Bases de Données : Procédures pour le modèle physique (PL/SQL)

Stéphane Devismes

Université Grenoble Alpes

26 août 2020

Plan

Contraintes transactionnelles

Pour les contraintes qui ne peuvent pas être traduites en SQL, on conseille d'utiliser le langage de programmation PL/SQL.

SQL est un langage de gestion de bases de données, **ce n'est pas un langage de programmation** :

SQL ne permet pas d'écrire des tests conditionnels, des boucles, ...

Mais, il est généralement « englobé » dans un langage de programmation.

Pour Oracle, il s'agit de PL/SQL.

PL/SQL

PL/SQL (Procedural Language / Structured Query Language) est un langage (influencé par Ada) qui permet de combiner des instructions procédurales (boucles, conditions...) et des ordres SQL.

Le but de PL/SQL est essentiellement de définir des contraintes (celles qui ne sont pas exprimables en SQL) et de vérifier qu'elles sont satisfaites lors de la modification des données.

On présente d'abord PL/SQL en tant que langage de programmation, ensuite on voit ce qui est spécifique aux bases de données, en particulier **les curseurs et les déclencheurs**. Enfin, on propose des solutions générales pour **traiter les contraintes générées par le modèle relationel**.

Un programme PL/SQL

Un programme PL/SQL est un **bloc**, typiquement sa forme est :

```
DECLARE
      déclarations
BEGIN
      instructions
END;
/
```

où les instructions peuvent comporter à la fois des instructions dans le style de Ada et des ordres SQL.

Attention : Le « / » sur la dernière ligne n'est pas du code PL/SQL. Mais il indique à Oracle que ce qui précède est du code PL/SQL qui doit être exécuté. Le « / » doit être en début de ligne.

Conditions

PL/SQL a des branchements conditionnels de style ADA :

```
IF ... THEN
...
ELSE
...
END IF;
```


Boucles

PL/SQL a plusieurs sortes de boucles, en particulier :

```
WHILE ... LOOP
...
END LOOP;
```


Fonctions

```
CREATE [OR REPLACE] FUNCTION nom-fonction
[(paramètre [IN | OUT | IN OUT] type [, ...])]
RETURN type
IS
      déclarations-de-variables
BEGIN
      instructions
END;
/
```

Affichage

PL/SQL a des fonctionnalités d'entrée-sortie sommaires.

Par exemple, pour les sorties, après avoir tapé :

```
SET SERVEROUTPUT ON
```

On peut alors utiliser des commandes comme :

```
DBMS_OUTPUT.PUT_LINE(' La valeur de la variable x est ' || x);
DBMS_OUTPUT.NEW_LINE;
```


Exceptions

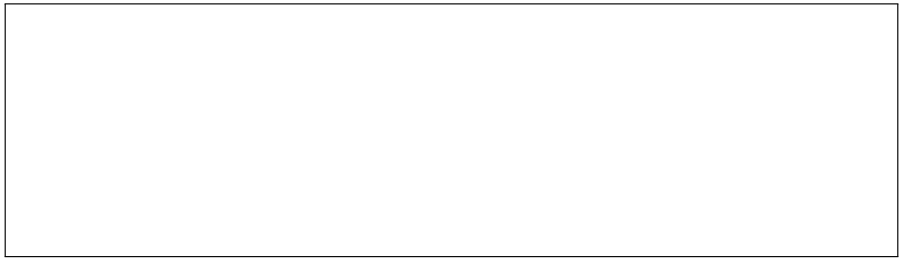
Le traitement des exceptions est décrit à la fin du bloc :

```
EXCEPTION
      traitement des exceptions
END;
/
```

Il y a des exceptions prédéfinies, comme NO_DATA_FOUND lorsqu'une requête renvoie un résultat vide, et des exceptions définies par l'utilisateur dans la zone de déclaration. Les mots-clés sont :

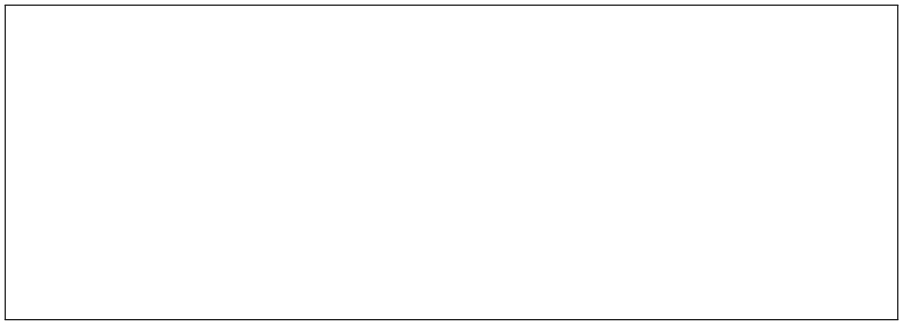
- EXCEPTION pour typer une exception (l'exception doit être déclarée).
- RAISE ... pour lever une exception.
- WHEN ... THEN ... pour traiter une exception.

