Bases de données avancées
PL/SQL

Dan VODISLAV

CY Cergy Paris Université

Master Informatique M1

Cours BDA

Plan

- PL/SQL: éléments de base du langage
- Curseurs
- Déclencheurs (triggers)
- Exceptions

PL/SQL

- Rappels
  - Langage procédural pour écrire des programmes *dans* Oracle
  - Langage propriétaire proche de la norme PSM (Persistent Storing Modules)
  - Syntaxe inspirée du langage ADA
- Syntaxe générale
  - Blocs DECLARE … BEGIN … EXCEPTION … END
    - Les blocs peuvent être imbriqués

```
DECLARE
  -- section optionnelle de déclarations
BEGIN
  -- section obligatoire d'instructions
EXCEPTION
  -- section optionnelle de gestion d'erreurs produites par l'exécution
END;
```


Variables

- Déclarées dans la section DECLARE
  - Syntaxe simplifiée: `nomVariable [CONSTANT] type [:= valeur];`
  - Autre façon d'affecter une valeur à une variable:

    `SELECT … INTO nomVariable FROM … WHERE … ;`
    - La requête doit retourner *un seul* résultat !!
- Types
  - Scalaires: CHAR, NUMBER, DATE, VARCHAR2, ….
  - Composés: ligne, table, enregistrement
- Constructions %TYPE et %ROWTYPE
  - *nomTable.nomColonne*%TYPE : le type des valeurs de la colonne
  - *nomVariable*%TYPE : le type de la variable (déclarée auparavant)
  - *nomTable*%ROWTYPE : type (composé) des lignes de la table

Exemple

- Schéma relationnel (les clés sont soulignées)

```
immeuble (Adr, NbEtg, DateConstr, NomGerant)
appart (Adr, Num, Type, Superficie, Etg, NbOccup)
personne (Nom, Age, CodeProf)
occupant (Adr, NumApp, NomOccup, DateArrivee, DateDepart)
propriete (Adr, NomProprietaire, QuotePart)
```

- Déclarations de variables

```
DECLARE
  nomPersonne CHAR(10) := 'Guillaume';
  monAdr CONSTANT CHAR(20):= '2 rue de la Paix';
  etage appart.Etg%TYPE;
  monImmeuble immeuble%ROWTYPE;
BEGIN
  SELECT * INTO monImmeuble FROM immeuble WHERE Adr=monAdr;
  SELECT Etg INTO etage FROM appart
   WHERE Adr=monImmeuble.Adr AND Num=10;
  INSERT INTO personne VALUES(nomPersonne, 35, 'ING');
END;
```


Entrées / sorties

- Paquetage DBMS_OUTPUT

  ENABLE; -- permet les entrées/sorties
  DISABLE; -- désactive les entrées/sorties
  PUT (in *valeur*); -- affichage valeur chaine de caractères/ numérique/ date
  NEW_LINE; -- retour à la ligne
  PUT_LINE (in *valeur*); -- même chose avec retour à la ligne
  GET_LINE (out *varChaine*, out *état*); -- lecture d'une ligne dans *varChaine*
  état est 0 si tout est en règle
  GET_LINES (out *varTabChaines*, in out *nbLignes*); -- lecture de plusieurs lignes

- Exemple

```
DECLARE
  entree VARCHAR2(50);
  etat INTEGER := 0;
  age NUMBER(3);
BEGIN
  DBMS_OUTPUT.ENABLE;
  DBMS_OUTPUT.PUT_LINE('Entrez votre age');
  DBMS_OUTPUT.GET_LINE(entree, etat);
  age := TO_NUMBER(entree);
  DBMS_OUTPUT.PUT_LINE('Vous avez donc ' || age || ' ans!');
END;
```

Structures de contrôle conditionnelles

- IF *condition1* THEN *instructions1*;
  [ELSIF *condition2* THEN *instructions2*;]
  …
  [ELSE *instructions*;]
  END IF;
- CASE *variable*
  WHEN *valeur1* THEN *instructions1*;
  …
  [ELSE *instructions*]
  END CASE;
- CASE
  WHEN *condition1* THEN *instructions1*;
  …
  [ELSE *instructions*]
  END CASE;

