Introduction à PL-SQL

(PL/pgSQL avec PostgreSQL)

D. Gonzalez

Université Lille 3 - Charles de Gaulle

Janvier 2004

Sommaire V-PL / PGSQL

Chapter 50

PL/pgSQL : présentation

50.1 Origine

Les pages qui concernent PL/pgSQL sont fortement inspirées[1] du chapitre qui lui est consacré dans une traduction de la documentation officielle qu'on peut trouver à cette adresse :

`http://pgsql-fr.tuxfamily.org/pgsql-fr/plpgsql.html`

Je n'ai fait qu'en modifier la présentation.

50.2 PL/pgSQL ?

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

- peut être utilisé pour créer des procédures fonctions et déclencheurs,
- ajoute des structures de contrôle au langageSQL,
- peut effectuer des traitements complexes,
- hérite de tous les types, fonctions et opérateurs définis par les utilisateurs,
- peut être défini comme digne de confiance par le serveur,
- est facile à utiliser.

50.3 Installer PL/pgSQL

Pour être utilisé, PL/pgSQL doit être installé. Cela peut se faire au niveau du serveur (en fait dans la base `template1`), et il sera automatiquement utilisable dans les tables créées ensuite.
Si ce n'est pas le cas :

```
CREATE FUNCTION plpgsql_call_handler () RETURNS OPAQUE AS
      '$libdir/plpgsql' LANGUAGE C;
CREATE TRUSTED PROCEDURAL LANGUAGE plpgsql
      HANDLER plpgsql_call_handler;
```

La première commande dit au serveur de bases de données où trouver les informations nécessaires pour manipuler les appels de fonctions pour PL/pgSQL.
La deuxième commande définit l'objet précedemment déclaré comme devant manipuler des objets dont l'attribut langage sera `plpgsql`.

[1] C'est une manière hypocrite de ne pas dire « intégralement recopiées »...

50.4 Survol

Le gestionnaire d'appel PL/pgSQL découpe le texte source de la fonction et produit un arbre d'instructions binaires internes la première fois que la fonction est appelée (au sein de chaque session). L'arbre d'instructions traduit complètement la structure de l'expression PL/pgSQL, mais les expressions SQL individuelles et les commandes SQL utilisées dans la fonction ne sont pas traduites immédiatement.
Chaque expression et commande SQL étant d'abord utilisée dans la fonction, l'interpréteur PL/-pgSQL crée un plan d'exécution élaboré (en utilisant les fonctions `SPI_prepare` et `SPI_saveplan` du gestionnaire SPI). Les visites suivantes à cette expression ou commande reutilisent le plan élaboré. Ainsi, une fonction avec du code conditionnel qui contient de nombreuses expressions pour lesquelles des plans d'exécution pourraient être nécessaires ne feront que préparer et sauvegarder ces plans, qui ne sont réellement utilisés que durant le temps que la connexion à la base de données vivra. Ceci peut réduire substantiellement le temps total nécessaire à l'analyse syntaxique, et générer des plans d'exécution pour les expressions d'une fonction PL/pgSQL. Un inconvénient est que les erreurs d'une expression ou commande particulière peuvent ne pas être détectées jusqu'à ce que cette partie de la fonction soit atteinte au cours de l'exécution.
Une fois que PL/pgSQL a créé un plan d'exécution pour une commande de fonction particulière, il réutilisera ce plan pour le temps que durera la connexion à la base de données. C'est généralement un gain de performances, mais cela peut causer quelques problèmes si vous modifiez dynamiquement votre schéma de base de données. Par exemple :

```
CREATE FUNCTION populate() RETURNS integer AS '
DECLARE
    -- declarations
BEGIN
    PERFORM my_function();
END;
' LANGUAGE plpgsql;
```

Si vous exécutez la fonction ci-dessus, l'OID de `my_function()` référencé dans le plan d'exécution est produit pour l'expression `PERFORM`. Par la suite, si vous détruisez et recréez `my_function()`, `populate()` ne sera plus en mesure de trouver `my_function()`. Vous auriez alors à recréer la fonction `populate()`, ou au moins à lancer une nouvelle connexion à la base de donnée pour faire en sorte de la compiler à nouveau. Un autre moyen d'éviter ce probème est d'utiliser `CREATE OR REPLACE FUNCTION` lors de la mise à jour de la définition de `my_function` (quand une fonction est "remplacée", son OID n'est pas changé).
Comme PL/pgSQL sauvegarde les plans d'exécution de cette façon, les commandes SQL qui apparaissent directement dans une fonction PL/pgSQL doivent se référer aux mêmes tables et colonnes pour chaque exception ; en fait, vous ne pouvez pas utiliser un paramètre tel que le nom d'une table ou d'une colonne dans une commande SQL. Pour contourner cette restriction, vous pouvez construire des commandes dynamiques en utilisant l'expression PL/pgSQL `EXECUTE` (au prix de la construction d'un nouveau plan d'exécution pour chaque exécution).
Remarque.

> L'expression PL/pgSQL `EXECUTE` n'a pas de rapport avec l'expression `EXECUTE` supportée par le serveur PostgreSQL. L'expression `EXECUTE` du serveur ne peut pas être utilisée au sein des fonctions PL/pgSQL (et n'est pas nécessaire).

Exception faites des conversions d'entrées/sorties et des fonctions de traitement pour des types définis par l'utlisateur, tout ce qui peut être défini dans les fonctions du langage C peut aussi être fait avec PL/pgSQL. Par exemple il est possible de créer des fonctions de traitement conditionnel complexes et par la suite les utiliser pour définir des opérateurs ou les utiliser dans des expressions d'index.

50.4.1 Avantages de l'utilisation de PL/pgSQL

SQL est le langage que PostgreSQL (et la plupart des autres bases de données relationnelles) utilise comme langage de requête. Il est portable et facile à apprendre. Mais chaque expression SQL doit être exécutée individuellement par le serveur de bases de données.
Cela signifie que votre application client doit envoyer chaque requête au serveur de bases de données, attendre que celui-ci la traite, recevoir les résultats, faire quelques traitements, et enfin envoyer d'autres requêtes au serveur. Tout ceci induit des communications interprocessus et peut aussi induire une surcharge du réseau si votre client est sur une machine différente du serveur de bases de données.
Grâce à PL/pgSQL vous pouvez grouper un bloc de traitement et une série de requêtes au sein du serveur de bases de données, et bénéficier ainsi de la puissance d'un langage procédural, tout en gagnant du temps puisque vous évitez toute la charge de la communication client/serveur. Ceci peut permettre un gain de performances considérable.
Ainsi, avec PL/pgSQL vous pouvez utiliser tous les types de données, opérateurs et fonctions du SQL.

50.4.2 Arguments supportés et types de données résultats

Les fonctions écrites en PL/pgSQL peuvent accepter comme argument n'importe quel type de données supporté par le serveur, et peuvent renvoyer un résultat de n'importe lequel de ces types. Elles peuvent aussi accepter ou renvoyer n'importe quel type composite (type ligne) spécifié par nom. Il est aussi possible de déclarer une fonction PL/pgSQL renvoyant un type `record`, signifiant que le résultat est un type ligne dont les colonnes sont déterminées par spécification dans la requête appelante.
Les fonctions PL/pgSQL peuvent aussi être déclarées comme acceptant et renvoyant les types « polymorphes », `anyelement` et `anyarray`. Le type de données réel géré par une fonction polymorphe peut varier d'appel en appel.
Les fonctions PL/pgSQL peuvent aussi être déclarées comme devant renvoyer un « `set` » ou une table de n'importe lequel des type de données dont elles peuvent renvoyer une instance unique. De telles fonctions génèrent leur sortie en exécutant `RETURN NEXT` pour chaque élément désiré de l'ensemble résultat.
Enfin, une fonction PL/pgSQL peut être déclarée comme renvoyant `void` si elle n'a pas de valeur de retour utile.

