Support d'introduction à PL/SQL

1. Introduction à la surcouche procédurale dans les SGBDRs

Nous allons aborder la notion de surcouche procédurale telle que proposée au sein d'Oracle (PL/SQL) ou encore de PostgreSQL (PL/pgSQL) au travers de trois grands aspects :

- renforcement de la sécurité des bases de données
- supervision et évaluation de la performance
- gestion de l'applicatif au sein même du SGBD

Dans ce sens, les SGBDs relationnels vont proposer la gestion non seulement des données mais aussi des traitements qui peuvent être appliqués à ces données. Ainsi, des structures logiques (procédures stockées, déclencheurs (triggers), assertions, curseurs) vont venir s'ajouter aux structures habituelles telles que les tables, les index ou encore les vues et seront stockées au sein de la base de données. Ces structures vont permettre d'effectuer des calculs sur le contenu de la base ou encore de renforcer l'intégrité de ce même contenu lorsque surviennent des évènements à prohiber (rôle par exemple des déclencheurs). PL/SQL est également mis à profit par les administrateurs de bases de données (DBA) pour tout ce qui concerne la supervision et la maintenance des bases de données dont ils ont la charge. Les exemples traités portent sur le schéma de base Employe dont le schéma est donné ci-dessous. Quelques exemples supplémentaires portent sur la consultation du dictionnaire de données (appropriés au métier de DBA).

1.1 Schéma Employé

```
DEP (num_d number , nom_d varchar(22), adresse varchar(22))
FONCTION (nom_f varchar(20), salaire_min float, salaire_max float)
EMP (num number, nom varchar(15), prenom varchar(15), fonction varchar(20), salaire
    float,
commission float, date_embauche date, n_sup number, n_dep number)
```

Listing 1 – Relations

1.2 Sécurité et intégrité des données : contraintes et déclencheurs

La modification des informations contenues dans une base de données peut s'avérer dangereuse (erreur, donnée de mauvaise qualité, perte de cohérence ...). L'intégrité des BD est traditionnellement assurée notamment via les contraintes intra ou inter-relations qui ont déjà été abordées dans vos cours précédents et qui sont mises en place au niveau du schéma de BD :

- contraintes intra-tables
  - clés primaires (unicité + non nullité)
  - contraintes sur les attributs ou sur les tuples (unicité, non nullité et autres contraintes de domaine)

– contraintes inter-tables : clés étrangères

L'idée va être d'aller au delà et de mettre en place des programmes d'application qui vont vérifier les ordres d'insertion, de modification ou de délétion de tuples afin d'en garantir la cohérence. Il s'agit de considérer les contraintes sur les attributs, les tuples et les relations comme un tout en définissant des assertions (contraintes inter-relations mais indisponibles sous Oracle) et des déclencheurs (ou triggers) (= éléments actifs qui vont jouer leur rôle lors de la venue de certains évènements : insertion, modification ou délétion au sein d'une relation). Les assertions et les triggers vont être des éléments actifs et font partie du schéma de la base au même titre que les relations ou les vues.

Assertion : expression SQL booléenne qui doit toujours être vraie. Un exemple d'assertion (assure que le président est l'employé le mieux payé) vous est donné :

```
create assertion PresidentLePlusRiche check (not exists (select
* from employe where fonction<>'president' and salaire >= (select
salaire from employe where fonction='president')));

drop assertion PresidentLePlusRiche;
```

Listing 2 – Exemple d'assertion

Trigger ou déclencheur : La notion de déclencheur s'inscrit dans de la programmation evènementielle et s'apparente à une action différée. Un déclencheur est vu comme une règle évènement-condition-action ou règle ECA. Le trigger s'active lorsqu'un évènement de type insertion, modification ou délétion survient. Le trigger prend en charge non seulement l'évènement mais aussi le contexte dans lequel il survient (conditions) ainsi que la réponse à y apporter. Il possède dans ce sens les caractéristiques suivantes :