Structures de contrôle de répétition

- WHILE *condition* LOOP
  instructions;
  END LOOP;
- LOOP
  [*instructions1*;]
  EXIT [WHEN *condition*];
  [*instructions2*;]
  END LOOP;
- FOR compteur IN [REVERSE] *valInf* .. *valSup* LOOP
  instructions;
  END LOOP;

Sous-programmes

- Sous-programme PL/SQL = bloc nommé et paramétré
  - Deux types: fonctions et procédures
- Procédures

  PROCEDURE *nomProc*
  [(*param1* [IN|OUT|IN OUT] *type1*, …)]
  IS|AS [*déclarations*]
  BEGIN
  instructions
  [EXCEPTION
  traitementException]
  END [*nomProc*];
- Fonctions

  FUNCTION *nomFonc*
  [(*param1* [IN|OUT|IN OUT] *type1*, …)]
  RETURN *typeRetour*
  IS|AS [*déclarations*]
  BEGIN
  instructions
  [EXCEPTION
  traitementException]
  END [*nomFonc*];

Sous-programmes (suite)

- Instruction RETURN [*valeur*]
  - Termine une fonction en retournant une valeur de type *typeRetour*
  - Peut terminer une procédure (sans retourner de valeur)
- Catégories de paramètres
  - IN: non modifiable, seule sa valeur compte
  - OUT: on ne peut que lui affecter une valeur de sortie, pas de le consulter
  - IN OUT: on peut le consulter et le modifier
  - Remarques
    - OUT et IN OUT ne peuvent être que des variables
    - Pour les fonctions on devrait utiliser seulement des paramètres IN
- Appel:
  - *nomProc*(*listeParamEffectifs*);
  - *nomFonc*(*listeParamEffectifs*);

Définition et manipulation de sous-programmes

- Définition
  - Dans un bloc PL/SQL: à la fin de la section DECLARE

    DECLARE …
      PROCEDURE *nomProc* … END *nomProc*;
      FUNCTION *nomFonc* … END *nomFonc*;

    BEGIN … END;
  - Sous-programme stocké par le SGBD

    CREATE [OR REPLACE] PROCEDURE *nomProc* … END *nomProc*;

    CREATE [OR REPLACE] FUNCTION *nomFonc* … END *nomFonc*;
- Compilation
  - Automatique: à la création ou quand des objets dont il dépend ont été modifiés
  - Manuelle: ALTER PROCEDURE|FUNCTION *nom* COMPILE;
- Suppression: DROP PROCEDURE|FUNCTION *nom*

Curseurs

- Interaction avec la base de données: commandes SQL
  - Consultation: données récupérées dans des variables

    SELECT *listeColonnes* INTO *listeVariables* FROM … WHERE …
  - Modification: utilisant des valeurs de variables

    INSERT, UPDATE, DELETE
- Problème à la consultation: la requête doit retourner *exactement une ligne résultat*, qui est affectée à la (aux) variable(s) INTO

  → Comment faire pour consulter un résultat de plusieurs lignes?
- Solution: *les curseurs* = zones de travail
  - Stockant plusieurs enregistrements suite à une requête SQL
  - Gérant l'accès à ces enregistrements

Types de curseurs

- Curseurs implicites
  - Pour la *consultation à un résultat* et pour les *modifications*
  - Accessibles à travers le nom réservé SQL
  - Attributs qui offrent des informations sur la dernière commande exécutée
    - SQL%FOUND : la dernière commande a affecté au moins une ligne
    - SQL%NOTFOUND : la dernière commande n'a affecté aucune ligne
    - SQL%ROWCOUNT : nombre de lignes affectées par la dernière commande

```
UPDATE immeuble SET NomGerant='Toto' WHERE NbEtg>10;
DBMS_OUTPUT.PUT_LINE('Toto gère ' || SQL%ROWCOUNT || '
  immeubles de plus de 10 étages')
```

