SGBD : EXPLOITATION D'UNE BASE DE DONNÉES [R206]

TD(TP) $N^0 3 - 4$ - PROGRAMMATION PLPG/SQL

Objectifs

- Programmation en PL/pgSQL
- Gestion des Exceptions

Enoncés

Exercice I : case, array

Question 1.1. *Décrire une fonction* `nboccur` *qui prend en entrée un tableau et une valeur et retourne le nombre d'occurences de la valeur dans le tableau.*

```
CREATE or REPLACE function nboccur
 (  int[],
    int
 )
  returns int as
$$
   -- corps
$$ language plpgsql;
```

Question 1.2. *Décrire une fonction* `tribbule` *qui prend en entrée un tableau et retourne le tableau trié en utilisant la méthode de tri à bulle.*

```
CREATE or REPLACE function tribulle
 ( inout int[]
 ) as
$$
    -- corps
$$ language plpgsql;
```

Le principe du tri à bulle est résumé dans le code suivant pour un tableau `t` *ayant* `n` *éléments :*

Date: March 8, 2022.
Hocine ABIR - IUT Villetaneuse .

```
for (int i=n; i<=2;i--)
{
    for (int j=2;j<=i;j++)
    {
         if (t[j-1] > t[j]) -- ordre decroissant
         {
              temp=t[j];     -- permuter
              t[j]=t[j-1];
              t[j-1]=temp;
         }
    }
}
```

Question 1.3. *Décrire une fonction* `dicocherch` *qui prend en entrée un tableau et une valeur et retour vrai si la valeur est dans le tableau (faux sinon)en utilisant une recherche dichotomique après avoir trier le tableau par la fonction* `tribulle` *(de la question précédente).*

```
CREATE or REPLACE function dicocherch
 ( int[],
   int
 )
  returns bool as
$$
   -- corps
$$ language plpgsql;
```

Le principe de la recherche dichotomique est résumé dans le code suivant pour un tableau `t` *trié ayant* `n` *éléments :*

```
        inf=1;
        sup=n;
        while(1){
             milieu:=(inf+sup)/2;
             if  Tab[milieu]=val
                  break;
             if  (Tab[milieu]>val
                   sup:=milieu-1;
             else
                   inf:=milieu+1;

             if  inf>sup
                   break;
        };
        IF  Tab[milieu]=val
```

```
16              trouve=1;    -- true
17          else
18              trouve=0;     -- false;
```

Exercice II : En mathématiques, la factorielle d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à n, notée $n!$. Soit :

$$n! = 1 \times 2 \times 3 \times ... \times (n-1) \times n \tag{1}$$

ou récursivement:

$$factorielle(n) = 1 \quad si \quad n = 1 \tag{2}$$

$$= n \times fact(n-1) \quad si \quad n > 1 \tag{3}$$

Question 2.1. *En utilisant la définition (2) et (3), compléter le corps de la fonction suivante*

```
1  CREATE FUNCTION fact_rec(int)
2  returns int as
3  $$
4   -- Corps
5  $$ language SQL;
```

Question 2.2. *En utilisant la définition (1), compléter le corps de la fonction suivante par une requête* `WITH` *(CTE) :*

```
1  CREATE or replace FUNCTION fact_ect(int)
2  returns int as
3  $$
4   -- Corps
5  $$ language SQL;
```

Question 2.3. *En utilisant la définition (1), compléter le corps de la fonction suivante par une boucle* `FOR ... LOOP`*:*

```
1  CREATE OR REPLACE FUNCTION fact_pl(
2                      n integer )
3   RETURNS integer AS
4   $$
5      -- corps
6   $$ LANGUAGE plpgsql;
```

Question 2.4. *La fonction factorielle est* `IMMUABLE`*, c'est à dire qu'elle retourne toujours le même résultat pour les mêmes arguments. On propose la table cache suivante :*

```
CREATE TABLE fact_cache (
    num integer PRIMARY KEY,
    fact integer NOT NULL
  );
```

Pour stocker toutes les factorielles déjà calculées. Utiliser le corps de la fonction 2.3 et le compléter pour définir le corps de la fonction suivante:

```
CREATE FUNCTION fact_cache(
                  n integer )
 RETURNS integer AS
 $$
    -- corps
 $$ LANGUAGE plpgsql;
```

Exercice III : Le SIREN (Système Informatique du Répertoire des Entreprises) est un code Insee unique qui sert à identifier une entreprise française. Le numéro SIREN est composé de huit chiffres, plus un chiffre de contrôle qui permet de vérifier la validité du numéro. La clé de contrôle utilisée pour vérifier l'exactitude d'un identifiant est une clé "1-2 ", suivant l'algorithme de Luhn.

Le principe est le suivant : on multiplie les chiffres de rang impair à partir de la droite par 1, ceux de rang pair par 2 ; la somme des chiffres (et non des nombres) obtenus doit être congrue à zéro modulo 10, c'est-à-dire qu'elle doit être multiple de 10.

