Le langage procédural PL-PGSQL

PL/pgSQL est un langage procédural utilisé dans le système de gestion de bases de données PostgreSQL.

Objectifs :

- créer des fonctions standards et triggers,
- ajouter des structures de contrôle au langage SQL,
- effectuer des traitements complexes,
- pouvoir utiliser tous les types, les fonctions et les opérateurs définis par les utilisateurs
- faciliter à utiliser.

1. Quelques éléments

1.1. Avantages de l'utilisation de PL/pgSQL

SQL est le langage standard que le SGBD PostgreSQL et la plupart des autres SGBD relationnels utilisent 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.

L'application client envoie chaque requête au serveur de bases de données, attend que celui-ci la traite, reçoive et traite les résultats, faire quelques traitements, et envoie d'autres requêtes au serveur.

Ceci induit des communications inter-processus et induit aussi une surcharge du réseau si le client est sur une machine différente du serveur de bases de données.

Grâce à PL/pgSQL, on peut grouper un bloc de traitement et un ensemble de requêtes *au sein* du serveur de bases de données, et bénéficier ainsi de la puissance d'un langage procédural, avec beaucoup de gains car il n'y a plus la surcharge nécessaire à la communication client/serveur, notamment :

- Élimination des allers/retours entre le client et le serveur
- pas de nécessité de traiter ou transférer entre le client et le serveur les résultats intermédiaires dont le client n'a pas besoin
- pas de nécessité de s'occuper du va_et_vient des analyses de requêtes

Ceci peut permettre une augmentation considérable des performances en comparaison à une application qui n'utilise pas les procédures stockées.

Ainsi, avec PL/pgSQL, on peut utiliser tous les types de données, opérateurs et fonctions du SQL.

1.2. Arguments supportés et types de données résultats

Les fonctions écrites en PL/pgSQL peuvent accepter en 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 (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 « ensemble » 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.

– Une fonction PL/pgSQL peut être déclarée comme renvoyant le type *void* si elle n'a pas de valeur de retour utile.

–

Rmq

Les fonctions PL/pgSQL peuvent aussi être déclarées avec des paramètres en sortie à la place de la spécification explicite du code de retour. Ceci n'ajoute pas de fonctionnalité fondamentale au langage, mais c'est un moyen pour renvoyer plusieurs valeurs.

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 [ label ];
```

Initialisation des variables

Les variables déclarées dans la section **DECLARE** précédant un bloc sont initialisées à leur valeur par défaut chaque fois qu'on entre dans un bloc et pas seulement à chaque appel de fonction. Par exemple :

```
CREATE FUNCTION toto() RETURNS integer AS
$$
DECLARE
    quantité integer := 30;
BEGIN
    RAISE NOTICE 'quantité vaut ici %', quantité;  -- quantité vaut ici 30
    quantité := 50;
    --
    -- Crée un sous-bloc
    --
    DECLARE
        quantité integer := 80;
    BEGIN
        RAISE NOTICE 'quantité vaut ici %', quantité;  -- quantité vaut ici 80
    END;

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

    RETURN quantité;
END;
$$ LANGUAGE plpgsql;
```

```
=> fichier test.sql
```

Chargement :

```
> \i test.sql
```

Exécution :

```
> select * from toto();
```

Déclarations

Toutes les variables utilisées dans un bloc doivent être déclarées dans la section déclaration du bloc, sauf la variable de boucle d'une boucle FOR (déclarée automatiquement comme variable de type integer).

Les variables PL/pgSQL :sont de n'importe quel type de données (integer, varchar, ...)

Exemples de déclarations de variables :

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

Syntaxe générale d'une déclaration de variable est :

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

DEFAULT : valeur initiale assignée à la variable.

Si la clause DEFAULT n'est pas indiquée, la variable est initialisée à NULL.

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'assignation d'une valeur NULL aboutira à une erreur d'exécution.

Les valeurs par défaut des variables déclarées NOT NULL doivent être spécifiées non NULL.

Exemples :

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

Alias de paramètres de fonctions

Les paramètres passés aux fonctions sont nommés par les identifiants $1, $2, etc. (avant la vesrion 8 et supérieures). Des alias peuvent être déclarés, éventuellement, 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.

Il existe deux façons de créer un alias :

- La façon préférée est de donner un nom au paramètre dans la commande **CREATE FUNCTION**, par exemple :

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

- L'autre façon, la seule disponible pour les versions antérieures de PostgreSQL™ 8.0, est de déclarer explicitement un alias en utilisant la syntaxe de déclaration

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

Le même exemple dans ce style ressemble à :

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

Autres exemples :

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


CREATE FUNCTION concat_champs_selectionnes(in_t un_nom_de_table) RETURNS text AS
$$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

Quand une fonction PL/pgSQL est déclarée avec des paramètres en sortie, ces derniers se voient attribuer les noms $n et des alias optionnels de la même façon que les paramètres en entrée. Un paramètre en sortie est une variable qui commence avec la valeur NULL ; il devrait se voir attribuer une valeur lors de l'exécution de la fonction. La valeur finale du paramètre est ce qui est renvoyée. Exemple, taxe_ventes peut s'écrire de cette façon :

