UNIVERSITÉ de la NOUVELLE-CALÉDONIE

Chapitre 1: Manipuler les données

Programmation procédurale

Frédéric Flouvat

(dérivé du cours du Pr. Jeffrey Ullman, Stanford University
et du cours du Pr. Christian Retoré, Université de Bordeaux)

Université de la Nouvelle-Calédonie

frederic.flouvat@univ-nc.nc

Le SQL dans de vrais programmes

- Pour l'instant, le SQL est utilisé comme interface générique pour faire des requêtes
  - un environnement où l'utilisateur est face à un terminal et fait des requêtes à une base de données.
- La réalité est différente: des programmes conventionnels interagissent avec le SQL.
- Les options possibles:
  - le code dans un langage spécifique est stocké dans la base de données (PSM, PL/SQL).
  - les instructions SQL sont englobées dans un langage hôte (p.ex. le C) et transformées/exécutées en ligne par un préprocesseur (Embedded SQL).
  - des outils de connections sont utilisés pour permettre à un langage conventionnel d'accéder à la base de données (p.ex. CLI, JDBC, PHP)

Les Procédures Stockées

- PSM, ou “persistent stored modules,” permet de stocker des procédures comme des éléments du schéma de la base de données.
  - SQL/PSM un standard international (ISO)

- PSM = un mélange d’instructions conventionnelles (if, while, etc.) et de SQL.

- Avantages:
  - Améliore la sécurité (via une couche intermédiaire)
  - Améliore l’intégrité des données (des modifications cohérentes)
  - Améliore la conception (plus de modularité du code)
  - Améliore les performances (instructions préparées une seule fois)
  - Limite les communications client/serveur

- Attention: en pratique, des implémentations légèrement différentes en fonction des éditeurs
  - p.ex. PL/SQL dans Oracle, **PL/pgSQL dans PostgreSQL**

Structure de base d'une procedure PL/pgSQL

```
CREATE [OR REPLACE] FUNCTION <name> (<arguments>) RETURNS <type> AS $$
DECLARE
    <optional declarations>
BEGIN
    <PL/SQL statements>
END;
$$ LANGUAGE plpgsql;
```

instructions obligatoires

- Les paramètres/arguments:
  - syntaxe **nom mode type** (différent de PSM), séparés par des virgules, avec mode égal à :
    - IN = variable en entrée, non modifiée par la procédure.
    - OUT = variable retournée en sortie de l'algorithme.
    - INOUT = les deux.
- Type de retour (RETURNS): mettre VOID pour une procédure
- Déclaration des variables locales (DECLARE): optionnelle
- Corps de la procédure (BEGIN … END;): séparer les instructions par des points virgules

Exemple de procédure sockée PL/pgSQL

- Ecrivons une procédure avec deux paramètres b et p, et qui ajoute un tuple ayant bar = 'Joe''s Bar', beer = b, et price = p, à la relation Sells(bar, beer, price).
  - utilisée par Joe pour ajouter des bières à sa carte plus facilement.

```
CREATE  FUNCTION JoeMenu( b IN VARCHAR(10), p IN INTEGER )
 RETURNS VOID AS $$
BEGIN
        INSERT INTO Sells
        VALUES ('Joe''s Bar', b, p);
END;
$$ LANGUAGE plpgsql;
```

les paramètres sont tous les deux en lecture seule, i.e. non modifiés

le corps ---
une seule insertion

Invoquer des Procédures PL/pgSQL

SELECT * FROM <name>(<arguments>) ;

ou SELECT <name>(<arguments>) ;

- Exemple:

  SELECT JoeMenu('Moosedrool', 5.00);

- Les fonctions peuvent être utilisées dans des expressions SQL, à condition que le type de la valeur retournée soit approprié.
  - Attention au type de retour RECORD qui n'a pas de structure prédéfinie et qui peut donc nécessiter un transtypage (*cast*) pour pouvoir être utilisé dans une autre expression SQL

