Cinquième Partie

PROGRAMMATION PROCEDURALE

Programmation Procédurale
Sommaire

- Introduction
- PL/SQL
  - Structures de contrôle
  - Intégration des opérations SQL
  - Opérations simples
  - Requêtes et curseurs
  - Procédures stockées
  - Gestion des erreurs
  - Autres possibilités
- Triggers
  - Définition
  - Triggers de ligne
  - Triggers INSTEAD OF
- Conclusion
- Portage PL/SQL vers PL/PGSQL

Programmation Procédurale
Introduction

- Problèmes du modèle SGBD/SQL <-> client
  - faiblesse des contraintes exprimables en SQL
  - logique du client alourdie
  - dispersion du code, répétition
  - SQL est un langage non procédural

  donc : fragilité

- Les langages procéduraux permettent de placer du code (quelconque) dans la base de données, pour :
  - assurer la cohérence avec le modèle de données
  - simplifier/abstraire l'interface d'accès
  - alléger le client
  - enrichir le modèle SQL simple

- C'est le SGBD qui exécute le code !

Programmation Procédurale
Introduction (suite)

- A priori, n'importe quel langage généraliste convient

- Cependant, un langage spécialisé a beaucoup d'avantages
  - cohérence des types de données
  - Echange de variable entre les code SQL
  - Utilisation de fonctions/procédures
  - gestion des erreurs homogènes
  - vérification des requêtes/opérations imbriquées
  - optimisation possible
  - multi-clients

- Exemple :
  - PL/SQL (Oracle)
  - TransactSQL (SQL Server etc.)
  - PL/pgSQL (PostgreSQL)

Programmation Procédurale
Introduction (suite)

- Nous utiliserons l'exemple de PL/SQL dans Oracle

- PL/SQL intègre SQL
  - opérations élémentaires
  - requêtes
  - SQL DDL (mais peu adapté)

- C'est un langage fortement typé (inspiré de Ada)

- Fortement structuré (packages)
  - Muni d'un compilateur
  - vérification du code SQL au moment de la création
  - totalement intégré au SGBD
  - produit (éventuellement) du code natif (pas de code interprété)

Programmation Procédurale
PL/SQL

- La structure de base de PL/SQL est le bloc :

```
[DECLARE
  declarations]
BEGIN
  instructions
[EXCEPTION
  gestionnaires]
END
```

- Découpage : déclaration de variables, corps, gestionnaires d'exception

- Les blocs peuvent être imbriqués

Programmation Procédurale
PL/SQL (suite)

- Les déclarations de variable sont de la forme :

  `nom type ;`

  où `type` est n'importe quel type SQL Oracle (plus `BOOLEAN` et `CURSOR`)

- On peut utiliser les types des éléments du schéma :

  `table.colonne%TYPE` : type d'une colonne

  `table%ROWTYPE` : type record, ligne de table

  -> garantit la cohérence des types

  Les attributs %TYPE et %ROWTYPE permettent de déclarer des variables similaires a des colonnes de la base de données sans connaitre le type de la colonne.

PL/SQL
Structures de contrôle

- PL/SQL propose des structures de contrôle classiques :

```
IF condition THEN
        instructions
[ELSIF condition
        instructions]
[ELSE
        instructions]
END IF;
```

```
LOOP
...
EXIT [WHEN condition]
...
END LOOP ;
```

```
WHILE condition LOOP
instructions
END LOOP;
```

```
FOR var IN low .. high
LOOP
instructions
END LOOP;
```

PL/SQL
Intégration des opérations SQL

- Les instructions SQL de base s'écrivent littéralement, et peuvent utiliser les variables PL/SQL déclarées

```
DECLARE
   nomemp employe.nom%TYPE ;
   prenomemp employe.prenom%TYPE ;
BEGIN
   nomemp := 'Jean' ;
   UPDATE employe SET salaire=salaire*2
     WHERE nom LIKE ('%' || nomemp || '%') ;
   prenomep := 'Robert' ;
   DELETE FROM employe
     WHERE prenom = prenomemp ;
   END ;
```

