37.7. Structures de contrôle

Les structures de contrôle sont probablement la partie la plus utile (et importante) de PL/pgSQL. Grâce aux structures de contrôle de PL/pgSQL, vous pouvez manipuler les données PostgreSQL™ de façon très flexible et puissante.

37.7.1. Retour d'une fonction

Il y a deux commandes disponibles qui vous permettent de renvoyer des données d'une fonction : **RETURN** et **RETURN NEXT**.

37.7.1.1. **RETURN**

```
RETURN expression;
```

RETURN accompagné d'une expression termine la fonction et renvoie le valeur de l'*expression* à l'appelant. Cette forme est à utiliser avec des fonctions PL/pgSQL qui ne renvoient pas d'ensemble de valeurs.

Lorsqu'elle renvoie un type scalaire, n'importe quelle expression peut être utilisée. Le résultat de l'expression sera automatiquement converti vers le type de retour de la fonction, comme décrit pour les affectations. Pour renvoyer une valeur composite (ligne), vous devez écrire une variable record ou ligne comme *expression*.

Si vous déclarez la fonction avec des paramètres en sortie, écrivez seulement **RETURN** sans expression. Les valeurs courantes des paramètres en sortie seront renvoyées.

Si vous déclarez que la fonction renvoie void, une instruction **RETURN** peut être utilisée pour quitter rapidement la fonction ; mais n'écrivez pas d'expression après **RETURN**.

La valeur de retour d'une fonction ne peut pas être laissée indéfinie. Si le contrôle atteint la fin du bloc de haut niveau de la fonction, sans parvenir à une instruction **RETURN**, une erreur d'exécution survient. Néanmoins, cette restriction ne s'applique pas aux fonctions sans paramètre de sortie et aux fonctions renvoyant void. Dans ces cas, une instruction **RETURN** est automatiquement exécutée si le bloc de haut niveau est terminé.

37.7.1.2. **RETURN NEXT**

```
RETURN NEXT expression;
```

Lorsqu'une fonction PL/pgSQL est déclarée renvoyer `SETOF` *type quelconque*, la procédure à suivre est légèrement différente. Dans ce cas, les éléments 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 être utilisé avec des types scalaires et des types composites de données ; avec un type de résultat composite, une « table » entière de résultats sera renvoyée.

RETURN NEXT ne sort pas vraiment de la fonction -- il met simplement de côté la valeur de l'expression. Puis, l'exécution continue avec la prochaine instruction de la fonction PL/pgSQL. Au fur et à mesure que des commandes **RETURN NEXT** successives sont exécutées, l'ensemble de résultat est construit. Un **RETURN** final, qui pourrait ne pas avoir d'argument, fait sortir de la fonction (mais vous pouvez aussi atteindre la fin de la fonction).

Si vous avez déclaré la fonction avec des paramètres en sortie, écrivez simplement **RETURN NEXT** sans expression. Les valeurs en cours de(s) paramètre(s) en sortie seront sauvegardées pour un retour éventuel. Notez que vous devez déclarer la fonction comme renvoyant `SETOF record` s'il y a plusieurs paramètres en sortie ou `SETOF` *`untype`* quand il y a un seul paramètre en sortie de type *`untype`*, pour créer une fonction renvoyant un ensemble avec les paramètres en sortie.

Les fonctions qui utilisent **RETURN NEXT** devraient être appelées d'après le modèle suivant :

```
SELECT * FROM une_fonction();
```

En fait, la fonction doit être utilisée comme une table source dans une clause `FROM`.

Note

L'implémentation actuelle de **RETURN NEXT** pour PL/pgSQL emmagasine la totalité de l'ensemble des résultats avant d'effectuer le retour de la fonction, comme vu plus haut. Cela signifie que si une fonction PL/pgSQL produit une structure résultat très grande, les performances peuvent être faibles : les données seront écrites sur le disque pour éviter un épuisement de la mémoire mais la fonction en elle-même ne renverra rien jusqu'à ce que l'ensemble des résultats entier soit généré. Une version future de PL/pgSQL pourra permettre aux utilisateurs de définir des fonctions renvoyant des ensembles qui n'auront pas cette limitation. Actuellement, le point auquel les données commencent à être écrites sur le disque est contrôlé par la variable de configuration work_mem. Les administrateurs ayant une mémoire suffisante pour enregistrer des ensembles de résultats plus importants en mémoire devraient envisager l'augmentation de ce paramètre.

