R2.06 - Exploitation d'une Base de Données

Langage Procédural et Triggers

BUT Informatique

Département Informatique
IUT de Lannion
Université de Rennes

12 février 2024

Plan du cours

Motivations

Besoin : définir un ensemble de traitements complexes, soit effectués à la demande, soit déclenchés automatiquement par le SGBD losque certains phénomènes se produisent.
Outils :

- Un langage procédural : PL/pgSQL
  - spécifique à PostgreSQL
  - pas unique : PL/tcl, PL/Perl, PL/Python disponibles
- Les triggers ou déclencheurs
  - sont enregistrés dans la base (et non-pas dans l'application),
  - garantissent la la cohérence globale du S. I.,
  - dépassent largement la mise en place de contraintes d'intégrité.

PL/pgSQL : Introduction

PL/pgSQL est un langage de procédures chargeable pour le système de bases de données PostgreSQL™. Les objectifs de la conception de PL/pgSQL ont été de créer un langage de procédures chargeable qui

- est utilisé pour créer des fonctions standards et triggers,
- ajoute des structures de contrôle au langage SQL,
- permet d'effectuer des traitements complexes,
- hérite de tous les types, fonctions et opérateurs définis par les utilisateurs,
- est défini comme digne de confiance par le serveur,
- est facile à utiliser.

À partir de la version 9.0 de PostgreSQL™, PL/pgSQL est installé par défaut. Il reste toutefois un module chargeable et les administrateurs craignant pour la sécurité de leur instance pourront le retirer. Documentaton disponible ici.

Plan : 2 - Langage procédural PL/pgSQL

PL/pgSQL : Structure du langage

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>> ]
[ DECLARE
        déclarations ]
BEGIN
        instructions
END [ label ];
```

Chaque déclaration et chaque expression au sein du bloc est terminé par un point-virgule.

PL/pgSQL : Structure du langage - fonctions

```
CREATE FUNCTION une_fonction() RETURNS integer AS $$
<< blocexterne >>
   DECLARE
    quantite integer := 30;
BEGIN
    RAISE NOTICE 'quantité vaut ici %', quantite;
    quantite := 50;
    -- Crée un sous-bloc
    DECLARE
        quantite integer := 80;
    BEGIN
        RAISE NOTICE 'quantite vaut ici %', quantite;
        RAISE NOTICE 'la quantité externe vaut ici %',
              blocexterne.quantite;
    END;

    RAISE NOTICE 'quantité vaut ici %', quantite;

    RETURN quantite;
END;
$$ LANGUAGE plpgsql;
```

Plan : 2 - Langage procédural PL/pgSQL

PL/pgSQL : Déclarations

Toutes les variables utilisées dans un bloc doivent être déclarées dans la section déclaration du bloc.
La syntaxe générale d'une déclaration de variable est :

```
nom [ CONSTANT ] type [ COLLATE nom_collationnement ]
    [ { DEFAULT | := } expression ];
```

Exemples :

```
id_utilisateur integer;
quantité numeric(5);
url varchar;
ma_ligne nom_table%ROWTYPE;
mon_champ nom_table.nom_colonne%TYPE;
une_ligne RECORD;
```

Plan : 2 - Langage procédural PL/pgSQL

PL/pgSQL : Instructions de base

L'affectation d'une valeur à une variable PL/pgSQL s'écrit ainsi :

```
variable := expression;
```

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. Ceci se fait en écrivant la commande SQL de base et en ajoutant une clause INTO. Par exemple,

```
SELECT expressions_select INTO [STRICT] cible FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] cible;
UPDATE ... RETURNING expressions INTO [STRICT] cible;
DELETE ... RETURNING expressions INTO [STRICT] cible;
```

où `cible` peut être une variable de type record, row ou une liste de variables ou de champs record/row séparées par des virgules.

PL/pgSQL : Instructions de base

Vous pouvez vérifier la valeur de la variable spéciale *FOUND* pour déterminer si une ligne a été renvoyée :

```
SELECT * INTO monrec FROM emp WHERE nom = mon_nom;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employé % introuvable', mon_nom;
END IF;
```

PL/pgSQL : Instructions de base

Créer dynamiquement des requêtes SQL est un besoin habituel dans les fonctions PL/pgSQL, par exemple des requêtes qui impliquent différentes tables ou différents types de données à chaque fois qu'elles sont exécutées. L'instruction EXECUTE est proposée :