- La pseudo-variable SQL permet de connaître l'effet de la dernière instruction SQL :
  - `SQL%FOUND` : l'instruction a affecté au moins une ligne if found ou if not found
  - `SQL%ROWCOUNT` : nombre de lignes affectées GET DIAGNOSTICS my_var = ROW_COUNT;

PL/SQL
Opérations simples

- Pour le cas des instructions affectant une seule ligne, différentes colonnes peuvent être récupérées au vol :

```
UPDATE employe SET salaire = salaire*1.5
  WHERE id = 42
  RETURNING nom, prenom INTO nomemp, prenomemp;

INSERT INTO journal(op,desc)
  VALUES ('Augmentation',(nomemp||' '||prenomemp));
```

- C'est vrai également pour les requêtes renvoyant une seule ligne :

```
SELECT nom, prenom FROM employe
  INTO nomemp, prenomemp
  WHERE id = 42;
```

PL/SQL
Requêtes et curseurs

- Mais les curseurs donnent également la possibilité de parcourir les lignes progressivement :

```
DECLARE
   n employe.nom%TYPE ;
   p employe.prenom%TYPE ;
   CURSOR c IS SELECT nom,prenom
            FROM employe WHERE salaire > 2000 ;
   BEGIN
    OPEN c ;
    LOOP
            FETCH c INTO n,p ;
            EXIT WHEN c%NOTFOUND ;
            email(n,p,17) ;
    END LOOP ;
   CLOSE c ;
END;
```

- Les curseurs sont des dispositifs incrémentaux (syntaxe différente en postgres)

PL/SQL
Requêtes et curseurs (suite)

- Il y a une syntaxe particulière de FOR pour les requêtes à résultat multiple (un record à la fois) :

```
FOR e IN (SELECT nom,prenom FROM
  employe WHERE salaire > 2000)
LOOP
  email(e.nom,e.prenom,17) ;
END LOOP
```

- Ou l'équivalent, en utilisant un curseur déclaré

```
DECLARE
  CURSOR c IS SELECT nom,prenom
          FROM employe WHERE salaire > 2000 ;
BEGIN
  FOR e IN c LOOP
   email(e.nom,e.prenom,17) ;
  END LOOP ;
END ;
```

- On peut également passer les curseurs en paramètres, etc.

PL/SQL
Requêtes et curseurs (suite)

- On peut paramétrer un curseur :

```
DECLARE
  CURSOR c(max NUMBER) IS SELECT nom,prenom
   FROM employe WHERE salaire > max ;
BEGIN
  FOR e IN c(2000) LOOP
   email(e.nom,e.prenom,17) ;
  END LOOP ;
END ;
```

- Les instructions et requêtes sont compilées, donc :
  - vérification syntaxique et sémantique
  - éventuellement, préparation des plans d'exécution

  mais aussi
  - les blocs dépendent des structures qu'ils utilisent

PL/SQL
Procédures stockées

- Un bloc PL/SQL peut être muni d'un nom, de paramètres, et être stocké dans la base de données (pas de procédure sous postgres):

```
CREATE [OR REPLACE] PROCEDURE nom(params)
  declarations
BEGIN
  instructions
[EXCEPTION
  gestionnaires]
END nom ;
```

- Une fonction est similaire mais renvoie une valeur :

```
CREATE [OR REPLACE] FUNCTION nom(params)
  RETURN type AS
  ...
```

Le code doit contenir : `RETURN expression`

- Les fonctions PL/SQL peuvent être utilisées dans les clauses `WHERE` des requêtes (`SELECT, UPDATE, DELETE`)

PL/SQL
Procédures stockées (suite)

- Les procédures/fonctions stockées peuvent être regroupées en packages. Un package est décrit par deux éléments : la déclaration et la définition

```
CREATE PACKAGE biblio AS
   PROCEDURE emprunter(lect lecteur.id%TYPE,liv
                       livre.id%TYPE);
END biblio ;
...
CREATE PACKAGE BODY biblio AS
   PROCEDURE emprunter(lect lecteur.id%TYPE,liv
                       livre.id%TYPE)
   BEGIN
   ...
   END emprunter ;
END biblio ;
```

