Initiation à PL/pgSQL

Le langage procédural pour SQL de PostgreSQL

Nathalie Camelin – 2012/2013

Université du Maine

Institut Claude Chappe

D'après le cours de C. Fredouille et Y. Estève
Et toujours la doc postgre ...

Rappel sur le langage SQL

- SQL langage de communication avec la BDR

  → Possibilité de requêtes longues et complexes

- Architecture client/serveur
  - ex. serveur web / SGBDR: le client doit envoyer une ou plusieurs requêtes au serveur, attend les résultats, les traite, et renvoie si nécessaire d'autres requêtes au serveur
    - Allers/retours de requêtes entre le client et le serveur
    - Surcharge du réseau
    - Attente du client
- Ne contient pas les fonctionnalités d'un langage évolué

Les langages procéduraux pour SGBD

- Comprend les commandes DML
  - Interrogation avec SELECT
  - Manipulation avec INSERT, UPDATE, DELETE

!! Mais pas du tout : les commandes DDL

- Et aussi : une partie procédurale
  - Boucles
  - Conditions
  - Variables
  - Affectation
  - ...

Les langages procéduraux pour SGBD

- Les + du langage procédural
  - Spécification d'une suite d'étapes
    → Créer des blocs d'instructions
  - Dans l'architecture client/serveur, amélioration de la qualité de l'application
    - Traitement centralisé au niveau du serveur
      → réduction du nombre d'allers/retours
      → seuls les résultats nécessaires sont renvoyés
- Exemples de langages procéduraux SQL:
  - Oracle : PL/SQL
  - PostgreSQL : PL/pgSQL
  - Microsoft SQL Server : Transact SQL
  - DB2 : SQLPL

Structure en blocs

- Code PL/pgSQL organisé en bloc
- Structure en bloc :
  - **DECLARE** : Partie déclarative (facultatif)
  - **BEGIN** : Corps principal (obligatoire)
  - **EXCEPTION** : Traitement des « erreurs » (facultatif)
  - **END;** : Fin de bloc (obligatoire)
- Toute instruction se termine par un ;

Structure en blocs

- Syntaxe

```
[ << label >> ]
[DECLARE
   déclarations ]
BEGIN
   instructions
EXCEPTION
   traitements
END [label];
```

- Possibilité de nommer les blocs
- Possibilité d'imbriquer les blocs

PL/pgSQL et fonctions

- Placement des blocs dans une fonction
- CREATE [OR REPLACE] FUNCTION
  - Créé et nomme la fonction
  - Définit les paramètres d'entrée
  - Définit le type de retour
- $$: Délimiteur de début et fin de définition de la fonction
- Se termine par : *language plpgsql;*

Structure en blocs

```
CREATE OR REPLACE FUNCTION nom_fonction (paramètres)
RETURNS type AS $$
   DECLARE
     -- déclarations
   BEGIN
     -- sous-bloc
     DECLARE
         -- déclarations
     BEGIN
         -- corps
     END; -- fin du sous-bloc
   END; -- fin du bloc principal
$$ LANGUAGE plpgsql;
```

- Appel de la fonction dans une commande SQL

Exemple

```
CREATE OR REPLACE FUNCTION
        mafonction (param1 integer, param2 integer, param3 varchar(20))
RETURNS integer AS $$
DECLARE
    res integer ;
BEGIN
    RAISE notice '%',param3 ;
    res := param1+param2 ;
    RETURN res ;
END ;
$$ LANGUAGE plpgsql;
```

- Appel de la fonction dans une commande SQL

```
select  mafonction(5, 10, 'toto');
```

Paramètres et alias

- $n : n définit le numéro d'ordre du paramètre d'entrée

  $1 pour le premier paramètre, $2 pour le second …

- Donner un alias à un paramètre
  - Exemple avant la version 8.0

```
CREATE OR REPLACE FUNCTION taxe_ventes(real) RETURNS real AS $$
DECLARE
    sous_total ALIAS FOR $1 ;
BEGIN
    RETURN sous_total * 0.06;
END ;
$$ LANGUAGE plpgsql;
```

  - Exemple depuis la version 8.0

```
CREATE OR REPLACE FUNCTION taxe_ventes(sous_total real)
RETURNS real AS $$
BEGIN
    RETURN sous_total * 0.06;
END ;
$$ LANGUAGE plpgsql;
```

Paramètres de sortie

- Mot clé RETURN
- Exemple :

```
CREATE OR REPLACE FUNCTION somme_n_produits(x int, y int)
RETURNS RECORD AS $$
DECLARE
    somme int;
    produit int;
    mon_rec record;
BEGIN
    somme := x + y;
    produit := x * y;
    select somme,produit into mon_rec;
    RETURN mon_rec;
END;
$$ LANGUAGE plpgsql;
```

