Quatrième partie

Focus sur Oracle

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

©2008-2009 - JACQUES THOORENS - HTTP://WWW.THOORENS.NET

Chapitre 12

Programmation PL/SQL

SQL est un standard employé depuis plus de 20 ans. Il se base sur l'algèbre relationnelle. Nous avons vu qu'au-delà du standard accepté par tous les systèmes actuels, un certain nombre de commandes ont été ajoutées par chaque éditeur pour faciliter les manipulations et pour tirer parti de certaines innovations par lesquelles on peut se démarquer de la concurrence.

Lorsqu'on veut construire un système très performant, il devient nécessaire de permettre à un certain nombre de traitements de s'effectuer à l'intérieur de la base de données sans passer par des échanges avec des clients. Cela se justifie par

- des impératifs de performances : un échange avec un programme-client mobilise souvent des ressources du réseau
- des impératifs de sécurité : le traitement interne nous met à l'abri de l'exécution d'un programme externe mal conçu ou ne prenant pas en compte les dernières modifications.

Chaque éditeur propose donc son langage de programmation interne original pour réaliser ces traitements. Il est évidemment possible d'utiliser ce langage pour exécuter des scripts au départ d'un client. La distinction entre différents programmes tient surtout à l'endroit où ils sont enregistrés et à la manière dont ils démarrent :

les scripts sont des programmes stockés sur support externe et lancés généralement par des administrateurs. Ils concernent le plus souvent des opérations ponctuelles comme la création des tables ou des données ;

les procédures et fonctions stockées (*store procedures*) figurent sur le serveur et sont lancées par des appels explicites à la console, dans des scripts, d'autres procédures ou dans des déclencheurs.

les déclencheurs (*triggers*) figurent également sur le serveur mais sont automatiquement lancés lorsque surviennent certains événements (modification des données ou événements divers comme la connexion d'un utilisateur).

La suite de ce chapitre va décrire sommairement le langage PL/SQL, une extension propriétaire de SQL par Oracle. Il est employé par plusieurs dizaines de milliers de programmeurs dans le monde. Rappelons qu'il ne peut tourner que sur un serveur. Oracle propose également l'emploi du langage Java pour l'écriture de certaines procédures. La nécessité de présenter les bases du langage avant son utilisation à l'intérieur d'Oracle rend son exposition incompatible avec le temps dont nous disposons. Il faut néanmoins noter que Java joue un rôle croissant dans l'évolution d'Oracle :

- dans la version 10, le serveur Web Apache a été remplacé par un serveur écrit en Java.
- les programmes d'installation du serveur, les logiciels destinées aux administrateurs sont également des programmes Java.
- les procédures stockées peuvent être rédigées pour la plupart en Java.
- l'interface de gestion privilégiée est maintenant SQL-Developer, une application Java.

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

– Oracle distribue également gratuitement un IDE de programmation Java totalement adapté à son environnement : *JDeveloper*.

Dans le cadre d'un projet d'entreprise, on peut envisager d'utiliser Java au niveau du serveur de données et au niveau du serveur d'application. Cela permet de rentabiliser un maximum les formations et l'expertise en Java.

12.1 Structure d'un programme PL/SQL

Le langage PL/SQL présente un mélange de caractéristiques procédurales (découpages en sous-routines, passages de paramètres, déclarations de variables, alternatives et boucles) et ensemblistes (les instructions DML de SQL). Il rappelle par certains côtés le langage Pascal, auquel il emprunte les marqueurs de blocs `BEGIN` et `END`. Il est assez proche de Transact SQL, le langage équivalent de Microsoft, et du dialecte DSQL d'InterBase.

Les exemples donnés dans cette section fonctionnent comme des scripts, les procédures et les triggers étant plus lourds à mettre en œuvre.

12.1.1 Structure de bloc

Un bloc est une structure récursive qui comporte plusieurs parties :

– une zone optionnelle de déclaration de variables (suivant le mot `AS` ou commençant par le mot `DECLARE` dans le cas des triggers) ;
– une zone d'instructions commençant par le mot `BEGIN` ;
– une zone optionnelle de traitement des erreurs commençant par le mot `EXCEPTION` ;
– une marque de fin de bloc composée du mot `END` suivi d'un point-virgule.