- Curseurs explicites
  - Pour les *consultations*
  - Doivent être déclarés, ouverts, consultés et fermés

Opérations sur les curseurs explicites

- *Déclaration*: dans la section DECLARE
  - CURSOR *nomCurseur* IS SELECT … FROM … WHERE …;
- *Ouverture*: exécution de la requête et initialisation
  - OPEN *nomCurseur*;
- *Chargement*: récupération d'une ligne de résultat dans des variables et positionnement sur la ligne suivante
  - FETCH *nomCurseur* INTO *listeVariables*;
- *Fermeture*: libération de la zone mémoire
  - CLOSE *nomCurseur*;

Attributs des curseurs explicites

- *nomCurseur*%ISOPEN
  - Vrai si le curseur est ouvert
- *nomCurseur*%FOUND
  - Vrai si le dernier FETCH a trouvé une ligne
- *nomCurseur*%NOTFOUND
  - Vrai si le dernier FETCH n'a plus trouvé de ligne (fin du résultat)
- *nomCurseur*%ROWCOUNT
  - Nombre de lignes trouvées par la requête

Exemple

- Calculer le nombre d'occupants d'un immeuble

```
DECLARE
  monAdr CONSTANT CHAR(20):= '2 rue de la Paix';
  CURSOR cOccupants IS
     SELECT NbOccup FROM appart WHERE Adr=monAdr;
  occupApart appart.NbOccup%TYPE;
  total INTEGER := 0;
BEGIN
  OPEN cOccupants;
  LOOP
     FETCH cOccupants INTO occupAppart;
     EXIT WHEN cOccupants%NOTFOUND;
     total := total + occupAppart;
  END LOOP;
  CLOSE cOccupants;
  DBMS_OUTPUT.PUT_LINE(total || ' occupants dans
  l'immeuble');
END;
```

Utilisation de la boucle FOR

- Avantage: OPEN, FETCH et CLOSE sont implicites
  - OPEN et CLOSE automatiques au début et à la fin de la boucle
  - Variable de boucle de type *curseur*%ROWTYPE déclarée automatiquement
  - FETCH automatique dans la variable de boucle à chaque itération
- Pour le même exemple

```
DECLARE
  monAdr CONSTANT CHAR(20):= '2 rue de la Paix';
  CURSOR cOccupants IS
     SELECT NbOccup FROM appart WHERE Adr=monAdr;
  total INTEGER := 0;
BEGIN
  FOR occupAppart IN cOccupants LOOP
    total := total + occupAppart;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(total || ' occupants dans l'immeuble');
END;
```


Curseurs et modifications

- On peut aussi *modifier* les lignes parcourues par le curseur
- Déclaration: ajout d'une clause FOR UPDATE

```
CURSOR nomCurseur IS
  SELECT … FROM … WHERE …
  FOR UPDATE [OF listeColonnes] [NOWAIT | WAIT durée];
```

- Effet: verrouillage des colonnes en attendant (WAIT durée) ou non (NOWAIT) qu'elles soient disponibles pour la modification
  - Par défaut: toutes les colonnes et attente indéfinie
- Modification: UPDATE ou DELETE sur la ligne courante
  -
```
UPDATE table SET modifications
WHERE CURRENT OF nomCurseur;
```
  -
```
DELETE FROM table
WHERE CURRENT OF nomCurseur;
```

Exemple modification

- Diminuer la superficie des appartements sous les combles (3ème étage) d'une maison donnée

```
DECLARE
  adrMaison CHAR(20) := '5 rue de la Seine';
  CURSOR cModif IS
     SELECT * FROM appart WHERE Adr=adrMaison AND Etg=3
     FOR UPDATE;
BEGIN
  FOR appartMaison IN cModif LOOP
     UPDATE appart
     SET Superficie = appartMaison.Superficie - 3
     WHERE CURRENT OF cModif;
  END LOOP;
END;
```


