BD dynamique / Active

Laisser au SGBD la gestion des contraintes ou comment programmer des contraintes…

Plan

- Définitions
- Déclencheurs et règles
- Utilisation des règles
  - intégrité
  - vues
- Définir des règles sous Oracle
  - Introduction
  - Programmation en PL/SQL
  - Développement de triggers (déclencheurs)
  - Use case

Définitions

- SGBD passif
  - stockage de l'information
  - restitution de l'information
  - les activités de contrôle sont réalisées dans les programmes d'application
- SGBD actif
  - stockage et restitution de l'information
  - description des événements pertinents et des réactions à activer lorsque ces événements se produisent

Limitation SQL

- Gestion de contraintes simples dans la définition du schéma:
  - valeur nulle impossible (NOT NULL)
  - unicité de l'attribut (UNIQUE)
  - contrainte référentielle (FOREIGN KEY ... REFERENCES ...)
  - contrainte générale (CHECK)
- Réactions prédéfinies aux opérations de suppression et de mise à jour (ON DELETE, ON UPDATE) liées aux contraintes référentielles (NO ACTION, CASCADE, SET DEFAULT, SET NULL)

Création de schéma: CREATE TABLE

```
CREATE TABLE  nomTable (
    colonne1 type1 [DEFAULT valeur1] [NOT NULL],
    colonne2 type2 [DEFAULT valeur2] [NOT NULL],
    [CONSTRAINT nomContrainte1 typeContrainte1 paramContrainte1] …
    );
```

- NomTable: le nom de la relation comportant des lettres, des chiffres et les symboles classiques (_,$,#,etc.).
- Colonnei: le nom d'un attribut de la relation,
- Typei: un type (Oracle) qui définit le domaine d'un attribut.
- CONSTRAINT …: une contrainte pour cette relation (détaillé plus loin),
- DEFAULT: permet de préciser une valeur par défaut (si aucune n'est précisée lors de l'insertion d'un tuple)
- NOT NULL: précise que la valeur NULL (absence de valeur) est interdite pour un attribut.

Types (Oracle) disponibles

- Type chaîne de caractères
  - de longueur fixe : **`CHAR(longueur)`**
    Par défaut la valeur est d'1 caractère et au maximum de 2000 caractères.
  - de longueur variable : **`VARCHAR2(longueur)`**
    Par défaut 1 caractère et au maximum 4000 caractères.
    Le stockage dépend de la taille réelle de la chaîne.
    La norme définit le type ***char(longueur)****.*
  - de flot de caractères: **`CLOB(longueur) jusqu'à 4Go.`**
  - de caractères Unicode: **`NCHAR(l), NVARCHAR2(l), NCLOB(l)`**

Types (Oracle) disponibles

- Type numérique décimale :
  `NUMBER(précision, échelle)`
  - `précision` : nombre entier de chiffres significatifs (1 à 38).
  - `échelle` : nombre de chiffre à droite de la virgule (-84 à 127). Une valeur négative arrondit le nombre.

  ex: VAL NUMBER(8,2) définit la donnée VAL avec 8 chiffres significatif dont deux après la virgule (-999999,99 à +999999,99)
  VAL NUMBER(8,-2) arrondit la donnée à la centaine.
  - *La norme définit les types **decimal(precision, echelle)** et **integer**.*

Types (Oracle) disponibles

- Type date/heure:
  - **`DATE`**: le format standard d'une date qui permet de stocker le siècle, l'année, le mois, le jour, l'heure, les minutes et les secondes. Le format par défaut dépend de la langue (le stockage est indépendant).
  - **`TIMESTAMP`**(précision): date et heure incluant des fractions de secondes (précision de 0 à 9).
  - **`INTERVAL YEAR`** (précision) **`TO MONTH`**: permet d'extraire une différence entre deux moments avec une précision mois/année (précision)
  - **`INTERVAL DAY`** (précision1) **`TO SECOND`** (précision2): différence plus précise entre deux moments (précision1 pour les jours et précision2 pour les fractions de secondes).
- Type binaires:
  - **`BLOB`**: données non structurées jusqu'à 4Go.
  - **`BFILE`**: données stockées dans un fichier externe à la base.

CREATE TABLE (exemple)

```
/* Description des Personnes */
CREATE TABLE Personne (nP CHAR(10), nom VACHAR2(15), adr
    VARCHAR2(50));
/* Description des prénoms
    d'une personne */
CREATE TABLE PersonnePrénoms(
    num CHAR(10),
    prénom VARCHAR2(15) );
-- Description des étudiants
-- qui sont par ailleurs des personnes.
CREATE TABLE Etudiant(nP CHAR(10), nE CHAR(10), dateN DATE);
/* Description des études réalisées par un étudiant */
CREATE TABLE EtudiantEtudes(nE CHAR(10), année NUMBER(1), diplôme
    VARCHAR2(30));
-- Fin du script SQL pour Oracle.
```

Gestion des contraintes

- Les contraintes sont déclarées de deux manières:
  - **mode *inline***: c'est-à-dire en même temps que l'attribut qu'elle contraint (contrainte monocolonne).
  - **mode *out-of-line***: la définition apparaît à la suite des définitions d'attributs, introduit avec le mot clé CONSTRAINT. Dans ce cas la contrainte peut être nommée (plus flexible) et peut porter sur plusieurs attributs (ou colonnes).
- On distingue 4 types de contraintes:
  - **UNIQUE** : impose une valeur distinct d'un attribut pour chacun des tuples.
  - **PRIMARY KEY** : déclare l'identifiant de la relation (ou clé de la table), la contrainte UNIQUE et l'option NOT NULL est sous-entendu pour les attributs concernés.
  - **FOREIGN KEY** : déclare un identifiant externe (ou clé étrangère) ainsi que la méthode employée pour assurer l'intégrité référentielle.
  - **CHECK** : permet de spécifier une condition simple que doit suivre le ou les attributs.

Exemple de schéma (amélioré)

```
/* Description des Personnes */
CREATE TABLE Personne (
    nP CHAR(10) NOT NULL UNIQUE, -- contrainte UNIQUE inline
    nom VACHAR2(15) NOT NULL,
    adr VARCHAR2(50));
/* Description des prénoms
    d'une personne */
CREATE TABLE PersonnePrénoms(
    num CHAR(10) NOT NULL,
    prénom VARCHAR2(15) NOT NULL,
    -- contrainte out-of-line nommée
    CONSTRAINT une_fois_prenom UNIQUE (num,prénom) );
```

On peut faire beaucoup mieux en introduisant les identifiants !