– l'action est réalisée soit avant, soit après l'évènement déclencheur
– l'action est réalisée pour chaque tuple concerné ou bien de manière globale, pour l'ensemble des tuples concernés
– la réponse peut se rapporter aux anciennes comme aux nouvelles valeurs des attributs du tuple affecté lors de l'évènement (insert : nouvelles valeurs, update : anciennes et nouvelles valeurs, delete : anciennes valeurs)
– pour une opération update, l'evènement peut ne concerner qu'un attribut ou un ensemble d'attributs
– Les conditions peuvent être spécifiées dans le corps du programme ou bien dans une clause WHEN. Dans ce sens, il faut la conjonction de l'évènement déclenchant le trigger et de la satisfaction de la condition pour que l'action puisse avoir lieu.

```
create [or replace] trigger [schema.]nom trigger
before|after
delete|insert|update [of colonne [,colonne]...] [or
delete|insert|update [of colonne [,colonne]...]...]
on [schema.]table [[reference old [as] ancien new [as] nouveau] |
new [as] nouveau old [as] ancien]]
[for each row]
[when (condition)]
bloc pl/sql
```

Listing 3 – Un trigger en notation BNF

Un trigger de la base est supprimé par l'ordre : `DROP TRIGGER nomTrigger`. Un trigger peut être activé ou désactivé par les ordres : `ALTER TRIGGER nomTrigger ENABLE` ou `ALTER TRIGGER`

`nomTrigger DISABLE`. Un trigger va être un objet stocké dans la base tout comme une table ou un procédure et les informations le concernant sont disponibles dans la vue `USER_TRIGGERS` du catalogue. Quelques exemples mettant en œuvre différents contrôles, sont présentés ci-dessous :

Création d'un déclencheur Alert Le trigger Alert afffie un message "fin de l'Operation" après toute insertion, modification ou suppression effectuée sur la table EMP.

```
create or replace trigger Alert
after delete or insert or update
on EMP for each row
begin
dbms_output.put_line('fin de l''Operation');
end ;
/
```

Listing 4 – Message

Création d'un déclencheur Ouvrable On souhaite que toute mise à jour de la table EMP ne soit effectuée que durant les jours ouvrables, si la mise à jour a lieu le samedi ou le dimanche, une exception est alors levée avant l'exécution de l'opération.

```
create trigger Ouvrable
before delete or insert or update on EMP
begin
if (to_char(sysdate,'DY')='SAT') or (to_char(sysdate,'DY')='SUN')
then
raise_application_error(-20010, 'Modification interdite le '||to_char(sysdate,'DAY') ) ;
end if ;
end ;
/
```

Listing 5 – Ouvrable

Création d'un déclencheur Verif_salaire Lors de toute insertion ou de toute modification concernant le salaire d'un employé occupant un emploi autre que 'president', on veut vérifier que le salaire est compris entre 1000 et 5000 euros. Si cette condition n'est pas vérifiée, un message approprié est retourné (salaire trop bas ou trop haut). Les nouvelles comme les anciennes valeurs prises par le salaire des employés sont manipulables grâce aux préfixes NEW et OLD (`:`NEW et `:`OLD dans le corps du trigger, NEW et OLD dans la condition when).

```
create trigger verif_salaire
before insert or update of salaire on EMP
for each row
when (new.fonction != 'president')
begin
if ( :new.salaire < 1000 )
then
raise_application_error
(-20022, :new.nom||' n''est pas assez paye.') ;
elsif ( :new.salaire > 5000 )
then
raise_application_error
(-20023, :new.nom||' est trop paye.') ;
end if ;
```

```
end ;
/
```