```
EXECUTE command-string [ INTO [STRICT] target ]
    [ USING expression [, ...] ];
```

Exemple :

```
EXECUTE 'SELECT count(*) FROM matable '
        || 'WHERE insere_par = $1 AND insere <= $2'
INTO c
USING utilisateur_verifie, date_verifiee;
```

Plan : 2 - Langage procédural PL/pgSQL

PL/pgSQL : Structures de contrôle - Contrôles conditionnels

Les instructions IF et CASE vous permettent d'exécuter des commandes basées sur certaines conditions. PL/pgSQL a trois formes de IF :

- `IF ... THEN`
- `IF ... THEN ... ELSE`
- `IF ... THEN ... ELSIF ... THEN ... ELSE`

qui se terminent toujours par un `END IF;`

et deux formes de CASE :

- `CASE ... WHEN ... THEN ... ELSE ... END CASE`
- `CASE WHEN ... THEN ... ELSE ... END CASE`

PL/pgSQL : Structures de contrôle - Boucles while

```
[<<label>>]
WHILE expression-booléenne LOOP
    instructions
END LOOP [ label ];
```

L'instruction WHILE répète une séquence d'instructions aussi longtemps que expression-booléenne est évaluée à vrai. L'expression est vérifiée juste avant chaque entrée dans le corps de la boucle.
Par exemple :

```
WHILE montant_possede > 0 AND balance_cadeau > 0 LOOP
    -- quelques traitements ici
END LOOP;

WHILE NOT termine LOOP
    -- quelques traitements ici
END LOOP;
```

PL/pgSQL : Structures de contrôle - Boucles for sur entier

```
[<<label>>]
FOR nom IN [ REVERSE ] expression .. expression [ BY e
    instruction
END LOOP [ label ];
```

Cette forme de FOR crée une boucle qui effectue une itération sur une plage de valeurs entières. La variable nom est automatiquement définie comme un type integer et n'existe que dans la boucle.

PL/pgSQL : Structures de contrôle - Boucles dans les résultats de requêtes

```
[<<label>>]
FOR cible IN requête LOOP
    instructions
END LOOP [ label ];
```

La cible est une variable de type record, row ou une liste de variables scalaires séparées par une virgule. La cible est affectée successivement à chaque ligne résultant de la requête et le corps de la boucle est exécuté pour chaque ligne.

PL/pgSQL : Structures de contrôle - Boucles dans les résultats de requêtes

Voici un exemple :

```
\begin{verbatim}
CREATE FUNCTION cs_rafraichir_vuemat() RETURNS integer AS $$
DECLARE
    vues_mat RECORD;
BEGIN
    RAISE NOTICE 'Rafraichissement des vues matérialisées...';
    FOR vues_mat IN
       SELECT * FROM cs_vues_materialisees ORDER BY cle_tri LOOP
        -- À présent vues_mat contient un enregistrement de
        -- cs_vues_materialisees
            RAISE NOTICE 'Rafraichissement de la vue matérialisée %s ...',
                    quote_ident(mviews.mv_name);
            EXECUTE 'TRUNCATE TABLE  ' || quote_ident(vues_mat.vm_nom);
            EXECUTE 'INSERT INTO '
                    || quote_ident(vues_mat.vm_nom) || ' '
                    || vues_mat.vm_requete;
    END LOOP;
    RAISE NOTICE 'Fin du rafraichissement des vues matérialisées.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Plan : 3 - Déclencheurs : utilité et définition

Types de contraintes dynamiques définissables

- contraintes concernant le passage de la base d'un état à un autre
- génération automatique de clé primaire
- résoudre le problème de mise à jour en cascade
- enregistrer les accès à une table
- gérer automatiquement les redondances
- confidentialité : empêcher certaines modifications
- mettre en oeuvre des règles de fonctionnement compliquées

Plan : 3 - Déclencheurs : utilité et définition

Syntaxe de création PostgreSQL

```
CREATE TRIGGER <nom_trigger>
  { BEFORE | AFTER | INSTEAD OF}
  { DELETE | INSERT | UPDATE }
  ON <nom_table>
  [FOR EACH { ROW | STATEMENT }]
  [WHEN <condition>]
  EXECUTE PROCEDURE <nom_fonction> (<arguments>);
```

Explications :

- `{ BEFORE | AFTER | INSTEAD OF}` : détermine si la fonction est appelée avant ou après l'événement, ou à la place de l'événement.
- `{ DELETE | INSERT | UPDATE }` : précise l'événement qui active le déclencheur.