Contrainte PRIMARY KEY

```
/* Description des Personnes */
CREATE TABLE Personne (
    nP CHAR(10) PRIMARY KEY, -- contrainte inline
    nom VACHAR2(15) NOT NULL,
    adr VARCHAR2(50));
/* Description des prénoms
    d'une personne */
CREATE TABLE PersonnePrénoms(
    num CHAR(10),
    prénom VARCHAR2(15),
    -- contrainte out-of-line nommée
    CONSTRAINT pk_PersonnePrénoms PRIMARY KEY (num,prénom) );
```

- Dans la première table on peut utiliser une contrainte *inline* où *out-of-line*.
- Dans la seconde table, la contrainte *out-of-line* est obligatoire: elle porte sur deux colonnes, l'identifiant étant composé de deux attributs.

Contrainte Foreign Key

- Cette contrainte est le noyau de la gestion de cohérence d'une base de données relationnelle: l'intégrité référentielle.
  - basée sur une relation entre identifiant (clé primaire/candidate) et identifiant externe (clé étrangère): "Un étudiant est une personne qui doit exister dans la base".
  - on parle souvent de tables:
    - **père**: cette table possède la clé primaire référencée (ex: Personne).
    - **fils**: cette table possède une clé étrangère qui référence la clé primaire/candidate d'une table père (ex: Etudiant).
- Les problèmes résolus/évités par un SGBD en activant cette cohérence:
  - **fils vers père**: impossible d'ajouter un tuple fils qui ne puisse pas se rattacher à un tuple père (ex: pas d'étudiant sans être une personne),
  - **père vers fils**: impossible de supprimer un tuple père ayant encore des tuples fils qui lui sont rattachés (ex: plus d'étudiants si plus de personnes).

Contrainte Foreign Key (suite)

- Il faut donc gérer cette contrainte qui assure l'intégrité référentielle du coté père et du coté fils:
  - du coté père: on s'assure d'avoir une clé primaire (PRIMARY KEY) pour pouvoir référencer les tuples de la relation père.
  - du coté fils: on définit la clé étrangère à l'aide d'une contrainte FOREIGN KEY (qui précise le ou les attributs du fils qui référencent la clé primaire/candidate du père).
- CONSTRAINT contrainte FOREIGN KEY(f_attr1, f_attr2) REFERENCES p_table(p_attr1,p_attr2)
  - contrainte: nom de la contrainte pour faciliter sa gestion,
  - f_attr_i: nom des attributs de la table fils composant la clé étrangère,
  - p_table: nom de la table père référencée,
  - p_attr_i: nom des attributs de la table père p_table composant sa clé primaire/candidate qui sont référencés par f_attr_i.

Contrainte Foreign Key (exemple)

```
/* Description des Personnes: table père */
CREATE TABLE Personne (
    nP CHAR(10) PRIMARY KEY, -- définition de l'identifiant chez le père
    nom VACHAR2(15) NOT NULL,
    adr VARCHAR2(50));
/* Description des Etudiants: table fils */
CREATE TABLE Etudiant(nE CHAR(10) PRIMARY KEY,
    nP CHAR(10),
    dateN DATE,
    -- définition de l'identifiant externe
    CONSTRAINT fk_Etudiant_Personne FOREIGN KEY(nP)
        REFERENCES Personne(nP) );
```

Contrainte Foreign Key (gestion de la cohérence père -> fils)

- La suppression d'un tuple du père donnent trois alternatives:
  - on interdit la suppression de ce tuple s'il est encore référencé par des tuples des fils: il faut supprimer tous les tuples fils avant. On ne doit pas survivre à ces enfants!
  - on accepte la suppression de ce tuple mais cette suppression est automatiquement propager aux tuples des fils. Ainsi la cohérence est respectée. La clause optionnelle ON DELETE CASCADE doit être ajoutée.
  - on accepte cette suppression de tuple mais sans propagation aux tuples des fils. Les composants de la clé étrangère des tuples des fils prennent la valeur NULL. La clause optionnelle ON DELETE SET NULL doit être ajoutée (et ne pas contredire les contraintes NOT NULL).
- Exemple:

```
CREATE TABLE Etudiant(nE CHAR(10) PRIMARY KEY,
    nP CHAR(10),DateN DATE,
    -- définition de l'identifiant externe
    CONSTRAINT fk_Etudiant_Personne FOREIGN KEY(nP)
          REFERENCES Personne(nP) ON DELETE CASCADE);
```

Contrainte Check(condition)

- Cette contrainte contrôle l'insertion ou la modification d'un tuple en fonction d'une condition explicite:
  - impose un domaine de valeurs (restriction plus précise qu'un typage),
  - Les contraintes plus complexes se programment via des "déclencheurs" .
- Exemple:

CREATE TABLE Etudiant(nE CHAR(10) PRIMARY KEY, nP CHAR(10), DateN DATE,
CONSTRAINT ck_DateN CHECK ((SYSDATE-DateN)/365>18));

- La condition accepte divers opérateurs:
  - de comparaison (>,=,<,>=,<=,<>)
  - logiques (NOT, AND, OR)
  - mathématiques (+,-,*,/)
  - des fonctions (cf. fin de ce cours)
  - des opérateurs intégrés (BETWEEN, IN, LIKE, IS NULL)

Contrainte CHECK (Suite)

- Les opérateurs intégrés (comme le format de condition) sont très utilisés en SQL (en mode interrogation).

CREATE TABLE Personne (
nP CHAR(10) ***PRIMARY KEY****,*
nom VACHAR2(15),
adr VARCHAR2(50)

  - Opérateur BETWEEN … AND … teste l'appartenance à un intervalle de valeurs, renvoie un booléen (VRAI/FAUX).

CONSTRAINT ck_nP1 CHECK (nP BETWEEN 1 and 10),

  - Opérateur IN (…) teste l'appartenance à un ensemble de valeurs.

CONSTRAINT ck_nP2 CHECK (nP IN (1,2,3,4,5,6,7,8,9,10)),

  - Opérateur LIKE … compare une expression à une chaîne de caractère type (% pour un ou plusieurs caractères, _ pour un seul caractère).