12.1.2 Définition des variables

Comme dans les langages modernes, les variables doivent être explicitement déclarées. Certaines le sont au moment de la définition des paramètres (uniquement dans le cas des procédures et des fonctions stockées). La déclaration des variables se fait dans une section de bloc. Cette section commence par les mots `IS`, `AS` ou `DECLARE` et est implicitement terminée par le mot `BEGIN`. Chaque déclaration se compose d'un nom de variable suivi de son type et d'un point-virgule. Le nom de la variable obéit aux règles habituelles des langages de programmation, mais on conseille d'utiliser les préfixes `v_` et `c_` pour les variables normales et celles qui sont destinées à contenir la valeur d'une colonne de table. Les types sont ceux de SQL (ou plus exactement ceux d'Oracle). Notons que les variables de type `CHAR()` sont comblées avec des espaces, à la différence des `VARCHAR2()` qui ne contiennent que les caractères explicitement affectés.

```
DECLARE
   v_Age NUMBER(3);
   v_Nom VARCHAR2(40);
   v_Prenom CHAR(15);
```

Quelques types particuliers sont à mentionner :

– le type `BOOLEAN` offre la possibilité de mémoriser des valeurs de vérité (`TRUE` ou `FALSE`).
– le type `ROWID` permet de mémoriser un identifiant unique de ligne d'une table (voir chapitre précédent pour une utilisation).

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

– le type `%ROWTYPE` s'utilise avec un nom de table et correspond à une ligne complète de table. Par exemple,

```
v_enreg Clients%ROWTYPE;
```

crée une variable complexe comportant les champs de la table *Clients*, auxquels on accède au moyen de la notation traditionnelle :

```
v_enreg.Nom := 'Dupont';
v_enreg.Prenom := 'Albert';
```

– le type `%TYPE` s'emploie avec un nom de colonne et crée une variable de type identique à celui de la colonne du même nom. Cela présente l'avantage d'avoir un programme susceptible de s'adapter aux éventuelles modifications de la table. Avouons aussi que cela permet au programmeur de gagner du temps !

```
DECLARE
   c_nom Clients.Nom%TYPE;
   c_prenom Clients.Prenom%TYPE;
```

D'autres types, notamment les curseurs, seront évoqués plus loin.

12.1.3 Instructions d'affectation

PL/SQL dispose d'un opérateur d'affectation semblable à celui de Pascal (:=). Les types complexes comme les dates bénéficient d'une conversion implicite.

```
v_Verifie = TRUE;
v_Duree = 5;
v_DateEntre = '15-mai-1997';
```

Il existe une forme d'affectation beaucoup plus puissante qui utilise une instruction `SELECT` modifiée. Elle implique que le résultat de la requête possède une et une seule ligne[1].

```
DECLARE
   c_idClient INT;
   c_Nom VARCHAR(20);
BEGIN
  SELECT idClient,Nom INTO c_idClient,c_Nom
  FROM CLient
  WHERE idclient=1001;
END;
```

12.1.4 Affichage

PL/SQL n'a pas été conçu pour écrire un programme de traitement de texte et il n'affiche donc pas grand chose d'autre qu'un rassurant 'Appel terminé' quand tout va bien. Lors de la mise au point d'un programme, on a parfois besoin de vérifier certains résultats à l'écran pour trouver une erreur. Oracle dispose de routines qui l'autorisent, mais il faut activer un paramètre dans la console ou la fenêtre du navigateur pour que les messages nous parviennent.

[1] Si on veut manipuler des requêtes à plusieurs lignes, il devient nécessaire de définir des *curseurs* dont on parlera plus loin dans ce chapitre.

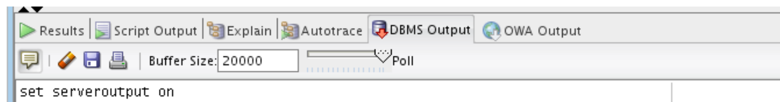
©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

```
set serveroutput on
```

L'appel de routine système suivant placé dans une sous-routine affiche un petit message à l'écran :

```
dbms_ouput.put_line('Tout␣va␣bien!!!');
```

Pour obtenir les messages dans SQL Developer, il convient d'activer l'affichage de messages DBMS.Output. Au-dessus de la fenêtre de résultats, on trouvera une petite icône permettant cette activation.



12.1.5 Traitement des erreurs

Comme la plupart des programmes PL/SQL s'exécute sur le serveur, il est indispensable de prévoir une gestion des erreurs bien pensée afin de ne pas avoir de surprises. Heureusement, bon nombre d'erreurs sont détectées lors de l'écriture du programme : noms de table ou de champ incorrects, erreurs de syntaxe et autres. Il reste à traiter les erreurs d'exécution.