Chapter 51

Astuces pour développer en PL/pgSQL

51.1 Utilisez un éditeurs de texte !

Un bon moyen de développer en PL/pgSQL est d'utiliser l'éditeur de texte de votre choix pour créer vos fonctions, et d'utiliser `psql` dans une autre fenêtre pour charger et tester ces fonctions. Si vous procédez ainsi, une bonne idée est d'écrire la fonction en utilisant `CREATE OR REPLACE FUNCTION`. De cette façon vous pouvez recharger le fichier seulement pour mettre à jour la définition de la fonction. Par exemple :

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS '
          ....
end;
' LANGUAGE plpgsql;
```

Pendant que `psql` tourne, vous pouvez charger ou recharger de telles définitions de fonction avec :

```
\i filename.sql
```

et alors immédiatement soumettre des commandes SQL pour tester la fonction.

51.2 pgaccess

Un autre bon moyen de développer en PL/pgSQL est d'utiliser un outil d'accès à la base de données muni d'une interface graphique qui facilite le développement dans un langage procédural. Un exemple d'un tel outil est *PgAccess*, bien que d'autres existent. Ces outils fournissent souvent des fonctionnalités pratiques telles que la détection des guillemets ouverts et facilitent la re-création et le débuggage des fonctions.

51.3 Utilisation des guillemets simples (quotes)

Puisque le code d'une fonction PL/pgSQL est spécifié dans la commande `CREATE FUNCTION` comme une chaîne de caractères, chaque guillemet simple (*quote*) à l'intérieur de la fonction doit être précédé d'un autre guillemet simple (caractère d'échappement). Ceci peut conduire parfois à un code assez compliqué, surtout si vous écrivez une fonction générant d'autres fonctions. Ce tableau peut être utile comme résumé du nombre de quotes nécessaire dans diverses situations.

1 guillemet simple
: Pour commencer et terminer le corps de la fonction, par exemple :

```
CREATE FUNCTION foo() RETURNS integer AS '...'
    LANGUAGE plpgsql;
```

Partout au sein du corps de la fonction, les guillemets simples doivent aller par paires.

2 guillemets simples
Pour les chaînes de caractères à l'intérieur du corps de la fonction, par exemple :

```
a_output := ''Blah'';
SELECT * FROM users WHERE f_name=''foobar'';
```

4 guillemets simples
Quand vous avez besoin d'un guillemet simple dans une chaîne constante à l'interieur du corps de la fonction, par exemple :

```
a_output := a_output || '' AND name LIKE ''''foobar'''' AND xyz''
```

La valeur effectivement concaténée à `a_output` est : `AND name LIKE 'foobar' AND xyz`.

6 guillemets simples
Quand un simple guillemet dans une chaîne à l'intérieur du corps d'une fonction est adjacent à la fin de cette chaîne constante, par exemple :

```
a_output := a_output || '' AND name LIKE ''''foobar''''''
```

La valeur effectivement concaténée à `a_output` est alors : `AND name LIKE 'foobar'`.

10 guillemets simples
Lorsque vous voulez 2 guillemets simples dans une chaîne constante (qui compte pour 8 guillemets simples) et qu'elle est adjacente à la fin de cette chaîne constante (2 de plus). Vous n'aurez probablement besoin de ceci que si vous écrivez une fonction qui génère d'autres fonctions. Par exemple :

```
a_output := a_output || '' if v_'' ||
    referrer_keys.kind || '' like ''''''''''
    || referrer_keys.key_string || ''''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

La valeur de `a_output` sera alors :

```
if v_... like ''...'' then return ''...''; end if;
```

Une approche différente est d'échapper les guillemets du corps de la fonction avec un *backslash* plutôt qu'en les doublant. Avec cette méthode, vous vous verrez écrire des choses telles que `\'\'` au lieu de `''''`. Certains trouveront ceci plus lisible, d'autres non.

Chapter 52

Structure de 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;
```

Chaque déclaration et chaque expression au sein du bloc est terminée par un point-virgule.
Tous les mots clés et identifiants peuvent être écrits en majuscules et minuscules mélangées. Les identifiants sont implicitement convertis en minuscules à moins d'être entourés de guillemets doubles.
Il y a deux types de commentaires dans PL/pgSQL. Un double tiret (--) débute une ligne de commentaire qui s'étend jusqu'à la fin de la ligne. Un /* débute un bloc de commentaire qui s'étend jusqu'à la prochaîne occurence de */. Les blocs de commentaires ne peuvent pas être imbriqués, mais les commentaires de lignes (double tiret) peuvent être contenus dans un bloc de commentaire et un double tiret peut cacher les délimiteurs du bloc de commentaire /* et */.
Chaque expression de la section expression d'un bloc peut être un sous-bloc. Les sous-blocs peuvent être utilisés pour des groupements logiques ou pour localiser des variables locales à un petit groupe d'instructions.
Les variables déclarées dans la section déclaration précédant un bloc sont initialisées à leur valeur par défaut chaque fois qu'on entre dans un bloc et pas seulement une fois à chaque appel de fonction. Par exemple :

```
CREATE FUNCTION somefunc() RETURNS integer AS ''
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE ''Quantity here is %'', quantity;  -- Quantity here is 30
    quantity := 50;
    --
    -- Crée un sous-bloc
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE ''Quantity here is %'', quantity;  -- Quantity here is 80
    END;
```

```
    RAISE NOTICE ''Quantity here is %'', quantity;  -- Quantity here is 50

    RETURN quantity;
END;
' LANGUAGE plpgsql;
```

Il est important de ne pas confondre l'utilisation de `BEGIN/END` pour grouper les instructions dans PL/pgSQL avec les commandes de bases de données pour le contrôle des transactions. Les `BEGIN/END` de PL/pgSQL ne servent qu'au groupement ; ils ne débutent ni ne terminent une transaction. Les procédures fonctions et déclencheurs sont toujours exécutées à l'intérieur d'une transaction établie par une requête extérieure (ils ne peuvent pas être utilisés pour commencer ou effectuer un `commit` sur une transaction puisque PostgreSQL ne gère pas les transactions imbriquées).

Chapter 53

PL/pgSQL : déclarations

Toutes les variables utilisées dans un bloc doivent être déclarées dans la section déclaration du bloc. (La seule exception est que la variable de boucle d'une boucle `FOR` effectuant une itération sur des valeurs entières est automatiquement déclarée comme variable entière.
Les variables PL/pgSQL peuvent être de n'importe quel type de données tels que `integer`, `varchar`, et `char`.
Voici quelques exemples de déclarations de variables :

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

La syntaxe générale d'une déclaration de variable est :

```
name [ CONSTANT ] type [ NOT NULL ] [ { DEFAULT | := } expression ];
```

La clause `DEFAULT`, si indiquée, spécifie la valeur initiale assignée à la variable quand on entre dans le bloc. Si la clause `DEFAULT` n'est pas indiquée, la variable est initialisée à la valeur SQL `null`. L'option `CONSTANT` empêche l'assignation de la variable, de sorte que sa valeur reste constante pour la durée du bloc. Si `NOT NULL` est spécifié, l'assignement d'une valeur `null` aboutira à une erreur d'exécution. Les valeurs par défaut de toutes les variables déclarées `NOT NULL` doivent être spécifiées non `null`.
La valeur par défaut est évaluée à chaque entrée du bloc. Ainsi, par exemple, l'assignation de '`now`' à une variable de type timestamp donnera à la variable l'heure de l'appel de la fonction courante, et non l'heure au moment où la fonction a été précompilée.
Exemples :

```
quantity integer DEFAULT 32;
url varchar := ''http://mysite.com'';
user_id CONSTANT integer := 10;
```

53.1 Alias de paramètres de fonctions

