Fonctions (plpgSQL)

BD4 2015-16 L3 MIASHS M1 ISIFAR

SB AD FC NGdR

22 Mars 1016

Fonctions plpgsql (I)

Pourquoi intégrer un langage de programmation comme `plpgsql` dans un SGBD ?

Trois objectifs

- Automatisation de taches répétitives (administration)
  - On veut répéter une même tache sur une collection de schémas.
  - On veut traiter une collection de roles.
- Calculs impossibles réaliser en SQL
  Exemple : calculer la *fermeture transitive* d'une relation comme `film_actors`
- Triggers
  Certaines contraintes ne peuvent pas être mises en place avec les seules constructions `primary key`, `unique`, `foreign key`, `check` et `exclude` (notamment des contraintes d'exclusion qui mettent en jeu plusieurs tables).
  On peut les maintenir à l'aide de traitements spéciaux les `triggers`.
  Les triggers reposent sur des fonctions spéciales.

`plpgsql`

↪ Programming Language PostGres SQL

Automatisation des tâches répétitives : deux outils nécessaires

Lorsqu'on administre une base,

on doit souvent engendrer des *requêtes dynamiques* à l'intérieur d'une fonction PL/pgSQL, c'est à dire des commandes qui vont concerner des tables ou des types différents à chaque exécution. Les *requêtes préparées* sont alors très utilement combinées avec les *structures de contrôle* (itérations, alternatives) pour automatiser les tâches !

2 outils

- requêtes dynamiques `EXECUTE, PREPARE`
- structures de contrôles `IF, LOOP, ...`

↪ Ces deux constructions ne font pas partie des instructions de base de SQL

Un premier cas

Une tâche de surveillance/maintenance

Objectifs

Déterminer pour chaque usager (*schéma*) le nombre de tuples dans la table `ville_pays`

Pour chaque schéma `schema`

on veut évaluer une requête

```
SELECT COUNT(*) FROM schema.ville_pays ;
```

Ici `schema` doit être calculé en interrogeant le SGBD

situation inédite :

- comment déterminer les schémas pertinents ?

La métabase : `information_schema` et `pg_catalog`

Informations de base

- chaque usager correspond à un `role` et ce rôle correspond dans notre cas à un schema créé à partir du rôle via l'instruction `CREATE SCHEMA AUTHORIZATION user_name ;`
- On a envie d'écrire une requête comme `SELECT COUNT(*) FROM username.tournaments ;` où `username` est collectée à partir de `SELECT usename FROM pg_catalog.pg_user ;`.

Deux schémas pour l'administration : la **métabase**

`information_schema`

Ce schema contient l'information sur les `schémas` du cluster/catalogue :

- les définitions de tables, de vues, de colonnes, les contraintes, ...
- il est formé de tables et surtout de (très nombreuses) vues
- les instructions, `CREATE`, `ALTER`, `DROP` modifient le contenu de ce schéma (une seule instruction `ALTER TABLE` peut engendrer plusieurs mises à jours dans les tables de `information_schema`)

`pg_catalog`

Ce schema contient lui aussi beaucoup de tables et vues utiles au fonctionnement du serveur

Exemple de vue de `pg_catalog`

```
BD4-2015=# \d pg_catalog.pg_user
 View "pg_catalog.pg_user"
   Column    |   Type    | Modifiers
-------------+-----------+-----------
 usename     | name      |
 usesysid    | oid       |
 usecreatedb | boolean   |
 usesuper    | boolean   |
 usecatupd   | boolean   |
 userepl     | boolean   |
 passwd      | text      |
 valuntil    | abstime   |
 useconfig   | text[]    |
```

`pg_user` nous renseigne sur

- les usagers (`usename`)
- leur statut (`usesuper` : super-utilisateur ou pas)
- leurs privilèges (`createdb`: peut créer une base ou non)
- ...