PL/pgSQL Déclarations et Affectations

- Déclarer une variable/un paramètre: <name> <type>
  - Les types SQL.
  - De nouveaux types:
    - p.ex. type composite

```
CREATE TYPE element_inventaire AS (
        nom                 text,
        id_fournisseur      integer,
        prix                numeric
);
```

- Affecter une valeur: <variable> := <expression>;
  - Exemple: b := 'Bud';

Les variables de type attribut et variables de type tuple

- PL/pgSQL permet à une variable d'avoir le même type qu'un attribut d'une relation ou la même structure qu'un tuple de ses tuples.
  - Attention: ne fonctionne pas avec les types composites (CREATE TYPE)

- x R.a%TYPE donne a x le même type que l'attribut a de R.
- x R%ROWTYPE donne à x le type des tuples de R.
  - x.a donne la valeur de l'attribut a du tuple x.

- Exemple: Reprendre JoeMenu(b,p) en utilisant Sells.beer et Sells.price.

```
CREATE  FUNCTION JoeMenu( b IN "Sells".beer%TYPE, p IN "Sells".price%TYPE)
 RETURNS VOID AS $$
BEGIN
                    INSERT INTO Sells
                    VALUES (' Joe' ' s Bar' , b, p);
END;
$$ LANGUAGE plpgsql;
```

Instructions conditionnelles IF et CASE

- Les IF
  - IF <condition> THEN
    <statement(s)>
    END IF;
  - IF . . . THEN . . . ELSE . . . END IF;
  - IF … THEN … ELSIF … … THEN … ELSE … END IF;
- Les CASE
  - CASE <*search-expression*>
    WHEN <*expression*> *[, <expression> [...]]* THEN <statement(s)>
    WHEN … THEN …
    ELSE ...
    END CASE;
  - CASE
    WHEN <*boolean-expression*> THEN <statement(s)>
    ...
    END CASE;

Exemple: IF

- Classer les bars en function de leur nombre de clients en utilisant la relation Frequents(drinker,bar).
  - <100 clients: ‘unpopular’.
  - 100-199 clients: ‘average’.
  - >= 200 clients: ‘popular’.

- La fonction Rate(b) classe le bar b.

```
CREATE FUNCTION Rate( b IN CHAR(20) ) RETURNS CHAR(10) AS $$
  DECLARE
    cust INTEGER;
  BEGIN
    SELECT COUNT(*) INTO cust FROM "Frequents" WHERE bar = b;
    IF cust < 100 THEN RETURN ’ unpopular’ ;
    ELSIF cust < 200 THEN RETURN ’ average’ ;
    ELSE RETURN ’ popular’ ;
    END IF;
    RETURN;
  END;
$$ LANGUAGE plpgsql;
```

nombre de clients du bar b

IF imbriqués

retourne la valeur ici et non lors des autres RETURN

Les boucles

- Forme basique:

  [<loop label>] LOOP <statements>

  END LOOP [<loop name>];

- Interrompre une boucle:

  EXIT [<loop label>] [WHEN <condition>]

- Exemple:

```
loop1 LOOP
        . . .
        EXIT loop1;   <-- si cette instruction est exécutée
        . . .
END LOOP loop1;       <-- fin de la boucle
```

Les boucles

- [<while label>] WHILE <condition> LOOP
  <statements>
  END LOOP [<while label>] ;

- [<for label>] FOR <variable> IN [REVERSE] <expression> .. <expression> [BY <expression>]
  LOOP
  <statements>
  END LOOP [<for label>];

Exemple:

```
FOR i IN 1 .. 10 BY 2 LOOP
…
END LOOP;
```

Exécuter des requêtes dans une procédure

- Les requêtes générales SELECT-FROM-WHERE ne peuvent pas être exécutées directement.
- Seules les requêtes ne retournant aucun résultat peuvent être directement invoquées (p.ex. INSERT).
- Il y a trois principales méthodes pour manipuler le résultat d'une requête SELECT:
  - SELECT . . . INTO…
  - Parcourir le résultat de la requête dans une boucle FOR.
  - Les curseurs.

SELECT . . . INTO

- Un moyen de récupérer la valeur d'une requête qui retourne **un unique** tuple est d'utiliser INTO <variable> après la clause SELECT.