CONSTRAINT ck_adr CHECK (adr LIKE '% Grenoble %'),

  - Opérateur IS NULL teste si une expression correspond à la valeur NULL.

CONSTRAINT ck_nom CHECK (nom NOT IS NULL));

Évolution de schéma

- La commande DROP (pour supprimer une table)
  - DROP TABLE nom_table [CASCADE CONSTRAINTS];
  - CASCADE CONSTRAINTS
    - Supprime toutes les contraintes référençant une clé primaire (primary key) ou une clé unique (UNIQUE) (permet de supprimer une table sans être gêné par ses fils)
    - Si on cherche à détruire une table dont certains attributs sont référencés sans spécifier CASCADE CONSTRAINT, Oracle retourne une erreur.
  - Avant de créer un schéma il est toujours préférable d'avoir un script qui le détruit (pour éviter les erreurs de cohérence):
    - La destruction suit un chemin précis: d'abord les fils, ensuite les relations pères (inverse de la création).

DROP TABLE EtudiantEtudes;
DROP TABLE Etudiant;
DROP TABLE PersonnePrénoms
DROP TABLE Personne;

Manipulation de données

- Après la création d'un schéma, il faut peupler les tables, c'est-à-dire insérer des tuples.
- La commande INSERT INTO permet d'insérer ligne par ligne de nouveaux tuples:

  INSERT INTO table VALUES …

  - table: le nom de la table où insérer un tuple.
  - on peut renseigner ensuite toutes les valeurs d'un tuple:

    INSERT INTO Personne VALUES (1,'Dupont', 'Grenoble');
  - ou bien ne renseigner que certaines valeurs:

    INSERT INTO Personne(nP, nom) VALUES (2,'Aubry');
    La valeur de l'attribut Adr est alors implicitement NULL.
- Problème avec la date:
  - les formats '01/09/04', '01-09-04' et '01-09-2004' sont équivalents
  - sinon il faut utiliser la fonction TO_DATE(texte,format) qui transforme une chaîne de caractères en une date en fonction d'une expression de formatage:

    INSERT INTO Etudiant VALUES (1,1001,'22-01-71');
    ou INSERT INTO Etudiant VALUES(1,1001,TO_DATE('22 Janvier 71','DD MONTH YY');

Limitation SQL

- Prémisse d'une gestion active: la commande CREATE ASSERTION permet de tester des conditions avant ou après certains événements.

```
CREATE ASSERTION <nom contrainte>
    [BEFORE COMMIT |
    AFTER {INSERT|DELETE|UPDATE[OF (Attribut+)]} ON <Relation>]
    CHECK <Condition>
    [FOR [EACH ROW OF] <Relation>]
```

Exemple :

```
CREATE ASSERTION Mindegre
    BEFORE COMMIT
    CHECK (SELECT MIN(Degre) FROM Vins >10)
    FOR Vins;
```

! Implémentation réduite dans les SGBD !

Règle active

règles ECA

- Event: LORSQUE l'événement *E* se produit
- Condition: SI la condition *C* est satisfaite
- Action: ALORS exécuter l'action *A*

Les événements

L'événement spécifie la cause qui déclenche une règle.

- Modification des données

  insert, delete, update, méthode (BDOO uniquement)

- Interrogation

  select, méthode (BDOO)

- Temporel

  le 1er janvier 2000 / tous les jours à 12h00 /
  toutes les 10 minutes / ....

- Définis par l'application (température trop élevée, login-utilisateur, ...) : cet événement doit être correctement déclaré et son monitoring bien défini.

Événements composés

- opérateurs logiques
  - AND, OR, NOT
- séquences logiques
  - avant, après, ...
- séquences temporelles
  - 5 secondes après e1
  - 10 fois toutes les 30’ après e2

Conditions

La condition définie une condition supplémentaire pour valider l'action après déclenchement d'une règle.

- prédicats BD
  - WHERE SQL
- requête SQL
  - résultat vide / non vide
- fonctions d'agrégation, ...
- procédures ad-hoc (retourne Vrai / Faux)

Actions

Action exécutée quand une règle est déclenchée et sa condition OK.

- opérations de modification des données
  - insert, delete, update
  - appel de méthodes
- interrogation de la base de données
  - select
- autres
  - rollback, commit, abort, grant /revoke privilèges, ...
- procédures ad-hoc
- déclenchement d'autres règles

Exécution de l'action

- immédiate
- à la fin de la transaction
- check point
- autre moment explicitement défini

Algorithme de déclenchement

TANT QUE il y a une règle à déclencher

PRENDRE une règle déclenchable

EVALUER la condition

SI la condition est vérifiée ALORS

EXECUTER l'action

FIN TANT QUE

Organisation des règles

- le même événement peut déclencher plusieurs règles
  - priorités entre règles
  - priorités statiques: 1ère, 2ème, ...
  - priorités relatives: celle-ci avant celle-là
- structuration de l'ensemble des règles (create / delete / modify / enable / disable, ...)

Quand examiner les règles ?

- dès qu'un événement survient
- à la fin de l'exécution d'une requête SQL
- à la fin de la transaction
- périodiquement
- à la demande

Quand examiner les règles ? (Oracle)

- Mode classique: gestion **immédiate**, par défaut les contraintes sur les schémas sont testées pour chaque opération.
  - *Set constraint Nom_contrainte immediate*
- Mode relâchée: gestion **différée**, les contraintes ne sont testées qu'au moment de la validation d'une transaction.
  - *Set constraint Nom_contrainte deffered* si la contrainte a été définie ***defferable***.

Séquentiel versus parallèle

- Exécution séquentielle
  - choisir une règle, l'exécuter
  - choisir une autres règle, l'exécuter
- Exécution parallèle
  - choisir une règle, l'exécuter
  - pendant l'exécution, choisir une autres règle et l'exécuter
- Exécution groupée
  - toutes les règles déclenchables sont exécutées en parallèle
  - **Choix de l'ordre de déclenchement et exécution ?**

Terminaison

- L'algorithme de déclenchement peut ne pas terminer (boucle dans le graphe de déclenchement)
- solutions :
  - laisser faire
  - admettre un nombre maximal de règles exécutées après un événement
  - interdire l'activation d'une règle par une autre règle
  - restreindre les activations en chaîne au sein d'un groupe de règles

Implémentation

- mémorisation et gestion des règles (création, suppression, modification, activation, persistance)
- Concurrence (sur les règles, les événements, les conditions et les actions)
- Fiabilité (en cas de déclenchement d'une règle pendant un crash)
- Autorisations d'accès / privilèges sur les règles
- Gestion d'erreurs à différents niveaux
- Tracer les règles pour le debugging
- Performances des règles
- ....