Listing 6 – Salaire

Création de déclencheurs Before_D et After_D Il s'agit ici d'illustrer les mérités comparés de la syntaxe before ou after au sein du déclencheur. L'ordre before rend possible, la modification des valeurs des attributs des tuples considérés, dans le corps du déclencheur. Avec after, l'action est réalisée après le déclenchement. En conséquence, il est impossible de modifier les valeurs des attributs des tuples concernés. Vous pouvez vérifier en TP, cette réalité, à partir du déclencheur proposé ci-dessous (la version after doit logiquement lever une erreur) :

```
create or replace trigger Before_D
before insert on EMP for each row
begin
:new.nom := upper( :new.nom);
dbms_output.put_line('bravo vous venez d''inserer le tuple '|| :new.nom) ;
end ;
/
```

Listing 7 – Before

```
create or replace trigger After_D
after insert on EMP for each row
begin
:new.nom := upper( :new.nom);
dbms_output.put_line('bravo vous venez d''inserer le tuple '|| :new.nom) ;
end ;
/
```

Listing 8 – After

Création d'un déclencheur Global_update Un déclencheur peut aussi ne s'exécuter qu'une fois, quel que soit le nombre de tuples affectés par l'évènement déclencheur. Le déclencheur Global_update va ainsi insérer un tuple dans une table journal (log) relatant de la mise à jour de la table EMP

```
create table log (
txt varchar2(20), date_maj date, user_maj varchar2(15) ) ;

create or replace trigger Global_update
before update on EMP
begin
insert into log values ('update trigger', sysdate, user) ;
end ;
/

update EMP set nom = nom || 't'
where substr(nom,1,1) = 'M' ;

-- un seul enregistrement dans log quel que soit le nombre de tuples modifiés.

select * from log ;

drop table log ;
```

Listing 9 – MAJ globale

Création d'un déclencheur Historique Un déclencheur peut également s'assurer du type d'évènement qui survient (insertion, mise à jour, suppression) et exécuter des actions adaptées en conséquence. Dans l'exemple suivant, une table historique est définie et va être alimentée par un déclencheur chargé de conserver une trace de toutes les activités opérées sur la table EMP.

```
create table historique (dateOperation date, typeOperation
varchar(15), nomUser varchar(15), AnciennumEmploye number,
NouveauNumEmploye number) ;

create or replace trigger monitor_historique
after insert or delete or update on emp for each row
declare
typeOp varchar(15);
BEGIN
if inserting then
typeOp := 'Insertion';
elsif updating then
typeOp := 'Modification';
elsif deleting then
typeOp := 'Suppression';
end if ;
insert into historique values (sysdate, typeOp,
user, :old.num, :new.num);
end ;
/
```

Listing 10 – MAJ globale

2. Les principes de base de PL/SQL

PL/SQL est également le langage transactionnel d'Oracle, il permet en effet de grouper un ensemble de commandes et de les soumettre au noyau comme un bloc indécomposable de traitements. En résumé, PL/SQL apporte à la fois les traitements procéduraux (structures de contrôle : traitements conditionnels et itérations) et les traitements de transactions. De manière générale, un programme PL/SQL est envisagé comme une unité de traitements et va être appelé un bloc (manipulable à partir des outils d'Oracle et notamment du client SQLPlus).

2.1 Structure d'un bloc

Nous allons tout d'abord nous intéresser à la structure d'un bloc PL/SQL qui se compose de deux parties : une section déclarative (qui débute par DECLARE) et une section corps de programme (qui débute par BEGIN et s'achève par END. Une sous-section optionnelle (qui débute par EXCEPTION) peut être également trouvée dans le corps de programme ; cette section va permettre la gestion des erreurs.

```
[DECLARE
-- declarations variables, curseurs,
-- constantes, exceptions]
```

```
BEGIN
-- commandes exécutables
[EXCEPTION -- gestion des erreurs ]
END ;
/
```

Listing 11 – Syntaxe d'un bloc

Un corps de programme PL/SQL peut comprendre des :
- commandes SQL
- opérateurs de comparaison
- instructions de contrôle conditionnel
- instructions de contrôle itératif
- gestion des curseurs
- gestion des erreurs

Tout bloc PL/SQL doit se terminer par une ligne ne comportant qu'un "." ou encore un "/" (dans ce dernier cas, la vérification du code est alors suivie d'une exécution immédiate).

