Compte rendu – PostgrlSQL – SLAM3.5

Introduction

Objectif : L'objectif de ce TD est d'utiliser PostgrlSQL et de comprendre le formalisme du code.

Résolution : Nous réalisons les exercices qui nous sont proposés.

Conception de la solution

Exercice 1 : Écrire la fonction *calculer_longueur_max()* qui calcule la longueur de deux chaînes de caractères fournies en argument et qui retourne la longueur la plus longue.

Conception :

```
CREATE FUNCTION  calculer_longueur_max(var1 TEXT, var2 TEXT) RETURNS TEXT
LANGUAGE plpgsql
AS $plpgsql$
DECLARE
i1 INT;
i2 INT;
x TEXT;

BEGIN
  i1 = CHAR_LENGTH(var1);
  i2 = CHAR_LENGTH(var2);

  IF i2>i1 THEN
    x='i2 est long';
ELSE
    x='i1 est long';
END IF;

  RETURN x;
END
$plpgsql$;


SELECT calcul('DUPONT','DELAVAL');
```

Résultat : On obtient alors comme résultat en comparant ‘DUPONT’ et ‘DELAVAL’ que *i2 est long*.

*Remarque : la méthode **CHAR_LENGTH(**variable**)** permet de calculer la longueur d’une chaîne.*

Exercice 2 : Écrire la fonction *nb_occurrences()* qui compte et retourne le nombre d'occurrences d'un caractère dans un intervalle d'une chaîne de caractères. L'intervalle est indiqué par une position de départ et de fin dans la chaîne.

Conception :

```
CREATE OR REPLACE FUNCTION nb_occurrences3(c CHAR, chaine VARCHAR, i1 INT, i2 INT) RETURNS CHAR
LANGUAGE plpgsql
AS $plpgsql$
DECLARE
  rest VARCHAR;
  comp INT;
  i INT;
  list CHAR[];
BEGIN
  comp = 0;
  list = STRING_TO_ARRAY(chaine,NULL);
  FOR i IN i1..i2 LOOP
    IF list[i] = c THEN
      comp = comp+1;
      END IF;
  END LOOP;

  RETURN comp;
END;
$plpgsql$;



SELECT nb_occurrences3('i','voici un test', 2, 6);
```

Résultat : Le morceau de chaine analysé est rogné ainsi : « *ici u* ». La fonction retourne alors 2 pour le nombre de *i* contenus dans la chaîne.

Remarque : la méthode ***STRING_TO_ARRAY(****variable, NULL****)*** permet de stocker les caractères d'une chaine dans un tableau.

Exercice 3 : Écrire la fonction *getNbJoursParMois()* qui calcule le nombre de jours dans un mois pour une date fournie en paramètre.

Conception :

```
CREATE OR REPLACE FUNCTION  getNbJoursParMois3(datee date) RETURNS DATE AS
$$
  SELECT (date_trunc('MONTH', $1) + INTERVAL '1 MONTH - 1 day')::DATE;

$$
LANGUAGE 'sql';



SELECT date_part('day',getNbJoursParMois3('2020-01-01')) ;
```

Résultat : On obtient pour la date entrée en paramètre *31* car le mois de janvier contient 31 jours.

Remarque : la méthode ***date_part(****'day', function(parmètres)****)*** permet d'extraire un élément d'une date. Ici on extrait le jour mais on aurait aussi pu extraire le mois (*month*) ou l'année (*year*).

Exercice 4 : Écrire la fonction *dateSqlToDatefr()* qui vous permet de convertir la date fournie en paramètre au format SQL en une date au format JJ/MM/AA.

Conception :

```
CREATE OR REPLACE FUNCTION  datesqltodateJour(datee date) RETURNS DATE
LANGUAGE plpgsql
AS $plpgsql$
  BEGIN
    RETURN TO_CHAR(datee, 'DD/MON/YYYY') ;
END;
$plpgsql$;



SELECT datesqltodateJour('2020-01-01') ;
```

Résultat : La fonction retourne *01/01/2020* au lieu de *2020-01-01*.

Remarque : la méthode ***TO_CHAR(****variable, 'format'****)*** permet de changer le format de la chaine de caractères et donc de la date ici.

Exercice 5 : Écrire la fonction *getNomJour()* qui vous permet de retourner le nom du jour de la semaine correspondant à la date fournie en paramètre.