Syntaxe de création PostgreSQL

```
CREATE TRIGGER <nom_trigger>
  { BEFORE | AFTER | INSTEAD OF}
  { DELETE | INSERT | UPDATE }
  ON <nom_table>
  [FOR EACH { ROW | STATEMENT }]
  [WHEN <condition>]
  EXECUTE PROCEDURE <nom_fonction> (<arguments>);
```

Explications :

- `FOR EACH {ROW|STATEMENT}` : précise si la procédure du déclencheur doit être lancée pour chaque ligne affectée par l'événement ou simplement pour chaque instruction SQL. L'absence de clause spécifie un déclencheur de niveau commande : exécution une seule fois.

Syntaxe de création PostgreSQL

```
CREATE TRIGGER <nom_trigger>
  { BEFORE | AFTER | INSTEAD OF}
  { DELETE | INSERT | UPDATE }
  ON <nom_table>
  [FOR EACH { ROW | STATEMENT }]
  [WHEN <condition>]
  EXECUTE PROCEDURE <nom_fonction> (<arguments>);
```

Explications :

- `<nom_fonction>` : nom d'une fonction utilisateur renvoyant le type `TRIGGER`.
- `WHEN` : précision d'une condition supplémentaire sur l'exécution de la fonction. Par exemple, vérifier lors d'une mise à jour que la nouvelle valeur dépasse l'ancienne.

Syntaxe (compléments) PostgreSQL

- Pour supprimer la définition d'un déclencheur on utilise :

```
DROP TRIGGER <nom_trigger>
  ON <nom_table>;
```

- Remarque : Les triggers font partie du standard SQL3. PostgreSQL implante un sous-ensemble du standard.
- Conception :
  - Sur quelle table ? Quel type (instruction ou n-uplet) ? Avant, après, au lieu de ?
  - Construire une fonction préalable qui réalisera le traitement.

PL/pgSQL

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>> ]
[ DECLARE
    déclarations ]
BEGIN
    instructions
END [ label ];
```

Fonctions PL/pgSQL

```
CREATE FUNCTION une_fonction() RETURNS integer AS $$
<< blocexterne >>
DECLARE
    quantite integer := 30;
BEGIN
    RAISE NOTICE 'quantité vaut ici %', quantite;  -- affiche 30
    quantite := 50;
    --
    -- Crée un sous-bloc
    --
    DECLARE
        quantite integer := 80;
    BEGIN
        RAISE NOTICE 'quantite vaut ici %', quantite;  -- affiche 80
        RAISE NOTICE 'la quantité externe vaut ici %', blocexterne.quantite;  -- affiche 50
    END;

    RAISE NOTICE 'quantité vaut ici %', quantite;  -- affiche 50

    RETURN quantite;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL : Erreurs, exceptions et messages

Lever une exception, rapporter une erreur ou un message :

```
RAISE { DEBUG | LOG | INFO | NOTICE |
        WARNING | EXCEPTION }
    <'message'> [,<expression> [,...] ];
```

où

- `DEBUG|LOG|INFO|NOTICE|WARNING|EXCEPTION` permettent de générer un message à leur niveau de priorité respectif. Pour envoyer un message sur la sortie standard, il faut utiliser `RAISE NOTICE`.
- `RAISE EXCEPTION` lève une exception avec le message d'erreur `<'message'>` et arrête l'exécution de la fonction.

PL/pgSQL : Erreurs, exceptions et messages

Récupérer une exception :

```
[ DECLARE
    <déclarations> ]
BEGIN
    <instructions>
EXCEPTION
    WHEN <exception> [OR <exception> [...] ]
    THEN
       <instruction_gestion_exception>
    [ WHEN <exception> [OR <exception> [...] ]
      THEN
       <instruction_gestion_exception>
    [...] ]
END [ label ];
```

Plan : 3 - Déclencheurs : utilité et définition

Visibilité des modifications des données I

Si, dans la fonction, vous accédez à la table pour laquelle vous créez le déclencheur, il faut connaître les règles de visibilité des données car elles déterminent si les commandes SQL voient les modifications de données pour lesquelles est exécuté le déclencheur.