```
nom ALIAS FOR $n;
```

Les paramètres passés aux fonctions sont nommés par les identifiants `$1`, `$2`, etc. Éventuellement des alias peuvent être déclarés pour les noms de paramètres de type `$n` afin d'améliorer la lisibilité. L'alias ou l'identifiant numérique peuvent être utilisés indifféremment pour se référer à la valeur du paramètre. Quelques exemples :

```
CREATE FUNCTION sales_tax(real) RETURNS real AS '
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
' LANGUAGE plpgsql;

CREATE FUNCTION instr(varchar, integer) RETURNS integer AS '
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- quelques traitements
END;
' LANGUAGE plpgsql;

CREATE FUNCTION concat_selected_fields(tablename) RETURNS text AS '
DECLARE
    in_t ALIAS FOR $1;
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
' LANGUAGE plpgsql;
```

Lorsque le type de retour d'une fonction PL/pgSQL est déclaré comme type polymorphe (`anyelement` ou `anyarray`), un paramètre spécial `$0` est créé. Son type de donnée est le type effectif de retour de la fonction, déduit d'après les types d'entrée. Ceci permet à la fonction d'accéder à son type de retour réel. `$0` est initialisé à `NULL` et peut être modifié par la fonction, de sorte qu'il peut être utilisé pour contenir la variable de retour si besoin est, bien que ca ne soit pas requis. On peut aussi donner à `$0` un alias. Par exemple, cette fonction fonctionne comme un opérateur `+` pour n'importe quel type de données.

```
CREATE FUNCTION add_three_values(anyelement, anyelement, anyelement)
RETURNS anyelement AS '
DECLARE
    result ALIAS FOR $0;
    first ALIAS FOR $1;
    second ALIAS FOR $2;
    third ALIAS FOR $3;
BEGIN
    result := first + second + third;
    RETURN result;
END;
' LANGUAGE plpgsql;
```

53.2 Copie de types

`variable%TYPE`

`%TYPE` fournit le type de données d'une variable ou d'une colonne de table. Vous pouvez l'utiliser pour déclarer des variables qui contiendront des valeurs de bases de données. Par exemple, disons que vous avez une colonne nommée `user_id` dans votre table `users`. Pour déclarer une variable du même type de données que `users.user_id` vous pouvez écrire :

```
user_id users.user_id%TYPE;
```

En utilisant %TYPE vous n'avez pas besoin de connaître le type de données de la structure à laquelle vous faites référence, et plus important, si le type de données de l'objet référencé change dans le futur (par exemple : vous changez le type de user_id de integer à real), vous pouvez ne pas avoir besoin de changer votre définition de fonction.
%TYPE est particulièrement utile dans le cas de fonctions polymorphes, puisque les types de données nécessaires aux variables internes peuvent changer d'un appel à l'autre. Des variables appropriées peuvent être créées en appliquant%TYPE aux arguments de la fonction ou à la la variable fictive de résultat.

53.3 Types ligne

```
name table_name%ROWTYPE;
name composite_type_name;
```

Une variable de type composite est appelée variable ligne (ou variable row-type). Une telle variable peut contenir une ligne entière de résultat de requête SELECT ou FOR, du moment que l'ensemble de colonnes de la requête correspond au type déclaré de la variable. Les champs individuels de la valeur row sont accessibles en utilisant la notation pointée, par exemple rowvar.field.
Une variable ligne peut être déclarée de façon à avoir le même type que les lignes d'une table ou vue existante, en utilisant la notation table_name%ROWTYPE ou elle peut être déclarée en donnant un nom de type composite. (Chaque table ayant un type de données associé du même nom, il importe peu dans PostgreSQL que vous écriviez %ROWTYPE ou pas. Cependant la forme utilisant %ROWTYPE est plus portable.)
Les paramètres d'une fonction peuvent être des types composites (lignes complètes de tables). En ce cas, l'identifiant correspondant $n sera une variable ligne, à partir de laquelle les champs peuvent être sélectionnés, par exemple $1.user_id.
Seules les colonnes définies par l'utilisateur d'une ligne de table sont accessibles dans une variable de type ligne, et non l'OID ou d'autres colonnes systèmes (parce que la ligne pourrait être issue d'une vue). Les champs du type ligne héritent des tailles des champs de la table ou de leur précision pour les types de données tels que char(n).
Voici un exemple d'utilisation des types composites :

```
CREATE FUNCTION use_two_tables(tablename) RETURNS text AS '
DECLARE
    in_t ALIAS FOR $1;
    use_t table2name%ROWTYPE;
BEGIN
    SELECT * INTO use_t FROM table2name WHERE ... ;
    RETURN in_t.f1 || use_t.f3 || in_t.f5 || use_t.f7;
END;
' LANGUAGE plpgsql;
```

53.4 Types record

```
nom RECORD;
```

Les variables record sont similaires aux variables de type ligne, mais n'ont pas de structure prédéfinie. Elles empruntent la structure effective de type ligne de la ligne à laquelle elles sont assignées durant une commande SELECT ou FOR. La sous-structure d'une variable record peut changer à chaque fois qu'on l'assigne. Une conséquence de cela est que jusqu'à ce qu'elle ait été assignée, elle n'a pas de sous-structure, et toutes les tentatives pour accéder à un de ses champs entraineront une erreur d'exécution.

Notez que `RECORD` n'est pas un vrai type de données mais seulement un paramètre fictif (`placeholder`). Il faut aussi realiser que quand une fonction PL/pgSQL est déclarée renvoyer un type `record`, il ne s'agit pas tout à fait du même concept qu'une variable `record`, même si une telle fonction peut aussi utiliser une variable `record` pour contenir son résultat. Dans les deux cas la structure réelle de la ligne n'est pas connue quand la fonction est écrite, mais dans le cas d'une fonction renvoyant un type `record` la structure réelle est déterminée quand la requête appelante est analysée, alors qu'une variable record peut changer sa structure de ligne à la volée.

53.5 RENAME

```
RENAME ancien nom TO nouveau nom;
```

En utilisant la déclaration `RENAME`, vous pouvez changer le nom d'une variable, d'un `record` ou d'un `row` (ligne). C'est particulièrement utile si `NEW` ou `OLD` doivent être référencés par un autre nom dans une procédure déclencheur. Voir aussi `ALIAS`.
Exemples :

```
RENAME id TO user_id;
RENAME this_var TO that_var;
```

Remarque.

> `RENAME` semble ne pas fonctionner dans PostgreSQL 7.3. Cette correction est de faible priorité, `ALIAS` couvrant la plupart des utilisations partiques de `RENAME`.

Chapter 54

PL/pgSQL : expressions

Toutes les expressions utilisées dans les instructions PL/pgSQL sont traitées par l'exécuteur SQL classique du serveur. Les expressions qui apparaissent contenir des constantes peuvent en fait nécessiter une évaluation pendant l'exécution (par exemple, 'now' pour le type `timestamp`) ainsi il est impossible pour l'analyseur syntaxique PL/pgSQL d'identifier les valeurs réelles des constantes autres que le mot clé `NULL`. Toutes les expressions sont evaluées de façon interne en exécutant une requête `SELECT expression` en utilisant le gestionnaire `SPI`. Pour l'évaluation, les ocurrences des identifiants de variables PL/pgSQL sont remplacées par des paramètres, et les valeurs réelles des variables sont passées à l'exécuteur dans le tableau des paramètres. Ceci permet au plan de requêtes pour le `SELECT` de n'être élaboré qu'une fois et réutilisé pour les évaluations postérieures. L'évaluation faite par l'analyseur syntaxique principal de PostgreSQL a quelques effets de bord sur l'interprétation des valeurs constantes. Plus précisément, il y a une différence entre ce que font ces deux fonctions :

```
CREATE FUNCTION logfunc1(text) RETURNS timestamp AS '
    DECLARE
        logtxt ALIAS FOR $1;
    BEGIN
        INSERT INTO logtable VALUES (logtxt, ''now'');
        RETURN ''now'';
    END;
' LANGUAGE plpgsql;
```

et