Déclencheurs (triggers)

- Programmes dont l'exécution est déclenchée par un événement
  - Un déclencheur n'est pas appelé explicitement
- Avantage: valables pour toute la base de données, quel que soit le programme qui produit l'événement déclencheur
- Événements déclencheurs :
  - Instructions de modification de données : INSERT, UPDATE, DELETE
  - Instructions de modification de schéma : CREATE, ALTER, DROP
  - Démarrage ou arrêt de la base
  - Connexion ou déconnexion d'utilisateur
  - Erreur d'exécution
- Utilisation habituelle: pour réaliser des règles de gestion non exprimables par les contraintes au niveau des tables

Création d'un déclencheur

- Déclencheur sur modification des données

```
CREATE [OR REPLACE] TRIGGER nomDeclencheur
{BEFORE | AFTER | INSTEAD OF}
{DELETE | INSERT | UPDATE [OF colonne1, …] [OR …]}
ON {nomTable | nomVue}
[REFERENCING {OLD [AS] nomAncien | NEW [AS] nomNouveau
              | PARENT [AS] nomParent } …]
[FOR EACH ROW]
[WHEN conditionSupplementaire]
{[DECLARE …] BEGIN … [EXCEPTION …] END;
| CALL nomSousProgramme(listeParametres)}
```

- Structure :
  - Description de l'événement déclencheur
  - Éventuelle condition supplémentaire à satisfaire pour déclenchement
  - Description du traitement à réaliser après déclenchement

Description de l'événement déclencheur

- Description de l'événement
  - Type de modification (DELETE, INSERT, UPDATE)
    - Un ou plusieurs (séparés par OR si plusieurs)
  - Éventuelles colonnes spécifiques ([OF colonne1, …])
  - Le nom de la table (ou vue) (ON {nomTable | nomVue})
- Moment d'exécution du déclencheur
  - Avant l'événement : BEFORE
  - Après l'événement : AFTER
  - À la place de l'événement : INSTEAD OF (uniquement pour vues multi-tables)

Description de l'événement déclencheur (suite)

- Changement des noms par défaut : clause REFERENCING
  - :OLD désigne un enregistrement à effacer (déclencheur sur DELETE, UPDATE)
    - REFERENCING OLD AS *nomAncien*: permet d'utiliser *nomAncien* à la place de :OLD
  - :NEW désigne un enregistrement à insérer (déclencheur sur INSERT, UPDATE)
    - REFERENCING NEW AS *nomNouveau*: utiliser *nomNouveau* à la place de :NEW
  - :PARENT pour des tables imbriquées
    - REFERENCING PARENT AS *nomParent*
- Clause FOR EACH ROW
  - Avec FOR EACH ROW, une exécution *par ligne* concernée par l'instruction DELETE/INSERT/UPDATE ("*row trigger*")
  - Sans FOR EACH ROW, une exécution *par instruction* DELETE/INSERT/UPDATE ("*statement trigger*")

Exemple de contraintes à satisfaire

- Clé étrangère (intégrité référentielle): contraintes dans le schéma

  Ex. Dans la table `propriete`, à chaque insertion/modification de ligne
  CONSTRAINT prop_pers FOREIGN KEY (NomProprietaire)
  REFERENCES personne (Nom)
- Condition entre colonnes: contraintes dans le schéma

  Ex. Dans la table `occupant`, à chaque insertion/modification de ligne
  CONSTRAINT dates CHECK (DateArrivee < DateDepart)
- Autres règles non exprimables par des contraintes dans le schéma → *déclencheurs*

  Ex. somme des quotes-parts d'une propriété = 100,
  date de construction d'immeuble < date d'arrivée d'un occupant

