varchar
AS
```

```
    $$ SELECT NomClient FROM Sejour; $$
```

```
LANGUAGE SQL;
```

→ Renvoie un résultat de type *varchar*, affiche le premier nom de client obtenu (ordre ?)

```
CREATE OR REPLACE FUNCTION f2 () RETURNS SETOF varchar
AS
```

```
    $$ SELECT NomClient FROM Sejour; $$
```

```
LANGUAGE SQL;
```

→ Renvoie et affiche la totalité des noms des clients (type *SETOF varchar*)

Le type enregistrement : **record**

Pour construire des lignes (**sans structure prédéfinie**)

Autre exemple – type record

```
CREATE OR REPLACE FUNCTION mafct(varchar)
```

```
RETURNS record
```

```
AS $$
```

```
    SELECT NomS, TarifUnité
```

```
    FROM Sport
```

```
    WHERE NomS=$1;
```

```
$$ LANGUAGE SQL;
```

```
SELECT * FROM mafct("Tennis") AS (NomSport varchar, Prix float);
```

On précise la structure à donner au résultat lors de l'appel de fonction



Maintien de l'intégrité

- Déclencheur – Trigger
- Un déclencheur est une règle, dite **active**, de la forme:
 - «événement - condition -action»
 - L'action est déclenchée à la suite de l'événement, si la condition est vérifiée.
 - Une action peut être une vérification ou une mise à jour

Le trigger se compose :

D'un évènement qui va le déclencher

D'une action à exécuter au déclenchement

Exemples de trigger

COLLECTIONS (codeColl, intitule, #codeEdition, theme)

EDITIONS (codeEdition, nom, adresse, dateCreation...)

LIVRES (isbn, titre, prix, depotLegal, #CodeColl...)

Trigger 1 : vérifier lors de l'insertion ou mise à jour d'un livre que la date de dépôt légal est postérieure à la date de création de la maison d'édition qui le publie

Trigger 2 : après insertion, mise à jour ou suppression d'un livre, afficher le nouveau prix moyen global des livres



Déclencheurs/ triggers

- Programme déclenché par un évènement
- Il est associé à une relation
 - L'évènement qui le déclenche est une opération sur cette relation
 - Insertion, mise à jour, suppression
 - L'action est spécifiée **par une procédure PL/pgSQL**



PL/PGSQL

- PL/PGSQL = Programming Language / postgresSQL
- C'est un langage qui intègre SQL et permet de programmer de manière procédurale
- Procédures stockées
 - un ensemble d'instructions précompilées, stockées dans une base de données et exécutées sur demande par le SGBD qui manipule la base de données
 - Les procédures stockées peuvent être lancées par un utilisateur, un administrateur ou encore de façon automatique par un événement déclencheur (trigger)



Ecrire un trigger : deux étapes

1. Écrire une fonction sans arguments qui renvoie un résultat de type TRIGGER

```
CREATE OR REPLACE FUNCTION <func> ()
```

```
RETURNS TRIGGER
```

```
AS... définition de l'action associée au trigger
```

2. Puis définir le déclencheur

```
CREATE TRIGGER <name>
```

```
{ BEFORE | AFTER } { <event> [OR ...] }
```

```
ON <table> FOR EACH { ROW | STATEMENT }
```

```
EXECUTE PROCEDURE <func> ()
```



Exemple

COLLECTIONS (codeColl, intitule, #codeEdition, theme)

EDITIONS (codeEdition, nom, adresse, **dateCreation**...)

LIVRES (isbn, titre, prix, **depotLegal**, #CodeColl...)

Trigger 1 : **vérifier lors de l'insertion ou mise à jour d'un livre** que la date de dépôt légal est postérieure à la date de création de la maison d'édition qui le publie



Trigger appelant une fonction – exemple 1

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

Souvent, le nom du trigger et de sa fonction associée sont les mêmes

```
INSERT INTO LIVRES  
VALUES ('123', 'blabla', 10, '2017/12/12', 'ABC' ...)  
  