- Ici la procédure s'appelle `biblio.emprunter`

Pas de package en postgres

PL/SQL
Procédures stockées (suite)

- Exemple : un lecteur ne peut emprunter que 5 livres simultanément

```
PROCEDURE emprunter(lect lecteur.id%TYPE,
                    liv livre.id%TYPE)
BEGIN
  SELECT COUNT(*) INTO N FROM emprunt
         WHERE idlecteur = lect ;
  IF n >= 5 THEN
   RAISE_APPLICATION_ERROR(-20109,'Trop
  d''emprunts') ;
  ELSE
   INSERT INTO emprunt VALUES(lect,liv,...) ;
  END IF ;
END emprunter ;
```

La procédure teste les règles de gestion, et INSERT les contraintes d'intégrité référentielle

PL/SQL
Procédures stockées (suite)

- Les avantages des procédures cataloguées incluent:
  - une maintenance plus aisée(centralisée dans la base de données, 1 seul endroit),
  - des applications plus petites.
- Une exécution
  - plus rapide (précompilé)
  - et de plus grandes économies de mémoire.
- Une utilisation des procédures depuis de nombreuses applications Oracle différentes
  - pro*c,
  - forms,
  - trigger, autre procédure,…
- Les procédures/fonctions stockées sont des objets du schéma : un utilisateur peut recevoir (ou pas) le privilège de les invoquer

PL/SQL
Gestion des erreurs

- La gestion des erreurs est basée sur un mécanisme d'exception : il faut distinguer
  - une erreur : problème dans l'exécution du code
  - une exception : un objet PL/SQL représentant une situation exceptionnelle

  (une exception ne représente pas forcément une erreur)

- Une exception logique est :
  - déclarée comme une variable :

    ```
    interdit EXCEPTION ;
    ```
  - est déclenchée explicitement :

    ```
    RAISE interdit ;
    ```
  - peut être interceptée par un gestionnaire d'exception :

    ```
    WHEN interdit THEN
    ...
    ```

PL/SQL
Gestion des erreurs (suite)

- Le mécanisme est similaire aux exceptions des langages “modernes” (remontée dans la pile, jusqu’à un gestionnaire)

- Certaines exceptions correspondant à des erreurs SQL sont prédéfinies :

  - `ZERO_DIVIDE (DIVISION_BY_ZERO)`
  - `NO_DATA_FOUND (SELECT INTO ... ne renvoie rien)`
  - `TOO_MANY_ROWS (SELECT INTO ... Renvoie plusieurs lignes)`
  - `...`

  (Ce sont les instructions SQL qui déclenchent ces exceptions)

Remplacement des sorties de curseur EXIT WHEN ... %NOTFOUND par IF NOT FOUND THEN EXIT; END IF;

PL/SQL
Gestion des erreurs (suite)

- Premier cas particulier : une erreur Oracle doit être interceptée comme une exception (on attrape un erreur, et on lève une exception)

```
DECLARE
  e EXCEPTION ;
  PRAGMA EXCEPTION_INIT(e,-51); --range of:-20000
  to -20999 are supported in postgres
BEGIN
  ...
EXCEPTION
  WHEN e THEN
    ... - il y a eu timeout ...
  ...
END ;
```

- Second cas particulier : une erreur logique doit être traitée comme une erreur (et peut-être remonter au client)

```
RAISE_APPLICATION_ERROR(-20100,'Faute !') ;
```

PL/SQL
Autres possibilités

- Requêtes dynamiques : production à la volée d'instructions SQL

```
-> EXECUTE IMMEDIATE ... (sans vérification)
```

- Exemple : recherche par 1, 2 ou 3 mots-clés, la requête est construite dans une chaîne de caractères

```
PROCEDURE recherche(m1 VARCHAR2, m2 VARCHAR2, m3
  VARCHAR2)
  ...
BEGIN
  w := '1=1' ;
  IF m1 IS NOT NULL THEN
          w := w || ' AND titre LIKE ''%' || m1 ||
  '%''' ;
  END IF ;
  ... - idem avec m2 et m3
  EXECUTE IMMEDIATE 'SELECT * FROM livre WHERE ' || w ;
  ...
END recherche ;
```