Exemple de déclencheur sur insertion

- Pour un nouvel occupant, vérifie si
  `occupant.DateArrivee > immeuble.DateConstr`

```
CREATE TRIGGER verifDate
  BEFORE INSERT ON occupant
  FOR EACH ROW
DECLARE
  Imm immeuble%ROWTYPE;
BEGIN
  SELECT * INTO Imm FROM immeuble
                    WHERE immeuble.Adr = :NEW.Adr;
  IF :NEW.DateArrivee < Imm.DateConstr THEN
    RAISE_APPLICATION_ERROR(-20100, :NEW.Nom || ' arrivé
  avant la construction immeuble ' || Imm.Adr);
  END IF;
END;
```


Déclencheur sur insertion (suite)

- Autre exemple: insérer automatiquement un appartement dans tout nouvel immeuble construit

```
CREATE TRIGGER premierAppart
  AFTER INSERT ON immeuble
  FOR EACH ROW
BEGIN
  INSERT INTO appart (Adr, Num, NbOccup)
         VALUES (:NEW.Adr, 1, 0);
END;
```

Déclencheur sur suppression

- Au départ d'un occupant, décrémenter le nombre d'occupants de l'appartement en question

```
CREATE TRIGGER departOccupant
  AFTER DELETE ON occupant
  FOR EACH ROW
BEGIN
  UPDATE appart
    SET NbOccup = NbOccup-1
    WHERE appart.Adr = :OLD.Adr AND
          appart.Num = :OLD.NumApp;
END;
```


Déclencheur sur modification

- En cas de modification sur un occupant, modifier le nombre d'occupants des appartements concernés (si la modification concerne l'immeuble/l'appartement)

```
CREATE TRIGGER modifOccupant
  AFTER UPDATE ON occupant
  FOR EACH ROW
BEGIN
 IF :OLD.Adr<>:NEW.Adr OR :OLD.NumApp<>:NEW.NumApp THEN
   UPDATE appart SET NbOccup = NbOccup-1
   WHERE appart.Adr=:OLD.Adr AND appart.Num=:OLD.NumApp;
   UPDATE appart SET NbOccup = NbOccup+1
   WHERE appart.Adr=:NEW.Adr AND appart.Num=:NEW.NumApp;
 END IF;
END;
```

Déclencheur sur conditions multiples

- Exemple: un seul déclencheur, qui regroupe toutes les actions sur `occupant` et qui actualise le nombre d'occupants des appartements

```
CREATE TRIGGER touteModifOccupant
  AFTER INSERT OR DELETE OR UPDATE ON occupant
  FOR EACH ROW
BEGIN
  IF INSERTING THEN ...
  ELSIF DELETING THEN ...
  ELSIF UPDATING THEN ...
  END IF;
END;
```


Déclencheurs sur modification de schéma, droits…

- Syntaxe

```
CREATE [OR REPLACE] TRIGGER nomDeclencheur
{BEFORE | AFTER} action [OR action …]
ON {[nomSchema.]SCHEMA | DATABASE}
{[DECLARE …] BEGIN … [EXCEPTION …] END;
| CALL nomSousProgramme(listeParametres)}
```

- Quelques actions possibles :
  - CREATE, RENAME, ALTER, DROP sur un objet du dictionnaire de données
  - GRANT, REVOKE droits pour un utilisateur

Exemple de déclencheur sur schéma

- Enregistrer dans une table les changements de nom des objets du dictionnaire de données

```
CREATE TRIGGER changementNom
  AFTER RENAME ON DATABASE
BEGIN
  -- On se sert de 2 attributs système
  -- ora_dict_obj_name : nom objet affecté
  -- ora_dict_obj_owner : propriétaire objet affecté
  INSERT INTO changementsNoms
    VALUES (SYSDATE,
            ora_dict_obj_name,
            ora_dict_obj_owner);
END;
```


Déclencheurs d'instance

- Événements concernant le démarrage/arrêt de la base, les connexions des utilisateurs, les erreurs
- Syntaxe

```
CREATE [OR REPLACE] TRIGGER nomDeclencheur
{BEFORE | AFTER} evenement [OR evenement …]
ON {[nomSchema.]SCHEMA | DATABASE}
{[DECLARE …] BEGIN … [EXCEPTION …] END;
| CALL nomSousProgramme(listeParametres)}
```