LIVRES(isbn, titre, prix, depotLegal, #CodeColl ... )
```



Définition d'un déclencheur

```
CREATE TRIGGER nomtrigger  
  { BEFORE | AFTER } { evenement [ OR ... ] } ON table  
  [ FOR [ EACH ] { ROW | STATEMENT } ]  
  [ WHEN (<condition>)]  
EXECUTE PROCEDURE foncTrigger()
```

- FOR EACH ROW :
 - action réalisée pour chaque n -uplet inséré ou mis à jour
- FOR EACH STATEMENT :
 - Action réalisée une seule fois pour la requête



Pour définir un déclencheur

- Le déclencheur est activé par une requête de mise à jour
 - spécifier l'événement qui déclenche l'action en indiquant
 - le type de la mise à jour (INSERT, UPDATE, DELETE),
 - le nom de la relation
 - et éventuellement le nom des attributs mis à jour
 - indiquer si l'action est réalisée *avant, après ou à la place* de la mise à jour
 - indiquer si l'action est réalisée pour chaque n -uplet mis à jour ou une seule fois pour la requête
 - **décrire l'action à réaliser sous la forme d'une fonction/procédure PL/pgSQL**
 - ...

```
INSERT INTO LIVRES
VALUES ('123', 'blabla', 10, '2017/12/12', 'ABC' ...)
```



Ex 1

```
LIVRES(isbn, titre, prix, depotLegal, #CodeColl ... )
EDITIONS(codeEdition, ..., dateCreation...)
COLLECTIONS(codeColl, ... #codeEdition, ...)
```

```
CREATE OR REPLACE FUNCTION VerifDatesEditionPublication()
RETURNS TRIGGER
```

```
AS $$
```

```
DECLARE
```

```
-- déclarations de variables --
```

```
BEGIN
```

```
-- corps de l'action --
```

```
END;
```

```
$$ LANGUAGE PLPGSQL;
```



Variables spéciales (de type record)

Il faut aussi donner un nom à l'ancien et au nouveau n -uplet pour le manipuler

NEW

- Pour désigner l'enregistrement à insérer

OLD

- Pour désigner un enregistrement à effacer


```
INSERT INTO LIVRES  
VALUES ('123', 'blabla', 10, '2017/12/12', 'ABC' ...)
```



Ex 1

```
LIVRES(isbn, titre, prix, depotLegal, #CodeColl ... )  
EDITIONS(codeEdition, ..., dateCreation...)  
COLLECTIONS(codeColl, ... #codeEdition, ...)
```

```
CREATE OR REPLACE FUNCTION VerifDatesEditionPublication()
```

```
RETURNS TRIGGER
```

```
AS $$
```

```
DECLARE      dates_ok boolean := false;
```

```
BEGIN
```

```
    SELECT new.depotLegal >= dateCreation
```

```
    FROM EDITIONS, COLLECTIONS
```

```
    WHERE EDITIONS.codeEdition = COLLECTIONS.codeEdition
```

```
    AND CodeColl = new.CodeColl
```

```
    INTO dates_ok :
```

```
END;
```

```
$$ LANGUAGE PLPGSQL;
```

```
INSERT INTO LIVRES
VALUES ('123','blabla',10, '2017/12/12','ABC'...)
```



Ex 1

```
LIVRES(isbn, titre, prix, depotLegal, #CodeColl ... )
EDITIONS(codeEdition, ..., dateCreation...)
COLLECTIONS(codeColl, ... #codeEdition, ...)
```

```
CREATE OR REPLACE FUNCTION VerifDatesEditionPublication()
```

```
RETURNS TRIGGER
```

```
AS $$
```

```
DECLARE      dates_ok boolean := false;
```

```
BEGIN
```

```
    SELECT new.depotLegal >= dateCreation
```

```
    FROM EDITIONS
```

```
    WHERE codeEdition = (SELECT codeEdition
```

```
                        FROM COLLECTIONS
```

```
                        WHERE CodeColl =
```

```
                            new.CodeColl)
```

```
    INTO dates_ok
```

```
END;
```

```
$$ LANGUAGE PLPGSQL;
```



Exemple 1

```
CREATE OR REPLACE FUNCTION VerifDatesEditionPublication()  
RETURNS TRIGGER  
AS $$  
DECLARE          dates_ok boolean := false;  
BEGIN  
    -- requête de spécification --  
    IF (dates_ok) THEN RETURN new;  
    ELSE RETURN NULL;  
        -- ne renvoie rien, insertion abandonnée --  
    ENDIF;  
END;  
$$ LANGUAGE PLPGSQL;
```



Variables spéciales

TG_NAME : variable qui contient le nom du trigger lancé

TG_WHEN: une chaîne (BEFORE ou AFTER) en fc définition du trigger

TG_LEVEL: une chaîne (ROW ou STATEMENT) en fc définition du trigger

TG_OP : une chaîne (INSERT, UPDATE ou DELETE) l'opération pour laquelle le trigger a été lancé



Exemple 1