```
CREATE FUNCTION logfunc2(text) RETURNS timestamp AS '
    DECLARE
        logtxt ALIAS FOR $1;
        curtime timestamp;
    BEGIN
        curtime := ''now'';
        INSERT INTO logtable VALUES (logtxt, curtime);
        RETURN curtime;
    END;
' LANGUAGE plpgsql;
```

Dans le cas de `logfunc1`, l'analyseur syntaxique principal de PostgreSQL sait, quand il élabore le plan pour l'`INSERT`, que la chaîne 'now' doit être interprétée comme un `timestamp` parce que la colonne cible de logtable est de ce type. Ainsi, il en fera une constante à ce moment et cette valeur constante sera alors utilisée dans toutes les invocations de logfunc1 pendant le temps que durera la session. Il va sans dire que ce n'est pas ce que le programmeur voulait.

Dans le cas de `logfunc2`, l'analyseur principal de PostgreSQL ne sait pas quel type '`now`' doit devenir, et par conséquent, il renvoie une valeur de type `text` contenant la chaîne `now`. Durant l'assignation consécutive de la variable locale `curtime`, l'interpréteur PL/pgSQL transtype cette chaîne en type `timestamp` en appelant les fonctions `text_out` et `timestamp_in` pour la conversion. Ainsi, l'horodateur est mis à jour à chaque exécution comme l'attend le programmeur.
La nature modifiable des variables record présente un problème lors de cette connexion. Quand les champs d'une variable record sont utilisés dans les expressions ou les instructions, les types de données des champs ne doivent pas changer entre les appels de deux expressions identiques, puisque l'expression sera planifiée en utilisant le type de données présent quand l'expression est atteinte pour la première fois. Gardez ceci à l'esprit quand vous écrivez des procédures déclencheurs qui gèrent des évènements pour plus d'une table. (`EXECUTE` peut être utilisé pour contourner le problème si nécessaire).

Chapter 55

PL/pgSQL : instructions de base

Dans cette section ainsi que les suivantes, nous décrirons tous les types d'instructions explicitement compris par PL/pgSQL. Tout ce qui n'est pas reconnu comme l'un de ces types d'instruction est présumé être une commande SQL et est envoyé au moteur principal de bases de données pour être exécutée (après substitution de chaque variable PL/pgSQL utilisée dans l'instruction). Ainsi, par exemple, les commandes SQL `INSERT`, `UPDATE`, et `DELETE` peuvent être considérées comme des instructions de PL/pgSQL, mais ne sont pas spécifiquement listées ici.

55.1 Assignation

L'assignation d'une valeur à une variable ou à un champ `row/record` est écrite ainsi :

```
identifiant := expression;
```

Comme expliqué plus haut, l'expression dans un telle instruction est évaluée au moyen de la commande SQL `SELECT` envoyée au moteur principal de bases de données. L'expression ne doit manier qu'une seule valeur.
Si le type de données du résultat de l'expression ne correspond pas au type de donnée de la variable, ou que la variable a une taille ou une précision (comme `char(20)`), la valeur résulat sera implicitement convertie par l'interpréteur PL/pgSQL en utilisant la fonction d'écriture (`output-function`) du type du résultat, et la fonction d'entrée (`input-function`) du type de la variable. Notez que cela pourrait potentiellement conduire des erreurs d'exécution générées par la fonction d'entrée, si la forme de la chaîne de la valeur résultat n'est pas acceptable par la fonction d'entrée.
Exemples :

```
user_id := 20;
tax := subtotal * 0.06;
```

55.2 SELECT INTO

Le résultat d'une commande `SELECT` manipulant plusieurs colonnes (mais une seule ligne) peut être assignée à une variable de type `record` ou ligne, ou une liste de valeurs scalaires. Ceci est fait via :

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

où cible peut être une variable `record`, une variable ligne, ou une liste, séparées de virgules, de simples variables de champs `record`/ligne.
Notez que cela est assez différent de l'interprétation normale par PostgreSQL de `SELECT INTO`, où la cible de `INTO` est une table nouvellement créée. Si vous voulez créer une table à partir du résultat d'un `SELECT` d'une fonction PL/pgSQL, utilisez la syntaxe `CREATE TABLE ... AS SELECT`.

Si une ligne ou une liste de variable est utilisée comme cible, les valeurs sélectionnées doivent correspondre exactement à la structure de la cible, ou une erreur d'exécution se produira. Quand une variable `record` est la cible, elle se configure seule automatiquement au type ligne formé par les colonnes résultant de la requête.
À l'exception de la clause `INTO`, l'instruction `SELECT` est identique à la commande SQL `SELECT` normale et peut en utiliser toute la puissance.
Si la requête ne renvoit aucune ligne, des valeurs `null` sont assignées au(x) cibles(s). Si la requête renvoit plusieurs lignes, la première ligne est assignées au(x) cible(s) et le reste est rejeté. (Notez que « la première ligne » n'est pas correctement définie à moins d'utiliser `ORDER BY`.)
Actuellement, la clause `INTO` peut apparaître presque n'importe où dans l'instruction `SELECT`, mais il est recommandé de la placer immédiatement apres le mot clé `SELECT` comme décrit plus haut. Les version futures de PL/pgSQL pourront être moins laxistes sur le placement de la clause `INTO`.
Vous pouvez utiliser `FOUND` immédiatement après une instruction `SELECT INTO` pour déterminer si l'assignation a réussi (c'est à dire qu'au moins une ligne a été renvoyée par la requête). Par exemple :

```
SELECT INTO myrec * FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION ''employee % not found'', myname;
END IF;
```

Pour tester si un résultat `record`/ligne est `null`, vous pouvez utiliser la conditionelle `IS NULL`. Il n'y a cependant aucun moyen de dire si une ou plusieurs lignes additionnelles ont été rejetées. Voici un exemple qui traite le cas où aucune ligne n'a été renvoyée.

```
DECLARE
    users_rec RECORD;
    full_name varchar;
BEGIN
    SELECT INTO users_rec * FROM users WHERE user_id=3;

    IF users_rec.homepage IS NULL THEN
        -- l'utilisateur n'a entré aucune page, renvoyer "http://"
        RETURN ''http://'';
    END IF;
END;
```

55.3 Exécuter une expression ou requête sans résultat

Quelquefois l'on souhaite évaluer une expression ou une requête mais rejeter le résultat (généralement parce que l'on appelle une fonction qui a des effets de bords utiles mais pas de résultat utile). Pour ce faire dans PL/pgSQL, utilisez l'instruction `PERFORM` :

```
PERFORM requête;
```

Ceci exécute `requête`, qui doit être une instruction `SELECT`, et rejette le résultat. Les variables PL/pgSQL sont normalement remplacées dans la requête. Par ailleurs, la variable spéciale `FOUND` est positionnée à `true` si la requête produit au moins une ligne ou `false` si elle n'en produit aucune.
Remarque.

> On pourrait s'attendre à ce qu'un `SELECT` sans clause `INTO` aboutisse à ce résultat, mais en réalité la seule façon acceptée de faire cela est `PERFORM`.

Un exemple :

```
PERFORM create_mv(''cs_session_page_requests_mv'', my_query);
```

55.4 Exécuter des commandes dynamiques

Souvent vous voudrez générer des commandes dynamiques dans vos fonctions PL/pgSQL, c'est à dire, des commandes qui impliquent différentes tables ou différents types de données à chaque fois qu'elles sont exécutées. Les tentatives normales de PL/pgSQL pour garder en cache les planification des commandes ne marcheront pas dans de tels scénarios. Pour gérer ce type de problème, l'instruction `EXECUTE` est fournie :