```
CREATE FUNCTION taxe_ventes(sous_total real, OUT taxe real) AS $$
BEGIN
    taxe := sous_total * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Notez que nous avons omis RETURNS real. Nous aurions pu l'inclure mais cela aurait été redondant (avec out ...).

Les paramètres en sortie sont encore plus utiles lors du retour de plusieurs valeurs. Un exemple trivial est :

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

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 cela ne soit pas requis. On peut aussi donner un alias à $0.

Par exemple, cette fonction s'exécute comme un opérateur + (addition) pour n'importe quel type de données.

```
CREATE FUNCTION ajoute_trois_valeurs(v1 anyelement, v2 anyelement, v3
anyelement)
RETURNS anyelement AS $$
DECLARE
    resultat ALIAS FOR $0;
BEGIN
    resultat := v1 + v2 + v3;
    RETURN resultat;
END;
$$ LANGUAGE plpgsql;
```

Le même effet peut être obtenu en déclarant un ou plusieurs paramètres en sortie comme anyelement ou anyarray. Dans ce cas, le paramètre spécial `$0` n'est pas utilisé ; les paramètres en sortie servent ce même but. Par exemple :

```
CREATE FUNCTION ajoute_trois_valeurs(v1 anyelement, v2 anyelement, v3
anyelement, OUT somme anyelement)
AS $$
BEGIN
    somme := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

36.4.2. Copie de types (que le type de colonne référencée)

```
variable%TYPE
```

%TYPE fournit le type de données d'une variable ou d'une colonne de table. Par exemple, disons qu'on a une colonne nommée `id_utilisateur` dans une table `utilisateurs`. Pour déclarer une variable du même type de données que `utilisateurs.id_utilisateur`, on peut écrire :

```
id_utilisateur utilisateurs.id_utilisateur%TYPE;
```

En utilisant %TYPE von n'a pas besoin de connaître le type de données de la structure à laquelle on fait référence et, plus important, si le type de données de l'objet référencé change dans le futur (par exemple : on change le type de `id_utilisateur` de integer à real), on peut ne pas avoir besoin de changer la définition de la 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 variable fictive de résultat.

36.4.3. Types *ligne*

```
nom nom_table%ROWTYPE;
nom nom_type_composite;
```

Variable de type composite : appelée variable *ligne* (ou variable *row-type*).

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 `varligne.champ`.

Une variable ligne peut être déclarée de façon à avoir le même type que les lignes d'une table ou d'une vue existante, en utilisant la notation `nom_table %ROWTYPE`.

Elle peut aussi 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). Dans ce cas, l'identifiant correspondant $n sera une variable ligne à partir de laquelle les champs peuvent être sélectionnés avec la notation pointée, par exemple `$1.id_utilisateur`.

Seules les colonnes définies par l'utilisateur 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).

Exemple d'utilisation des types composites.

table1 et table2 sont des tables ayant au moins les champs mentionnés :

```
CREATE FUNCTION assemble_champs(t_ligne table1) RETURNS text AS $$
DECLARE
    t2_ligne table2%ROWTYPE;

BEGIN
    SELECT * INTO t2_ligne FROM table2 WHERE ... ;
    RETURN t_ligne.f1 || t2_ligne.f3 || t_ligne.f5 || t2_ligne.f7;
END;
$$ LANGUAGE plpgsql;