Autre exemple d'usage de la métabase

```
SELECT datname, application_name, client_addr, backend_start, state
FROM pg_catalog.pg_stat_activity ;

 datname | application_name | client_addr |         backend_start
---------+------------------+-------------+-------------------------------
 bd_2016 | psql             | 127.0.0.1   | 2016-03-24 08:20:05.383029+01
 bd_2016 |                  | 127.0.0.1   | 2016-03-24 08:22:29.552843+01
 bd_2016 |                  | 127.0.0.1   | 2016-03-24 08:22:29.292983+01
 bd_2016 | sqltabs          | 127.0.0.1   | 2016-03-24 08:23:01.63413+01
 bd_2016 |                  | 127.0.0.1   | 2016-03-24 08:23:31.035217+01
 bd_2016 |                  | 127.0.0.1   | 2016-03-24 08:23:31.2528+01
 bd_2016 |                  | 127.0.0.1   | 2016-03-24 08:23:31.671399+01
 bd_2016 |                  | 127.0.0.1   | 2016-03-24 08:23:31.855202+01
```

Renseigne sur les utilisateurs ayant une session en cours

Tentative

On engendre *dynamiquement* une série de requêtes

par une instruction de la forme suivante :

Requête

```
SELECT 'SELECT '|| quote_literal(usename) || ', COUNT(*) FROM '||
FROM pg_catalog.pg_user  ;
```

A quoi sert `quote_literal()` ?

Le résultat est

↪ une table de chaines de caractères

Tel quel, cela ne fonctionnera pas !

- Il faut pouvoir confier ces chaines de caractères à l'évaluateur de requêtes
- Il faut pouvoir le faire pour chacune des chaines de caractères produites par la requête (itérer)

Mode opératoire

```
CREATE OR REPLACE FUNCTION taille_ville_pays()
LANGUAGE plpgsql RETURNS  TEXT AS
$$
DECLARE
  stmt CHARACTER VARYING ;
  usename CHARACTER VARYING ;
  result CHARACTER VARYING    := '' ;
  resp CHARACTER VARYING := '' ;
BEGIN
FOR usename IN SELECT u.usename
  FROM pg_catalog.pg_user AS u JOIN
  information_schema.tables t ON
  (u.usename=t.table_schema and t.table_name ='ville_pays')
LOOP
  stmt = 'SELECT CAST(COUNT(*) AS VARCHAR) FROM '|| usename||'.ville_pays;
  EXECUTE stmt INTO resp ;
  result := result || usename || ' :  ' || resp || ';  ' ;
END LOOP ;
RETURN  result  ;
END ; $$  ;
```

Remarques

Itérations

Dans cette fonction, nous utilisons une des structures itératives de plpgSQL :

```
FOR v IN expression LOOP instructions END LOOP ;
```

Ici, on itère sur les tuples de la requête `SELECT u.usename ...`
Voir Documentation officielle pour les autres constructions possibles

Exécution dynamique

Dans le corps de la boucle `LOOP`, on fabrique une requête : `stmt := ...` Puis on exécute cette requête : `EXECUTE stmt INTO resp ;`
Le résultat de la requête engendrée dynamiquement est affecté à la variable locale `stmt`
Le contenu est accumulé dans `result`

Renvoi du résultat

plpgSQL propose une variété de manières de construire et de renvoyer le résultat d'une fonction.
Voir Documentation officielle

Fonctions issues du schéma `sakila`

Fonction inventory_in_stock

Objectif

calculer si un dvd est en stock ou pas

Un DVD est en stock

- s'il n'a jamais été loué
- OU
- si toutes les locations de ce DVD sont déjà terminées (return_date n'est pas nul)

Signature de la fonction inventory_in_stock

```
CREATE FUNCTION inventory_in_stock(p_inventory_id integer)
 RETURNS boolean
 LANGUAGE plpgsql
AS $function$
DECLARE
v_rentals INTEGER;
v_out     INTEGER;
BEGIN
-- AN ITEM IS IN-STOCK IF THERE ARE
-- EITHER NO ROWS IN THE rental TABLE FOR THE ITEM
-- OR ALL ROWS HAVE return_date POPULATED


END $function$
;
```

Corps de la fonction inventory_in_stock

```
SELECT count(*) INTO v_rentals
FROM rental
WHERE inventory_id = p_inventory_id;

IF v_rentals = 0 THEN
  RETURN TRUE;
END IF;

SELECT COUNT(rental_id) INTO v_out
FROM inventory LEFT JOIN rental USING(inventory_id)
WHERE inventory.inventory_id = p_inventory_id
AND rental.return_date IS NULL;

IF v_out > 0 THEN
  RETURN FALSE;
ELSE
  RETURN TRUE;
END IF;
```