2.2 Types de données

PL/SQL offre des types de données scalaires (char, varchar, long, boolean, integer, number, float, etc) ou composés (record et table de la couche relationnelle-objet). Il est également possible de définir une variable en la déclarant du même type qu'une colonne d'une table (nom table.nom attribut%type) ou d'une ligne d'une table (nom table.nom%rowtype).

2.2.1 Illustration

Ce programme n'a pas d'intérêt en soi, mais donne quelques perspectives sur la manipulation d'instructions simples en PL/SQL.

```
set serveroutput on
declare
x number ;
y number ;
compteur number ;
-- constante
z constant integer := 4 ;
-- date du jour auj date default sysdate
aujourdhui date := sysdate ;
begin
-- affectation
x :=2 ;
-- traitement conditionnel
if x>0 then y :=x ;
else y :=-x ;
end if ;
dbms_output.put_line('x et y valent '||x||' et '||y) ;
-- traitement itératif
for compteur in 1..z
loop
dbms_output.put_line('histoire d''essayer');
end loop ;
end ;
/
```

Listing 12 – Syntaxe d'un bloc

2.3 Les paquetages ("packages") standards

Il est possible d'appuyer le travail de programmation sur un ensemble de packages prédéfinis. Seul, le package DBMS_OUTPUT est abordé ici. Pour afficher à l'écran divers messages ou variables, il est nécessaire d'utiliser la librairie DBMS_OUTPUT qui fournit les méthodes d'écriture des diverses variables sur le périphérique de sortie standard [1] Les procédures PUT et PUT_LINE permettent d'écrire dans le buffer (avec possiblité de retour à la ligne).

```
PROCEDURE PUT(A VARCHAR2)
PROCEDURE PUT(A NUMBER)
PROCEDURE PUT(A DATE)
PROCEDURE PUT_LINE(A VARCHAR2)
PROCEDURE PUT_LINE(A NUMBER)
PROCEDURE PUT_LINE(A DATE)
```

Listing 13 – Signatures PUT

Il est possible de concaténer les diverses variables par ||.

2.4 Les curseurs

Les curseurs (CURSOR) permettent de traiter les résultats d'une requête en vue d'analyses ou d'interprétations diverses. Ils peuvent être comparés à des tables temporaires.

2.4.1 Les curseurs implicites

Un ordre SQL peut ne retourner qu'un n-uplet, il n'est pas nécessaire de définir un curseur de manière explicite, on parle alors de curseur implicite.

```
declare
name emp.nom%type;
numero emp.num%type;
begin
select num, nom into numero, name from emp where num=79 ;
dbms_output.put_line('j''affiche le numero et nom '||numero||' '||name) ;
end ;
/
```

Listing 14 – Curseur implicite

2.4.2 Les curseurs explicites

Dès que l'ordre SELECT SQL retourne une réponse comprenant plus de deux n-uplets, il faut se résoudre à définir des curseurs de manière explicite. un curseur est alors déclaré dans la section DECLARE, le curseur est nommé et associé à une requête, à l'exemple de cursor `mon_curseur is select nom, num, fonction from emp;` Il reste alors à ouvrir ce curseur dans le corps du programme par la commande OPEN, puis à faire défiler l'extraction des données par la commande FETCH en utilisant une boucle et une instruction conditionnelle pour quitter la boucle. Une fois les traitements finis, il suffit de fermer le curseur (CLOSE). Des attributs de curseur permettent d'accéder à la zone de contexte du curseur :

- %FOUND : retourne NULL quand le curseur est ouvert mais non lu, TRUE quand une ligne est renvoyée après lecture (FETCH), FALSE sinon,