SELECT assemble_champs(t.*) FROM table1 t WHERE ... ;
```

Types record

```
nom RECORD;
```

Variables record : similaires aux variables de type ligne (mais pas de structure prédéfinie.)

Elles empruntent la structure effective de type ligne de la ligne à laquelle elles sont affectées durant une commande **SELECT** ou **FOR**.

La sous-structure d'une variable record peut changer à chaque fois qu'on l'affecte ==> elle n'a pas de sous-structure jusqu'à ce qu'elle ait été affectée, et toutes les tentatives pour accéder à un de ses champs entraînent une erreur d'exécution.

Rmq.

RECORD n'est pas un vrai type de données (seulement un paramètre fictif (placeholder)).

Lorsqu'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.

RENAME

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

Changer le nom d'une variable, d'un record ou d'un ligne (ROW).

C'est particulièrement utile si NEW ou OLD doivent être référencés par un autre nom dans une procédure trigger.

Exemples :

```
RENAME id TO id_utilisateur;
RENAME cette_var TO cette_autre_var;
```

Instructions de base

=> 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).

1. Assignation

Assignation d'une valeur à une variable ou à un champ row/record :

```
identifiant := expression;
```

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 résultat 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 à 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 :

```
id_utilisateur := 20;
tax := sous_total * 0.06;
```

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.

L'expression `select_expressions` et le reste de la commande sont identiques à du SQL standard.

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 on veut créer une table à partir du résultat d'un **SELECT** d'une fonction PL/pgSQL, utiliser 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. 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.

La clause `INTO` peut apparaître pratiquement partout dans l'instruction **SELECT**. De façon personnalisé, il est écrit soit juste après SELECT comme indiqué ci-dessus soit juste avant `FROM` -- c'est-à-dire soit juste avant soit juste après la liste de `select_expressions`.

Si la requête ne renvoie aucune ligne, des valeurs NULL sont assignées au(x) cibles(s). Si la requête renvoie plusieurs lignes, la première ligne est assignée au(x) cible(s) et le reste est rejeté.

Actuellement, la clause `INTO` peut apparaître presque n'importe où dans l'instruction **SELECT** mais il est recommandé de la placer immédiatement après le mot clé SELECT comme décrit plus haut.

On peut vérifier la variable spéciale `FOUND` après une instruction **SELECT INTO** pour déterminer si l'affectation est réussie, c'est-à-dire si au moins une ligne a été renvoyée par la requête. Par exemple :

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

Pour tester si un résultat record/ligne est NULL, on peut utiliser la conditionnelle `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
    enreg_utilisateurs RECORD;
BEGIN
    SELECT INTO enreg_utilisateurs * FROM utilisateurs WHERE id_utilisateur=3;

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

3. Exécuter une expression ou requête sans résultat

Quelque fois, on souhaite évaluer une expression ou une requête mais rejeter le résultat (généralement parce qu'on appelle une fonction qui a des effets de bords utiles mais pas de résultat utile) => on utilise l'instruction **PERFORM** :

```
PERFORM requête;
```

=> exécute requête et annule le résultat. Écrire *requête* de la même façon que dans une commande SQL **SELECT** mais en remplaçant le mot clé initial **SELECT** par **PERFORM**.

Les variables PL/pgSQL seront substituées dans la requête comme d'habitude. 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.

Note

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 creer_vuemat('cs_session_page_requests_mv', ma_requete);
```

4. Ne rien faire

Quelque fois, une instruction qui ne fait rien est utile. Par exemple, il indique qu'une partie de la chaîne *if/then/else* est délibérément vide. Pour cela, utiliser l'instruction :

```
NULL;
```

Par exemple, les deux fragments de code suivants sont équivalents :

```
BEGIN
  y := x / 0;
  EXCEPTION
  WHEN division_by_zero THEN
    NULL;  -- ignore l'erreur
  END;

BEGIN
  y := x / 0;
  EXCEPTION
  WHEN division_by_zero THEN  -- ignore l'erreur
  END;
```

5. Exécuter des commandes dynamiques

Souvent, on voudrait générer des commandes dynamiques dans les fonctions PL/pgSQL, c'est-à-dire des commandes en paramètres.

