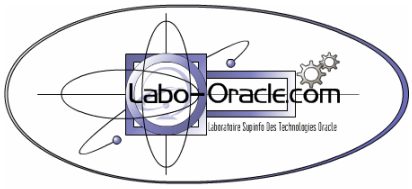

Module n°1
PL/SQL Avancé

ORACLE

Programme de Formation de Supinfo

Laboratoire Supinfo des Technologies Oracle
Auteur : Thibault Blanchard
Date : 01/01/1601 01:00 - Version 1.2
Nombre de Page : 101
http://www.labo-oracle.com

Ecole Supérieure d'Informatique
23, rue Château Landon
75010 PARIS
http://www.supinfo.com

1 APERÇU DU PL/SQL 7

1.1 La solution complète d'Oracle 7

1.2 Les programmes en PL/SQL 7
1.2.1 Les modèles de programme en PL/SQL 7
1.2.2 Structure d'un bloc PL/SQL anonyme 7
1.2.3 Structure d'un sous-programme PL/SQL 8
1.2.4 Avantages des sous-programmes 8

1.3 Les environnements de développement 8
1.3.1 SQL*Plus et Procedure Builder 8
1.3.2 Développer des fonctions et procédures avec SQL*Plus 9
1.3.3 Développer en utilisant Oracle Procedure Builder 9

1.4 Appel de fonctions et de procédures 9

2 UTILISATION DE PROCEDURE BUILDER 10

2.1 Procedure Builder 10
2.1.1 Les composants de Procedure Builder 10
2.1.2 Développer des unités de programmes et des unités de programmes stockés 10
2.1.3 Le navigateur d'objets 10
2.1.4 L'éditeur d'unités de programmes 11
2.1.5 L'éditeur d'unités de programme stockées 12

2.2 Utilisation de Procedure Builder 12
2.2.1 Création d'une unité de programme cliente 12
2.2.2 Création d'une unité de programme serveur 12
2.2.3 Transférer les programmes entre le client et le serveur 12

2.3 L'interpréteur PL/SQL 12

2.4 Le package TEXT_IO 13

3 CREATION DE PROCEDURES 14

3.1 Aperçu des procédures 14

3.2 Créer des procédures 14
3.2.1 La syntaxe de création de procédures 14
3.2.2 Les modes de paramètres de la procédure 14
3.2.3 Développer des procédures stockées 15
3.2.4 Développer une procédure en utilisant SQL*Plus 15
3.2.5 Développer une procédure en utilisant procedure builder 15

3.3 Procédures et paramètres 16
3.3.1 Création de procédures avec des paramètres 16
3.3.2 Le paramètre IN 17
3.3.3 Le paramètre OUT 17
3.3.4 Le paramètre IN OUT 18
3.3.5 Passer des paramètres 20

3.4 Les sous programmes 21
3.4.1 Déclarer des sous-programmes 21
3.4.2 Invoquer une procédure depuis un bloc anonyme 22
3.4.3 Invoquer une procédure depuis une procédure stockée 22

3.5 Gestion des exceptions 22
3.5.1 Exceptions traitées 22
3.5.2 Exceptions non traitées 23

3.6 Supprimer des procédures 23

3.7 Supprimer des procédures serveur 23

3.8 Supprimer des procédures client 23

4 CREATION DE FONCTIONS 24

4.1 Les fonctions 24
4.1.1 Aperçu des fonctions stockées 24
4.1.2 Syntaxe pour la création de fonctions 24
4.1.3 Création de fonction 24

4.2 Les fonctions dans SQL*Plus 25
4.2.1 Création de fonctions stockées 25
4.2.2 Exécuter des fonctions 25

4.3 Les fonctions dans Procedure Builder 26
4.3.1 Création de fonction 26
4.3.2 Exemple de création de fonction 26
4.3.3 Exécuter des fonctions 26

4.4 Les fonctions définies par l'utilisateur dans du SQL 27
4.4.1 Avantages des fonctions dans des expressions SQL 27
4.4.2 Emplacements d'où appeler les fonctions 27
4.4.3 Appel de fonctions : restrictions 27