Exemples (1/3)



Exemples (2/3)

Exemples (3/3)



Exemple

Récupérer une donnée

Pour récupérer des données fournies par une requête SQL retournant une seule ligne, on peut utiliser INTO à l'intérieur de la requête SQL :

```
DECLARE
    a, b, c NUMBER;
BEGIN
    SELECT ... INTO a, b, c
    FROM ... WHERE ...;
    instructions
END;
/
```


Curseurs

On vient de voir comment récupérer le résultat d'une requête, lorsque ce résultat est formé d'une seule ligne : une à plusieurs colonnes comportant exactement une ligne.

Plus généralement, un curseur permet de récupérer et d'**exploiter ligne à ligne** le résultat d'une requête, quel que soit le nombre de colonnes et le nombre de lignes de ce résultat.

En fait, un curseur est une relation obtenue par une requête SQL.

Dans la partie déclaration du programme PL/SQL on déclare le curseur et on le définit comme le résultat d'une requête SQL.

```
DECLARE
   CURSOR MonCurseur IS
   SELECT ...;
```

Parcourir un curseur

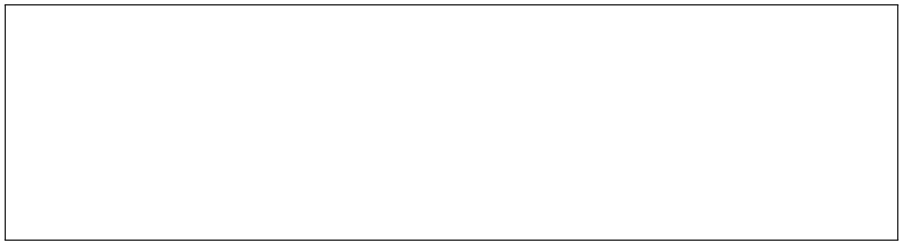
Une variable permettant de parcourir chacunes des lignes du curseur est utilisée dans une boucle FOR.

Il est inutile de déclarer cette variable, PL/SQL la déclare automatiquement avec le bon type, à partir du schéma du curseur.

```
FOR Cur IN MonCurseur LOOP ... END LOOP;
```

Si A est un attribut de la relation MonCurseur, alors dans la boucle l'identificateur Cur.A a pour valeur la valeur de l'attribut A dans la ligne courante.

Exemple



Définition

Un déclencheur (trigger) peut être vu comme une instruction conditionnelle dans laquelle la condition est une modification des données.

Cette instruction est exécutée automatiquement à chaque fois que la condition se produit dans la base de données.

Les déclencheurs PL/SQL permettent d'implanter des contraintes qui ne peuvent pas être exprimées en SQL.

Créer un déclencheur

```
CREATE OR REPLACE TRIGGER nom-du-déclencheur
{BEFORE|AFTER}
{INSERT|DELETE|UPDATE}
ON nom-de-table
[FOR EACH ROW [WHEN...]]
corps-du-déclencheur
```

Supprimer un déclencheur

```
DROP TRIGGER MonDeclencheur;
```


Désactiver/réactiver un déclencheur

Pour un déclencheur :

```
ALTER TRIGGER MonDeclencheur DISABLE;
ALTER TRIGGER MonDeclencheur ENABLE;
```

Pour désactiver puis réactiver tous les déclencheurs définis sur une table :

```
ALTER TABLE MaTable DISABLE ALL TRIGGERS;
ALTER TABLE MaTable ENABLE ALL TRIGGERS;
```


BEFORE et AFTER

- `AFTER` : le corps du trigger est exécuté à la fin de l'instruction (`INSERT` ou `DELETE` ou `UPDATE`), pour compléter les traitements (n'échoue jamais, pas de `RAISE_APPLICATION_ERROR`).

  Exemple : si une cage du zoo devient vide, supprimer les gardiens affectés à cette cage.

- `BEFORE` : le corps du trigger est exécuté au début de l'instruction (`INSERT` ou `DELETE` ou `UPDATE`), pour autoriser ou interdire des traitements (`RAISE_APPLICATION_ERROR`).