Pour gérer ce type de problème, l'instruction **EXECUTE** est fournie :

```
EXECUTE chaîne-commande [ INTO cible ];
```

où chaîne-commande est une expression manipulant une chaîne (de type text) contenant la commande à être exécutée et cible est une variable record ou ligne ou une liste de variables simples ou de champs de lignes/enregistrements séparées par des virgules.

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.

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 tables ou colonnes différentes.

La clause INTO spécifie où les résultats d'une commande **SELECT** devraient être affectés.

Si une ligne ou une liste de variable est fournie, elle doit correspondre exactement à la structure des résultats produits par le **SELECT** (quand une variable de type record est utilisée, elle se configurera toute seule pour correspondre à la structure du résultat).

Si plusieurs lignes sont renvoyées, alors seule la première sera assignée à la variable INTO.

Si aucune ligne n'est renvoyée, NULL est affectée à la variable INTO. Si aucune clause INTO n'est spécifiée, les résultats d'une commande **SELECT** sont annulées.

SELECT INTO n'est pas supporté actuellement à l'intérieur de **EXECUTE**.

En travaillant avec des commandes dynamiques, vous aurez souvent à gérer des échappements de guillemets simples. La méthode recommandée pour mettre entre guillemets un texte fixe dans le corps de votre fonction est d'utiliser les guillemets dollar.

Les valeurs dynamiques qui sont à insérer dans la requête construite requièrent une gestion spéciale car elles pourraient elles-même contenir des guillemets. Un exemple (ceci suppose que l'on utilise les guillemets dollar pour la fonction dans sa globalité, du coup les guillemets n'ont pas besoin d'être doublés) :

```
EXECUTE 'UPDATE tbl SET '
    || quote_ident(nom_colonne)
    || ' = '
    || quote_literal(nouvelle_valeur)
    || ' WHERE cle = '
    || quote_literal(valeur_cle);
```

Cet exemple démontre l'utilisation des fonctions quote_ident et quote_literal. Pour plus de sûreté, les expressions contenant les identifiants des colonnes et des tables doivent être passés à la fonction quote_ident. Les expressions contenant les valeurs devant être des chaînes dans la commande construite devraient être passées à quote_literal. Les deux font les étapes appropriées pour renvoyer le texte en entrée entouré par des guillemets doubles ou simples respectivement, avec tout caractère intégré spécial proprement échappé.

Noter que les guillemets dollar sont souvent utiles pour placer un texte fixe entre guillemets. Ce serait une très mauvaise idée d'essayer de faire l'exemple ci-dessus de cette façon

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

car cela casserait si le contenu de newvalue pouvait contenir $$. La même objection s'appliquerait à tout délimiteur dollar que vous pourriez choisir. Donc, pour mettre un texte inconnu entre guillemets de façon sûr, vous *devez* utiliser quote_literal.

6. Statut du résultat

Plusieurs moyens pour déterminer l'effet d'une commande.

La première méthode est d'utiliser **GET DIAGNOSTICS** :

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

=> permet la récupération des indicateurs d'état du système. Chaque élément est un mot clé identifiant une valeur d'état devant être affectée à la variable indiquée (qui doit être du bon type de donnée pour que l'affectation puisse se faire sans erreur.)

Les éléments 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.

Noter que RESULT_OID n'est utile qu'après une commande **INSERT** dans une table contenant des OID.

Exemple :

```
GET DIAGNOSTICS var_entier = ROW_COUNT;
```

La seconde méthode permettant de déterminer les effets d'une commande est la variable spéciale nommée FOUND de type boolean. La variable FOUND est initialisée à false au début de chaque fonction PL/pgSQL. Elle est positionnée par chacun 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, faux 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 positionnée de cette façon que quand la boucle **FOR** s'achève ; dans l'exécution de la chaîne, FOUND n'est pas modifiée par l'instruction **FOR**, bien qu'elle puisse être modifié par l'exécution d'autres instructions situées dans le corps de la boucle.

FOUND est une variable locale à l'intérieur de chaque fonction PL/pgSQL ; chaque changement qui y est fait n'affecte que la fonction en cours.

Structures de contrôle

Partie la plus importante de PL/pgSQL. Manipuler les données PostgreSQL™ de façon très flexible et puissante.