Utilisations des règles

- vérification de contraintes d'intégrité => Gestion Intégrité
  - avantage: on ne vérifie que les données en cause
  - possibilité d'action corrective
  - contraintes statiques: facile
  - contraintes dynamiques : événements composés, ...
- maintien de la cohérence des vues (virtuelles, matérialisées, instantanées)
  => Gestion de données dérivées
- gestion de versions => Gestion de réplication
  - propagation des modifications de version
  - règle d'évolution des versions
- évolution de schéma

Intégrité: Génération de règles

- Peut être semi-automatique
  - (1) formule déclarative de la CI $\Rightarrow$ condition
  - (2) causes possibles de la violation $\Rightarrow$ événements
  - (3) action réparatrices $\Rightarrow$ actions

- Approche simple: CI encodées comme règles d'avortement
  - Activées par tout événement qui pourrait violer la contrainte
  - Précondition teste l'occurrence des violations
  - Si violation: commande ROLLBACK
  - Inconvénient: cause beaucoup de ROLLBACK

Intégrité: Génération de règles, cont.

- Autre approche: CI encodées comme règles de réparation

  - CI spécifiées avec actions à réaliser pour restaurer l'intégrité
  - Ex: intégrité référentielle SQL-92

  * même pré-condition et événements que règles d 'avortement

  * actions contiennent des manipulations de la BD

  - Ex: actions de réparation pour l'intégrité référentielle reflètent la sémantique de CASCADE, RESTRICT, SET NULL, SET DEFAULT

Intégrité : Analyse des contraintes

- Les CI conservent la base dans un état cohérent
- Mais les CI sont elles correctes:
  - sont-elles cohérentes entre elles ?

```
CREATE ASSERTION
    AFTER INSERT ON Vins CHECK Degre>12;
CREATE ASSERTION
    AFTER INSERT ON Vins CHECK Degre<11;
```

  - Ne sont-elles pas redondantes ?

```
CREATE ASSERTION
    AFTER INSERT ON Vins CHECK Degre>12;
CREATE ASSERTION
    AFTER INSERT ON Vins CHECK Degre>11;
```

- sont-elles minimales ou peut on les simplifier ?

Les règles dans les systèmes existants

- SQL 3 : standardisation de la syntaxe des déclencheurs (règles)
- Oracle:
  - les règles ne peuvent pas manipuler la relation en cause
  - gestion de la règle au niveau table ou au niveau ligne.

Oracle et les triggers

- Un trigger est un « programme résident » dans une BD
  - Associé à un événement (insertion, modification, suppression)
  - qui porte sur une table
  - Déclenché automatiquement
  - Et programmé en PL/SQL
- Un trigger permet par exemple
  - De définir une politique de sécurité complexe
  - De modifier la valeur d'un attribut en fonction de la valeur d'autres attributs de différentes tables
  - De valider des valeurs d'attribut en fonction d'un calcul complexe relatif aux valeurs d'autres attributs
  - D'archiver des informations automatiquement.

Trigger

- De manière générale un trigger assure:
  - L'implémentation des règles de gestion complexes,
  - La déportation des contraintes au niveau du serveur de données,
  - La programmation de l'intégrité référentielle
  - L'archivage de données
- Cycle de vie d'un trigger
  - Spécifier et analyser les règles de gestion
  - Coder en PL/SQL les règles à implémenter au niveau du serveur de données
  - Compiler chaque trigger (stocké dans la base)
  - Activation/Désactivation des triggers

Les bases de PL/SQL

- Oracle fournit une extension à SQL, sous la forme d'un langage procédural mêlé à des commandes SQL
  - Script PL/SQL pour éteindre les possibilités du SQL
  - Procédure stockée
  - Déclencheur pour la gestion de règles
- Avantages de PL/SQL:
  - Blocs d'instructions très modulaire
  - Indépendance de l'OS et du langage de l'application
  - Intégration à SQL

Structure d'un programme PL/SQL

```
DECLARE
    -- Déclarations des variables, types, exceptions
    …

BEGIN
    -- Code PL/SQL avec (ou pas) des directives SQL
    …

EXCEPTION
    /* Traitement des erreurs du SGBD */
    …

END;
/ -- (symbole nécessaire pour interpréter un bloc en tant que bloc PL/SQL)
```



```
DECLARE
    Déclarations…

BEGIN
    Code PL/SQL…

EXCEPTION
    Gestion des erreurs…

END;
```

Variables et typage en PL/SQL

- Variables scalaires
  - D'un type de base (number, char, boolean, varchar2, date, timestamp, interval, blob, etc.)
  - D'un type prédéfini:
    - Binary_integer (-2^31 à 2^31), natural (0 à 2^31), positive (1 à 2^31), pls_integer
    - Decimal, integer, float

NomVariable [CONSTANT] type [NOT NULL]
[:=|DEFAULT expression];

Ex: dateNaiss DATE:=sys.date();

Variables et typage en PL/SQL

- Exemple très simple:

```
SQL>create table etudiant(      num integer,
                                nom varchar2(20),
                                prenom varchar2(20),
                                dateNaiss DATE,
                                dateInscr DATE);
SQL>declare
      i int;
Begin
      for i in 1..100 loop
            insert into etudiant(num,dateInscr)
            values (i, sysdate);
      end loop;
      Commit;
End;
/
```

Variables et typage en PL/SQL

- Type indirect (%TYPE)
  - Le type d'une variable est déclarée comme le type d'un attribut d'une table existante.

  NomVariable nomTable.nomColonne%TYPE

Ex: name etudiant.nom%type;

- Type complexe implicite (%ROWTYPE)
  - Défini une variable comme une structure dont le schéma est celui d'une table.

Ex: MonEtudiant etudiant%ROWTYPE;

Variables et typage en PL/SQL

- Exemple de déclaration indirect/implicite:

```
Declare
        lastname etudiant.nom%type='Doe';
        firstname etudiant.prenom%type='John';
        MonEtudiant etudiant%ROWTYPE;
        i int;
Begin
        update etudiant
        set nom=lastname, prenom=firstname
        where nom is NULL;

        MonEtudiant.nom='Smith'; MonEtudiant.prenom='John';
        MonEtudiant.DateNaiss=NULL;
        for i in 1..100 loop
                MonEtudiant.num=i;MonEtudiant.DateInscr=sysdate;
                insert into etudiant values MonEtudiant;
        end loop;
        Commit;
End;
/
```

Variables et typage en PL/SQL

- Type structurée RECORD
  - Une variable peut être structurée et explicitement définie à partir d'un type complexe.