4.5 Supprimer des fonctions 28
4.5.1 Supprimer des fonctions serveur 28
4.5.2 Supprimer des fonctions client 28

4.6 Procédure ou fonction ? 28
4.6.1 Récapitulatif 28
4.6.2 Comparaison entre procédures et fonctions 28
4.6.3 Les avantages des procédures et fonctions stockées 29

5 CREATION DE PACKAGES 30

5.1 Les packages 30
5.1.1 Aperçu des packages 30
5.1.2 Les composantes d'un packages 30
5.1.3 Référencer les objets d'un package 31

5.2 Créer des packages 31
5.2.1 Développement d'un package 31
5.2.2 Création des spécifications du package 32
5.2.3 Déclaration d'éléments publics 32
5.2.4 Création de spécification de package : exemple 32
5.2.5 Création du corps du package 32
5.2.6 Eléments publics et privés 33
5.2.7 Création de corps de package : exemple 34
5.2.8 Directives pour développer des packages 34
5.2.9 Les variables globales 34

5.3 Manipuler les packages 35
5.3.1 Exécuter une procédure publique d'un package 35
5.3.2 Invoquer des éléments de packages 35
5.3.3 Référencer une variable publique à partir d'une procédure autonome 36
5.3.4 Supprimer des packages 37
5.3.5 Avantages des packages 37

6 COMPLEMENTS SUR LES PACKAGES 39

6.1 La surcharge 39

6.2 Les déclarations anticipées 40

6.3 Création d'une procédure à usage unique 41

6.4 Restrictions sur les fonctions de package en SQL 41

6.5 Invoquer une fonction d'un package définit par l'utilisateur dans un ordre SQL 42

6.6 L'état persistant 42

6.6.1 Les packages de variables 42
6.6.2 Les packages de curseurs 43
6.6.3 Les packages de tables et records PL/SQL 44

7 PACKAGES FOURNIS PAR ORACLE 46

7.1 Les packages fournit par Oracle 46

7.2 Le package DBMS_PIPE 46
7.2.1 La composition de DBMS_PIPE 46
7.2.2 Les fonctions de DBMS_PIPE 46
7.2.3 Exemple d'utilisation de DBMS_PIPE 46

7.3 Le SQL dynamique 47
7.3.1 Définition 47
7.3.2 Le flux d'exécution 47
7.3.3 Le package DBMS_SQL 48
7.3.4 Utilisation de DMBS_SQL 48
7.3.5 EXECUTE IMMEDIATE 49
7.3.6 49
7.3.7 Utilisation de EXECUTE IMMEDIATE 50

7.4 Les autres packages disponibles 50
7.4.1 Le package DBMS_DDL 50
7.4.2 Le package DBMS_JOB 50
7.4.3 Le package DBMS_OUTPUT. 51
7.4.4 D'autres packages fournit par Oracle 51

8 CREATION DE TRIGGERS DE BASE DE DONNEES 52

8.1 Les triggers de base de données 52
8.1.1 Aperçu des triggers 52
8.1.2 Directives de conception des triggers 52
8.1.3 Exemple de trigger de base de données 52
8.1.4 Création de triggers 53

8.2 Les composantes d'un trigger 53
8.2.1 La synchronisation du trigger 53
8.2.2 L'évènement déclenchant 53
8.2.3 Le type de trigger 54
8.2.4 Le corps du trigger 54

8.3 La séquence de déclenchement 54

8.4 Création de Statement trigger 54
8.4.1 Syntaxe de création de Statement triggers 54
8.4.2 Création avec SQL*Plus 55
8.4.3 Création avec Procedure Builder 55
8.4.4 Test de SECURE_EMP 56
8.4.5 Utilisation d'attributs conditionnels 57