```
EXECUTE chaîne-commande;
```

où chaîne-commande est une expression manipulant une chaîne (de type `text`) contenant la commande à être exécutée. Cette chaîne est littéralement donnée à manger au moteur SQL.
Notez en particulier qu'aucune substitution de variable PL/pgSQL n'est faite sur la chaîne-commande. Les valeurs des variables doivent être insérées dans la chaîne de commande lors de sa construction.
Lorsque vous travaillez avec des commandes dynamiques vous avez à faire face à l'échappement des guillemets simples dans PL/pgSQL. Référez vous au survol dans 51.3, qui peut vous épargner quelque effort.
À la différence de toutes les autres commandes dans PL/pgSQL, une commande lancée par une instruction `EXECUTE` n'est pas préparée ni sauvée une seule fois pendant la durée de la session. À la place, la commande est préparée à chaque fois que l'instruction est lancée. La chaîne-commande peut être dynamiquement créée à l'intérieur de la fonction pour agir sur des variables tables ou colonnes.
Les résultats des commandes `SELECT` sont rejetés par `EXECUTE`, et `SELECT INTO` n'est pas actuellement géré à l'intérieur d'une instruction `EXECUTE`. Ainsi, la seule façon d'extraire le résultat d'un `SELECT` créé dynamiquement est d'utiliser la forme `FOR-IN-EXECUTE` décrite plus loin.
Exemple :

```
EXECUTE ''UPDATE tbl SET ''
        || quote_ident(colname)
        || '' = ''
        || quote_literal(newvalue)
        || '' WHERE ...'';
```

Cet exemple montre l'usage des fonctions `quote_ident(text)` et `quote_literal(text)`. Les variables contenant les identifiants de colonne et de table devraient être passées à la fonction `quote_ident`. Les variables contenant les valeurs qui devraient être des chaînes de caractères dans la commande construite devraient être passées à `quote_literal`. Ces deux fonctions effectuent les traitements appropriés pour renvoyer le texte entré enfermé entre doubles ou simples guillemets respectivement, chaque caractère spécial correctement échappé.
Voici un exemple beaucoup plus important de commande dynamique et d'utilisation d'`EXECUTE` :

```
CREATE FUNCTION cs_update_referrer_type_proc() RETURNS integer AS '
DECLARE
    referrer_keys RECORD;  --déclare un record générique à utiliser dans un FOR
    a_output varchar(4000);
BEGIN
   a_output := ''CREATE FUNCTION cs_find_referrer_type(varchar,varchar,varchar)
                 RETURNS varchar AS ''''
                    DECLARE
                        v_host ALIAS FOR $1;
                        v_domain ALIAS FOR $2;
                        v_url ALIAS FOR $3;
                    BEGIN '';

   -- Notez comment nous scannons les résultats d'une requête dans une boucle
```

```
   -- FOR en utilisant la construction FOR <record>.

   FOR referrer_keys IN SELECT * FROM cs_referrer_keys ORDER BY try_order LOOP
       a_output := a_output || '' IF v_'' || referrer_keys.kind
               || '' LIKE ''''''''''' || referrer_keys.key_string
               || ''''''''''' THEN RETURN ''''''
               || referrer_keys.referrer_type || ''''''; END IF;'';
   END LOOP;

   a_output := a_output || '' RETURN NULL; END; '''' LANGUAGE plpgsql;'';

   EXECUTE a_output;
END;
' LANGUAGE plpgsql;
```

55.5 Obtention du statut du résultat

Il y a plusieurs moyens de déterminer l'effet d'une commande. La première méthode est d'utiliser `GET DIAGNOSTICS`, qui a la forme suivante :

```
GET DIAGNOSTICS variable = item [ , ... ] ;
```

Cette commande permet la récupération des indicateurs de l'état du système. Chaque item est un mot clé identifiant une valeur d'état devant être assignée à la variable indiquée (qui devrait être du bon type de données pour pouvoir la recevoir). Les items d'état actuellement disponibles sont `ROW_COUNT`, le nombre de lignes traitées par la dernière commande SQL envoyée au moteur SQL, et `RESULT_OID`, l'OID de la dernière ligne insérée par la commande SQL la plus récente. Notez que `RESULT_OID` n'est utile qu'après une commande `INSERT`.
Exemple :

```
GET DIAGNOSTICS var_integer = ROW_COUNT;
```

La seconde méthode pour déterminer les effets d'une commande est la variable spéciale nommée `FOUND` de type `boolean`. `FOUND` commence par être `false` dans chaque fonction PL/pgSQL. Elle est positionnée par chacune des types d'instructions suivants.
Une instruction `SELECT INTO` positionne `FOUND` à `true` si elle renvoie une ligne, false si aucune ligne n'est renvoyée.
Une instruction `PERFORM` positionne `FOUND` à `true` si elle produit (rejette) une ligne, `false` si aucune ligne n'est produite.
Les instructions `UPDATE`, `INSERT`, et `DELETE` positionnent `FOUND` à `true` si au moins une ligne est affectée, `false` si aucune ligne n'est affectée.
Une instruction `FETCH` positionne `FOUND` à `true` si elle renvoie une ligne, `false` si aucune ligne n'est renvoyée.
La commande `FOR` positionne `FOUND` à `true` si elle effectue une itération une ou plusieurs fois, sinon elle renvoie `false`. Ceci s'applique aux trois variantes de l'instruction `FOR` (boucles `FOR integer`, `FOR record-set`, et `FOR record-set` dynamique). `FOUND` n'est positionné que quand la boucle `FOR` s'achève : dans l'exécution de la chaîne, `FOUND` n'est pas modifiée par l'instruction `FOR`, bien qu'il puisse être modifié par l'exécution d'autres instructions situées dans le corps de la boucle.
`FOUND` est une variable locale ; chaque changement qui y est fait n'affecte que la fonction PL/pgSQL courante.

Chapter 56

PL/pgSQL : 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.

56.1 Retour d'une fonction

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

56.1.1 `RETURN`

```
RETURN expression;
```

`RETURN` accompagné d'une expression termine la fonction et renvoit le valeur d'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 transtypé vers le type de retour de la fonction, comme décrit pour les assignations. Pour renvoyer une valeur composite (ligne), vous devez écrire une variable `record` ou ligne comme expression.
La valeur de retour d'une fonction ne peut pas être laissée indéfinie. Si le contrôle atteint la fin du bloc de premier niveau sans avoir rencontré d'instruction `RETURN` une erreur d'éxécution sera lancée. Notez que si vous avez déclaré la fonction comme renvoyant `void`, une instruction `RETURN` doit être quand même spécifiée ; l'expression suivant la commande `RETURN` est cependant optionnelle et sera ignorée dans tous les cas.

56.1.2 `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 items 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 ; dans ce dernier cas, une « table » entière de résultats sera renvoyée.
Les fonctions qui utilisent `RETURN NEXT` devraient être appelées d'après le modèle suivant :

```
SELECT * FROM some_func();
```

En fait la fonction est utilisée comme table source dans une clause `FROM`.
`RETURN NEXT` n'effectue pas vraiment de renvoi ; il sauvegarde simplement les valeurs des expressions (ou variables de type `record` ou ligne, d'après le type approprié au type de données renvoyé). L'exécution continue alors avec la prochaîne instruction dans la fonction PL/pgSQL. Lorsque des commandes `RETURN NEXT` successives sont renvoyées, l'ensemble des résultats est élaboré. Un `RETURN` final, qui ne devrait pas avoir d'argument, provoque la sortie du contrôle de la fonction.
Remarque.

> 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'a 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 controlé par la variable de configuration `sort_mem`. Les administrateurs ayant une mémoire suffisante pour enregistrer des ensembles de résultats plus importants en mémoire devraient envisager l'augmentation de ce paramètre.

56.2 Contrôles conditionnels

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

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

56.2.1 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 `true`. Autrement, ils seront négligés.
Exemple :

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

56.2.2 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 ensemble d'instructions alternatif à exécuter si la condition est evaluée à false.
Exemples :

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

IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN ''t'';
ELSE
    RETURN ''f'';
END IF;
```

56.2.3 IF-THEN-ELSE IF

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

```
IF demo_row.sex = ''m'' THEN
    pretty_sex := ''man'';
ELSE
    IF demo_row.sex = ''f'' THEN
        pretty_sex := ''woman'';
    END IF;
END IF;
```

Lorsque vous utilisez cette forme, vous imbriquez une instruction IF dans la partie ELSE d'une instruction IF extérieure. Ansi 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.