Nous avons vu que tout bloc pouvait avoir une zone réservée au traitement des erreurs. Elle commence par le mot `EXCEPTION` et autorise un traitement en fonction du type d'erreur constaté. Le tableau suivant reprend les principaux codes d'erreur.

numéro	Constante	numéro	Constante
ORA-00001	DUP_VAL_ON_INDEX	ORA-00051	TIMEOUT_ON_RESOURCE
ORA-01001	INVALID_CURSOR	ORA-01012	NOT_LOGGED_ON
ORA-01017	LOGIN_DENIED	ORA-01403	NO_DATA_FOUND
ORA-01410	SYS_INVALID_ROWID	ORA-01422	TOO_MANY_ROWS
ORA-01476	ZERO_DIVIDE	ORA-01722	INVALID_NUMBER
ORA-06500	STORAGE_ERROR	ORA-06501	PROGRAM_ERROR
ORA-06502	VALUE_ERROR	ORA-06504	ROWTYPE_MISMATCH
ORA-06511	CURSOR_ALREADY_OPEN	ORA-06530	ACCESS_INTO_NULL
ORA-06531	COLLECTION_IS_NULL	ORA-06532	SUBSCRIPT_OUTSIDE_LIMIT
ORA-06533	SUBSCRIPT_BEYOND_COUNT		

Nous allons doter notre petit script de test d'affectation d'un traitement d'erreur en distinguant les cas où il n'y a pas de données, de ceux où il y a trop de données. Enfin, nous considérerons toutes les autres erreurs (par exemple, une valeur caractère dans la comparaison). On notera la syntaxe : `WHEN` *`type d'erreur`* `THEN`

```
DECLARE
   c_idClient INT;
  c_Nom VARCHAR(20);
BEGIN
  SELECT idClient,Nom INTO c_idClient,c_Nom FROM CLient
  where idclient>1;
EXCEPTION
   when NO_DATA_FOUND then
```

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

```
      dbms_output.put_line('Aucune donnée ne répond à la question');
    when TOO_MANY_ROWS then
      dbms_output.put_line('Plusieurs données répondent à la question');
    when OTHERS then
      dbms_output.put_line('Erreur inconnue');
END;
```

Remarquons qu'il est possible de définir soi-même ses propres types d'erreur, mais ce mécanisme dépasse le niveau de cette petite introduction.

12.1.6 Alternatives

PL/SQL dispose d'une instruction structurée permettant des tests multiples :

```
DECLARE
   v INT;
BEGIN
 v := 0;
 IF v < 0 THEN
    dbms_output.put_line('négatif');
    dbms_output.put_line('vraiment négatif');
 ELSIF v > 0 THEN
    dbms_output.put_line('positif');
 ELSE
    dbms_output.put_line('nul');
 END IF;
END;
```

Il peut y avoir plusieurs instructions dans les parties alternatives. L'emploi de `ELSIF` permet de ne pas imbriquer plusieurs structures alternatives (un seul `END IF`).

12.1.7 Boucles

On retrouve les grands classiques : boucle infinie avec une instruction `EXIT WHEN` un peu primitive, une boucle à test initial et une boucle avec compteur (sa variable n'a besoin ni d'être déclarée ni incrémentée, le pas est toujours de 1 et le parcours croissant).

```
DECLARE
   v_I int;
BEGIN
  v_I := 1;
  -- boucle à sortie quelconque
  LOOP
    dbms_output.put_line('Test loop/exit when '||v_I);
  EXIT WHEN v_I=10;
    v_I := v_I+1;
  END LOOP;
  -- boucle à test initial
  WHILE v_I <= 20
  LOOP
    dbms_output.put_line('Test while/loop '||v_I);
    v_I := v_I+1;
```

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

```
    END LOOP;
    -- boucle avec compteur
    FOR v_I2 IN 1..10
    LOOP
      dbms_output.put_line('test␣for␣'||v_I2);
    END LOOP;
  END;
```

12.2 Les déclencheurs ou *triggers*

Après avoir vu les rudiments du langage, nous sommes en mesure de définir le premier type de programmes exécutés sur le serveur. Les déclencheurs (*triggers*) sont des procédures automatiques et non paramétrées qui se déclenchent lors des événements notamment associés à une table donnée. Pour définir un déclencheur, il faut donc :

- lui donner un nom
- l'associer à une table précise
- spécifier l'événement ou les événements qui le déclenchent
- signaler si le code doit s'exécuter avant ou après l'événement
- spécifier les instructions qui vont s'exécuter

Nous verrons ailleurs que le déclencheurs peuvent s'appliquer à d'autres types d'événement. Nous allons d'abord examiner ceux qui concernent les tables, historiquement les plus anciens et encore les plus répandus dans l'ensemble des autres SGBD.