```
CREATE OR REPLACE FUNCTION VerifDatesEditionPublication()  
RETURNS TRIGGER  
AS $$  
DECLARE          dates_ok boolean := false;  
BEGIN  
    -- requête de spécification -  
  
    RAISE INFO 'déclencheur provoqué par %', TG_OP;  
    RAISE INFO 'dates correctes ?%', dates_ok;  
END;  
$$ LANGUAGE PLPGSQL
```



Trigger appellant cette fonction – exemple 1

```
CREATE TRIGGER VerifDatesEditionPublication  
    BEFORE INSERT OR UPDATE  
    ON Livres  
    FOR EACH ROW  
    EXECUTE PROCEDURE VerifDatesEditionPublication()
```



Exemple

```
COLLECTIONS (codeColl, intitule, #codeEdition, theme)
EDITIONS (codeEdition, nom, adresse, dateCreation...)
LIVRES (isbn, titre, prix, depotLegal, #CodeColl...)
```

Trigger 2 : **après insertion, mise à jour ou suppression d'un livre, afficher le nouveau prix moyen global des livres**

Trigger de type instruction (statement)



Trigger– exemple 2

```
CREATE TRIGGER PrixMoyenGlobal  
    AFTER INSERT OR UPDATE OR DELETE  
    ON Livres  
    FOR EACH STATEMENT  
    EXECUTE PROCEDURE PrixMoyenGlobal()
```




Action - Exemple 2

```
CREATE OR REPLACE FUNCTION PrixMoyenGlobal()  
RETURNS TRIGGER AS $$  
    DECLARE    prix_moy numeric(4,2);  
    BEGIN  
        SELECT avg(prix) FROM livres  
        INTO prix_moy;  
        IF (prix_moy IS NULL) THEN  
            RAISE INFO 'Prix moyen des livres inconnu';  
        ELSE  
            RAISE INFO 'Prix moyen des livres : % €' , prix-moy;  
        ENDIF;  
        RETURN NULL;  
    END;  
$$ LANGUAGE PLPGSQL;
```

Procédures stockées



Procédures SQL
Quelques éléments



Procédures stockées en PL/PGSQL

- PL/PGSQL = Programming Language / postgresSQL
- C'est un langage qui intègre SQL et permet de programmer de manière procédurale
- Les procédures stockées peuvent être lancées par un utilisateur, un administrateur ou encore de façon automatique par un événement déclencheur (trigger)
- Les fonctions écrites en PL/pgSQL accepte en argument n'importe quel type de données supportées par le serveur et peuvent renvoyer un résultat de n'importe lequel de ces types



Fonctionnement

- Les requêtes envoyées à un serveur font l'objet d'une analyse syntaxique puis d'une interprétation avant d'être exécutée
 - Ces étapes sont très lourdes si l'on envoie plusieurs requêtes complexes.
- Procédures stockées
 - une requête n'est envoyée qu'une unique fois sur le réseau puis analysée, interprétée et stockée sur le serveur sous forme exécutable (pré-compilée).
 - Pour qu'elle soit exécutée, le client n'a qu'à envoyer une requête comportant le nom de la procédure stockée
 - passage de paramètres lors de son appel



Exemple simple

```
CREATE OR REPLACE FUNCTION somme (a INTEGER, b INTEGER)
  RETURNS INTEGER
  AS $$
    DECLARE
      resultat integer;
    BEGIN
      resultat := a+b;
      return resultat;
    END;
  $$ LANGUAGE plpgsql;
```



Exemple simple suite

Appel

```
SELECT somme (2, 3) ;
```

5

(1 ligne)

Liste des procédures

\df



PL/PGSQL

- PL/PGSQL permet :
 - de déclarer des variables (DECLARE)
 - d'avoir une forme spéciale de SELECT qui autorise à stocker le résultat dans des variables (SELECT INTO)
 - des structures de contrôle (IF, WHILE)
 - de quitter et de retourner un résultat (RETURN)



Types de variables

- Les types simples `integer`, `char`, `varchar(x)`, `text`
...