PL/SQL
Autres possibilités (suite)

- La requête est construite à l'exécution :
  - pas de vérification à la compilation
  - reconstruction à chaque appel de la procédure

- Problèmes de sécurité (injection de code)

```
recherche(' ; delete * from emprunt',NULL,NULL)
```

  (ne dispense pas de valider les entrées)

- Dans le cas d'une requête : comment obtenir les résultats ?
  - solution naïve : utiliser une table temporaire...
  - ... mais problème de concurrence

- À manier avec beaucoup de précautions

Triggers

Triggers

- Les triggers (déclencheurs) sont des morceaux de code associés aux opérations standards (`INSERT, UPDATE, DELETE`)

  ->déclenchement implicite

- Les triggers permettent de :

  - gérer les « réflexes » (opérations automatiques) (par exemple : archivage, journalisation des opérations)

  - définir des contraintes d'intégrité non exprimables par ailleurs (en génie logiciel : pré- et post-conditions, invariants)

  - redéfinir complètement le comportement dans certains cas (par exemple : vues non modifiables)

Triggers
Définition

- Syntaxe :

```
CREATE [OR REPLACE] TRIGGER nom
{BEFORE|AFTER|INSTEAD OF} événements
ON table
[FOR EACH ROW [WHEN condition]]
Blocplsql
```

- Un trigger est nommé, et associé à une table ou vue

- Les événements sont :
  - `INSERT`
  - `UPDATE [OF col]`
  - `DELETE`

  On peut combiner : `INSERT OR DELETE`

Triggers

Définition (suite)

- Un trigger peut se déclencher :
  - soit globalement, pour une instruction (qu'elle affecte ou non des lignes)
  - soit localement, pour chaque ligne affectée (`FOR EACH ROW`)

  Dans ce cas, les lignes concernées peuvent être sélectionnées par une condition

- Le bloc PL/SQL est quelconque, mais ne peut contenir :
  - des instructions DDL
  - des instructions de contrôle de transaction (commit, rollback)

- Il peut appeler des procédures, avec les mêmes restrictions

- Si le trigger provoque une exception non interceptée, l'instruction échoue

Triggers

Triggers de ligne

- Un trigger de ligne (FOR EACH ROW) peut faire référence aux données en cours de traitement : par convention
  - `old` représente l'ancienne ligne (`UPDATE ou DELETE`)
  - `new` représente la nouvelle ligne (`UPDATE ou INSERT`)

(la condition peut aussi utiliser new et old)

```
CREATE TRIGGER verif_salaire
BEFORE INSERT OR UPDATE ON employe
FOR EACH ROW
DECLARE
   min NUMBER ;
   max NUMBER ;
BEGIN
   SELECT min,max INTO mini,maxi
     FROM grille WHERE fonction = :new.fonction ;
   IF :new.salaire NOT BETWEEN mini AND maxi THEN
     RAISE_APPLICATION_ERROR(-20101,'Salaire incorr.') ;
   END IF ;
END ;
```

Triggers

Triggers de ligne (suite)

- Exemple (bibliothèque)

```
CREATE TRIGGER verif_emprunt
BEFORE INSERT ON emprunt
FOR EACH ROW
DECLARE
  n NUMBER ;
BEGIN
  SELECT COUNT(*) INTO n
   FROM emprunt WHERE lecteur = :new.lecteur ;
  IF n >= 5 THEN
   RAISE_APPLICATION_ERROR(-20109,'Trop d''emprunts') ;
  END IF ;
END ;
```

- Avantage par rapport à une procédure : transparence

Triggers

Triggers de ligne (suite)

- Un trigger de ligne peut avoir un impact sérieux sur les performances

  (par exemple s'il effectue des requêtes sur d'autres tables)

- Il y a des restrictions sur les opérations possibles sur les tables dépendantes

  (par exemple, avec une contrainte d'intégrité référentielle)

- Triggers de ligne et contraintes d'intégrité sont deux choses différentes !