La syntaxe est donnée dans l'encadré suivant, sachant que les mots soulignés sont facultatifs, que `AFTER` et `BEFORE` sont alternatifs et que l'on peut choisir de placer le déclencheur sur une, deux ou trois actions.

```
CREATE OR REPLACE TRIGGER NomDéclencheur
AFTER|BEFORE INSERT OR UPDATE OR DELETE
ON Nom_de_la_table
FOR EACH ROW
DECLARE
Déclarations de variables
BEGIN
...
END;
```

La phrase `FOR EACH ROW` permet de spécifier si le traitement doit s'appliquer une seule fois (en cas d'insertion, modification ou suppression multiples) ou pour chaque ligne individuellement.

Exemple n° 1 : compter le nombre d'insertions dans une table

Le but est de compter combien d'insertions sont opérées dans une table, indépendamment d'éventuelles suppressions. On va utiliser la table *TestTrigger* et la table *CompteurTestTrigger* pour mémoriser le compteur.

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

```
CREATE TABLE TestTrigger
(idTT numeric(5) primary key);
CREATE TABLE CompteurTestTrigger
(Compteur numeric(5) primary key) ;
INSERT INTO CompteurTestTrigger VALUES (0);
```

Le traitement va se limiter à incrémenter le contenu du champ *COMPTEUR* . `CREATE OR REPLACE` permet de garder la même syntaxe pour la création et la modification d'un trigger. On peut ausssi utiliser `ALTER TRIGER`, mais il ne permet pas de changer toutes les caractéristiques du trigger.

```
CREATE OR REPLACE TRIGGER COMPTERINSTT
AFTER INSERT ON TESTTRIGGER
AFTER FOR EACH ROW
BEGIN
        UPDATE CompteurTestTrigger SET Compteur = Compteur + 1;
END;
```

Voici ce que donnent trois insertions suivies d'une insertion de plusieurs lignes au départ d'une autre table.

```
INSERT INTO TestTrigger  VALUES (-2);
INSERT INTO TestTrigger  VALUES (-1);
INSERT INTO TestTrigger  VALUES (0);
INSERT INTO TestTrigger SELECT N FROM Etudiants
SELECT * FROM CompteurTestTrigger;
```

La dernière requête affiche bien la valeur 592 (les trois premières insertions + 589 insertions correspondants aux numéros de tous les étudiants dans la table). On voit donc que la dernière insertion, qui insère 589 lignes dans la table, déclenche le trigger 589 fois. Si on n'avait pas précisé `FOR EACH ROW`, le résultat aurait été 4.

Exemple n°2 : historisation du salaire des employés

Dans une entreprise, on veut savoir qui manipule le salaire des employés dans la table *Salaires*. Nous allons écrire un déclencheur qui notera le moment, l'utilisateur, le numéro de l'employé, les valeurs avant et après la modification et le type de modification opérée. Nous devrons introduire de nouvelles notions pour cela :

- distinguer l'opération déclenchante à l'aide des conditions spéciales `IF INSERTING`, `IF DELETING` et `IF UPDATING`[2].
- déterminer les valeurs introduites, modifiées ou supprimées à l'aide des notations `:new.`*NomChamp* et `:old.`*NomChamp* (les deux sont utilisées en cas de modification).

Le déclencheur ainsi conçu suppose que les modifications d'un même employé ne se feront pas dans la même seconde (au risque de déclencher une erreur de clé double et par là même d'annuler la modification). Au cas où cela poserait problème, il faudrait proposer une autre clé primaire pour la table d'historisation.

[2]On aurait pu définir trois triggers, un par opération, mais on y aurait perdu la vision d'ensemble que permet notre version.