8.5 Création de Row Triggers 57
8.5.1 Syntaxe de création d'un Row Trigger 57
8.5.2 Création avec SQL*Plus 58
8.5.3 Création avec Procedure Builder 59
8.5.4 Utilisation des qualificatifs OLD et NEW 59
8.5.5 Valeurs de OLD et NEW 59
8.5.6 Restreindre un trigger de ligne 60

8.6 Trigger INSTEAD OF 60
8.6.1 Intérêt des triggers INSTEAD OF 60
8.6.2 Création d'un trigger INSTEAD OF 60

8.7 Différence entre les triggers et les procédures stockées 61

8.8 Gestion des triggers 61
8.8.1 Activation des triggers 61

8.8.2 Suppression de triggers 62
8.9 Test de triggers 62
8.9.1 Cas à tester 62
8.9.2 Modèle d'exécution de trigger et vérification de contrainte 62
8.10 Interactions 63
8.10.1 Une démonstration type 63
8.10.2 La table d'audit 63
8.10.3 Les triggers 63
8.10.4 Spécifications du package VAR_PACK 64
8.10.5 Procédure 65
9 COMPLEMENTS SUR LES TRIGGERS 66
9.1 Création de triggers sur des événements utilisateur 66
9.2 Création de triggers sur des évènements systèmes 66
9.3 Des exemples des trigger Log On et Log Off 67
9.4 La déclaration CALL 67
9.5 Les tables en cours de modifications 68
9.5.1 Lecture de donnée dans une table en cours de modifications 68
9.5.2 Exemple de table en cours de modifications 68
9.6 Fonctions des triggers 69
9.6.1 Sécuriser le serveur 69
9.6.2 Audit 69
9.6.3 Garantir l'intégrité des données 71
9.6.4 Garantir l'intégrité référentielle 71
9.6.5 Dupliquer les tables 72
9.6.6 Utiliser des données dérivées 73
9.6.7 Gérer les logs d'événements avec les triggers 74
9.6.8 Avantages des triggers de base de données 75
10 GERER LES SOUS PROGRAMMES ET LES TRIGGERS 76
10.1 Privilèges 76
10.1.1 Privilèges systèmes 76
10.1.2 Privilèges objets 76
10.2 Accorder des accès aux données 76
10.3 Spécifier les droits des utilisateurs 77
10.4 Gérer les objets PL/SQL stockés 77
10.5 Informations sur l'objet 78
10.5.1 USER_OBJECTS 78
10.5.2 Lister toutes les procédures et fonctions 78
10.6 Texte de la procédure 78
10.6.1 La vue du dictionnaire de données USER_SOURCE 78
10.6.2 Lister le code de toute les procédures et fonctions 79
10.6.3 Lister le code des procédures stockées avec procedure builder 79
10.7 Texte d'un trigger 79
10.7.1 USER_TRIGGERS 79
10.7.2 Lister le code des triggers 80
10.8 Les paramètres 80
10.8.1 DESCRIBE dans SQL*Plus 80
10.8.2 La commande .DESCRIBE sous procedure builder 81
10.9 Erreurs de compilation 81
10.9.1 Détecter les erreurs de compilation avec l'éditeur de programme stocké 81
10.9.2 USER_ERRORS 81

10.9.3 Détecter les erreurs de compilation : exemple 82

10.10 Informations sur le débuggage 83
10.10.1 Débugger en utilisant le package DBMS_OUTPUT 83
10.10.2 Débugger les sous programmes en utilisant procedure builder 84
10.10.3 Créer des breakpoints 84
10.10.4 Utiliser les niveaux de débuggages 85
10.10.5 Contrôler l'exécution des programmes 85

11 GERER LES DEPENDANCES 87

11.1 Objets dépendants et référencés 87
11.1.1 Comprendre les dépendances 87
11.1.2 Les dépendances directes et indirectes 87
11.1.3 Les dépendances locales 87
11.1.4 Les dépendances distantes 87
11.1.5 Un scénario de dépendances locales 88