37.7.2. Contrôles conditionnels

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

```
IF ... THEN

IF ... THEN ... ELSE

IF ... THEN ... ELSE IF

IF ... THEN ... ELSIF ... THEN ... ELSE

IF ... THEN ... ELSEIF ... THEN ... ELSE
```

37.7.2.1. `IF-THEN`

```
IF expression-booleenne THEN
    instructions
END IF;
```

Les instructions `IF-THEN` sont la forme la plus simple de `IF`. Les instructions entre `THEN` et `END IF` seront exécutées si la condition est vraie. Autrement, elles seront ignorées.

Exemple :

```
IF v_id_utilisateur <> 0 THEN
    UPDATE utilisateurs SET email = v_email WHERE id_utilisateur = v_id_utilisateur;
END IF;
```

37.7.2.2. `IF-THEN-ELSE`

```
IF expression-booleenne THEN
    instructions
ELSE
    instructions
END IF;
```

Les instructions IF-THEN-ELSE s'ajoutent au IF-THEN en vous permettant de spécifier un autre ensemble d'instructions à exécuter si la condition est fausse.

Exemples :

```
IF id_parent IS NULL OR id_parent = ''
THEN
    RETURN nom_complet;
ELSE
    RETURN hp_true_filename(id_parent) || '/' || nom_complet;
END IF;
```

```
IF v_nombre > 0 THEN
    INSERT INTO nombre_utilisateurs (nombre) VALUES (v_nombre);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

37.7.2.3. IF-THEN-ELSE IF

Les instructions IF peuvent être imbriquées, comme dans l'exemple suivant :

```
IF demo_ligne.sexe = 'm' THEN
    texte_sexe := ''homme'';
ELSE
    IF demo_ligne.sexe = 'f' THEN
        texte_sexe := 'femme';
    END IF;
END IF;
```

Lorsque vous utilisez cette forme, vous imbriquez une instruction IF dans la partie ELSE d'une instruction IF extérieure. Ainsi, vous avez besoin d'une instruction END IF pour chaque IF imbriqué et une pour le IF-ELSE parent. Ceci fonctionne mais devient fastidieux quand il y a de nombreuses alternatives à traiter. Considérez alors la forme suivante.

37.7.2.4. IF-THEN-ELSIF-ELSE

```
IF expression-booleenne THEN
    instructions
[ ELSIF expression-booleenne THEN
    instructions
[ ELSIF expression-booleenne THEN
    instructions
    ...
]
]
[ ELSE
    instructions ]
END IF;
```

IF-THEN-ELSIF-ELSE fournit une méthode plus pratique pour vérifier de nombreuses alternatives en une instruction. Elle est équivalente formellement aux commandes IF-THEN-ELSE-IF-THEN imbriquées, mais un seul END IF est nécessaire.

Voici un exemple :

```
IF nombre = 0 THEN
    resultat := 'zero';
ELSIF nombre > 0 THEN
    resultat := 'positif';
ELSIF nombre < 0 THEN
```

```
    resultat := 'negatif';
ELSE
    -- hmm, la seule possibilité est que le nombre soit NULL
    resultat := 'NULL';
END IF;
```

37.7.2.5. IF-THEN-ELSEIF-ELSE

ELSEIF est un alias pour ELSIF.

37.7.3. Boucles simples

Grâce aux instructions LOOP, EXIT, CONTINUE, WHILE et FOR, vous pouvez faire en sorte que vos fonctions PL/pgSQL répètent une série de commandes.

37.7.3.1. LOOP

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

LOOP définit une boucle inconditionnelle répétée indéfiniment jusqu'à ce qu'elle soit terminée par une instruction EXIT ou **RETURN**. Le *label* optionnel peut être utilisé par les instructions EXIT et CONTINUE dans le cas de boucles imbriquées pour définir la boucle impliquée.

37.7.3.2. EXIT

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

Si aucun *label* n'est donné, la boucle la plus imbriquée se termine et l'instruction suivant END LOOP est exécutée. Si un *label* est donné, ce doit être le label de la boucle, du bloc courant ou d'un niveau moins imbriqué. La boucle ou le bloc nommé se termine alors et le contrôle continue avec l'instruction située après le END de la boucle ou du bloc correspondant.

Si WHEN est spécifié, la sortie de boucle ne s'effectue que si *expression* vaut true. Sinon, le contrôle passe à l'instruction suivant le EXIT.

EXIT peut être utilisé pour tous les types de boucles ; il n'est pas limité aux boucles non conditionnelles. Lorsqu'il est utilisé avec un bloc BEGIN, EXIT passe le contrôle à la prochaine instruction après la fin du bloc.

Exemples :

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

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

BEGIN
    -- quelques traitements
    IF stocks > 100000 THEN
        EXIT;  -- cause la sortie (EXIT) du bloc BEGIN
    END IF;
END;
```

37.7.3.3. CONTINUE

```
CONTINUE [ label ] [ WHEN expression ];
```

Si aucun *label* n'est donné, la prochaine itération de la boucle interne est commencée. C'est-à-dire que le contrôle est redonné à l'expression de contrôle de la boucle (s'il y en a une) et le corps de la boucle est ré-évaluée. Si le *label* est présent, il spécifie le label de la boucle dont l'exécution va être continué.