- Exemples d'événements
  - Démarrage (STARTUP) ou arrêt (SHUTDOWN) de la base
  - Connexion (LOGON) ou déconnexion (LOGOFF) d'un utilisateur
  - Erreurs: SERVERERROR, NO_DATA_FOUND, …

Manipulation des déclencheurs

- Tout déclencheur est actif dès sa compilation!
- Re-compilation d'un déclencheur après modification :
  - ALTER TRIGGER nomDeclencheur COMPILE;
- Désactivation de déclencheurs :
  - ALTER TRIGGER nomDeclencheur DISABLE;
  - ALTER TABLE nomTable DISABLE ALL TRIGGERS;
- Réactivation de déclencheurs :
  - ALTER TRIGGER nomDeclencheur ENABLE;
  - ALTER TABLE nomTable ENABLE ALL TRIGGERS;
- Suppression d'un déclencheur :
  - DROP TRIGGER nomDeclencheur;

Exceptions

- Principe
  - L'exécution d'un programme PL/SQL peut produire des erreurs
  - PL/SQL propose un mécanisme de traitement des erreurs, permettant d'éviter l'arrêt systématique du programme
- Syntaxe

```
…
EXCEPTION
  WHEN nomException1 [OR nomException2 …] THEN
    instructions1;
  WHEN nomException3 [OR nomException4 …] THEN
    instructions3;
  WHEN OTHERS THEN
    instructionsAttrapeTout;
END;
```

Traitement d'une exception

- Une erreur apparaît à l'exécution d'une instruction dans un bloc
  - L'exécution du bloc PL/SQL courant est abandonnée
  - Le traitement de l'exception (spécifique ou attrape-tout) est recherché dans la section EXCEPTION associée au bloc courant
  - Si pas de traitement prévu: on cherche dans les blocs englobants (du plus proche vers le plus éloigné)
  - Si pas de traitement trouvé: on cherche dans le programme appelant, ensuite dans l'appelant de l'appelant, etc.
- Si un traitement est trouvé
  - Les instructions de traitement sont exécutées
  - L'exécution se poursuit normalement après le bloc *qui a traité* l'exception
- Sinon le programme s'arrête

Déclenchement des exceptions

- Déclenchement automatique suite à l'apparition d'une erreur prédéfinie Oracle
  - Exceptions nommées: VALUE_ERROR, ZERO_DIVIDE, ...
  - Exceptions non nommées: identifiées par un numéro d'erreur
- Déclaration de ses propres exceptions + déclenchement explicite
  - Déclaration (dans DECLARE) : `nomException EXCEPTION;`
  - Déclenchement (dans BEGIN) : `RAISE nomException;`

  ☞ On peut déclencher avec RAISE une exception prédéfinie Oracle!

  ```
  RAISE nomExceptionPreDefinie;
  ```
- Déclenchement d'exceptions non nommées

  ```
  RAISE_APPLICATION_ERROR(numErreur, msgErreur);
  ```
- Propagation explicite de l'exception vers le bloc/programme appelant

  ```
  WHEN nomException1 THEN
    …
    RAISE; -- la même exception nomException1
  WHEN autreException THEN …
  ```

Traitement des exceptions non nommées

- Traitement non spécifique grâce à l'attrape-tout :
  - Variables SQLCODE (code erreur) et SQLERRM (message)

  Exemple :

```
WHEN OTHERS THEN
  DBMS_OUTPUT.PUT_LINE(SQLERRM || '(' || SQLCODE || ')');
```

- Identification et traitement spécifique
  - Identification dans la section DECLARE :

```
nomAPrendre EXCEPTION;
PRAGMA EXCEPTION_INIT(nomAPrendre, numErrOracle);
```

  - Traitement spécifique :

```
WHEN nomAPrendre THEN instructions;
```


Source

- Transparents du cours "*Développement d'applications avec les bases de données*", Michel Crucianu, CNAM Paris
  `http://cedric.cnam.fr/~crucianm/abd.html`