11.2 Visualiser les dépendances 88
11.2.1 Afficher les dépendances directes avec USER_DEPENDENCIES 88
11.2.2 Afficher les dépendances directes et indirectes 89
11.2.3 Afficher les dépendances 89

11.3 Un scénario de dépendances de nom locales 89

11.4 Dépendances distantes 90
11.4.1 Recompilation des dépendances 90
11.4.2 Concept de dépendances distantes 90
11.4.3 Scénario de dépendance distante en mode Timestamp 91
11.4.4 Le mode de signature 91

11.5 Recompilation manuelle 91
11.5.1 Recompiler un programme PL/SQL 91
11.5.2 Echec de recompilation 92
11.5.3 Recompilation réussie 92
11.5.4 Recompiler les procédures 92

11.6 Packages et dépendances 92

12 MANIPULER LES LARGES OBJECTS 94

12.1 Les LOBs 94
12.1.1 Qu'est-ce qu'un LOB 94
12.1.2 Opposition entre les types de données LONG et LOB 94
12.1.3 Anatomie d'un LOB 94

12.2 LOBs internes et externes 95
12.2.1 Les LOBs internes 95
12.2.2 Gérer les LOBs internes 95
12.2.3 Les LOBs externes 95
12.2.4 Définir les BFILEs 95

12.3 Le package DBMS_LOB 96
12.3.1 Utilisation de DBMS_LOB 96
12.3.2 Fonctions et procédures de DBMS_LOB 96
12.3.3 DBMS_LOB READ et WRITE 96
12.3.4 Exemple de création de table avec des LOBs 97

12.4 Manipuler des LOBs en SQL 97
12.4.1 Insertion en utilisant SQL 97
12.4.2 Mise à jour des LOBs en utilisant SQL 98
12.4.3 Mise à jour en utilisant DBMS_LOB 98
12.4.4 Sélectionner les valeurs CLOB 99
12.4.5 Exemple de suppression de LOBs 101

12.5 Les LOBs temporaires 102

1 APERÇU DU PL/SQL

1.1 La solution complète d'Oracle

La solution Oracle est constituée de plusieurs outils reconnaissant et soumettant des ordres SQL et PL/SQL au serveur pour être exécutées. Ces outils possèdent leurs propres langage de commandes.

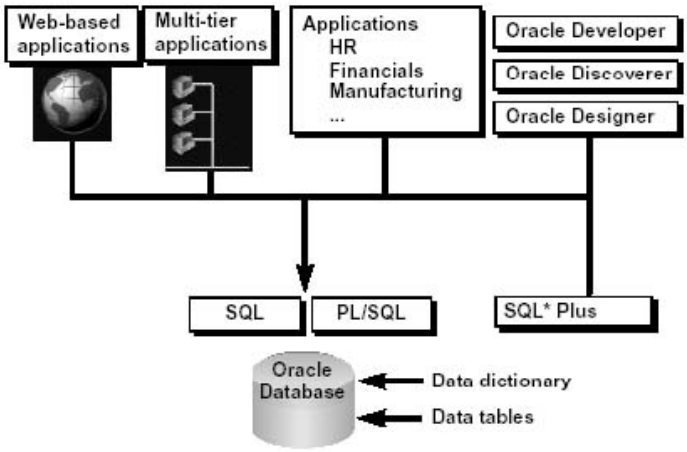

Figure 1 : Architecture de la solution complète Oracle

1.2 Les programmes en PL/SQL

1.2.1 Les modèles de programme en PL/SQL

Les programmes écrits en PL/SQL respectent tous une structure en blocs prédéterminés (Cf. Module n°4 « Notion de base du PL/SQL $1.3 « Les structures de programme PL/SQL »). Les différents modèles de programme (blocs anonymes, procédures et fonctions stockées, trigger, package…) sont donc construits suivant un même modèle.
Tous les blocs d'un programme PL/SQL peuvent être séparés et imbriqués les uns dans les autres. Donc un bloc peut représenter une petite partie d'un autre bloc qui est lui-même une partie du code du programme.