56.2.4 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 number = 0 THEN
    result := ''zero'';
ELSIF number > 0 THEN
```

```
    result := ''positif'';
ELSIF number < 0 THEN
    result := ''negatif'';
ELSE
    -- hmm, la seule possibilité est que le nombre soit null
    result := ''NULL'';
END IF;
```

56.3 Boucles Simples

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

56.3.1 LOOP

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

`LOOP` définit une boucle inconditionnelle répétée indéfiniement jusqu'a ce qu'elle soit terminée par une instruction `EXIT` ou `RETURN`. Le label optionnel peut être utilisé par les instructions `EXIT` dans le cas de boucles imbriquées pour définir quel niveau d'imbrication doit s'achever.

56.3.2 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 ensuite. Si un label est donné, ce doit être le label de la boucle ou 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 présent, la sortie de boucle ne s'effectue que si les conditions spécifiées sont `true`, autrement le contrôle passe à l'instruction suivant le `EXIT`.
Exemples :

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

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

BEGIN
    -- quelques traitements
    IF stocks > 100000 THEN
        EXIT;  -- invalide; on ne peut pas utiliser EXIT hors d'un LOOP
    END IF;
END;
```

56.3.3 WHILE

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

L'instruction `WHILE` repète une séquence d'instructions aussi longtemps que l'expression conditionnelle est évaluée à vrai. La condition est vérifée juste avant chaque entrée dans le corps de la boucle.
Par exemple :

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- quelques traitements ici
END LOOP;

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

56.3.4 FOR (variante avec entier)

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

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. Les deux expressions donnant les limites inférieures et supérieures de la plage sont evaluées une fois en entrant dans la boucle. Le pas de l'itération est normalement de `1`, mais est `-1` quand `REVERSE` is spécifié.
Quelques exemples de boucles `FOR` avec entiers :

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

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

56.4 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 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 et exécuté pour chaque ligne. Voici un exemple :

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS '
DECLARE
    mviews RECORD;
BEGIN
    PERFORM cs_log(''Refreshing materialized views...'');

    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP

        -- A présent "mviews" contient un enregistrement de
        -- cs_materialized_views

        PERFORM cs_log(''Refreshing materialized view ''
                       || quote_ident(mviews.mv_name) || ''...'');
        EXECUTE ''TRUNCATE TABLE  '' || quote_ident(mviews.mv_name);
        EXECUTE ''INSERT INTO '' || quote_ident(mviews.mv_name) || '' ''
                                 || mviews.mv_query;
    END LOOP;

    PERFORM cs_log(''Done refreshing materialized views.'');
    RETURN 1;
END;
' LANGUAGE plpgsql;
```

Si la boucle est terminée par une instruction `EXIT`, la dernière valeur ligne assignée est toujours accessible apres la boucle.
L'instruction `FOR-IN-EXECUTE` est un moyen d'itérer sur des enregistrements :

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

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

> L'analyseur PL/pgSQL fait actuellement la distinction entre les deux sortes de boucles `FOR` (avec entier ou résultat de requête) en verifiant si la variable cible mentionnée juste apres le `FOR` a été déclarée comme une variable record ou ligne. Si non, elle est présumée être une boucle `FOR` avec entier. Ceci peut produire des messages d'erreurs asser peu intuitifs quand le vrai problème est, disons, que l'on a mal orthographié le nom de la variable suivant le `FOR`.

Chapter 57

PL/pgSQL : exercices

Exercice 91

1. *Écrire une fonction* `moySalaire` *sans paramètre qui renvoie le salaire moyen des employés.*
2. *Utiliser ensuite cette fonction pour afficher les noms et le salaire des employés qui gagnent plus que le salaire moyen,*
3. *puis ceux dont le salaire est égal au salaire moyen à 10% près (c'est-à-dire ceux dont le salaire est compris entre 90% et 110% du salaire moyen).*

Exercice 92 *Écrire une fonction* `departement` *qui admette un numéro d'employé en paramètre et qui renvoie comme résultat le nom du département de l'employé.*

Exercice 93 *Écrire une fonction* `collegues` *qui admet un numéro d'employé en paramètre et qui renvoie comme résultat le nom et le prénom des ses collègues du même département, l'employé lui-même ne devant pas faire partie de la liste des collègues.*

Exercice 94 *Écrire une fonction* `numlignes` *qui renvoie le nombre de lignes de la table dont le nom est passé en paramètre.*

Exercice 95 *Écrire une fonction* `superieurs` *qui affiche les noms et prénoms DES supérieurs de l'employé dont le numéro est passé en paramètre.* **Remarque.**

> *Pour commencer, écrivez une fonction qui affiche les nom et prénom DU supérieur. Puis testez là.*

Exercice 96 *Détruire toutes les fonctions que vous avez créées pour cette fiche.*

Chapter 58

Correction des exercices en PL/pgSQL

Solution de l'exercice 91

1. La fonction :

   ```
   CREATE OR REPLACE FUNCTION moySalaire() RETURNS NUMERIC AS '
   DECLARE
       moyenne numeric;
   BEGIN
       SELECT INTO moyenne AVG(salaire) FROM emp;
       RETURN moyenne ;
   END;
   ' LANGUAGE plpgsql;
   ```

   Un exemple d'utilisation « en direct » :

   ```
   SELECT moySalaire();
   ```

2. Les noms et le salaire des employés qui gagnent plus que le salaire moyen :

   ```
   SELECT nom,salaire FROM emp
    WHERE salaire>moySalaire();
   ```

3. Ceux dont le salaire est égal au salaire moyen à 10% près :

   ```
   SELECT nom,salaire FROM emp
    WHERE ABS(salaire-moySalaire())<0.10*moySalaire();
   ```

Solution de l'exercice 92

```
CREATE OR REPLACE FUNCTION departement(integer) RETURNS VARCHAR AS '
DECLARE
    dpt varchar;
BEGIN
    SELECT INTO dpt d.nom FROM emp e,dept d
           WHERE e.nodept=d.nodept AND noemp=$1;
    RETURN dpt ;
END;
' LANGUAGE plpgsql;
```

Solution de l'exercice 93

```
CREATE OR REPLACE FUNCTION collegues(integer) RETURNS SETOF RECORD  AS '
DECLARE
   rec RECORD;
BEGIN
   FOR rec IN SELECT c.prenom ,c.nom
               FROM emp e, emp c
              WHERE e.nodept=c.nodept AND e.noemp=$1
     LOOP
       RETURN NEXT rec ;
     END LOOP ;
   RETURN ;
END;
' LANGUAGE plpgsql;
```

Exemple d'utilisation :

```
SELECT * FROM collegues(7) AS (prenom VARCHAR,nom VARCHAR);
```

Solution de l'exercice 94

```
CREATE OR REPLACE FUNCTION numlignes(TEXT) RETURNS INTEGER AS '
DECLARE
   table_name ALIAS FOR $1;
   rec RECORD;
BEGIN
   FOR rec IN EXECUTE ''SELECT COUNT(*) FROM '' || table_name LOOP
       RETURN rec.count;
   END LOOP;
   RETURN 0;
END;
' LANGUAGE plpgsql;
```

Solution de l'exercice 95

- Une fonction qui affiche LE supérieur d'un employé pourrait être :