Paramètres de sortie

- Mot clé : OUT
  - Paramètre nommé en $n comme ceux d'entrée
  - Initié à la valeur NULL
  - Non défini lors de l'appel de la fonction
- Exemple :

```
CREATE OR REPLACE FUNCTION somme_n_produits (x int, y int,
                                             OUT somme int, OUT produit int) AS $$
BEGIN
    somme := x + y;
    produit := x * y;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT somme_n_produits(3,4) ;
```

Commentaires et affichage

- Commenter
  - une ligne par --
  - plusieurs lignes par /* *bloc commenté* */
- Envoi de messages
  - Mot clé RAISE
  - Différents niveaux : DEBUG, LOG, INFO, NOTICE, WARNING, EXCEPTION
  - Possibilité d'afficher des valeurs de variables : %

```
RAISE NOTICE 'Hello world to %', var_prenom;
```

Variables

2 types de variables:

- Variables de type SQL (ou PostgreSQL)
  - char, numeric, integer
  - date, boolean …
- Variables propres à PL/pgSQL
  - Types composites : record ou table
  - Types références : cursor
  - Variable FOUND

Déclarations

- Syntaxe :

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

- Exemples
  - uneChaine varchar NOT NULL;
  - uneConstante CONSTANT integer := 13;
  - unNombre integer DEFAULT 36;
  - unChamp nomTable.nomColonne%TYPE;
    -- prend le type de nomTable.nomColonne
  - uneLigne nomTable%ROWTYPE;
    -- idem, mais pour une ligne entière
  - uneLigne2 RECORD;
    -- pas de structure prédéfinie : dépend du contexte

Affectation

- Avec l'opérateur **:=**
- Avec l'instruction **SELECT** *cible* **INTO** :
  - *cible* : une variable record, ligne, une liste de variables …
- Exemple

```
DECLARE
    i INTEGER ;
    emp_record RECORD;
BEGIN
    i := 12;
    SELECT * INTO emp_record FROM emp WHERE ename like
'JONES' ;
    RAISE NOTICE 'Nom:%', emp_record.ename ;
END ;
```

Les branchements conditionnels

- IF *condition* THEN *traitements* END IF ;
- Exemple

```
IF nombre = 0 THEN
    resultat := 'zero';
ELSIF nombre > 0 THEN
    resultat := 'positif';
ELSIF nombre < 0 THEN
    resultat := 'negatif';
ELSE
    resultat := 'NULL';
END IF;
```

Exemple d'utilisation de *FOUND*

```
SELECT * INTO mon_enreg FROM emp WHERE ename =
mon_nom;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employé % non trouvé', mon_nom;
END IF;
```

Exemple d'utilisation de *IS NULL*

```
DECLARE
    rec_a RECORD;
BEGIN
     SELECT * INTO rec_a FROM users WHERE id_user=3;
      IF rec_a.homepage IS NULL THEN
        -- l'utilisateur n'a entré aucune page, renvoyer "http://"
         RETURN 'http://';
      ELSE
            RETURN rec_a.homepage ;
      END IF;
END;
```

Les boucles : LOOP

- LOOP [label] *traitements* END LOOP [label];
- EXIT permet de sortir de la boucle

```
LOOP
    -- quelques traitements
    IF nombre > 0 THEN
        EXIT; -- sortie de boucle
    END IF;
END LOOP;
```

```
LOOP
    -- quelques traitements
    EXIT WHEN nombre > 0;
END LOOP;
```

Les boucles : WHILE

- WHILE *condition* LOOP *traitements* END LOOP;

```
WHILE param_a > 0 AND param_b > 0
LOOP
    -- quelques traitements ici
END LOOP;
```

```
WHILE NOT param_c<=0
LOOP
    -- quelques traitements ici
END LOOP;
```

- EXIT permet de sortir avant la fin de la boucle

Les boucles : FOR

- FOR *nom* IN [REVERSE] *expression .. expression* LOOP *traitements* END LOOP;

```
DECLARE
    maxi INTEGER : =$1;
BEGIN
    FOR i IN 1..maxi LOOP
        RAISE NOTICE 'i: %', i; -- affiche tous les entiers de 1 à $1
    END LOOP;
END;
```

```
FOR i IN REVERSE 10..1 LOOP
    RAISE NOTICE 'i: %', i; -- affiche tous les entiers de 10 à 1
END LOOP;
```

Les boucles : FOR

- FOR *var_record* IN *requête* LOOP *traitements* END LOOP;

```
CREATE OR REPLACE FUNCTION show_emp() AS $$
DECLARE
    rec_a RECORD;
BEGIN
    FOR rec_a IN select * from emp
    LOOP
        RAISE NOTICE 'nom: %, empno: %', rec_a.ename,
rec_a.empno;
    END LOOP;
END;
$$ LANGUAGE plpgsql ;
```

Gestion des erreurs

- Réserver un traitement spécifique à chaque erreur
- 2 types d'erreurs :
  - Les erreurs internes à Postgre
  - Les « erreurs » définies par l'utilisateurs
- Différents types d'erreurs Postgres prédéfinies
  - Connection failure
  - Data exception
  - Integrity constraint violation
  - Syntax error
  - ...