- Les déclencheurs niveau instruction suivent des règles de visibilité simples : aucune des modifications réalisées par une instruction n'est visible aux déclencheurs niveau instruction appelés avant l'instruction alors que toutes les modifications sont visibles aux déclencheurs AFTER niveau instruction.
- Les modifications de données (insertion, mise à jour ou suppression) lançant le déclencheur ne sont naturellement pas visibles aux commandes SQL exécutées dans un déclencheur BEFORE en mode ligne parce qu'elles ne sont pas encore survenues.

Visibilité des modifications des données II

- Néanmoins, les commandes SQL exécutées par un déclencheur BEFORE en mode ligne verront les effets des modifications de données pour les lignes précédemment traitées dans la même commande externe. Ceci requiert une grande attention car l'ordre des événements de modification n'est en général pas prévisible ; une commande SQL affectant plusieurs lignes pourrait visiter les lignes dans n'importe quel ordre.
- De façon similaire, un trigger niveau ligne de type INSTEAD OF verra les effets des modifications de données réalisées par l'exécution des autres triggers INSTEAD OF dans la même commande.
- Quand un déclencheur AFTER en mode ligne est exécuté, toutes les modifications de données réalisées par la commande externe sont déjà terminées et sont visibles par la fonction appelée par le déclencheur.

Plan : 4 - Déclencheurs : Exemples

Contrainte dynamique

Le salaire d'un employé ne peut pas diminuer.

```
CREATE FUNCTION salaire_invalide()
  RETURNS TRIGGER AS $BODY$
  DECLARE
       salaire_diminue EXCEPTION;
  BEGIN
       RAISE EXCEPTION 'Le salaire ne peut diminuer !';
  END ;
$BODY$
LANGUAGE 'plpgsql' ;

CREATE TRIGGER upd_salaire_personnel
BEFORE UPDATE OF salaire
ON personnel
FOR EACH ROW
WHEN (old.salaire > new.salaire)
EXECUTE PROCEDURE salaire_invalide() ;
```

Plan : 4 - Déclencheurs : Exemples

Contrainte à cheval sur plusieurs tables

Ne pas enregistrer d'inscription d'un étudiant à un cours si les crédits ECTS ne sont pas renseignés.

```
CREATE FUNCTION inscription_cours_valide()
    RETURNS TRIGGER AS $BODY$
DECLARE
    cred INTEGER;
BEGIN
    SELECT credit INTO cred FROM cours
    WHERE id_cours =  NEW.id_cours;
    IF cred IS NULL THEN
       RAISE EXCEPTION $$Inscription impossible au
       cours % car son nombre de crédits n'est pas
       défini$$, NEW.id_cours;
    ELSE RETURN NEW;
    END IF;
END;
$BODY$
LANGUAGE 'plpgsql';
```

Contrainte à cheval sur plusieurs tables

```
CREATE TRIGGER tg_inscription_cours_valide
BEFORE INSERT OR UPDATE
ON suivre
FOR EACH ROW
EXECUTE PROCEDURE inscription_cours_valide();
```

Plan : 4 - Déclencheurs : Exemples

Audit des modifications sur une table

```
CREATE FUNCTION audit_cours() RETURNS TRIGGER AS $BODY$
BEGIN
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO cours_audit
            SELECT 'D', NOW(), user, OLD.* ;
        RETURN OLD ;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO cours_audit
            SELECT 'U', NOW(), user, NEW.* ;
        RETURN OLD ;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO cours_audit
            SELECT 'I', NOW(), user, NEW.* ;
        RETURN OLD ;
    ENDIF;
    RETURN NULL ;
END;
$BODY$
LANGUAGE 'plpgsql';
```

Audit des modifications sur une table

```
CREATE TRIGGER tg_cours_audit
AFTER INSERT OR UPDATE OR DELETE
ON cours
FOR EACH ROW
EXECUTE PROCEDURE audit_cours();
```

Remarque : le retour de la fonction est ignoré car elle est appelée par un déclencheur AFTER.