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

```
CREATE TABLE SALAIRES
(IDEMPLOYE NUMBER(5) NOT NULL,
 SALAIRE NUMBER(7, 2),
 CONSTRAINT pkSalaires PRIMARY KEY(IDEMPLOYE)
);

CREATE TABLE HISTO_SALAIRE
(IDEMPLOYE NUMBER(5) NOT NULL,
 DATEMODIF TIMESTAMP NOT NULL,
 OPERATEUR VARCHAR2(20),
 OPERATION CHAR(3),
 ANCIEN NUMBER(7, 2),
 NOUVEAU NUMBER(7, 2),
 CONSTRAINT PK_HISTO_SALAIRE PRIMARY KEY(IDEMPLOYE,DATEMODIF)
);

CREATE OR REPLACE TRIGGER TR_HISTO_SALAIRE AFTER
INSERT OR UPDATE OR DELETE
ON SALAIRES
FOR EACH ROW
DECLARE

   v_idEmploye NUMERIC(5);
   v_DateModif DATE :=SYSDATE;
   v_OperateurVARCHAR(30):=USER;
   v_Operation CHAR(3);
   v_Ancien NUMERIC(7,2):=NULL;
   v_Nouveau NUMERIC(7,2):=NULL;

BEGIN
   IF (INSERTING) THEN
      v_idEmploye := :new.idEmploye;
      v_Operation := 'INS';
      v_Nouveau := :new.Salaire;
   ELSIF (DELETING) THEN
      v_idEmploye := :old.idEmploye;
      v_Operation := 'DEL';
      v_Ancien := :old.Salaire;
   ELSIF UPDATING('Salaire') THEN
      v_idEmploye := :old.idEmploye;
      v_Operation := 'UPD';
      v_Nouveau := :new.Salaire;
      v_Ancien := :old.Salaire;
   END IF;
   INSERT INTO Histo_Salaire
   VALUES(v_idEmploye, v_DateModif, v_Operateur, v_Operation,
   v_Ancien, v_Nouveau);
END;
```

Exemple n°3 : tracé des connexions à la base de données

L'exemple est un peu scolaire, dans la mesure où il entreprend la surveillance d'événements déjà sous le contrôle du système. Nous allons créer deux triggers qui vont noter dans une table

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

spéciale toutes les connexions et déconnexions d'utilisateurs. Pour cela, nous devons attribuer le trigger à un événement système[3]. C'est évidemment l'administrateur qui va réaliser cette tâche.

```
CREATE TABLE SYS.TRACELOG
  (    HEURE TIMESTAMP NOT NULL,
       UTILISATEUR VARCHAR2(50),
       ACTION VARCHAR2(10),
       CONSTRAINT pk_TRACELOG PRIMARY KEY (HEURE)
  ) ;
```

On utilisera l'événement `AFTER LOGON ON DATABASE` pour déclencher le trigger. Il suffira de noter l'heure système et le nom de l'utilisateur concerné.

```
CREATE OR REPLACE TRIGGER TRACELOGON
AFTER LOGON ON DATABASE
BEGIN
INSERT INTO TraceLog VALUES (SYSDATE, USER,'LOGON');
END;
```

L'opération inverse est quasiment identique. L'événement testé est ici `BEFORE LOGOFF ON DATABASE`. Comme on constate que SYSMAN semble se connecter et déconnecter régulièrement, il est intéressant de l'exclure de la surveillance. On modifiera l'autre trigger en conséquence.

```
CREATE OR REPLACE
TRIGGER TRACELOGOF
BEFORE LOGOFF ON DATABASE
BEGIN
  IF USER != 'SYSMAN' THEN
    INSERT INTO TraceLog
    VALUES (SYSDATE, USER,'LOGOFF');
  END IF;
END;
```

Remarques importantes

Lors de la mise au point des triggers, des procédures et des fonctions, l'écriture d'une routine incorrecte provoque le message « trigger/procédure/fonction compilée avec des erreurs ». Pour prendre connaissance des erreurs, il faut taper la commande

```
SHOW ERROR
```

L'édition des routines dans SQL Developer présente bien des avantages, en automatisant à l'aide d'un assistant une partie des déclarations et en fournissant un éditeur interactif qui permet de voir les messages d'erreur et de les localiser.

Dans certaines circonstances, la présence d'un trigger peut entraîner des inconvénients temporaires lors d'une mise au point ou d'une restructuration de la base de données. Au lieu de supprimer le trigger, on peut le désactiver temporairement et le réactiver à la fin de l'intervention.

[3]Ici encore, l'utilisation de SQL Developper nous fournit un assistant qui nous montre tous les événements auxquels nous pourrions associer des triggers.

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

```
ALTER TRIGGER TRACELOGOF disable
ALTER TRIGGER TRACELOGOF enable
```

12.3 Procédures stockées

Les procédures stockées sont également situées sur le serveur mais se distinguent des *triggers* par les caractéristiques suivantes :
- elles sont toujours appelées explicitement (par un utilisateur au moyen de les commandes EXECUTE ou CALL, par un *trigger* ou par une autre procédure ou fonction)
- elles peuvent posséder des paramètres
- elles ont une syntaxe différente : pas de mot DECLARE avant les déclarations.