1.2.2 Structure d'un bloc PL/SQL anonyme

Un bloc anonyme est un bloc ne possédant pas de noms. Ces blocs sont déclarés à l'endroit où ils vont être exécutés dans une application. Ils sont passés au moteur PL/SQL lors de l'exécution du programme.

```
[DECLARE]
[<section déclarative optionnelle>]
BEGIN
<section exécutable obligatoire>
[EXCEPTION]
<section de traitement des exceptions optionnelle>]
END;
```

Les mots clés DECLARE, BEGIN et EXCEPTION ne sont pas suivis d'un point virgule, seul END et les autres ordres PL/SQL nécessitent un point virgule.

1.2.3 Structure d'un sous-programme PL/SQL

Un sous programme PL/SQL est un block nommé qui peut prendre des paramètres et être invoqué dans d'autres blocs. Il existe deux types de sous programmes : les procédures et les fonctions.

```
Header
IS|AS
[<section déclarative optionnelle>]
BEGIN
<section exécutable obligatoire>
[EXCEPTION]
<section de traitement des exceptions optionnelle>]
END;
```

La section Header détermine la manière dont le sous programme va être appelé ou invoquée. Cette section détermine également le type du sous programme (procédure ou fonction), la liste des paramètres si il y en a, la clause RETURN qui s'applique uniquement aux fonctions.
Le mot clé IS est obligatoire. Il ne faut pas utiliser le mot clé DECLARE car la section déclarative se situe entre le IS et le BEGIN.
Pour le reste du code, il se comporte de la même manière que pour les blocs PL/SQL anonymes.

1.2.4 Avantages des sous-programmes

Les procédures et fonctions stockées ont des avantages en plus du développement modulaire des applications. Les sous programmes permettent d'améliorer :

- la maintenance :
  Modification des routines online sans interférer avec les autres utilisateurs
  Modification d'une routine pour agir sur plusieurs applications
  Modification d'une routine pour éliminer les tests en double
- l'intégrité et la sécurité des données :
  Contrôle des accès indirects aux objets de la base de données par des utilisateurs ne possédant pas des privilèges de sécurité
  S'assure que les actions liées sont exécutées ensemble, ou pas du tout, en centralisant l'activité des tables liées dans un seul répertoire
- les performances :
  Evite de re-parcourir les lignes pour différents utilisateurs en exploitant la zone SQL partagée
  Evite de parcourir le bloc PL/SQL pendant l'exécution en le parcourant lors de la compilation
  Réduit le nombre d'appels à la base de données et diminue le trafic réseau en envoyant les commandes par paquets

1.3 Les environnements de développement

1.3.1 SQL*Plus et Procedure Builder

Le PL/SQL n'est pas à proprement parlé un produit Oracle. C'est une technologie utilisée par le serveur Oracle et par certains outils de développement Oracle, les blocs de PL/SQL sont passés et traités par un moteur PL/SQL. Ce moteur peut être inclus dans l'outil de développement ou dans le serveur Oracle.
Les deux principaux outils de développement sont SQL*Plus et Procedure Builder.
SQL*Plus utilise le moteur PL/SQL du serveur Oracle alors que Procedure Builder utilise le moteur PL/SQL de l'outil client ou le moteur du serveur Oracle.

1.3.2 Développer des fonctions et procédures avec SQL*Plus

Il y a deux moyens pour écrire des blocs PL/SQL dans SQL*Plus. On peut les stocker dans le buffer SQL*Plus et ensuite l'exécuter à partir de SQL*Plus ou bien les stocker dans un script SQL*Plus et ensuite exécuter le fichier grâce à la commande EXECUTE.

1.3.3 Développer en utilisant Oracle Procedure Builder

Procedure Builder est un outil que l'on peut utiliser pour créer, exécuter et débugger des programmes PL/SQL utilisés dans vos application ou sur le serveur Oracle par le biais de son interface graphique.
L'environnement de développement Procedure Builder possède un éditeur intégré avec lequel il est possible de créer et d'éditer des sous programmes. Il est possible compiler, tester et débugger son code grâce à cet outil.