Ex: TYPE rec_etudiant IS RECORD (num integer,nom varchar2(20),prenom varchar2(20),dateNaiss DATE,dateInscr DATE);
MonEtudiant rec_etudiant;

- Type tableau TABLE
  - Une variable peut être un tableau dynamique (pas de dimensionnement initial). Un tableau est une table (relationnelle) composé d'un identifiant et d'un contenu.
  - Un ensemble de fonctions permettent de gérer les tableaux.

Ex: TYPE tab_etudiant IS TABLE OF rec_etudiant INDEX BY BINARY_INTEGER;
LesEtudiants tad_etudiant;
…
LesEtudiants(-1)=MonEtudiant;
LesEtudiants(50)=MonEtudiant;
…

Variables et typage en PL/SQL

- Fonctions de gestion des tableaux
  - EXISTS(x) : retourne TRUE si le xieme élément existe
  - COUNT : retourne le nombre d'éléments du tableau
  - FIRST/LAST : retourne le premier/dernier indice du tableau
  - PRIOR(x)/NEXT(x) : retourne l'élément suivant/précédent le xième élément
  - DELETE, DELETE(x), DELETE(x,y) : supprime un ou plusieurs éléments d'un tableau.
  - Fonctions non utilisables dans une commande SQL !
- Opérateurs utiles sur les variables:
  - :=, IS NULL, IS NOT NULL
- Variables de session (= variable globale/statique)
  - Mot clé VARIABLE au début d'un bloc SQL/PLUS
  - Affichage par PRINT

Ex: Variable compteur number;

Begin :compteur:=:compteur+1; End;/

Structure de contrôle PL/SQL

- Structures conditionnelles:

```
If cond then
  instructions;
End if;
```

```
If cond then
  instructions;
Else
  instructions;
End if;
```

```
If cond1 then
  instructions;
Elsif cond2 then
  instructions;
Else
  instructions;
End if;
```

```
Case
  when cond1 then instructions1;
  when cond2 then instructions2;
  …
  else instructions;
End case;
```

Structure de contrôle PL/SQL

- Structures répétitives:

While cond loop
  instructions;
End loop;
	Loop
	  instructions;
	  exit when cond;
	End loop;
		For compt in [reverse] Vinf..VSup loop
		  instructions;
		End loop;

Manipulation de données en PL/SQL

- PL/SQL permet de récupérer les résultats d'une requête dans des variables.
  - Requête SELECT FROM WHERE classique
  - Mot clé INTO pour transférer les valeurs
  - La requête ne doit renvoyer qu'un seul tuple (résultat mono-valué)
  - Une requête SELECT est obligatoirement associé à INTO

Ex:
```
declare
    nb integer;
Begin
    select count(*) INTO nb from etudiant;
    …
End;
/
```

- Pour gérer des résultats multi-valués, on utilise des curseurs (cf. cours IBD)

Manipulation de données en PL/SQL

- Une requête SELECT qui ne renvoie aucun résultat retourne une exception:
  - Erreur NO_DATA_FOUND (ORA-01403)
  - Il faut capturer l'exception
- Insertion,Modification,Suppression
  - Des variables peuvent être utilisées dans les clauses WHERE de la requête
  - Une requête de MAJ ne produit aucune erreur si aucun tuple modifié/supprimé
  - Variables curseurs "implicites" utiles:
    - SLQ%ROWCOUNT = nombre de tuples modifiés/supprimés
    - SQL%FOUND = TRUE si au moins un tuple modifié/supprimé
    - SQL%NOTFOUND = TRUE si contraire SQL%FOUND

Notion de curseur

- Gestion d'une requête multivaluée
  - déclaration du curseur: association à une requête SQL (CURSOR)
  - Ouverture du curseur: préparation du résultat (OPEN)
  - Parcours des tuples résultats (FETCH)
  - Libération des ressources du curseur (CLOSE)
- Exemple: parcourir la table étudiant limitée aux étudiants de plus de 30ans
  - tuple = (nom, prenom)
  - affichage (nom,prenom) de chaque étudiant

Exemple curseur

```
DECLARE
        CURSOR c1 IS SELECT nom, prenom FROM etudiant WHERE age>30;
        UnTuple c1%ROWTYPE;
        BEGIN
                OPEN c1;
                FETCH c1 INTO UnTuple;
                WHILE (c1%FOUND) LOOP
                        DBMS_OUTPUT.PUT_LINE('identité: '||c1.nom||c1.prenom);
                        FETCH c1 INTO UnTuple;
        END LOOP;
        CLOSE c1;
        END;
/
```

- **Idem avec une structure de répétition FOR**

```
BEGIN
        FOR UnTuple IN c1 LOOP
                DBMS_OUTPUT.PUT_LINE('identité: '||c1.nom||c1.prenom);
        END LOOP:
END;
/
```

Curseur paramétré

- Un curseur peut spécifier des paramètres :
  - permet de paramétrer la requête
  - instancié au moment de l'ouverture du curseur

```
DECLARE
        CURSOR c1(p_age IN INTEGER) IS SELECT nom, prenom FROM etudiant WHERE age>p_age;
        UnTuple c1%ROWTYPE;
BEGIN
        OPEN c1(30);
        FETCH c1 INTO UnTuple;
        WHILE (c1%FOUND) LOOP
                DBMS_OUTPUT.PUT_LINE('identité: '||UnTuple.nom||UnTuple.prenom);
                FETCH c1 INTO UnTuple;
        END LOOP;
        CLOSE c1;
        FOR UnTuple IN c1(25) LOOP DBMS_OUTPUT.PUT_LINE('identité:
'||UnTuple.nom||UnTuple.prenom);
        END LOOP:
END;
/
```

Modification via un curseur

- Modifier des tuples accéder par un curseur:
  - verrouillage de la table:

    FOR UPDATE [OF colonne] [WAIT/NOWAIT]
  - utilisation de clause SQL de MàJ
- Exemple: Ajout d'un commentaire aux étudiants >30ans et suppression des étudiants <16ans

```
DECLARE
        CURSOR c1 IS SELECT * FROM etudiant FOR UPDATE OF remarque;
        UnTuple c1%ROWTYPE;
BEGIN
        OPEN c1;
        FETCH c1 INTO UnTuple;
        WHILE (c1%FOUND) LOOP
                IF UnTuple.age>30 THEN UPDATE etudiant SET remarque='Vieux' WHERE CURRENT OF c1;
                ELSIF UnTuple.age<16 THEN DELETE FROM etudiant WHERE CURRENT OF c1;
                END IF;
        FETCH c1 INTO UnTuple;
        END LOOP;
        CLOSE c1;
END;
/
```

Aspect Transactionnel

- Un Bloc PL/SQL supporte la gestion des transactions:
  - COMMIT : termine une transaction
  - ROLLBACK : annule une transaction
  - ROLLBACK TO savepointLabel : annule jusqu'à un point de sauvegarde
- Possibilité d'imbriquer des transactions dans des blocs PL/SQL imbriqués.

Procédure stockée

- Une procédure stockée (ou cataloguée) est une procédure/fonction stockée au sein du SGBD
  - Compilation lors de la définition du bloc PL/SQL "procédure"
  - Recompilation automatique en cas de modification de schéma
  - Appel à l'aide de l'instruction EXECUTE
- Avantage
  - Gestion des droits par le SGBD
  - Traitement au sein du SGBD

Procédure stockée

```
Create procedure nomProcedure(
    nomParam [IN|OUT| IN OUT] typeSQL, …)
    [AUTHID CURRENT_USER|DEFINER]
    [AS PRAGMA AUTONOMOUS_TRANSACTION]
    Zone des variables locales…
BEGIN
    instructions…
END;
/
```

- IN,OUT,IN OUT = paramètres en entrée, sortie, les deux
- AUTHID = change les droits à l'exécution (propriétaire ou utilisateur)
- Une transaction autonome met en pause la transaction principale et lui rend la main après terminaison
- Des procédures imbriquées peuvent être définies dans la zone de déclaration d'une procédure

Procédure stockée

- **Exemple 1**: Corriger les étudiants dont la date d'inscription est < à la date de naissance

```
Create or replace procedure VerifDate
is
Begin
      update etudiants
      set dateNaiss=NULL; dateInscr=NULL;
      WHERE dateInscr<dateNaiss;
End;
/
```

- **Exemple 2**: retourne le nombre d'étudiants inscrits à une date donnée.

```
Create or replace function NbEtud (dinscr in date) return number
Is
Resultat number:=0;
Begin
      select count(*) into resultat from etudiants
      where dateInscr=dinscr;
      return resultat;
End;
/
```

Procédure stockée

- Compilation: automatique lors de la création
- Affichage détaillée des erreurs:

  Show errors (sur la console SQL*PLUS)
- Appel d'une procedure/fonction:

Exemple:

```
Variable nb number;
Begin
    nb:=nbEtud(sysdate);
    verifDate();
    …
    select dateInscr, nbEtud(dateInscr) from etudiants
    groupe by dateInscr;
End;
```

Interaction

- Paquetage DBMS_OUTPUT
  - Pour l'activer: set serveroutput on (sur la console)
  - Procedures disponibles:
    - ENABLE / DISABLE
    - PUT(message IN): affiche un message
    - PUT_LINE(message IN): + saut de ligne
    - GET_LINE(entree OUT): saisi d'un texte

Exemple:

```
DBMS_OUTPUT.ENABLE;
DBMS_OUTPUT.PUT_LINE('Bonjour');
```

Gestion des exceptions

- La gestion des exceptions est classique:
  - Une erreur lève une exception
  - La gestion se fait dans la section EXCEPTION d'un bloc PL/SQL
  - Sinon elle est propagée…

```
EXCEPTION
   WHEN except1 THEN instructions1;
   WHEN except2 OR except3 THEN instructions2;
   WHEN OTHERS THEN instructions;
```

- Exceptions système et Exceptions utilisateurs

Gestion des exceptions

- Exemple:

```
Create function percentEtud(dinscr IN date)
IS
    total number:=0;
    resultat number:=0;
    pas_d_etudiant exception;
Begin
    select count(*) into total from etudiants;
    select count(*)/total into resultat from etudiants
    where dateInscr=dinscr;
    if resultat=0 then raise pas_d_etudiant;
Exception
    when ZERO_DIVIDE then dbms_output.put_line('Aucun Etudiant dans la base!');
    when pas_d_etudiant then dbms_output.put_line('Aucun Etudiant inscrit le
    '||dinscr);
End;
/
```

Gestion des exceptions

- Résultat de requête vide
  - Une requête sans résultat lève une exception NO_DATA_FOUND
  - Comment identifier la requête sans résultat si plusieurs requêtes n'ont pas de résultat

Exemple:

```
dinscr date; dnaiss date;
Requete number;
BEGIN
      requete:=1;
      select dateInscr into dinscr from etudiants where name='Dupont';
      requete:=2;
      select dateNaiss into dnaiss from etudiants where name='Durand';
      …
Exception
      when NO_DATA_FOUND then
                if requete=1 then …
                else …
End;
```

Programmer un déclencheur

- Un trigger est construit sur la base d'une procédure déclenchée par un événement de MAJ:
  - Insert, delete, update sur une table: déclencheurs LMD
  - Create, alter, drop sur un objet (table, index, etc.): déclencheurs LDD
  - Startup/shutdown, ouverture/fermeture session: déclencheurs d'instances
- Nous étudierons les triggers LMD !

Structure d'un trigger LMD

- La définition d'un déclencheur se compose de 7 parties principales
  - Sa commande de création

Create or replace trigger nomTrigger

  - L'information temporelle de déclenchement relatif à l'événement déclencheur

BEFORE | AFTER

  - Le type d'événement déclencheur

DELETE | INSERT | UPDATE [OF nomAttribut,…]
(disjonction possible: delete OR insert OR update)

  - La table sur laquelle porte l'événement

ON nomTable (une seule à la fois !!!)