Gestion des erreurs

- Lors d'une erreur
  - Arrêt de l'exécution du bloc principal
  - Exécution de bloc d'exception correspondant
- Le traitement reprend APRÈS le bloc exception
  - Tout de qui est « entre » est annulé
- Si une exception est levée, toutes les modifications induites par le bloc qui lève l'exception sont annulées
  - La base de données revient à son état initial
- D'où l'intérêt de bien gérer les blocs/sous-blocs
  - Effectuer certains traitements malgré l'échec d'autres

Gestion des erreurs

```
DECLARE
BEGIN
    BEGIN
    -- Bloc traitements pouvant lever une exception
        instruction1;
        instruction2    -- lève l'exception;
        instruction3    -- ne sera pas exécutée;
    EXCEPTION
        WHEN condition1 THEN traitement1;
        WHEN condition2 THEN traitement2;
        ...
    END;
    instruction4        -- exécutée;
END;
```

Exemple de gestion des erreurs

```
CREATE OR REPLACE FUNCTION test() RETURNS VOID AS $$
DECLARE
      x INTEGER :=0 ;
      y INTEGER :=0 ;
BEGIN
     INSERT INTO mon_tableau(prenom,nom) VALUES('Tom', 'Jones');
     BEGIN
          UPDATE mon_tableau SET prenom = 'Joe' WHERE nom = 'Jones';
          x := x + 1;
          x := x / y;
     EXCEPTION
          WHEN division_by_zero THEN
               RAISE NOTICE 'récupération de l''erreur division_by_zero';
      END;
      RETURN x;
END ;
$$ language plpgsql;
```

Gestion des erreurs

```
CREATE TABLE toto (a INT PRIMARY KEY, b TEXT);
```

```
CREATE FUNCTION maj(cle INT, donnee TEXT)
RETURNS VOID AS $$
BEGIN
    LOOP
        UPDATE toto SET b = donnee WHERE a = cle;
        IF FOUND THEN
            RETURN;
        END IF;
        BEGIN
            INSERT INTO toto VALUES (cle, donnee);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            NULL ; -- ne rien faire
        END;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT maj(1, 'david');
SELECT maj(1, 'dennis');
```

Les curseurs

- Structure permettant de manipuler les résultats de requêtes ligne par ligne
- 3 types de curseurs
  - Curseurs non liés
  - Curseurs liés
  - Curseurs liés paramétrés
- 4 étapes
  - Déclaration
  - Ouverture
  - Manipulation
  - Fermeture

Déclaration des curseurs

- Dans le bloc DECLARE
  - Dépend du type de curseur
- 3 types de curseurs
  - Curseurs non liés

```
nom_curseur_non_lie REFCURSOR ;
```

  - Curseurs liés

```
nom_curseur_lie CURSOR FOR SELECT * FROM nom_table;
```

  - Curseurs liés paramétrés

```
nom_curseur_lie_param CURSOR (nom_param type_param)
FOR SELECT * FROM nom_table
        WHERE nom_colonne = nom_param;
```

Ouverture des curseurs

- Dans le bloc BEGIN
  - Dépend du type de curseur
- 3 types de curseurs
  - Curseurs non liés

```
OPEN nom_curseur_non_lie FOR SELECT * FROM nom_table;
```

  - Curseurs liés

```
OPEN nom_curseur_lie ;
```

  - Curseurs liés paramétrés

```
OPEN nom_curseur_lie_param(valeur_param) ;
```

Manipulation des curseurs

- Traitement ligne à ligne des curseurs

```
FETCH curseur INTO cible ;
```

- Rapatriement de l'enregistrement suivant depuis le curseur vers une cible
- La cible est une variable de type :
  - ligne, record, liste de variables
- Exemple

```
FETCH curseur INTO nom_record;
```

```
FETCH curseur INTO nom_var1, nom_var2;
```

Fermeture des curseurs

- Mot clé : CLOSE

```
CLOSE nom_curseur_non_lie ;
```

```
CLOSE nom_curseur_lie;
```

```
CLOSE nom_curseur_lie_param;
```