1. Valeur de retour d'une fonction

Permettent de renvoyer des données d'une fonction : **RETURN** et **RETURN NEXT**.

1.1. RETURN

```
RETURN expression;
```

RETURN accompagné d'une expression termine la fonction et renvoie le valeur de l'expression à l'appelant. A 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), on écrit une variable record ou ligne comme expression.

Si on déclare la fonction avec des paramètres en sortie, écrire seulement **RETURN** sans expression. Les valeurs courantes des paramètres en sortie seront renvoyées.

Si on déclare que la fonction renvoie void, une instruction **RETURN** peut être utilisée pour quitter rapidement la fonction ; mais ne pas écrire d'expression après **RETURN**.

La valeur de retour d'une fonction ne peut pas être laissée indéfinie. 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.

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 : 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 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.

Si fonction avec des paramètres en sortie, écrire simplement **RETURN NEXT** sans expression. Les valeurs en cours de(s) paramètre(s) en sortie seront sauvegardées pour un retour éventuel.

Noter

- qu'on doit 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();
```

Rmq :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 stocke l'ensemble des résultats avant d'effectuer le retour de la fonction =>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.

2. Contrôles conditionnels

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

2.1. IF-THEN

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

Instructions entre THEN et END IF exécutées si condition vraie.

Exemple :

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

2.2. IF-THEN-ELSE

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

Spécifie 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;
```

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;
```

=> Besoin d'une instruction END IF pour chaque IF imbriqué et une pour le IF-ELSE parent : fastidieux quand il y a de nombreuses alternatives à traiter.

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;
```

=> Equivalente formellement aux commandes IF-THEN-ELSE-IF-THEN imbriquées, mais un seul END IF est nécessaire.

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;
```

2.5. IF-THEN-ELSEIF-ELSE

ELSEIF est un alias pour ELSIF.

7.3. Boucles simples

LOOP, EXIT, CONTINUE, WHILE et FOR : les fonctions PL/pgSQL répètent une série de commandes.

3.1. LOOP

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

LOOP définit une boucle inconditionnelle répétée indéfiniment jusqu'à EXIT ou **RETURN**. Le label optionnel utilisé par EXIT et CONTINUE pour les boucles imbriquées pour définir la boucle impliquée.

3.2. EXIT

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

Si aucun label, la boucle la plus imbriquée se termine et l'instruction suivant END LOOP est exécutée.

Si label, 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;
```

3.3. CONTINUE

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

Si aucun label, 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 label est présent, il s'agit du label de la boucle dont l'exécution va être continuée.

Si WHEN est spécifié, la prochaine itération de la boucle est commencée seulement si l'expression vaut true.

Si WHEN absent, 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;
```

3.4. WHILE

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

Répète une séquence d'instructions aussi longtemps que l'expression conditionnelle est évaluée à vrai. La condition est vérifiée avant chaque entrée dans le corps de la boucle.

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;
```

3.5. FOR (variante avec entier)

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

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 évaluées une fois en entrant dans la boucle. Le pas de l'itération est normalement de 1 mais vaut -1 quand REVERSE est spécifié.

Quelques exemples de boucles FOR avec des 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;
```

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

4. Boucler dans les résultats de requêtes

En utilisant un type de FOR différent, on peut itérer au travers des résultats d'une requête (et par là-même manipuler ses données). Syntaxe :

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

La variable record ou ligne est successivement assignée à chaque ligne résultant de la requête (qui doit être une commande **SELECT**) et le corps de la boucle est exécuté pour chaque ligne.

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

        -- À 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 après la boucle.

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

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

Identique à la forme précédente, à ceci près que l'expression **SELECT** source est spécifiée comme une expression chaîne, évaluée et re-planifiée à chaque entrée dans la boucle FOR.

Permet au développeur de choisir entre la vitesse d'une requête préplanifiée et la flexibilité d'une requête dynamique, uniquement avec l'instruction **EXECUTE**.

Rmq :

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 » (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 on pourrait s'y attendre.

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. On peut 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.

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 correspondance trouvée, les instructions_gestion_erreurs correspondantes sont exécutées puis le contrôle est passé à l'instruction suivant le END.

Si aucune correspondance 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 le fichier pl-pgsql.htm. Un nom de catégorie correspond à toute erreur contenue dans cette catégorie.

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.

Exemple :

```
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.