  Exemple : si le nombre d'emprunts courant est égal à 5, alors interdire tout nouvel emprunt.

Un déclencheur a 2 parties : un en-tête et un corps.
Lorsqu'il est déclenché, avant de faire quoi que ce soit d'autre le système conserve l'ancienne ligne dans la variable `OLD` et la nouvelle ligne dans la variable `NEW`. Les variables `OLD et NEW` sont des variables de l'en-tête, si le corps veut utiliser ces variables il les fait précéder de « : ».

Erreurs de compilation

En cas d'erreur de compilation, les messages ne sont pas toujours compréhensibles, par exemple en essayant de compiler un déclencheur nommé **MonDeclencheur** :

```
- - Warning : Trigger created with compilation errors.
```

On peut essayer d'obtenir un message plus clair en tapant :

```
SHOW ERRORS TRIGGER MonDeclencheur;
```

Déclencheurs et transactions

Un déclencheur s'exécute dans le cadre d'une transaction.

Il ne peut donc pas contenir d'instruction `COMMIT` ou `ROLLBACK` ou toute instruction générant une fin de transaction implicite (comme un ordre `LDD`).

Exemple : Foot

$Pays(\underline{code}, nom\_pays, continent)$

$Joueurs(\underline{nom}, \underline{prenom}, nbselection, dateNais, id\_pays)$

$Competitions(\underline{id}, edition, nature, organisateur, nbQualifie)$

$Qualifie(\underline{id\_pays}, \underline{id\_compet})$

$Participations(\underline{id\_compet}, \underline{nom}, \underline{prenom})$

$Joueurs(id\_pays) \subseteq Pays(code)$

$Competitions(organisateur) \subseteq Pays(code)$

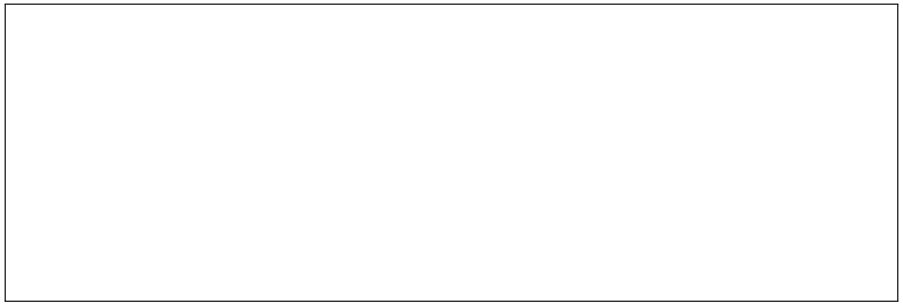
$Qualifie(id\_pays) \subseteq Pays(code)$

$Qualifie(id\_compet) \subseteq Competitions(id)$

$Participations(nom, prenom) \subseteq Joueurs(nom, prenom)$

$Participations(id\_compet) \subseteq Competitions(id)$

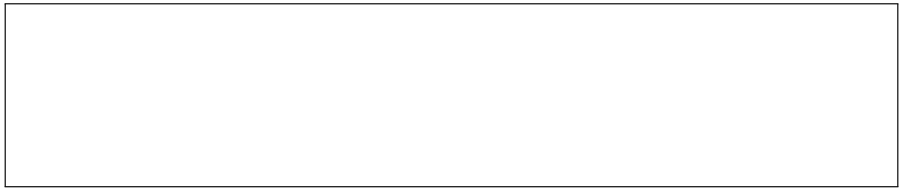
Exemples (1/4)



Exemples (2/4)

Exemples (3/4)

Exemples (4/4)



$R2[K1] \subseteq R1[K1]$

Contrainte de clé étrangère :

```
create table R1(
K1 number(2),
constraint R1_K1 primary key(K1)
);

create table R2(
K2 number(2),
K1 number(2),
constraint R2_K2 primary key(K2),
constraint R2_K1 foreign key(K1) references R1(K1)
);
```


$R1[K1] = R2[K1]$ (1/5)

On diffère la vérification de la clé étrangère dans $R2$ pour pouvoir créer une valeur pour $K1$ d'abord dans $R2$, puis dans $R1$. (Si une valeur de $R2[K1]$ n'existe pas dans $R1$ à la fin de la transaction, une erreur sera générée.)

```
create table R1(
K1 number(2),
constraint R1_K1 primary key(K1)
);

create table R2(
K2 number(2),
K1 number(2),
constraint R2_K2 primary key(K2),
constraint R2_K1 foreign key(K1) references R1(K1)
          deferrable initially deferred
);
```

$R1[K1] = R2[K1]$ (2/5)

La clé étrangère garantit $R2[K1] \subseteq R1[K1]$

Pour garantir $R1[K1] \subseteq R2[K1]$

- Il faut gérer les créations de valeur dans $R1[K1]$ par ajout ou modification : on utilise des triggers.
- Il faut gérer les effacements de valeur dans $R2[K1]$: on utilise un trigger.