12.3.1 En tête d'une procédure (paramètres d'entrée et de sortie)

```
CREATE OR REPLACE PROCEDURE NomProcédure
(Paramètre Mode Type,...) AS
Déclarations de variables
BEGIN
...
END;
```

La procédure commence par CREATE PROCEDURE ou mieux par CREATE OR REPLACE PROCEDURE. Les paramètres sont définis entre parenthèses, leur type est précédé de IN, OUT ou IN OUT. Les déclarations viennent après le mot IS ou AS.

```
CREATE OR REPLACE PROCEDURE NomProc
(Para1 IN Type, Para2 OUT Type...) AS
CREATE OR REPLACE PROCEDURE Factorielle
(Nombre IN NUMBER, FAC OUT NUMBER )AS
```

Il est important de noter que les paramètres en entrée ne peuvent être modifiés.

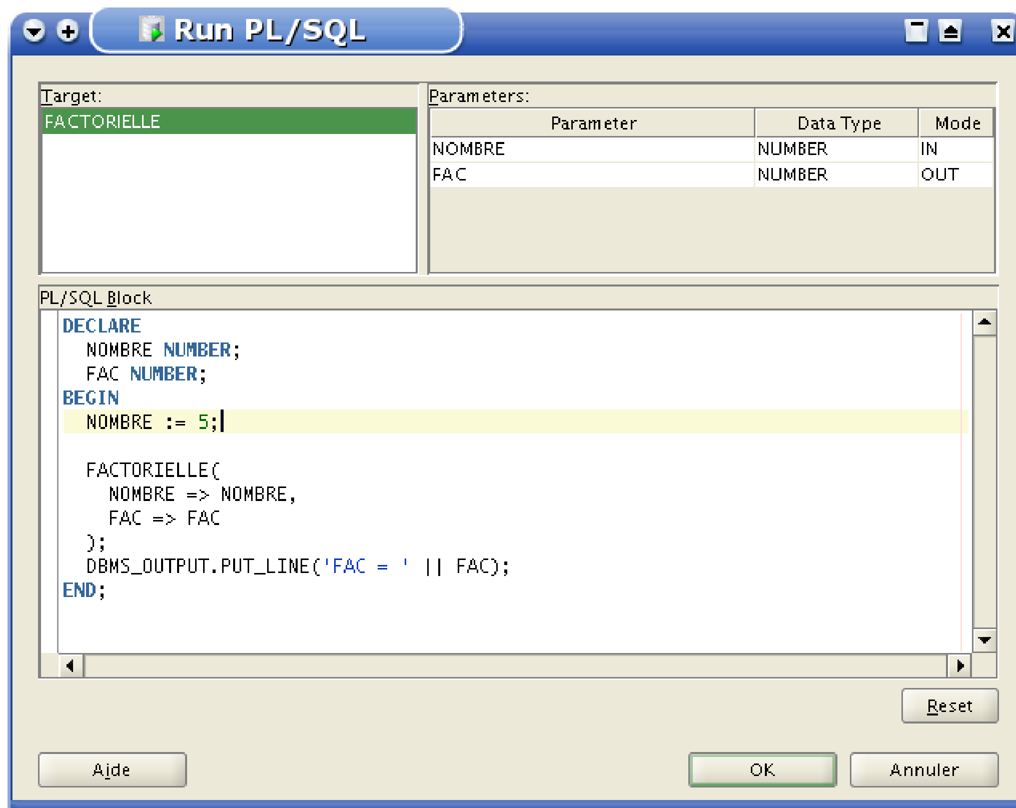
12.3.2 Exécution

L'exécution se réalise en utilisant la commande EXECUTE. Si les paramètres sont en entrée, il suffit de les placer dans les parenthèses, séparés par des virgules. S'ils sont en sortie, ils doivent être déclarés préalablement. En mode console, on devra utiliser la directive VARIABLE. Voici comment utiliser notre procédure Factorielle.

```
SQL> variable Brol NUMBER;
SQL> execute factorielle(6,:Brol);
Procédure PL/SQL terminée avec succès.
SQL> select :Brol from dual;
     :BROL
----------
       720
```

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

L'exécution dans SQL Developper est un peu plus complexe. Il faut utiliser la fenêtre d'exécution accessible par un clic droit sur le nom de la procédure. Une portion de code est ajoutée pour faciliter le passage des paramètres. Ici j'ai donné la valeur 5 au paramètre *Nombre*. Le paramètre de sortie est envoyé dans la fenêtre de log.