  - Le mode du trigger: mode ligne / mode table

[FOR EACH ROW]

  - La condition de déclenchement (expression SQL de type clause WHERE)

WHEN condition

  - Le code PL/SQL associé

DECLARE…BEGIN… END

Restriction du code PL/SQL

- Un trigger n'est pas une application complète : le code doit être court
  - Faire appel à des procédures
  - Éviter la récursivité
- Un trigger est un point critique dans lequel la base n'est pas cohérente
  - Pas de gestion de transaction dans le code d'un trigger
  - COMMIT, ROLLBACK, SAVEPOINT sont interdits
- La création d'un trigger l'active automatiquement
  - **Alter trigger nomTrigger DISABLE/ENABLE** pour activer/désactiver un trigger
  - **Alter table nomTable DISABLE/ENABLE ALL triggers** concerne tous les triggers d'une table
  - **Drop trigger** nomTrigger supprime un trigger

Trigger de type table

- La granularité de l'événement est la table:
  - L'ajout, l'insertion, la modification d'un ou plusieurs tuples dans une table
  - Le raisonnement du trigger (code) porte sur l'accès à une table, pas à un tuple particulier

Exemple: Interdiction d'accéder à la table étudiants pendant le weekend

```
Create trigger periodeOK
Before delete or update or insert on etudiants
Begin
    if to_char(sysdate,'DAY') in ('SAMEDI', 'DIMANCHE') then
            raise_application_error(-20101, 'Accès impossible le weekend!);
    end if;
End;
/
```

- Noter la manière de déclencher facilement une Exception utilisateur avec raise_application_error(numErreur, message).

Trigger de type table

- Utilisation de la clause WHEN
  - Pour éviter l'exécution inutile d'un trigger (couteux pour le système)
  - Pour remonter une condition hors du code

Exemple: La clause WHEN s'applique parfaitement à l'exemple précédent

```
Before delete or update or insert on etudiants
When to_char(sysdate,'DAY') in ('SAMEDI', 'DIMANCHE')
Begin
    raise_application_error(-20101, 'Accès impossible le weekend!);
End;
/
```

Trigger de type ligne

- La granularité de l'événement est la ligne:
  - L'ajout, l'insertion, la modification d'un ou plusieurs tuples dans une table déclenche l'évenement pour chaque tuple
  - Le raisonnement du trigger (code) porte sur l'accès à une ligne
  - Accès possible aux informations du tuple accéder
    - :NEW référence le tuple insérée ou modifiée
    - :OLD référence le tuple supprimer ou l'état du tuple avant modification
    - Les directives :NEW et :OLD s'utilisent "comme une table"
- Un trigger en mode ligne pose des verrous sur les données : Impossible d'accéder à la table ayant déclencher l'événement dans le code du trigger !

Trigger de type ligne

Exemple 1: recopie les étudiants supprimés dans une table archive.

```
Create trigger archiverEtudiants
After delete on etudiants
For each row
Begin
   insert into archive
   values(:old.num,:old.name,:old.prenom,:old.inscr);
End;
/
```

Trigger de type ligne

Exemple 2: Refuse l'ajout d'un étudiant qui a été précédemment supprimé (en regardant dans la table archive).

```
Create trigger TestEtudiant
before insert on etudiants
For each row
Declare
    vnum archive%num:=0;
Begin
    select num into Vnum
    from archive
    where num=:new.num;
    if vnum<>0 then raise_application_error(-20100,'Etudiant déjà supprimé!');
    end inf;
Exception
    when NO_DATA_FOUND then dbms_output.putline('OK');
End;
/
```

Le problème des tables mutantes

- Un trigger en mode ligne interdit l'accès à la table ayant déclenché l'événement
  - **La table est dite en mutation (mutating table)**
  - C'est un problème fréquent qu'il faut résoudre à l'aide d'un trigger en mode table!

Exemple: Empêcher l'ajout d'étudiant lorsqu'on a déjà 1000 étudiants dans la base.

```
Create trigger ControlAjout
before insert on etudiants
For each row
Declare
        nb integer;
Begin
        select count(*) into nb from etudiants;
        if nb>1000 then raise_application_error('Trop de monde dans la base !');
        end if;
End;
/
```

Si requête INSERT into etudiants values (….) arrive et qu'il y a déjà 1000 tuples dans la base alors on obtient
Mutating tables (erreur: ORA-04091: table etudiants en mutation)
Et pas le message attendu: le code du trigger a généré une erreur système !

Le problème des tables mutantes

- Résolution des tables mutantes
  - On passe le trigger en mode table
  - On contrôle l'état de la table
  - On génère l'erreur
- Problème
  - En mode table, pas d'accès possible au directives :NEW et :OLD
  - Il faut ruser… ou bien gérer la cohérence dans l'application (Rollback) !

Le problème des tables mutantes

- Réécriture d'un trigger ligne en trigger table

```
Create trigger ControlAjout2
before insert on etudiants
Declare
    nb integer;
Begin
    select count(*) into nb from etudiants;
    if nb=1000 then raise_application_error('Trop de monde dans la base !');
    end if;
End;
/
```

=> Mais ce n'est pas toujours aussi simple !

Le problème des tables mutantes

- Exemple plus difficile (le trigger en mode ligne se justifie)

Un étudiant ne peut pas être ajouté à la base s'il en existe déjà un avec le même nom à la même adresse (on oublie le prénom pour simplifier un peu). La clé primaire est ici son numéro.

```
Create trigger ControlAjout3
before insert on etudiants
For each row
Declare
    nb integer;
Begin
    select count(*) into nb from etudiants e (=> génère Mutating Table)
    where e.nom=:new.nom and e.adr=:new.adr
    if nb>0 then raise_application_error('L'étudiant existe déjà !');
    end if;
End;
/
```

Le problème des tables mutantes

- Réécriture de notre exemple complexe
  - On doit passer en mode table
  - Changer le moment d'évaluation du trigger (on laisse faire)
  - Et modifier la requête pour tester l'état de la base: incohérence ?

```
Create trigger ControlAjout3
after insert on etudiants
Declare
    nb integer;
Begin
    select count(*) into nb from etudiants e1, etudiants e2
    where e1.nom=:e2.nom and e1.adr=e2.adr and e1.np!=e2.np
    if nb>0 then raise_application_error('Plusieurs étudiants possèdent le même
    nom');
    end if;
End;
/
```

=> L'application traitera l'exception ou le système annulera la transaction (rollback)