Si WHEN est spécifié, la prochaine itération de la boucle est commencée seulement si l'*expression* vaut true. Sinon, le contrôle est passé à l'instruction après CONTINUE.

CONTINUE peut être utilisé avec tous les types de boucles ; il n'est pas limité à l'utilisation des boucles inconditionnelles.

Exemples :

```
LOOP
    -- quelques traitements
    EXIT WHEN nombre > 100;
    CONTINUE WHEN nombre < 50;
    -- quelques traitements pour nombre IN [50 .. 100]
END LOOP;
```

37.7.3.4. WHILE

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

L'instruction WHILE répète une séquence d'instructions aussi longtemps que l'expression conditionnelle est évaluée à vrai. La condition 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 expression_booleenne LOOP
    -- quelques traitements ici
END LOOP;
```

37.7.3.5. FOR (variante avec entier)

```
[<<label>>]
FOR nom IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    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 (toute définition de la variable est ignorée à l'intérieur de la boucle). Les deux expressions donnant les limites inférieures et supérieures de la plage sont évaluées une fois en entrant dans la boucle. Si la clause BY n'est pas spécifiée, l'étape d'itération est de 1, sinon elle est de la valeur spécifiée dans la clause BY. Si REVERSE est indiquée, alors la valeur de l'étape est considérée négative.

Quelques exemples de boucles FOR avec entiers :

```
FOR i IN 1..10 LOOP
    -- quelques calculs ici
    RAISE NOTICE 'i is %', i;
```

```
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- quelques calculs ici
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- quelques calculs ici
    RAISE NOTICE 'i is %', i;
END LOOP;
```

Si la limite basse est plus grande que la limite haute (ou moins grande dans le cas du REVERSE), le corps de la boucle n'est pas exécuté du tout. Aucune erreur n'est renvoyée.

37.7.4. Boucler dans les résultats de requêtes

En utilisant un type de FOR différent, vous pouvez itérer au travers des résultats d'une requête et par là-même manipuler ces données. La syntaxe est la suivante :

```
[<<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. Voici un exemple :

```
CREATE FUNCTION cs_rafraichir_vuemat() RETURNS integer AS $$
DECLARE
    vues_mat RECORD;
BEGIN
    PERFORM cs_log('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

        PERFORM cs_log('Rafraichissement de la vue matérialisée ' || quote_ident(vues_ma
        EXECUTE 'TRUNCATE TABLE  ' || quote_ident(vues_mat.vm_nom);
        EXECUTE 'INSERT INTO ' || quote_ident(vues_mat.vm_nom) || ' ' || vues_mat.vm_rec
    END LOOP;

    PERFORM cs_log('Fin du rafraichissement des vues matérialisées.');
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Si la boucle est terminée par une instruction EXIT, la dernière valeur ligne affectée est toujours accessible après la boucle.

La *requête* utilisée dans ce type d'instruction FOR peut être toute commande SQL qui renvoie des lignes à l'appelant : **SELECT** est le cas le plus commun mais vous pouvez aussi utiliser **INSERT**, **UPDATE** ou **DELETE** avec une clause RETURNING. Certaines commandes comme **EXPLAIN** fonctionnent aussi.

L'instruction FOR-IN-EXECUTE est un moyen d'itérer sur des lignes :

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

Ceci est identique à la forme précédente, à ceci près que l'expression de la requête source est spécifiée comme une expression chaîne, évaluée et replanifiée à chaque entrée dans la boucle FOR. Ceci permet au développeur de choisir entre la vitesse d'une requête préplanifiée et la flexibilité d'une requête dynamique, uniquement avec l'instruction **EXECUTE**.

Note

L'analyseur PL/pgSQL distingue actuellement deux types de boucles FOR (entier ou résultat d'une requête) en vérifiant si .. apparaît à l'extérieur des parenthèses entre IN et LOOP. Si .. n'est pas trouvé, la boucle est supposée être une boucle entre des lignes. Une mauvaise saisie de .. amènera donc une plainte du type « loop variable of loop over rows must be a record or row variable or list of scalar variables » (NdT : une variable de boucle d'une boucle sur des enregistrements doit être un enregistrement ou une variable de type ligne) plutôt qu'une simple erreur de syntaxe comme vous pourriez vous y attendre.