Ex de curseurs et boucles

```
DECLARE
    cur_emp CURSOR FOR SELECT * FROM emp;
    ligne_emp emp%ROWTYPE;
BEGIN
    OPEN cur_emp;
    LOOP
        FETCH cur_emp INTO ligne_emp ;
        EXIT WHEN NOT FOUND;
        RAISE NOTICE 'nom: %, job: %', ligne_emp.ename, ligne_emp.job;
    END LOOP;
    CLOSE cur_emp;
END;
```

Les triggers

- Triggeur = Déclencheur
  - Déclenchement lors d'une opération particulière de manipulation sur les tables (insert, update ou delete -- pas select)
- Associé à une fonction trigger
- 2 types de triggers :
  - Trigger par ligne (ROW)
    - Appel pour chaque ligne affectée par l'opération
  - Trigger par instruction (STATEMENT)
    - Appel une seule fois pour l'ensemble des lignes
- Exécution avant ou après l'opération (BEFORE/AFTER)

Les triggers

- Définition de la fonction trigger

```
CREATE [ OR REPLACE ] FUNCTION nom_fonction( )
RETURNS TRIGGER AS $$
...
$$ LANGUAGE plpgsql;
```

- Définition d'un trigger

```
CREATE TRIGGER nom_trigger {BEFORE | AFTER} {opération}
    ON nom_table [ FOR [ EACH] {ROW| STATEMENT}]
    EXECUTE PROCEDURE nom_fonction(arguments) ;
```

- Destruction d'un trigger

```
DROP TRIGGER nom_trigger ON nom_table;
```

Les triggers

- Définition de la fonction trigger
  - Argument : aucun possible
  - Type retour : obligatoirement trigger (=NULL ou record).
    - Trigger BEFORE par ligne :
      - Si retour NULL → INSERT/UPDATE annulé
      - Si record → ajout ou modification ok
      - Pas d'influence en DELETE
    - Trigger AFTER par ligne :
      - Pas d'influence par type de retour
    - Trigger par instruction :
      - Pas d'influence par type de retour
  - Si une exception est levée pendant l'exécution de la fonction → opération associée annulée

Les triggers

- Variables spéciales
  - Pré-définies, créées et affectées par le gestionnaire de triggers
- Exemples
  - NEW : nouvelle ligne utilisée par INSERT/UPDATE
  - OLD : ancienne ligne utilisée par UPDATE/DELETE
  - TG_NAME : nom du trigger déclenché
  - TG_WHEN : AFTER/BEFORE
  - TG_LEVEL : RAW/STATEMENT
  - TG_OP : INSERT/UPDATE/DELETE
  - …

Exemple d'un trigger

```
CREATE FUNCTION emp_stamp() RETURNS trigger AS $$
    BEGIN
        -- Verifie que ename et sal sont donnés
        IF NEW.ename IS NULL THEN
            RAISE EXCEPTION 'ename ne peut pas être NULL';
        END IF;
        IF NEW.sal OR NEW.sal<0 IS NULL THEN
            RAISE EXCEPTION '% doit avoir un salaire positif', NEW.ename;
        END IF;
        RETURN NEW;
    END;
$$ LANGUAGE plpgsql;
```

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

Le musée -- version SQL uniquement

ŒUVRE (id_oeuv, tit_oeuv, date_oeuv, type_oeuv, period_oeuv, style_oeuv)

AUTEUR (id_aut, nom_aut, prenom_aut, dnais_aut, ddec aut)

REALISATION (#id_oeuv, #id_aut)

SALLE_EXPO (id_salle, nom_salle, sol_salle, eclair_salle)

EXPOSITION (#id_salle, #id_oeuvre)

STOCKAGE (id_res, #id_oeuvre, num_etag)

PROPRIO (id_prop, nom_prop, adr_prop, type_prop)

EMPRUNT (#id_oeuvre, #id_prop, ddeb_pret, duree_pret, sens_pret)

7 – Saisissez les enregistrements adéquats : !! Attention, l'acquisition d'une statue et sa localisation doivent se faire « simultanément ».

Une nouvelle œuvre est acquise par le musée :

- La statue « La Vénus de Milo » de la fin de l'époque hellénistique, œuvre d'Alexandre Antioche réalisée vers 130-100 av. J.C. ;

Elle sera localisée dans la nouvelle salle du musée :

- 34 : Salle Cezanne, parquet et lumière incandescente « flood »

Le Louvre nous prête une œuvre :

- Le tableau « La Joconde » réalisé par Léonard de Vinci en 1503, période romantique

Cette œuvre sera également localisée dans la salle Cezanne