1. Important : positionner auparavant l'environnement par la commande SET SERVEROUTPUT ON

– %NOTFOUND : retourne NULL quand le curseur est ouvert mais non lu, TRUE quand aucune ligne n'est renvoyée après lecture, FALSE dans les autres cas.
– %ISOPEN : retourne TRUE si le curseur est ouvert, FALSE sinon.
– %ROWCOUNT : retourne un entier, 0 si le curseur est ouvert mais non lu, puis la valeur s'incrémente de 1 après chaque lecture.

```
declare
cursor mon_curseur is
select num, nom, fonction
from EMP ;
numero emp.num%type;
fnom emp.nom%type;
qualif emp.fonction%type;
begin
open mon_curseur ;
loop
fetch mon_curseur into numero, fnom, qualif ;
exit when mon_curseur%notfound;
dbms_output.put_line(numero||' '||fnom||' '||qualif);
end loop ;
close mon_curseur ;
end ;
/
```

Listing 15 – Curseur explicite

2.4.3 Les curseurs paramétrés

L'exemple ci-dessous affiche les employés ayant une fonction de designer

```
declare
t emp emp%rowtype;
cursor c (v_fonction in varchar)
is
select * from emp
where fonction=v_fonction ;
begin
for t emp in c('designer')
loop
dbms_output.put_line('designer : '||t emp.num||' '||t emp.nom||' '||t emp.salaire);
exit when c%notfound;
end loop ;
end ;
/
```

Listing 16 – Curseur avec argument

2.4.4 Utilisation simplifiée des curseurs

Le curseur for ...loop remplace les instructions open, fetch et close et déclare implicitement un index pour en faciliter le parcours.

```
declare
cursor mon_curseur is
select num, nom, fonction
from EMP ;
```

```
begin
For emp_rec in mon_curseur
loop
dbms_output.put_line(emp rec.num||' '||emp rec.nom||' '||emp rec.fonction);
end loop ;
end ;
/
```

Listing 17 – Simplification

2.5 Les exceptions

La gestion des exceptions dans un programme PL/SQL peut se faire en trois temps :

1. Déclarer une exception :
2. Lever une exception
3. Traiter une exception

Par exemple, si l'on reprend un des exemples précédents :

```
declare
name emp.nom%type;
numero emp.num%type;
mon_exception exception;
begin
select num, nom into numero, name from emp where num=79 ;
if name is null then raise mon_exception ;
else dbms_output.put_line('j''affiche le numero et nom '||numero||' '||name) ;
end if ;
exception
when mon_exception then raise_application_error(-20100,'le salarie
est non reference');
end ;
/
```

Listing 18 – Exemple exception

Un certain nombre d'exceptions sont prédéfinies comme NO_DATA_FOUND, CURSOR_ALREADY_OPEN, ZERO_DIVIDE ou encore VALUE_ERROR et peuvent s'utiliser "directement". La fonction RAISE_APPLICATION_E erreur dans les -20000, message à afficher) est de plus fort utile pour la gestion des exceptions.

2.6 Les conditionnelles

La syntaxe sera de type if ...then ...else ...end if ; Les if imbriqués sont possibles pour traiter les choix multiples (else if ou encore elsif) Une

```
IF sal < 1000 THEN prime:=300;
ELSIF sal < 1500 THEN prime:=300;
ELSIF sal < 2000 THEN prime:=200;
ELSE prime:=100;
END IF;
```

Listing 19 – Illustration IF

2.7 Les structures itératives

Plusieurs types d'instructions sont disponibles :
- boucle infinie loop séquence-de-commandes end loop avec une instruction exit pour sortir de la boucle
- for compteur in borneInf ...borneSup loop séquence-de-commandes end loop
- while condition-plsql loop séquence-de-commandes end loop