Trigger LDD

- Les évènements sont des modifications du dictionnaire
  - DROP : suppression d'un objet
  - CREATE : création d'un objet
  - etc
- Syntaxe identique, indiquer la base cible
  - BEFORE|AFTER DROP|CREATE ON nom_base.SCHEMA

```
 Create trigger trg100
after DROP on jouanotf.SCHEMA
Begin
    …
End;
/
```

Optimisation des triggers

- L'exécution d'un trigger doit être court (traitement dans un état transitoire):
  - Tests + exceptions
  - Répercutions de MAJ (attributs calculés ou liés)

- Limiter le nombre de triggers
  - Bien penser les triggers pour regrouper des contrôles de cohérence dans un même trigger
  - Utiliser le regroupement d'événements:

  BEFORE|AFTER insert or update or delete

  et dans le code du trigger:

  IF (INSERTING) THEN …

  IF (UPDATING ('nom_de_colonne') THEN …

  IF (DELETING) THEN …

Ordonnancement de l'exécution des triggers

- Il est très difficile de se baser sur un ordre prévisible d'exécution de triggers dont les événements déclencheurs arrivent au même instant.
- En théorie:
  - Tous les triggers table BEFORE
  - Analyse par le SGBD des lignes affectées par la requête
  - Tous les triggers ligne BEFORE
  - Verrouillage et vérification des Contraintes d'intégrités
  - Tous les triggers ligne AFTER
  - Vérifications des contraintes différées
  - Tous les triggers table AFTER
- En pratique on essaie de ne pas avoir besoin de prédire un ordre précis…

Use case: Le Zoo

- Ressortir les contraintes applicatives:
  - Un employé affecté à un rôle de gardien ne peut pas cumuler ce poste avec un rôle de responsable. On prendra en compte le fait qu'une affectation de responsable peut être ajoutée mais aussi modifiée.
  - Des animaux ne peuvent pas être placés dans une cage non gardée. On prendra en compte le fait que des animaux peuvent être ajoutés dans un cage mais aussi déplacés dans une autre cage.

Use case: Le Zoo

- Implémentation
  - Un employé affecté à un rôle de gardien ne peut pas cumuler ce poste avec un rôle de responsable. On prendra en compte le fait qu'une affectation de responsable peut être ajoutée mais aussi modifiée.

```
create or replace trigger Q2_3
before insert or update of NomE on LesResponsables
for each row
declare
   nb integer;
begin
   select count(*) into nb from LesGardiens where NomE=:new.NomE;
   if (nb > 0) then
      raise_application_error(-20008, 'un gardien ne peut pas etre egalement
responsable');
   end if;
end;/
```

Use case: Le Zoo

- Implémentation
  - Des animaux ne peuvent pas être placés dans une cage non gardée. On prendra en compte le fait que des animaux peuvent être ajoutés dans un cage mais aussi déplacés dans une autre cage.

```
create or replace trigger Q2_2
before insert or update of noCage on LesAnimaux
for each row
declare
   g integer;
begin
   select count(*) into g from LesGardiens where noCage = :new.noCage;
   if (g = 0) then
      raise_application_error(-20002, 'ajout d''un animal dans une cage non gardee');
   end if;
end;
/
```

Use case: Le Zoo

- Ressortir les contraintes applicatives:
  - Des animaux de type différent ne peuvent pas cohabiter dans une même cage. On prendra en compte le fait que des animaux peuvent être ajoutés dans une cage mais aussi déplacés dans une autre cage.
  - Si une cage gardée se retrouve sans animaux, un et un seul gardien peut lui être affecté (nettoyage). On prendra en compte le fait que des animaux peuvent être supprimés mais aussi déplacés dans une autre cage.

Use case: Le Zoo

- Implémentation:
  - Des animaux de type différent ne peuvent pas cohabiter dans une même cage. On prendra en compte le fait que des animaux peuvent être ajoutés dans une cage mais aussi déplacés dans une autre cage.

```
create or replace trigger Q2_1
after insert or update of noCage on LesAnimaux
declare
   nb integer;
begin
      select count(*) into nb from (
          select noCage, count(distinct type) from
             LesAnimaux
             group by noCage
             having count (distinct type) > 1   );
 if (nb > 0) then
            raise_application_error(-20001,
'cohabitation impossible entre animaux
de types different');
       end if;
end;
/
```

Use case: Le Zoo

- Implémentation:
  - Si une cage gardée se retrouve sans animaux, un et un seul gardien peut lui être affecté (nettoyage). On prendra en compte le fait que des animaux peuvent être supprimés mais aussi déplacés dans une autre cage.

```
create or replace trigger Q2_4
after delete or update of noCage on LesAnimaux
declare
    nb integer;
begin
select count(*) into nb from (
          select c.noCage, count(distinct nomE) from
             LesCages c, LesGardiens g
             where c.noCage = g.noCage
                and c.noCage not in (select distinct noCage from LesAnimaux)
             group by c.noCage
             having count (distinct nomE) > 1
    );
 if (nb > 0) then
          raise_application_error(-20003, 'un
gardien maximum pour une cage vide');
       end if;
end;
/
```

Use case: Le Zoo

- Ressortir les contraintes applicatives:
  - Un gardien ne peut pas être retiré de la surveillance d'une cage si les animaux qu'elle contient se retrouvent non gardés. On prendra en compte le fait que des gardiens peuvent être retirés mais aussi affectés à une autre cage.
  - Un employé ne peut pas se retrouver responsable d'une allée dont toutes les cages sont vides. On prendra en compte le fait que des animaux peuvent être supprimés d'une cage mais aussi déplacés dans une autre cage.
  - Lorsqu'un gardien voit l'une de ses affectations modifiées, son acienne affectation doit être conservée dans la table LesHistoiresAff.

Use case: Le Zoo

- implémentation:
  - Lorsqu'un gardien voit l'une de ses affectations modifiées, son acienne affectation doit être conservée dans la table LesHistoiresAff.

```
create or replace trigger Q2_7
before delete or update of noCage on LesGardiens
for each row
declare
   g integer;
begin
   insert into LesHistoiresAff values(:old.noCage, :old.nomE, sysdate);
end;
/
```

Use case: Le Zoo

- Fonctionnalité: Ajouter des animaux à une cage.
  - La cage peut être vide
    - Elle peut ne pas avoir de gardien
  - Les animaux peuvent être incompatibles
  - Choix d'implémentation:
    - Laisser les exceptions remonter

    => Nombreux rollback
    - Faire des tests en amont

    => Assurer l'isolation de la transaction