- Exemple:

```
SELECT price INTO p FROM Sells
        WHERE bar = 'Joe''s Bar' AND
        beer = 'Bud';
```

FOR. . . IN

- Un moyen de récupérer la valeur d'une requête qui retourne **un ensemble** de tuples est d'utiliser une boucle FOR … IN…

- Exemple:

```
CREATE OR REPLACE FUNCTION JoeGouge() RETURNS VOID AS $$
DECLARE
        r RECORD;
BEGIN
        FOR r IN SELECT beer, price FROM "Sells" WHERE bar='Joe"s Bar' LOOP
                IF r.price < 3.00 THEN
                        UPDATE "Sells" SET price = r.price+ 1.00
                                WHERE bar = 'Joe''s Bar' AND beer = r.beer;
                END IF;
        END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Les curseurs

- Un curseur est principalement une variable de type tuple qui stocke tous les tuples résultats d'une requête.

- Déclaration d'un curseur en PL/pgSQL :

  DECLARE <name> CURSOR [(<arguments>)] FOR <query>;

  Exemple:

  DECLARE c CURSOR (**b** varchar) FOR SELECT * FROM Sells WHERE beer=**b**;

- Utilisation d'un curseur :
  - initialisation du curseur: OPEN <name>[(<arguments>)] ;
    - la requête est évaluée, et le curseur pointe sur le premier tuple du résultat.
  - libération du curseur: CLOSE <name>;

Parcourir les tuples d'un Curseur

- Pour accéder au tuple suivant du curseur c, utiliser la commande:

  FETCH c INTO x1, x2,…,xn ;

  - les x sont des variables référençant chaque composant du tuple stocké dans c.
  - c est déplacé automatiquement au prochain tuple.

- Sortir d'une boucle d'un curseur
  - l'utilisation classique d'un curseur est de créer une boucle avec FETCH, et de faire un traitement pour chaque tuple parcouru.
  - mais comment sortir de la boucle quand tous les tuples ont été traités?

  ➢ EXIT WHEN NOT FOUND

Exemple: Curseur en PL/pgSQL (déclaration)

- La procédure JoeGouge() utilise un curseur pour parcourir les ventes de Joe's-Bar dans Sells(bar, beer, price), et augmente de $1 le prix des bières initialement vendue en dessous de $3 chez Joe's Bar.

Utilisé pour stocker les pairs beer-price lors du parcours du curseur

```
CREATE FUNCTION JoeGouge() RETURNS VOID AS $$
DECLARE
  theBeer "Sells".beer%TYPE;
  thePrice "Sells".price%TYPE;
  c CURSOR FOR
      SELECT beer, price FROM "Sells"
      WHERE bar = 'Joe''s Bar';
```

Retourne le menu de Joe

Exemple: Curseur en PL/pgSQL (corps de la procédure)

```
BEGIN
    OPEN c;
    LOOP
      FETCH c INTO theBeer, thePrice;
            EXIT WHEN NOT FOUND;
      IF thePrice < 3.00 THEN
            UPDATE "Sells" SET price = thePrice + 1.00
              WHERE bar = 'Joe''s Bar' AND beer = theBeer;
      END IF;
    END LOOP;
    CLOSE c;
END;
$$ LANGUAGE plpgsql;
```

condition d'arrêt de la boucle

Si Joe vend une bière moins $3, augmenter son prix de $1 à Joe's Bar.

Retourner un ensemble de valeurs (Déclaration)

- Deux options principalement pour **déclarer** le type de retour de la fonction:
  - Déclarer … RETURNS SETOF <sometype> …
  - Déclarer … TABLE(<col1> <type1>, <col2> <type2> …) …

- **Déclaration**: RETURNS SETOF
  - possibilité de retourner un ensemble de valeurs simples
    - p.ex. RETURNS SETOF integer
  - possibilité de retourner un ensemble de tuples en créant un nouveau type composite (ou en faisant référence à un type composite existant)
    - p.ex.