```
DECLARE id_utilisateur integer;
```

```
DECLARE quantité numeric(5);
```

```
DECLARE url varchar;
```




Types de variables

- Le type enregistrement : `record`
 - Pour stocker les lignes (sans structure prédéfinie)
- Clause `%TYPE` pour obtenir le type d'une colonne
`DECLARE mon_champ nom_table.nom_colonne%TYPE;`
- Clause `%rowType` pour obtenir le n-uple d'une table
`DECLARE ma_ligne nom_table%ROWTYPE;`
- Signature de fonction, type de retour :
 - `TRIGGER` pour les déclencheurs

Exemples de déclarations de variables et de leur type

```
DECLARE recordSport Sport%ROWTYPE
```

variable ligne (pour accéder aux champs individuels `ma_ligne.champ`)

le param d'1 fc peut être une ligne (`$i.champ` pour accéder à chaque champ)

```
DECLARE nom Sport.nomSport%TYPE ;
```

copie du type de la colonne référencée

```
DECLARE une_ligne RECORD;
```

similaire aux variables de type ligne mais sans structure prédéfinie, préciser la structure du résultat lors de l'appel



Instruction de base

Assignation

```
nom_variable := expression ;
```

Exécuter une requête avec une seule ligne de résultat

```
SELECT INTO <cible> <expressions> FROM ...;
```

Le résultat d'une commande SQL ne ramenant qu'une seule ligne (mais avec une ou plusieurs colonnes) peut être affecté à une variable de type record, row ou à une liste de variables scalaires



Exemple

```
DECLARE monCli client%ROWTYPE;
        nocli integer; nomcli varchar;

BEGIN

    select into monCli *
    from client
    where nom_client='dupond';

    -- ou

    select into nocli,nomcli *
    from client
    where nom_client='dupond';
```



Instruction de base

Exécution dynamique de commande

EXECUTE <commande> [INTO cible]

Exécution d'une commande dynamique (expression interprétable - chaîne contenant la commande à exécuter) qui contient des paramètres qui vont varier à chaque appel

Pas de substitution de variable dans <commande>

Exemple

```
CREATE OR REPLACE FUNCTION ComptageLignes(matable varchar)
RETURNS INTEGER
AS $$
    DECLARE
        nb_lignes integer := 0;
        chaine varchar;
    BEGIN
        chaine:= 'SELECT count(*) FROM ' || matable;
        RAISE NOTICE 'valeur %', chaine;

        EXECUTE chaine INTO nb_lignes;

        RAISE NOTICE 'valeur %', nb_lignes;
        RETURN nb_lignes;
    END;
$$ LANGUAGE PLPGSQL;
```



Structures de contrôle simple

```
RETURN expression;
```

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

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

```
IF expression THEN ... ELSEIF ... END IF;
```



Boucles

```
LOOP ... END LOOP;
```

```
EXIT [WHEN expression];
```

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

```
WHILE expression
```

```
    LOOP instructions END LOOP;
```

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



Boucles

```
FOR var in [REVERSE] exp1 .. exp2  
    LOOP instructions END LOOP;
```

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

La variable `record_ou_ligne` est successivement assignée à chaque ligne résultant de la requête (une commande `SELECT`) et le corps de la boucle est exécuté pour chaque ligne.



Curseurs

- Structure du langage qui permet de s'affranchir des ensembles pour manipuler les résultats de requêtes ligne par ligne.
- Fonctionne comme une tête de lecture sur les résultats de requêtes



Curseurs

- Déclarer un curseur :

```
DECLARE cursor REFCURSOR;
```

- Ouvrir un curseur :

```
OPEN cursor FOR query;
```

- Manipuler une ligne de résultat :

```
FETCH cursor INTO cible;
```

Avance le curseur sur la ligne suivante et indique via FOUND si elle existe

- Fermer un curseur et libérer les ressources associées : `CLOSE cursor;`

```
CREATE OR REPLACE FUNCTION fct() RETURNS void AS $$
```

```
DECLARE
```

```
x1 varchar;
```

```
x2 varchar;
```

```
curs refcursor;
```

```
BEGIN
```

```
EXECUTE 'CREATE TABLE
```

```
temp(idempl varchar(5), typ varchar(20))';
```

```
OPEN curs FOR SELECT numE,typeEmpl FROM Emplacement;
```

```
LOOP
```

```
FETCH curs INTO x1,x2;
```

```
EXIT WHEN NOT FOUND;
```

```
INSERT INTO temp VALUES (x1,x2);
```

```
END LOOP;
```

```
CLOSE curs;
```

```
END;
```

```
$$ LANGUAGE plpgsql;
```

Example