12.3.3 Exemple

```
CREATE OR REPLACE PROCEDURE Factorielle (Nombre IN NUMBER, FAC OUT NUMBER )
AS
Facteur NUMERIC(20);

BEGIN
FAC:=1;
-- Copie indispensable de Nombre (non modifiable)
Facteur:=Nombre;

WHILE Facteur>0 LOOP
   FAC:=FAC*Facteur;
   Facteur:=Facteur-1;
END LOOP;
END;
```

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

12.4 Fonctions

```
CREATE OR REPLACE FUNCTION NomFonction
(Paramètre Mode Type,...)
RETURN Type AS
Déclarations de variables
BEGIN
...
END;
```

La création d'une fonction diffère assez peu d'une procédure. Elle permet de renvoyer une valeur (indépendamment des paramètres déclarés en sortie) qu'on peut utiliser dans une expression ou dans une clause SELECT.

Nous allons reprendre le thème de la factorielle, que nous traiterons sous forme récursive.

```
CREATE OR REPLACE FUNCTION fFactorielle (Nombre IN NUMBER)
RETURN NUMBER
AS
BEGIN
IF Nombre < 2 THEN
   RETURN 1;
ELSE
   RETURN Nombre*fFactorielle(Nombre-1);
END IF;
END;
```

Le test de la fonction se fait de la même manière dans la console ou dans SQL Developer :

```
select ffactorielle(5) from dual;
```

12.5 Gestion des curseurs

Le langage SQL se montre très puissant pour réaliser des traitements qui demandent parfois des programmations lourdes avec les langages traditionnels. Il suffit de comparer un programme écrit en C et l'instruction SQL équivalente lorsqu'il s'agit par exemple de déterminer la date de naissance du plus vieil employé de l'entreprise (`select min(datenaiss) from employes;`). Par définition, SQL manipule les données comme des ensembles. Par contre, SQL se montre rétif à des opérations séquentielles. Un exemple simple l'illustrera.

On désire gérer un compte bancaire. Chaque opération correspondra à une ligne de la table suivante :

OPERATION	MONTANT	DESCRIPTION	SOLDE
1	100	Pas de description	
2	-50	Pas de description	
3	-15.25	Pas de description	
4	85	Pas de description	

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

Nous voudrions calculer le solde de chaque opération, qui correspond, sauf pour la première ligne, au solde précédent augmenté du montant, positif ou négatif, de l'opération. Aucune instruction SQL simple ne peut réaliser cela[4].

Un curseur est une variable qui stocke temporairement le résultat d'une requête comportant de zéro à plusieurs tuples et qui nous permet de manipuler ces tuples un par un. À l'aide d'une boucle, on peut alors réaliser un traitement séquentiel des données, par opposition aux traitements SQL, qui sont toujours globaux.

Un curseur se manipule au moyen de quatre opérations fondamentales :

1. la déclaration du curseur se place dans la zone de déclaration des variables et utilise le mot `CURSOR`, le nom du curseur et la requête qui le définit ;
2. l'ouverture du curseur fait usage de la commande `OPEN`
3. la récupération des données, dans des variables préalablement définies, se fait au moyen de l'instruction `FETCH`. Les opérateurs `%FOUND` et `%NOTFOUND` permettent de tester si des données ont été effectivement récupérées. Ils serviront à contrôler la boucle.
4. la fermeture du curseur libère les ressources et s'obtient par la commande `CLOSE`.

12.5.1 Déclaration explicite

Dans cette première version fonctionnelle du programme, nous ne modifions rien et nous nous contentons de renvoyer le solde final.

```
-- Calcul du solde final dans la table Compte version 1
CREATE OR REPLACE FUNCTION Calcule_Solde_Final return NUMBER
AS
SoldePrecedent Compte.Solde%TYPE;
v_Operation Compte.Operation%TYPE;
v_Montant Compte.Montant%TYPE;
v_Solde Compte.Solde%TYPE;

-- Déclaration du curseur
CURSOR mon_compte IS
   SELECT operation, montant, solde FROM Compte Order by operation;
BEGIN
SoldePrecedent:=0;
-- Ouverture du curseur
OPEN mon_compte;
-- Traitement
LOOP
```

[4]On peut toujours simuler. La requête suivante recalcule les soldes pour chaque ligne, mais ne tient pas compte du résultat de la ligne précédente. Notons que nous nous bornons à afficher. Si on veut modifier le champ `Solde`, nous ne sommes pas encore au bout de nos peines.

```
SELECT operation,montant,
(select sum(montant) FROM compte WHERE operation<=c.operation) AS NouveauSolde
FROM compte C;
 OPERATION    MONTANT NOUVEAUSOLDE
---------- ---------- ------------
         1        100          100
         2        -50           50
         3     -15,25        34,75
         4         85       119,75
```

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

```
   FETCH mon_compte INTO v_Operation, v_Montant, v_Solde;
   -- test de sortie de boucle
EXIT WHEN mon_compte%NOTFOUND;
   SoldePrecedent:=SoldePrecedent+v_Montant;
END LOOP;
-- Fermeture
CLOSE mon_compte;
Return SoldePrecedent;
END;
```