Des illustrations sont fournies : à supposer que l'on veuille insérer onze tuples dans la table EMP au moyen d'une boucle.

```
declare
i integer :=20000 ;
begin
while i <= 20010
loop
INSERT INTO emp (nom,num) VALUES ('pierrot',i);
i :=i+1 ;
end loop ;
---------------------------------
begin
for i in 20000..20010
loop
INSERT INTO emp (nom,num) VALUES ('pierrot',i);
end loop ;
---------------------------------
declare
i integer :=20000 ;
begin
loop
exit when i>20010 ;
INSERT INTO emp(nom,num) VALUES ('pierrot',i);
i :=i+1 ;
end loop ;
end ;
```

Listing 20 – Exemples boucle

2.8 Les procédures et les fonctions

Un des intérêts de PL/SQL porte notamment sur la possibilité de définir des objet fonctions qui seront cataloguées au niveau du méta-schéma et pourront de sorte être partagées par les utilisateurs (minimisation du code). De plus, les procédures comme les fonctions sont compilées et stockées dans la base de données, améliorant les performances du système à l'exécution. Les fonctions (procédures) sont consultables dans les vues du méta-schéma user_objects ou user_procedures et peuvent être supprimées à l'identique des autres objets via drop function nom_fonction (drop pro- cedure nom_procedure). Une fois, les procédures et les fonctions définies, elles peuvent être appelées dans un bloc PL/SQL.

```
create [or replace] procedure [schema.]nom_procedure [(liste d'arguments)]
{is|as}
bloc pl/sql
create [or replace] function [schema.]nom_fonction [(liste d'arguments)]
return type
{is|as}
bloc pl/sql
Arguments :
in : variable passée en entrée
```

```
out : variable renseignée par la procédure et renvoyée à l'appelant
in out : variable passée en entrée, renseignée par la procédure puis renvoyée
à l'appelant
```

Listing 21 – Construit PL/SQL

2.8.1 Quelques illustrations

conversion euros à francs : procédure et programme principal : une procédure de conversion monétaire

```
create or replace procedure P_CEF (euros IN number, francs OUT number) is
begin
francs :=euros*6.559;
end ;
/
declare
temp number ;
begin
P_CEF(100,temp);
DBMS_OUTPUT.PUT_LINE(temp);
end ;
/
```

Listing 22 – Proc et PP

conversion euros à francs : fonction et programme principal : une fonction de conversion monétaire

```
create or replace function F_CEF (euros in number) return number
is
francs number ;
begin
francs :=euros*6.559;
return (francs) ;
end ;
/
declare
result number ;
begin
result := F_CEF(100) ;
DBMS_OUTPUT.PUT_LINE(result);
end ;
/
```

Listing 23 – Fonction et PP

La fonction F_CEF peut être utilisée maintenant dans une requête SQL : `select salaire, f_cef(salaire) as salaire_en_francs from emp;`

Plus d'adéquation avec le monde des bases de données : une procédure qui permet d'effectuer une augmentation de salaire sur la table EMP

```
CREATE OR REPLACE PROCEDURE augmentation (numEmp IN NUMBER, taux IN
NUMBER) IS
```

```
salary number ;
comm number ;
BEGIN
select salaire, commission into salary, comm from emp where num =
numEmp ;
IF salary < 1200 AND comm IS NULL
THEN
update emp set salaire = salaire*(1+3*taux) where num = numEmp ;
ELSIF salary BETWEEN 1200 AND 2600 AND comm IS NULL
THEN
update emp set salaire = salaire*(1+2*taux) where num = numEmp ;
ELSIF salary > 2600 AND comm IS NULL
THEN update emp set salaire = salaire*(1+taux) where num = numEmp ;
END IF ;
EXCEPTION
WHEN NO DATA FOUND THEN raise_application_error(-20100, numEmp || '
non reference');
END ;
/
```

Listing 24 – Monde des BD

```
declare
begin
augmentation(78,0.5);
dbms output.put line('accompli');
end ;
/
```

Listing 25 – Monde des BD et PP