  ```
  CREATE OR REPLACE FUNCTION superieurs(integer) RETURNS SETOF RECORD AS '
  DECLARE
     rec RECORD;
     sup INTEGER;
  BEGIN
     SELECT INTO sup nosupr FROM emp WHERE noemp=$1;
     FOR rec IN SELECT noemp,nom,prenom,nosupr FROM emp WHERE noemp=sup
       LOOP
         RETURN NEXT rec ;
       END LOOP;
     RETURN ;
  END;
  ' LANGUAGE plpgsql;
  ```

  (Les colonnes `noemp` et `nosupr` ne sont pas vraiment nécessaires, mais elle nous permettront dans la version finale de contrôler ce que nous obiendrons.)

- On la teste ainsi :

```
 SELECT * FROM superieurs(12)
      AS (noemp NUMERIC(7,0),nom VARCHAR,prenom VARCHAR
          ,nosupr NUMERIC(7,0));
```

Ce qui nous aide à arriver à ça :

```
CREATE OR REPLACE FUNCTION superieurs(integer) RETURNS SETOF RECORD AS '
DECLARE
   rec RECORD;
   sup INTEGER;
BEGIN
   SELECT INTO sup nosupr FROM emp WHERE noemp=$1;
   IF sup IS NULL THEN
      RETURN NULL;
   END IF;
   FOR rec IN SELECT noemp,nom,prenom,nosupr FROM emp WHERE noemp=sup
     LOOP
       RETURN NEXT rec ;
     END LOOP;
   FOR rec IN SELECT * FROM
               superieurs(sup)
                    AS (noemp NUMERIC(7,0),
                        nom VARCHAR,
                        prenom VARCHAR,
                        nosupr NUMERIC(7,0))
     LOOP
       RETURN NEXT rec ;
     END LOOP;
   RETURN ;
END;
' LANGUAGE plpgsql;
```

Solution de l'exercice 96

```
DROP FUNCTION moySalaire() ;
DROP FUNCTION departement(integer) ;
DROP FUNCTION collegues(integer) ;
DROP FUNCTION numlignes(text) ;
DROP FUNCTION superieurs(integer) ;
```

Chapter 59

PL/pgSQL : curseurs

Plutôt que d'exécuter la totalité d'une requête à la fois, il est possible de créer un curseur qui encapsule la requête, puis en lit le résultat quelques lignes à la fois. Une des raisons pour faire de la sorte est d'éviter les surcharges de mémoire quand le résultat contient un grand nombre de lignes. (Cependant, les utilisateurs PL/pgSQL n'ont généralement pas besoin de se préoccupper de cela, puisque les boucles `FOR` utilisent automatiquement un curseur en interne pour éviter les problèmes de mémoire). Un usage plus intéressant est de renvoyer une référence à un curseur qu'elle a créé, permettant à l'appelant de lire les lignes. Ceci fournit un moyen efficace de renvoyer de grands ensembles de lignes à partir des fonctions.

59.1 Déclaration de variables curseur

Tous les accès aux curseurs dans PL/pgSQL se font par les variables curseur, qui sont toujours du type de données spécial `refcursor`. Un des moyens de créer une variable curseur est de simplement la déclarer comme une variable de type `refcursor`. Un autre moyen est d'utiliser la syntaxe de déclaration de curseur qui est en général :

```
nom CURSOR [ ( arguments ) ] FOR requête ;
```

(`FOR` peut être remplacé par `IS` pour la compatibilité avec *Oracle*.) `arguments`, si spécifié, est une liste de paires de nom type-de-donnée qui définit les noms devant être remplacés par les valeurs des paramètres dans la requête donnée. La valeur effective à subsituer pour ces noms sera spécifiée plus tard, lors de l'ouverture du curseur.
Quelques exemples :

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 = key;
```

Ces variables sont toutes trois du type de données `refcursor`, mais la première peut être utilisées avec n'importe quelle requête, alors que la seconde a une requête completement spécifiée qui lui est déjà liée, et la dernière est liée à une requête paramétrée. (`key` sera remplacée par un paramètre de valeur entière lors de l'ouverture du curseur.) La variable `curs1` est dite non liée puiqu'elle n'est pas liée a une requête particulière.

59.2 Ouverture de curseurs

Avant qu'un curseur puisse être utilsé pour rapatrier des lignes, il doit être ouvert. (C'est l'action equivalente de la commande SQL `DECLARE CURSOR`.) PL/pgSQL a trois formes pour l'instruction

OPEN, dont deux utilisent des variables curseur non liées et les autres utilisent une variable curseur liée.

59.2.1 OPEN FOR SELECT

```
OPEN curseur-non-lié FOR SELECT ...;
```

La variable curseur est ouverte et reçoit la requête spécifiée à exécuter. Le curseur ne peut pas être déjà ouvert, et il doit avoir été déclaré comme curseur non lié. (c'est à dire comme une simple variable `refcursor`). La requête `SELECT` est traitée de la même facon que les autres instructions `SELECT` dans PL/pgSQL : les noms de variables PL/pgSQL sont remplacés, et le plan de requête est mis en cache pour une possible réutilisation.
Exemple :

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

59.2.2 OPEN FOR EXECUTE

```
OPEN curseur-non-lié FOR EXECUTE chaîne-requête;
```

La variable curseur est ouverte et reçoit la requête spécifiée à exécuter. Le curseur ne peut pas être déjà ouvert, et il doit avoir été déclaré comme curseur non-lié (c'est a dire comme une simple variable `refcursor`. La requête est spécifiée comme une expression chaîne de la même facon que dans une commande `EXECUTE`. Comme d'habitude, ceci donne assez de flexibilité pour que la requête puisse changer d'une exécution à l'autre.
Exemple :

```
OPEN curs1 FOR EXECUTE ''SELECT * FROM '' || quote_ident($1);
```

59.2.3 Ouverture d'un curseur lié

```
OPEN curseur-lié [ ( arguments ) ];
```

Cette forme `d'OPEN` est utilisée pour ouvrir une variable curseur à laquelle la requête est liée au moment de la déclaration. Le curseur ne peut pas être déjà ouvert. Une liste des expressions arguments doit apparaître si et seulement si le curseur a été déclaré comme acceptant des arguments. Ces valeurs seront remplacées dans la requête. Le plan de requête pour un curseur lié est toujours considéré comme pouvant être mis en cache ; il n'y a pas d'équivalent de la commande `EXECUTE` dans ce cas.
Exemples :

```
OPEN curs2;
OPEN curs3(42);
```

59.3 Utilisation des curseurs

Une fois qu'un curseur a été ouvert, il peut être manipulé grâce aux instructions décrites ci-dessous. Ces manipulations n'ont pas besoin de se dérouler dans la même fonction que celle qui a ouvert le curseur. Vous pouvez renvoyer une valeur refcursor à partir d'une fonction et laisser l'appelant opérer sur le curseur. (D'un point de vue interne, une valeur refcursor est simplement la chaîne de caractères du nom d'un portail contenant la requête active pour le curseur. Ce nom peut être passé à d'autres, assigné à d'autres variables refcursor et ainsi de suite, sans déranger le portail.) Tous les portails sont implicitement fermés à la fin de la transaction. C'est pourquoi une valeur `refcursor` est utilisable pour référencer un cuseur ouvert seulement jusqu'à la fin de la transaction.