  (les CI sont toujours vérifiées, y compris à l'activation)

- Un trigger est activable/désactivable

  ```
  ALTER TRIGGER nom {ENABLE|DISABLE}
  ```

  (par exemple, audit/debugging)

Portage PL/SQL vers PL/PGSQL (différences)

Portage PL/SQL vers PL/PGSQL (différences)

- Il n'y a pas de valeurs par défaut pour les paramètres
- Vous ne pouvez pas utiliser des noms de paramètres identiques aux colonnes qui sont référencées dans la fonction. Oracle vous permet de le faire si vous qualifiez le nom du paramètre en utilisant nom_fonction.nom_paramètre.
- Vous pouvez surcharger les fonctions dans PostgreSQL™. C'est souvent utilisé pour contourner le manque de paramètres par défaut.
- Pas besoin de curseurs dans PL/pgSQL, mettez juste la requête dans l'instruction FOR
- Dans PostgreSQL™, le corps de la fonction doit être écrit comme une chaîne littérale. Du coup, vous avez besoin d'utiliser les guillemets dollar ou l'échappement des simples guillemets dans le corps de la fonction.

Portage PL/SQL vers PL/PGSQL (différences)

Voici une fonction en PL/SQL Oracle™ :

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar, v_version varchar)
RETURN varchar IS
BEGIN
        IF v_version IS NULL THEN
                RETURN v_name;
        END IF;
        RETURN v_name || '/' || v_version;
END;
/
show errors;
```

← n'existe pas en Postgresl (les erreurs sont par défaut affichées)

Voici de quoi aurait l'air cette fonction portée sous PostgreSQL™ :

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar, v_version varchar)
RETURNS varchar AS $$
BEGIN
        IF v_version IS NULL THEN
                return v_name;
        END IF;
        RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;
```

Portage PL/SQL vers PL/PGSOL (différences)

```
CREATE OR REPLACE PROCEDURE
cs_update_referrer_type_proc IS
CURSOR referrer_keys IS
SELECT * FROM cs_referrer_keys
ORDER BY try_order;
func_cmd VARCHAR(4000);
BEGIN
  func_cmd := 'CREATE OR REPLACE FUNCTION
cs_find_referrer_type(v_host IN VARCHAR, v_domain IN VARCHA
v_url IN VARCHAR) RETURN VARCHAR IS BEGIN';
  FOR referrer_key IN referrer_keys LOOP
    func_cmd := func_cmd ||
    ' IF v_' || referrer_key.kind
    || ' LIKE '''' || referrer_key.key_string
    || '''' THEN RETURN '''' ||referrer_key.referrer_type
    || ''''; END IF;';
  END LOOP;
  func_cmd := func_cmd || ' RETURN NULL; END;'; EXECUTE
IMMEDIATE func_cmd;
END;
/
show errors;
```

```
CREATE OR REPLACE FUNCTION cs_update_referrer_type_proc()
RETURNS void AS $func$

DECLARE referrer_key RECORD;
func_body text;
func_cmd text;
BEGIN
  func_body := 'BEGIN' ;

  FOR referrer_key IN SELECT * FROM cs_referrer_keys ORDER BY
try_order LOOP
    func_body := func_body ||
    ' IF v_' || referrer_key.kind
    || ' LIKE ' || quote_literal(referrer_key.key_string)
    || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
    || '; END IF;' ;
  END LOOP;
  func_body := func_body || ' RETURN NULL; END;';
  func_cmd :=
    'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host
varchar,v_domain varchar, v_url varchar)

    RETURNS varchar AS '
    || quote_literal(func_body)
    || ' LANGUAGE plpgsql;' ;
    EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;
```

Portage PL/SQL vers PL/PGSQL (différences)

http://docs.postgresql.fr/8.1/plpgsql-porting.html

TP 4 ET TP5

Réaliser le TP4 et le TP5