Il est possible de ne pas déclarer toutes les variables correspondant aux différents champs, en utilisant une variable de structure définie à l'aide de %ROWTYPE.

```
-- Calcul du solde dans la table Compte version 2
CREATE OR REPLACE FUNCTION Calcule_Solde_Final return NUMBER
AS
SoldePrecedent Compte.Solde%TYPE;
v_Ligne Compte%ROWTYPE;
-- Déclaration du curseur
CURSOR mon_compte IS
      SELECT * FROM Compte Order by operation;
BEGIN
SoldePrecedent:=0;
-- Ouverture du curseur
OPEN mon_compte;
-- Traitement
LOOP
   FETCH mon_compte INTO v_Ligne;
EXIT WHEN mon_compte%NOTFOUND;
   SoldePrecedent:=SoldePrecedent+v_Ligne.Montant;
END LOOP;
-- Fermeture
CLOSE mon_compte;
Return SoldePrecedent;
END;
```

La gestion de la boucle n'est pas particulièrement élégante (sortie au milieu). Un curseur peut encore se gérer d'une manière plus concise.

12.5.2 Déclaration implicite

On préférera une seconde manière de gérer un curseur, qui utilise un opérateur FOR modifié prenant en charge :

- la déclaration de la structure
- l'ouverture du curseur
- la lecture des données dans la structure
- le test de fin du parcours
- la fermeture du curseur en fin de boucle.

```
-- Calcul du solde dans la table Compte version 3
CREATE OR REPLACE FUNCTION Calcule_Solde_Final return NUMBER
AS
```

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

```
SoldePrecedent Compte.Solde%TYPE;
-- Déclaration du curseur
CURSOR mon_compte IS
      SELECT * FROM Compte Order by operation;
BEGIN
SoldePrecedent:=0;
-- Création de la structure, ouverture du curseur
-- Fetch automatique et contrôle de la boucle
FOR v_Ligne IN mon_compte LOOP
  SoldePrecedent:=SoldePrecedent+v_Ligne.Montant;
END LOOP;
-- Fermeture implicite
Return SoldePrecedent;
END;
```

12.5.3 Modification des données

Les trois versions de la fonction ne résolvent pas notre problème : la mise à jour du champ `Solde` de chaque ligne. Pour cela, il est nécessaire de déclarer un curseur capable de modifier les données. La modification se fait simplement :

- on doit spécifier lors de la déclaration du curseur les champs susceptibles d'être modifiés (`FOR UPDATE OF`).
- dans la boucle, on utilise une requête `UPDATE` pour effectuer la modification en spécifiant une condition `WHERE` particulière qui identifie la ligne en cours : `WHERE CURRENT OF` *NomCurseur*.
- il ne faut pas oublier de placer une commande `COMMIT` en fin de procédure pour valider la transaction, faute de quoi les modifications ne seront pas prises en compte définitivement.

```
-- Calcul du solde dans la table Compte (version finale)
CREATE OR REPLACE PROCEDURE MiseAJourSolde
AS
SoldePrecedent Compte.Solde%TYPE;
-- Déclaration du curseur pour modification de Solde
CURSOR mon_compte IS
      SELECT * FROM Compte Order by operation FOR UPDATE Of Solde;
BEGIN
SoldePrecedent:=0;
-- Gestion du curseur
FOR v_Ligne IN mon_compte LOOP
  SoldePrecedent:=SoldePrecedent+v_Ligne.Montant;
  UPDATE Compte
  SET Solde=SoldePrecedent
  WHERE CURRENT OF mon_compte;
END LOOP;
-- Valider la transaction
COMMIT;
END;
```

Cette fois, les données sont modifiées :

```
SQL> select * from compte;
```

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET

```
 OPERATION    MONTANT DESCRIPTION                                     SOLDE
---------- ---------- ------------------------------ ----------
         1        100 Pas de description
         2        -50 Pas de description
         3     -15,25 Pas de description
         4         85 Pas de description
SQL> execute MiseAJourSolde();
Procédure PL/SQL terminée avec succès.
SQL> select * from compte;
 OPERATION    MONTANT DESCRIPTION                                     SOLDE
---------- ---------- ------------------------------ ----------
         1        100 Pas de description                                100
         2        -50 Pas de description                                 50
         3     -15,25 Pas de description                              34,75
         4         85 Pas de description                             119,75
```

©2008-2009 -JACQUES THOORENS - HTTP://WWW.THOORENS.NET