$R1[K1] = R2[K1]$ (3/5)

Quand on insère une nouvelle valeur de $K1$ dans $R1$, elle doit exister dans $R2[K1]$.

```
CREATE OR REPLACE TRIGGER K1_insert_R1
BEFORE INSERT
ON R1
FOR EACH ROW
DECLARE
result integer;
BEGIN
SELECT count(*) into result FROM R2 WHERE K1=:NEW.K1;
IF result = 0 THEN
RAISE_APPLICATION_ERROR(-20018,'pb pour ' || :NEW.K1);
END IF;
END;
/
```


$R1[K1] = R2[K1]$ (4/5)

Il faut faire de même quand on crée une nouvelle valeur de $K1$ dans $R1$ lors d'une mise à jour.

```
CREATE OR REPLACE TRIGGER K1_update_R1
BEFORE UPDATE
ON R1
FOR EACH ROW WHEN (OLD.K1 != NEW.K1)
DECLARE
result integer;
BEGIN
SELECT count(*) into result FROM R2 WHERE K1=:NEW.K1;
IF result = 0 THEN
RAISE_APPLICATION_ERROR(-20019,' pb pour ' || :NEW.K1);
END IF;
END;
/
```


$R1[K1] = R2[K1]$ (5/5)

Si on efface toute occurence d'une valeur de $K1$ dans $R2$ lors d'une suppression ou d'une mise à jour, il faut supprimer cette valeur dans $R1$ aussi.

```
CREATE OR REPLACE TRIGGER K1_delete_R2
AFTER DELETE OR UPDATE
ON R2
BEGIN
DELETE FROM R1 WHERE K1 NOT IN (SELECT distinct K1 from R2);
END;
/
```

$K1$ non-nulle dans $R2$

```
create table R2(
K2 number(2),
K1 number(2),
constraint R2_K2 primary key(K2),
constraint R2_K1 foreign key(K1) references R1(K1),
constraint R2_K1bis check (K1 is not null)
);
```


$K1$ est unique $R2$

```
create table R2(
K2 number(2),
K1 number(2),
constraint R2_K2 primary key(K2),
constraint R2_K1 foreign key(K1) references R1(K1),
constraint R2_K1ter unique(K1)
);
```


Chaque valeur de $K1$ apparait au plus 5 fois dans $R2$ (1/2)

Si on a déjà 5 occurrences de la valeur v dans $R2[K1]$, on annule toute nouvelle insertion de v.

```
CREATE OR REPLACE TRIGGER compteK1_insert_2
BEFORE INSERT
ON R2
FOR EACH ROW
DECLARE
result integer;
BEGIN
SELECT count(*) into result FROM R2 WHERE K1=:NEW.K1;
IF result>=5 THEN
RAISE_APPLICATION_ERROR(-20011,' borne deja atteinte pour ' || :NEW.K1);
END IF;
END;
/
```


$K1$ apparait au plus 5 fois dans $R2$ (2/2)

Même chose si on crée une valeur lors d'une mise à jour !

```
CREATE OR REPLACE TRIGGER compteK1_update
BEFORE UPDATE
ON R2
FOR EACH ROW WHEN (OLD.K1 != NEW.K1)
DECLARE
result integer;
BEGIN
SELECT count(*) into result FROM R2 WHERE K1=:NEW.K1;
IF result>=5 THEN
RAISE_APPLICATION_ERROR(-20012,' borne deja atteinte pour ' || :NEW.K1);
END IF;
END;
/
```

$(isC2 = 0) \Rightarrow (A2$ est NULL$)$

Soit $R1(K1, A1, isC2, A2, ...)$.