Rmq :

Un bloc contenant une clause EXCEPTION est significativement plus coûteux en entrée et en sortie qu'un bloc sans. Du coup, ne pas utiliser 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.

La variable SQLERRM contient le message d'erreur associé avec l'exception. Ces variables sont indéfinies à l'extérieur des gestionnaires d'exceptions.

Exemple 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');
```

10. Procédures *Trigger*

Une procédure trigger est créée grâce à la commande **CREATE FUNCTION** utilisée comme fonction sans arguments ayant un type de retour trigger.

Noter 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 triggers 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 trigger.

TG_LEVEL

Type de données text ; une chaîne, soit ROW soit STATEMENT, selon la définition du trigger.

TG_OP

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

TG_RELID

Type de données oid ; l'ID de l'objet de la table qui a causé le déclenchement du trigger.

TG_RELNAME

Type de données nom ; le nom de la table qui a causé le déclenchement.

TG_NARGS

Type de données integer ; le nombre d'arguments donnés à la procédure trigger 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 NULL.

Une fonction trigger doit renvoyer soit NULL, soit une valeur record ayant exactement la structure de la table pour laquelle le trigger a été lancé.

Les triggers de niveau ligne lancés BEFORE peuvent renvoyer NULL pour indiquer au gestionnaire de trigger de sauter le reste de l'opération pour cette ligne (les triggers 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 (mais n'a pas d'effet sur le cas **DELETE**).

Pour modifier la ligne à stocker, il est possible de remplacer des valeurs seules directement dans NEW et de renvoyer NEW, ou de construire un nouveau record/ligne à renvoyer.

La valeur de retour d'un trigger de niveau instruction BEFORE ou AFTER ou un trigger de niveau ligne AFTER est toujours ignoré ; il pourrait aussi bien être NULL. Néanmoins, tous les types de triggers peuvent toujours annuler l'opération complète en envoyant une erreur.

Exemple 36.2. Une procédure trigger PL/pgSQL

Cet exemple de trigger assure qu'à chaque moment où une ligne est insérée ou mise à jour dans la table, le nom de l'utilisateur 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 (
    nom_employe text,
    salaire integer,
    date_dermodif timestamp,
    utilisateur_dermodif text
);


CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Verifie que nom_employe et salary sont donnés
        IF NEW.nom_employe IS NULL THEN
            RAISE EXCEPTION 'nom_employe ne peut pas être NULL';
        END IF;
        IF NEW.salaire IS NULL THEN
            RAISE EXCEPTION '% ne peut pas avoir un salaire', NEW.nom_employe;
        END IF;

        -- Qui travaille pour nous si la personne doit payer pour cela ?
        IF NEW.salaire < 0 THEN
            RAISE EXCEPTION '% ne peut pas avoir un salaire négatif',
NEW.nom_employe;
        END IF;

        -- Rappelons-nous qui a changé le salaire et quand
        NEW.date_dermodif := current_timestamp;
        NEW.date_dermodif := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

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

Une autre façon de tracer les modifications sur une table implique la création d'une nouvelle table qui contient une ligne pour chaque insertion, mise à jour ou suppression qui survient. Cette approche peut être vue comme un audit des modifications sur une table.

Exemple 36.3. Une procédure d'audit par trigger en PL/pgSQL

Cet exemple de trigger nous assure que toute insertion, modification ou suppression d'une ligne dans la table `emp` est enregistrée dans la table `emp_audit`. L'heure et le nom de l'utilisateur sont conservées dans la ligne avec le type d'opération réalisé.

```
CREATE TABLE emp (
    nom_employe       text NOT NULL,
    salaire           integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    tampon            timestamp NOT NULL,
    id_utilisateur    text      NOT NULL,
    nom_employe       text      NOT NULL,
    salaire           integer
);


CREATE OR REPLACE FUNCTION audit_employe() RETURNS TRIGGER AS $emp_audit$
BEGIN
    --
    -- Ajoute une ligne dans emp_audit pour refléter l'opération réalisée
    -- sur emp,
    -- utilise la variable spéciale TG_OP pour cette opération.
    --
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
        RETURN NEW;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
        RETURN NEW;
    END IF;
    RETURN NULL; -- le résultat est ignoré car il s'agit d'un trigger AFTER
END;
$emp_audit$ language plpgsql;