Exemple :

Code SIREN 732829320									
Position	9	8	7	6	5	4	3	2	1
Code SIREN	7	3	2	8	2	9	3	2	0
produit	7	6	2	16	2	18	3	4	0
Somme	7+	6+	2+	1+6+	2+	1+8+	3+	4+	0=40

Question 3.1. *En utilisant une requête* `WITH` *(ect), compléter le corps de la fonction suivante :*

```
CREATE FUNCTION tab_siren( siren varchar,
                  out i int,out c char)
returns setof record as
$$
 -- corps
$$ language SQL;
```

Pour produire le même résultat que la requête ci-dessous:

```
=> select 10-i as i,substr('732829320',i,1) as c
->           from  generate_series(1,9) as s(i);
 i | c
---+---
 9 | 7
 8 | 3
 7 | 2
 6 | 8
 5 | 2
 4 | 9
 3 | 3
 2 | 2
 1 | 0
(9 rows)
```

Question 3.2. *En utilisant la fonction* `tab_siren` *(de la question précedente), compléter le corps de la fonction suivante* `siren` *pour vérifier si un code siren est correct :*

```
1  CREATE FUNCTION siren( siren varchar)
2  returns boolean as
3  $$
4    -- corps
5  $$ language SQL;
```

Exercice IV : On considère le procédure stockée suivante :

```
1  CREATE FUNCTION except_intro()
2    RETURNS void AS
3  $$
4   DECLARE
5     nom varchar(5);
6   BEGIN
7      nom:='123456';
8   END;
9  $$ LANGUAGE plpgsql;
```

Question 4.1. *Déterminer le code d'erreur généré lors de son exécution.*

Dans la procédure stockée suivante, chaque ligne génère une erreur :

```
create table S
 (
    x int primary key,
    y int
 );

create table T
 (
    j int primary key,
    k int references S(x)
 );

insert into S values(1,10),(2,10);
insert into T values(2,2);

CREATE OR REPLACE FUNCTION caplante()
  RETURNS void AS
$$
 DECLARE
   i int;
   ligne int;
 BEGIN
       SELECT x INTO strict i FROM S WHERE y=10;
       SELECT x INTO strict i FROM S WHERE y=20;
       INSERT INTO s values(1,20);
       INSERT INTO t values(1,3);
       DELETE FROM S where y=10;
       UPDATE S set i=2;
       UPDATE S set x=2;
       UPDATE S set x=x-1;
       UPDATE T set k=3;
       insert into x values(2,3);
 END;
$$ LANGUAGE plpgsql;
```

Question 4.2. *Compléter cette procédure pour afficher chacune de ces erreurs et la ligne correpondante.*

Exercice V : On considère la procédure stockée suivante qui renvoie le nombre de tuples d'une table dont le nom est transmis en argument ou -1 si la table n'existe pas.

```
create function RowCount_Select(varchar)
 returns int as $$
 DECLARE
```

```
4     res int;
5  BEGIN
6     select into res  reltuples
7      from pg_class
8         where relkind='r'
9         and relname=$1;
10     if not found then
11       return -1;
12    end if;
13    return res;
14 END;
15 $$ language plpgsql strict;
```

Question 5.1. *Modifier cette procédure de sorte qu'elle génère une exception avec les données suivante :*

```
=> select RowCount_Select('x');
ERROR:  Table "x" Inexistante :
HINT:  Ca merde
```

Question 5.2. *Compléter le code ci-dessus pour intercepter l'exception et obtenir le résultat suivant :*

```
=> select RowCount_Select('x');
INFO:  test # Table "x" Inexistante :
 rowcount_select
-----------------
              -1
(1 row)
```

Exercice VI : On souhaite insérer ou modifier un tuple de la table `etudiant`:

```
1  CREATE TABLE etudiant
2  (
3      et_numero    int primary key,
4      et_nom       varchar,
5      et_prenom    varchar
6  );
```

en une seule commande et sans echec !! Si le tuple existe déjà alors le tuple est mis à jour sinon le tuple est inséré. Cela revient à fusionner les commandes `UPDATE` et `INSERT`, comme suit :

```
1   UPDATE etudiant
2   SET et_nom = in_nom, et_prenom=in_penom
3     WHERE et_numero = in_id;
4   -- moment critique !!
```

```
5  IF NOT FOUND THEN
6     INSERT INTO etudiant(et_numero,et_nom,et_prenom)
7          VALUES (in_id, in_nom, in_prenom);
8  END IF;
```

Sauf que ce bout de code est une section critique c'est à dire qu'il ne doit pas y avoir d'accès à la table etudiant entre les deux commandes : si une autre commande insère le même tuple entre les deux commandes alors elles echouent toutes les deux.

Pour garantir qu'il n'y aura pas d'echec des commandes `INSERT/UPDATE`, décriver une fonction `PLpgSQL` :

```
1       CREATE FUNCTION merge_etudiant(INT, TEXT, TEXT)
2       RETURNS VOID AS
3  $$
4      --corps
5
6  $$ LANGUAGE plpgsql;
```

qui intercepte l'éxception générée par un echec (s'il a lieu) et réitère l'opération jusqu'au succès comme dans l'exemple:

```
# select * from etudiant;
 et_numero |     et_nom     | et_prenom
-----------+----------------+------------
         1 | ABC VTTD YUUUF | Todto Titi
         2 |                | Todto Titi
(2 rows)

# select merge_etudiant(1,'alTinik','nEJDET');
# select merge_etudiant(2,'AZABGA','antnoy');
# select merge_etudiant(2,'AZAGBA','antony');
# select merge_etudiant(3,'YAPO','Franck');

# select * from etudiant;
 et_numero | et_nom  | et_prenom
-----------+---------+-----------
         1 | ALTINIK | Nejdet
         2 | AZAGBA  | Antony
         3 | YAPO    | Franck
(3 rows)
```