59.3.1 FETCH

```
FETCH curseur INTO target;
```

FETCH rapatrie le rang suivant depuis le curseur dans une cible, qui peut être une variable ligne, une variable record, ou une liste de simples variables séparées d'une virgule, exactement comme SELECT INTO. Comme pour SELECT INTO, la variable spéciale FOUND peut être verifiée pour voir si une ligne a été obtenue ou pas.
Exemple :

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
```

59.3.2 CLOSE

```
CLOSE curseur;
```

CLOSE ferme le portail sous-tendant un curseur ouvert. Ceci peut être utilisé pour libérer des ressources avant la fin de la transaction, ou de libérer la variable cuseur pour pouvoir la réouvrir.
Exemple :

```
CLOSE curs1;
```

59.3.3 Le renvoi de curseurs

Des PL/pgSQL peuvent renvoyer un curseur à l'appelant. Ceci est utilisé pour renvoyer plusieurs lignes ou colonnes d'une fonction. Pour ce faire, la fonction ouvre le curseur et renvoie le nom du curseur à l'appelant. L'appelant peut alors rapatrier des lignes du curseur. Le curseur peut être fermé par l'appelant, ou il peut être fermé automatiquement quand la transaction se termine.
Le nom du curseur renvoyé par la fonction peut être spécifié par l'appelant ou automatiquement généré. Les exemples suivants montrent comment un nom de curseur peut être fourni par l'appelant :

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

L'exemple suivant utilise la génération automatique du nom du curseur :

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;
```

```
BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

Chapter 60

PL/pgSQL : erreurs et messages

Utilisez l'instruction `RAISE` pour rapporter des messages et lever des erreurs.

```
RAISE niveau 'format' [, variable [, ...]];
```

Les niveaux possibles sont `DEBUG` (écrit le message sur le log du serveur), `LOG` (écrit le message sur le log du serveur avec une priorité plus élevée), `INFO`, `NOTICE` et `WARNING` (écrit le message sur le log du serveur et l'envoie au client avec des priorités respectivement supérieures), et `EXCEPTION` (lève une erreur et interrompt la transaction courante). Que les messages d'une priorité particulière soient rapportés au client, écrits dans le log du serveur, ou les deux est controlé par les variables de configuration `log_min_messages` et `client_min_messages`.
Au sein de la chaîne de formatage, `%` est remplacé par la représentation de la chaîne argument optionnelle suivante. Écrivez `%%` pour signifier un caractère `%`. Notez que les arguments optionnels doivent actuellement être de simples variables, non des expressions et que le format doit être une simple chaîne de caractères.
Dans cet exemple, la valeur de `v_job_id` remplacera le `%` dans la chaîne.

```
RAISE NOTICE ''Calling cs_create_job(%)'', v_job_id;
```

Cet exemple interrompra la transaction avec le message d'erreur donné.

```
RAISE EXCEPTION ''Inexistent ID --> %'', user_id;
```

PostgreSQL n'a pas un modèle d'exceptions très malin. Lorsque l'analyseur, planificateur/optimiseur ou exécuteur décide qu'une instruction ne peux pas continuer à être traitée, l'entière transaction s'interrompt et le système renvient à la boucle principale pour reçevoir la prochaîne commande de l'application client.
Il est possible de faire un crochet par le mécanisme d'erreurs pour examiner ce qui se passe. Mais actuellement il est impossible de dire ce qui a causé réellement l'interruption (erreur de format de type de données, erreur de virgule flottante, erreur de syntaxe, etc.). Et il est possible que le serveur de base de données soit dans un état inconscient à ce point, et que renvoyer à l'exécuteur supérieur ou traiter d'autres commandes puisse corrompre la base de données entière.
Ainsi, la seule chose que PL/pgSQL fait actuellement quand il rencontre une interruption dans une procédure fonction ou déclencheur est d'écrire quelques messages de log additionnels de niveau `NOTICE` indiquant dans quelle fonction et à quel endroit (numéro de ligne et type d'instruction) c'est arrivé. L'erreur arrête toujours l'exécution de la fonction.

Chapter 61

PL/pgSQL : procédures déclencheurs

PL/pgSQL peut être utilisé pour définir des procédures déclencheur. Une procédure déclencheur est créée grâce à la commande `CREATE FUNCTION` utilisée comme fonction sans arguments ayant un type de retour trigger. Notez que la fonction doit être déclarée avec aucun argument même si elle s'attend à recevoir les arguments spécifiés dans `CREATE TRIGGER` (les arguments trigger sont passés via `TG_ARGV`, comme décrit plus loin).
Quand une fonction PL/pgSQL est appelée en tant que trigger, plusieurs variables spéciales sont créées automatiquement dans le bloc de plus haut niveau. Ce sont :

`NEW`
: Type de données `RECORD` ; variable contenant la nouvelle ligne de base de données pour les opérations `INSERT`/`UPDATE` dans les déclencheurs de niveau ligne. Cette variable est `null` dans un trigger de niveau instruction.

`OLD`
: Type de données `RECORD` ; variable contenant l'ancienne ligne de base de données pour les opérations `UPDATE`/`DELETE` dans les triggers de niveau ligne. Cette variable est `null` dans les triggers de niveau instruction.

`TG_NAME`
: Type de données `name` ; variable qui contient le nom du trigger réellement lancé.

`TG_WHEN`
: Type de données `text` ; une chaîne, soit `BEFORE` soit `AFTER` selon la définition du déclencheur.

`TG_LEVEL`
: Type de données `text` ; une chaîne, soit `ROW` soit `STATEMENT` selon la définition du déclencheur.

`TG_OP`
: Type de données `text` ; une chaîne, `INSERT`, `UPDATE`, ou `DELETE` indiquant pour quelle opération le déclencheur a été lancé.

`TG_RELID`
: Type de données `oid` ; l'`ID` de l'objet de la table qui a causé l'invocation du trigger.

`TG_RELNAME`
: Type de données `name` ; le nom de la table qui a causé l'invocation du trigger.

`TG_NARGS`
: Type de données `integer` ; le nombre d'arguments donnés à la procédure déclencheur dans l'instruction `CREATE TRIGGER`.

`TG_ARGV[]`
: Type de donnée `text` ; les arguments de l'instruction `CREATE TRIGGER`. L'index débute à `0`. Les indices invalides (inférieurs à `0` ou supérieurs ou égaux à `tg_nargs`) auront une valeur nulle.

Une fonction déclencheur doit renvoyer soit `null` soit une valeur `record/ligne` ayant exactement la structure de la table pour laquelle le déclencheur a été lancé. La valeur de renvoi d'un déclencheur de niveau instruction `BEFORE` ou `AFTER` ou un déclencheur de niveau ligne `AFTER` est ignoré ; il peut être `null`. Cependant, n'importe lequel de ces déclencheurs peut interrompre l'opération déclencheur entière en levant une erreur.
Les déclencheurs de niveau ligne lancés `BEFORE` peuvent renvoyer null pour indiquer au gestionnaire de déclencheur de sauter le reste de l'opération pour cette ligne (les déclencheurs suivants ne sont pas lancés, et les `INSERT/UPDATE/DELETE` ne se font pas pour cette ligne). Si une valeur non null est renvoyée alors l'opération se déroule avec cette valeur ligne. Renvoyer une valeur ligne différente de la valeur originale de `NEW` modifie la ligne qui sera insérée ou mise à jour. Il est possible de remplacer des valeurs seules directement dans `NEW` et de renvoyer `NEW`, ou de construire un nouveau `record`/ligne à renvoyer.
L'exemple suivant montre un exemple d'une procédure déclencheur dans PL/pgSQL.
Cet exemple de déclencheur assure qu'à chaque moment où une ligne est insérée ou mise à jour dans la table, le nom d'utilisater courant et l'heure sont estampillés dans la ligne. Et cela assure qu'un nom d'employé est donné et que le salaire est une valeur positive.

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS '
    BEGIN
        -- Verifie que empname et salary sont donnés
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION ''empname cannot be null'';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION ''% cannot have null salary'', NEW.empname;
        END IF;

        -- Qui travaille pour nous quand elle doit payer pour cela ?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION ''% cannot have a negative salary'', NEW.empname;
        END IF;

        -- Rappelons nous qui a changé le payroll quand
        NEW.last_date := ''now'';
        NEW.last_user := current_user;
        RETURN NEW;
    END;
' LANGUAGE plpgsql;
```

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