37.7.5. Récupérer les erreurs

Par défaut, toute erreur survenant dans une fonction PL/pgSQL annule l'exécution de la fonction mais aussi de la transaction qui l'entoure. Vous pouvez récupérer les erreurs en utilisant un bloc **BEGIN** avec une clause EXCEPTION. La syntaxe est une extension de la syntaxe habituelle pour un bloc **BEGIN** :

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

Si aucune erreur ne survient, cette forme de bloc exécute simplement toutes les *instructions* puis passe le contrôle à l'instruction suivant END. Mais si une erreur survient à l'intérieur des *instructions*, le traitement en cours des *instructions* est abandonné et le contrôle est passé à la liste d'EXCEPTION. Une recherche est effectuée sur la liste pour la première *condition* correspondant à l'erreur survenue. Si une correspondance est trouvée, les *instructions_gestion_erreurs* correspondantes sont exécutées puis le contrôle est passé à l'instruction suivant le END. Si aucune correspondance n'est trouvée, l'erreur se propage comme si la clause EXCEPTION n'existait pas du tout : l'erreur peut être récupérée par un bloc l'enfermant avec EXCEPTION ou, s'il n'existe pas, elle annule le traitement de la fonction.

Les noms des *condition* sont indiquées dans l'Annexe A, Codes d'erreurs de PostgreSQL. Un nom de catégorie correspond à toute erreur contenue dans cette catégorie. Le nom de condition spéciale OTHERS correspond à tout type d'erreur sauf QUERY_CANCELED (il est possible, mais pas recommandé, de récupérer QUERY_CANCELED par son nom). Les noms des conditions ne sont pas sensibles à la casse.

Si une nouvelle erreur survient à l'intérieur des *instructions_gestion_erreurs* sélectionnées, elle ne peut pas être récupérée par cette clause EXCEPTION mais est propagée en dehors. Une clause EXCEPTION l'englobant pourrait la récupérer.

Quand une erreur est récupérée par une clause EXCEPTION, les variables locales de la fonction PL/pgSQL restent dans le même état qu'au moment où l'erreur est survenue mais toutes les modifications à l'état persistant de la base de données à l'intérieur du bloc sont annulées. Comme exemple, considérez ce fragment :

```
INSERT INTO mon_tableau(prenom, nom) VALUES('Tom', 'Jones');
BEGIN
  UPDATE mon_tableau SET prenom = 'Joe' WHERE nom = 'Jones';
  x := x + 1;
  y := x / 0;
  EXCEPTION
    WHEN division_by_zero THEN
      RAISE NOTICE 'récupération de l''erreur division_by_zero';
```

```
RETURN x;
END;
```

Quand le contrôle parvient à l'affectation de `y`, il échouera avec une erreur `division_by_zero`. Elle sera récupérée par la clause `EXCEPTION`. La valeur renvoyée par l'instruction **RETURN** sera la valeur incrémentée de `x` mais les effets de la commande **UPDATE** auront été annulés. La commande **INSERT** précédant le bloc ne sera pas annulée, du coup le résultat final est que la base de données contient `Tom Jones` et non pas `Joe Jones`.

Astuce

Un bloc contenant une clause `EXCEPTION` est significativement plus coûteuse en entrée et en sortie qu'un bloc sans. Du coup, n'utilisez pas `EXCEPTION` sans besoin.

À l'intérieur d'un gestionnaire d'exceptions, la variable `SQLSTATE` contient le code d'erreur correspondant à l'exception qui a été levée (référez-vous au Tableau A.1, « Codes d'erreur de PostgreSQL » pour une liste des codes d'erreurs possibles). La variable `SQLERRM` contient le message d'erreur associé avec l'exception. Ces variables sont indéfinies à l'extérieur des gestionnaires d'exceptions.

Exemple 37.1. Exceptions avec UPDATE/INSERT

Cet exemple utilise un gestionnaire d'exceptions pour réaliser soit un **UPDATE** soit un **INSERT**, comme approprié.

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

CREATE FUNCTION fusionne_base(cle INT, donnee TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
        UPDATE base SET b = donnee WHERE a = cle;
        IF found THEN
            RETURN;
        END IF;

        BEGIN
            INSERT INTO base(a,b) VALUES (cle, donnee);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- do nothing
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

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