A noter

`SELECT ... INTO ...`

Le résultat de la requête est ici un entier, il est affecté à une variable locale `v_rentals, v_out, ...`
En `plpgsql`, le résultat d'une requête doit être mémorisé (ou explicitement négligé en utilisant `PERFORM` plutôt que `SELECT`)

`IF ... THEN ...`

Alternative, comme dans un langage de programmation ordinaire

`RETURN`

retourne le résultat et termine l'exécution de la fonction

Un exemple plus complexe issu de sakila

rewards_report signature

```
CREATE FUNCTION rewards_report(min_monthly_purchases integer,
                               min_dollar_amount_purchased numeric)
 RETURNS SETOF customer LANGUAGE plpgsql  AS $$
DECLARE
last_month_start DATE;
last_month_end DATE;
rr RECORD;
tmpSQL TEXT;
BEGIN
    .....
END  $$ ;
```

Détermine la liste des bons clients qui beaucoup consommé durant le dernier mois écoulé.

A noter :

- la signature comporte une déclaration de type originale
- la déclaration des variables locales mentionne un type fourre-tout

`SETOF customer`

Table de même schéma que `customer` (mêmes colonnes)
Construction très très utile : `SETOF nom_de_table`

`RECORD`

Un type générique (fourre-tout) pour désigner les types *composés* (en particulier comme les types définis à partir des tables)

rewards_report corps I

```
/* Some sanity checks... */
IF min_monthly_purchases = 0 THEN
    RAISE EXCEPTION
        'Minimum monthly purchases parameter must be > 0';
END IF;
IF min_dollar_amount_purchased = 0.00 THEN
    RAISE EXCEPTION
 'Minimum monthly dollar amount purchased parameter must be > $0.00';
END IF;

last_month_start := CURRENT_DATE - '3 month'::interval;
last_month_start := to_date((extract(YEAR FROM last_month_start) || '-' ||
last_month_end := LAST_DAY(last_month_start);
```

A noter

`RAISE EXCEPTION 'msg'`

termine (en erreur) l'exécution de la fonction envoie un message d'erreur

Sur les dates

```
bd_2016=# SELECT quote_literal(now()) AS foo \gset _
bd_2016=# SELECT  :_foo::date ;
    date
------------
 2016-03-24
bd_2016=# SELECT  to_date((extract(YEAR FROM :_foo::date) || '-' |
                          extract(MONTH FROM :_foo::date) || '-01
  to_date
------------
 2016-03-01
```

rewards_report corps II

```
/*
Create a temporary storage area for Customer IDs.
*/
CREATE TEMPORARY TABLE tmpCustomer
      (customer_id INTEGER NOT NULL PRIMARY KEY);

/*
Find all customers meeting the monthly purchase requirements
*/

tmpSQL := 'INSERT INTO tmpCustomer (customer_id)
    SELECT p.customer_id
    FROM payment AS p
    WHERE DATE(p.payment_date) BETWEEN '||quote_literal(last_month_start)
          AND '|| quote_literal(last_month_end) || '
    GROUP BY customer_id
    HAVING SUM(p.amount) > '|| min_dollar_amount_purchased || '
    AND COUNT(customer_id) > ' || min_monthly_purchases ;

EXECUTE tmpSQL;
```

A noter

`CREATE TEMPORARY TABLE ...`

Crée une table (très simple ici) qui sera détruite avant la fin de l'exécution de la fonction. Si la fonction devait être interrompue, cette table ne survivrait pas à la session qui a invoqué la fonction.

Pourquoi utiliser une requête créée dynamiquement ?

- Est ce une nécessité ici ?
- Si non, quel est l'intérêt ?

rewards_report corps III

```
/*
Output ALL customer information of matching rewardees.
Customize output as needed.
*/
FOR rr IN
    EXECUTE
    'SELECT c.* FROM tmpCustomer AS t INNER JOIN customer AS c ON t.custom
LOOP
    RETURN NEXT rr;
END LOOP;

/* Clean up */
tmpSQL := 'DROP TABLE tmpCustomer';
EXECUTE tmpSQL;

RETURN;
END
$function$
;
```

A noter

Les requêtes dynamiques sont-elles vraiment nécessaires ?