Conception :

```
CREATE OR REPLACE FUNCTION getnomjour (a DATE) RETURNS VARCHAR
LANGUAGE PLPGSQL
AS $plpgsql$
DECLARE
  b INT;
  c TEXT [];
BEGIN
  b = EXTRACT(ISODOW FROM a);
  c = '{"Lundi","Mardi","Mercredi","Jeudi","Vendredi","Samedi","Dimanche"}';
  RETURN c[b];
END;
$plpgsql$;

SELECT getnomjour('2001-08-07') ;
```

Résultat : La fonction retourne le nom du jour correspondant. Pour le 07 août 2001 on obtient *mardi*.

Remarque : la méthode ***EXTRACT(ISODOW FROM** variable**)*** permet d'extraire un jour de la semaine du lundi (1) au dimanche (7). Les jours sont stockés dans un tableau.

Exercice 6 : Écrire une fonction qui retourne le nombre de clients débiteurs. Écrire une fonction qui retourne le nombre de clients habitant dans une ville. Le nom de la ville est un paramètre de la fonction.

Conception :

```
CREATE or REPLACE FUNCTION compteUser5() RETURNS INT
LANGUAGE plpgsql
AS $plpgsql$
BEGIN
  RETURN DISTINCT COUNT( DISTINCT client.num_client) FROM client, operation, compte, posseder WHERE
  operation.type_operation='DEBIT' AND client.num_client=posseder.num_client AND posseder.num_compte=compte.num_compte AND
  compte.num_compte=operation.num_compte;
END;
$plpgsql$;


SELECT compteUser5() ;
```

```
create or replace function countclient137(blbl character varying) returns integer
language plpgsql as $plpgsql$
BEGIN
  RETURN DISTINCT COUNT(DISTINCT client.num_client)
  FROM client
  WHERE client.adresse_client LIKE ('%' || blbl || '%');
END;
$plpgsql$;
```

Résultat : Grâce à une requête SQL on lie les différentes tables de la bdd et on obtient le compte des clients débiteurs, ici : *3* (*DELAVAL, HANOT, LEVY*). La seconde fonction retourne le nombre de client habitant dans une ville donnée.

Remarque : On aurait pu directement effectuer le compte des comptes associés à un débit mais nous avons choisi de compatibiliser le nom de clients par anticipation. On peut ainsi afficher le nom des clients.

Remarque : la méthode ***COUNT(DISTINCT** champ **FROM** table**)*** permet d'éviter les doublons. Par exemple si Monsieur Delaval est débiteur à deux reprises, il n'est compatabilisé qu'une seule fois.

Remarque : la condition ***LIKE ('%' || variable || '%')*** cherche dans les adresses des clients cette chaine de caractères. En effet, il n'y a qu'un seul champ qui contient la rue, la ville et le cp. Il faut donc faire une recherche dans chaque champ.

Exercice 7 : Écrire une fonction qui permet d'insérer un tuple dans la table client. Les valeurs des colonnes seront fournies en argument à la fonction sauf pour le numéro du client. Le numéro du client devra être calculé dans la fonction. La fonction récupère le dernier numéro et l'incrémente pour créer le nouveau numéro du client à enregistrer. Une information devra indiquer si le tuple a été bien enregistré ou pas.

Conception :

```
CREATE or REPLACE FUNCTION createUser(nom varchar, prenom VARCHAR,adresse VARCHAR, idinternet VARCHAR, mdpinternet varchar)
RETURNS INT
LANGUAGE plpgsql
AS $plpgsql$
  DECLARE
   var INT;
var2 INT;
BEGIN

  var = MAX(client.num_client) FROM client;
  var2 = var+1;
  INSERT INTO client (num_client, nom_client, prenom_client, adresse_client, identifiant_internet, mdp_internet) VALUES
  (var2, nom, prenom, adresse, idinternet, mdpinternet);
  RETURN var2;
END;
$plpgsql$;


SELECT createUser('Test', 'test','test','test', 'test') ;
```

Résultat : Le client est créé avec les informations entrées en paramètres. Pour vérifier sa création, on fait afficher l'identifiant maximum parmi les clients qui correspond au dernier client créé.

Remarque : Afin d'auto-incrémenter l'identifiant lors de sa création, on choisit de récupérer le dernier identifiant contenu dans la table client et on l'incrémente de 1 à chaque création.

Ressenti personnel

La syntaxe était vraiment difficile à comprendre au début mais une fois le système compris, les exercices sont rapidement réalisés.