On contrôle cette contrainte sur insertion et mise à jour.

```
CREATE OR REPLACE TRIGGER ImpliqueR1
BEFORE INSERT OR UPDATE
ON R1
FOR EACH ROW
BEGIN
IF :NEW.isC2 = 0 and :NEW.A2 IS NOT NULL THEN
RAISE_APPLICATION_ERROR(-20018,'pb: A2 doit être nulle');
END IF;
END;
/
```


$R2[K1] \cap R3[K1] = \emptyset$ (1/2)

Toute valeur insérée dans $R2[K1]$ ne doit pas déjà exister dans $R3[K1]$.

```
CREATE OR REPLACE TRIGGER K1_insert_dans_R2
BEFORE INSERT
ON R2
FOR EACH ROW
DECLARE
result integer;
BEGIN
SELECT count(*) into result FROM R3 WHERE K1=:NEW.K1;
IF result>0 THEN
RAISE_APPLICATION_ERROR(-20013,:NEW.K1 ||
      ' existe deja dans R3 ');
END IF;
END;
/
```

Idem pour $R3$

$R2[K1] \cap R3[K1] = \emptyset$ (2/2)

Même chose pour les valeurs créées lors de mises à jour

```
CREATE OR REPLACE TRIGGER  K1_update_dans_R2
BEFORE UPDATE
ON R2
FOR EACH ROW WHEN (OLD.K1 != NEW.K1)
DECLARE
result integer;
BEGIN
SELECT count(*) into result FROM R3 WHERE K1=:NEW.K1;
IF result>0 THEN
RAISE_APPLICATION_ERROR(-20014,:NEW.K1 ||
      ' existe deja dans R3 ');
END IF;
END;
/
```

Idem pour $R3$

$R1[K1] = R2[K1] \cup R3[K1]$ (1/5)

On diffère la vérification de la clé étrangère pour pouvoir créer une valeur pour $K1$ d'abord dans $R2$ et/ou $R3$, puis dans $R1$. (Si une valeur de $R2[K1] \cup R3[K1]$ n'existe pas dans $R1$ à la fin de la transaction, une erreur sera générée.)

```
create table R1(
K1 number(2),
constraint R1_K1 primary key(K1)
);

create table R2(
K1 number(2),
constraint R2_K1 primary key(K1),
constraint R2_K1bis foreign key(K1) references R1(K1)
          deferrable initially deferred);

create table R3(
K1 number(2),
constraint R3_K1 primary key(K1),
constraint R3_K1bis foreign key(K1) references R1(K1)
          deferrable initially deferred);
```

$R1[K1] = R2[K1] \cup R3[K1]$ (2/5)

Les deux clés étrangères garantissent $R2[K1] \cup R3[K1] \subseteq R1[K1]$

Pour garantir $R1[K1] \subseteq R2[K1] \cup R3[K1]$

- Il faut gérer les créations de valeur dans $R1[K1]$ par ajout ou modification : on utilise des triggers.
- Il faut gérer les effacements de valeur dans $R2[K1]$ et $R3[K1]$: on utilise des triggers.

(La méthode que nous allons présenter se généralise à des unions entre plus de deux ensembles.)

$R1[K1] = R2[K1] \cup R3[K1]$ (3/5)

Quand on insère une nouvelle valeur de $K1$ dans $R1$, elle doit exister dans $R2[K1] \cup R3[K1]$.

```
CREATE OR REPLACE TRIGGER K1_insert_R1
BEFORE INSERT
ON R1
FOR EACH ROW
DECLARE
result integer;
BEGIN
SELECT count(*) into result FROM
       (SELECT * FROM R2 UNION SELECT * FROM R3) WHERE K1=:NEW.K1;
IF result = 0 THEN
RAISE_APPLICATION_ERROR(-20018,'pb pour ' || :NEW.K1);
END IF;
END;
/
```


$R1[K1] = R2[K1] \cup R3[K1]$ (4/5)

Il faut faire de même quand on crée une nouvelle valeur de $K1$ dans $R1$ lors d'une mise à jour.

```
CREATE OR REPLACE TRIGGER K1_update_R1
BEFORE UPDATE
ON R1
FOR EACH ROW WHEN (OLD.K1 != NEW.K1)
DECLARE
result integer;
BEGIN
SELECT count(*) into result FROM
       (SELECT * FROM R2 UNION SELECT * FROM R3) WHERE K1=:NEW.K1;
IF result = 0 THEN
RAISE_APPLICATION_ERROR(-20019,' pb pour ' || :NEW.K1);
END IF;
END;
/
```


$R1[K1] = R2[K1] \cup R3[K1]$ (5/5)

Après des suppressions ou des mises à jour dans $R2$, on maintient la partition en supprimant dans $R1$.

```
CREATE OR REPLACE TRIGGER K1_delete_dans_R2
AFTER DELETE OR UPDATE
ON R2
BEGIN
DELETE FROM R1 WHERE R1.K1 NOT IN
       (SELECT K1 FROM R2 UNION SELECT K1 FROM R3);
END;
/
```

Idem pour $R3$