CREATE TRIGGER emp_audit
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE audit_employe();
```

Une utilisation des triggers est le maintien d'une table résumée d'une autre table. Le résumé résultant peut être utilisé à la place de la table originale pour certaines requêtes -- souvent avec des temps d'exécution bien réduits. Cette technique est souvent utilisée pour les statistiques de données où les tables de données mesurées ou observées (appelées des tables de faits) peuvent être extrêmement grandes.L'exemple montre un exemple d'une procédure trigger en PL/pgSQL maintenant une table résumée pour une table de faits dans un système de données (data warehouse).

Exemple 36.4. Une procédure trigger PL/pgSQL pour maintenir une table résumée

Le schéma détaillé ici est partiellement basé sur l'exemple du *Grocery Store* provenant de *The Data Warehouse Toolkit* par Ralph Kimball.

```
--
-- Tables principales - dimension du temps de ventes.
--
CREATE TABLE time_dimension (
  time_key                    integer NOT NULL,
  day_of_week                 integer NOT NULL,
  day_of_month                integer NOT NULL,
  month                       integer NOT NULL,
  quarter                     integer NOT NULL,
  year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
  time_key                    integer NOT NULL,
  product_key                 integer NOT NULL,
  store_key                   integer NOT NULL,
  amount_sold                 numeric(12,2) NOT NULL,
  units_sold                  integer NOT NULL,
  amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);
--
-- Table résumé - ventes sur le temps.
--
CREATE TABLE sales_summary_bytime (
  time_key                    integer NOT NULL,
  amount_sold                 numeric(15,2) NOT NULL,
  units_sold                  numeric(12) NOT NULL,
  amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);
--
-- Fonction et trigger pour amender les colonnes résumées
-- pour un UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER AS
$maint_sales_summary_bytime$
DECLARE
  delta_time_key          integer;
  delta_amount_sold       numeric(15,2);
  delta_units_sold        numeric(12);
  delta_amount_cost       numeric(15,2);
BEGIN

  -- Travaille sur l'ajout/la suppression de montant(s).
  IF (TG_OP = 'DELETE') THEN

    delta_time_key = OLD.time_key;
    delta_amount_sold = -1 * OLD.amount_sold;
    delta_units_sold = -1 * OLD.units_sold;
    delta_amount_cost = -1 * OLD.amount_cost;

  ELSIF (TG_OP = 'UPDATE') THEN

    -- interdit les mises à jour qui modifient time_key -
    -- (probablement pas trop cher, car DELETE + INSERT est la façon la plus
    -- probable de réaliser les modifications).
    IF ( OLD.time_key != NEW.time_key) THEN
      RAISE EXCEPTION 'Update of time_key : % -> % not allowed', OLD.time_key,
NEW.time_key;
    END IF;
```

```
    delta_time_key = OLD.time_key;
    delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
    delta_units_sold = NEW.units_sold - OLD.units_sold;
    delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

  ELSIF (TG_OP = 'INSERT') THEN

    delta_time_key = NEW.time_key;
    delta_amount_sold = NEW.amount_sold;
    delta_units_sold = NEW.units_sold;
    delta_amount_cost = NEW.amount_cost;
  END IF;


  -- Insertion ou mise à jour de la ligne de résumé avec les nouvelles valeurs.
  <<insert_update>>
  LOOP
  UPDATE sales_summary_bytime
  SET amount_sold = amount_sold + delta_amount_sold,
    units_sold = units_sold + delta_units_sold,
    amount_cost = amount_cost + delta_amount_cost
    WHERE time_key = delta_time_key;

    EXIT insert_update WHEN found;

    BEGIN
      INSERT INTO sales_summary_bytime (
        time_key,
        amount_sold,
        units_sold,
        amount_cost)
        VALUES (
        delta_time_key,
        delta_amount_sold,
        delta_units_sold,
        delta_amount_cost
        );
      EXIT insert_update;

      EXCEPTION
      WHEN UNIQUE_VIOLATION THEN
      -- do nothing
      END;
      END LOOP insert_update;

  RETURN NULL;

END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
  AFTER INSERT OR UPDATE OR DELETE ON sales_fact
  FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
```

```
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```