```
CREATE TYPE nomprenom AS ( nom VARCHAR(20), prenom
   VARCHAR(30) ) ;

CREATE FUNCTION clients() RETURNS SETOF nomprenom AS $$
…
```

Retourner un ensemble de valeurs (Déclaration)

- **Déclaration**: TABLE(<col1> <type1>, <col2> <type2> …)
  - syntaxe proche de la norme SQL/PSM
  - facilité d'interprétation
  - possibilité de retourner un ensemble de valeurs simples ou un ensemble de tuples
  - mais interdiction d'utiliser des paramètres en sortie (mode OUT)

    – p.ex.

```
CREATE FUNCTION clients() RETURNS TABLE( nom VARCHAR(20),
   prenom VARCHAR(30) ) AS $$
…
```

Retourner un ensemble de valeurs (Invocation)

- Deux options pour **retourner** des résultats dans le corps de la fonction:
  - RETURN QUERY <SQLquery> ;
  - RETURN NEXT [<expression>] ;

- **Retourner** : RETURN QUERY
  - compatible RETURNS SETOF et RETURNS TABLE
  - retourne le résultat d'une requête
  - possibilité de l'utiliser plusieurs fois
    - dans ce cas, chaque ensemble est ajouté à la suite de l'autre
  - résultat final uniquement retourné à la fin de la procédure ou au moment de l'appel RETURN;

Retourner un ensemble de valeurs (Invocation)

- Exemple: RETURNS TABLE et RETURN QUERY

```
CREATE OR REPLACE FUNCTION BarSallyJo ()
    RETURNS TABLE( bar TEXT, name VARCHAR(30) ) AS $$
BEGIN
    …
    RETURN QUERY SELECT bar, drinker FROM "Frequents" WHERE drinker = 'Sally';
    …
    RETURN QUERY SELECT bar, drinker FROM "Frequents" WHERE drinker = 'Jo';
    …
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

Retourner un ensemble de valeurs (Invocation)

- **Retourner** : RETURN NEXT
  - enregistre une valeur/tuple à la fois dans l'ensemble des solutions
  - généralement invoqué dans une boucle
  - son utilisation dépend du type de retour de la fonction:

    Si RETURNS SETOF <sometype> alors RETURN NEXT <expression>;
    - <expression> est classiquement une variable du type associé au SETOF

    Si RETURNS TABLE(<col1> <type1>, <col2> <type2> …) alors RETURN NEXT;
    - <col1>, <col2> … sont utilisées et initialisées comme des variables dans le corps de la procédure
    - chaque RETURN NEXT; enregistre dans l'ensemble des résultats un tuple initialisé à partir des valeurs en cours pour <col1>, <col2> …
  - résultat final uniquement retourné à la fin de la procédure ou au moment de l'appel RETURN;

Retourner un ensemble de valeurs (Invocation)

- Exemple: RETURNS TABLE et RETURN NEXT;

```
CREATE OR REPLACE FUNCTION BarSallyJo ()
    RETURNS TABLE( bar TEXT, name VARCHAR(30) ) AS $$
BEGIN
        FOR bar, name IN
            SELECT bar, drinker FROM "Frequents" WHERE drinker = 'Sally'
        LOOP
                RETURN NEXT;
        END LOOP;
        FOR bar, name IN
           SELECT bar, drinker FROM "Frequents" WHERE drinker = 'Jo'
        LOOP
                RETURN NEXT ;
        END LOOP;
        RETURN;
END;
$$ LANGUAGE plpgsql;
```

Retourner un ensemble de valeurs (Invocation)

- Exemple: RETURNS SETOF et RETURN NEXT;

```
CREATE TYPE infobar AS ( bar TEXT, name VARCHAR(30) );

CREATE OR REPLACE FUNCTION BarSallyJo ()  RETURNS SETOF infobar AS $$
DECLARE
    r infobar;
BEGIN
        FOR r IN
            SELECT bar, drinker FROM "Frequents" WHERE drinker = 'Sally'
        LOOP
                RETURN NEXT r ;
        END LOOP;
        …
        RETURN;
END;
$$ LANGUAGE plpgsql;
```