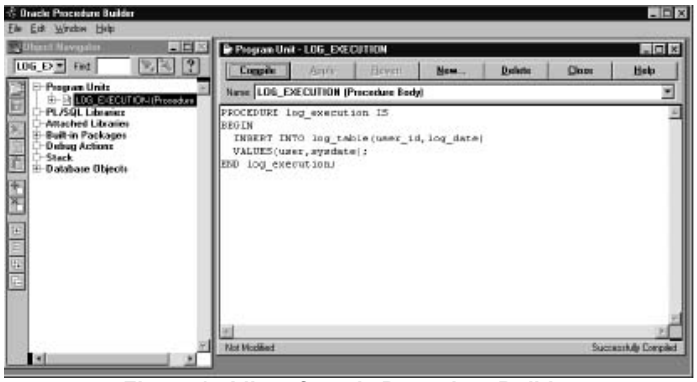

Figure 1 : L'interface de Procedure Builder

1.4 Appel de fonctions et de procédures

Les procédures et fonctions stockées peuvent être appelées depuis plusieurs environnements : SQL*Plus : Oracle Developer, Oracle Discoverer, WebDB , une autre procédure stockée et de nombreux outils Oracle et applications de précompilation.
Pour lancer une procédure ou fonction depuis SQL*Plus il faut utiliser la commande EXECUTE suivie du nom du fichier dans lequel elle est stockée. Avec Oracle discoverer et Oracle Developer il suffit de spécifier le nom du fichier script.
Pour exécuter une procédure à l'intérieur d'une autre il faut simplement donner le nom de la procédure au moment voulut.

Exemple :

```
CREATE OR REPLACE PROCEDURE leave_emp
(v_id IN emp.empno%TYPE)
IS
BEGIN
DELETE FROM emp
WHERE empno = v_id;
exec_proc;
END leave_emp;
```

-> Ce code crée une procédure qui efface les lignes dont le EMPNO corresponds au V_ID entré en paramètre. Après la suppression on fait appel à la procédure nommée EXEC_PROC. La structure de création de procédure sera expliquée plus loin dans ce module.

2 UTILISATION DE PROCEDURE BUILDER

2.1 Procedure Builder

2.1.1 Les composants de Procedure Builder

Procedure Builder est un environnement de développement intégré qui permet d'éditer, de compiler de tester et de débugger des unités de programmes PL/SQL client et serveur avec un seul et même outil. Toutes ces fonctionnalités sont possibles grâce aux différents composants intégrés dans Procedure Builder.

Composant	Utilisation
Explorateur d'objet	Permet de gérer les ensembles, et d'effectuer des opérations de débuggage
Interpréteur PL/SQL	Permet de débugger du code PL/SQL, et d'évaluer du code PL/SQL en temps réel
Editeur d'unités de programmes	Permet de créer et d'éditer du code source PL/SQL
Editeur d'unités de programmes stockés	Permet de créer et d'éditer du code source PL/SQL coté serveur
Editeur de trigger de base de données	Permet de créer et d'éditer des triggers de base de données

2.1.2 Développer des unités de programmes et des unités de programmes stockés

Procedure Builder permet de développer des sous programmes PL/SQL pouvant être utilisés dans des applications clientes ou serveurs. Les *unités de programmes* sont des sous programmes PL/SQL qui sont utilisées avec des applications clientes, tel que Oracle Developer. Les unités de programmes stockées sont des sous programmes PL/SQL que l'on peut utiliser avec toutes les applications, clientes ou serveurs.
Le code PL/SQL est perdu lorsque l'on ferme Procedure Builder sauf si l'on sauvegarde le code sur le serveur, dans la librairie PL/SQL ou si on l'exporte dans un fichier.
Il existe plusieurs façons de développer du code PL/SQL dans Procedure Builder :

Pour un code côté client on peut créer l'unité de programme en utilisant l'éditeur d'unité de programme ou faire un glisser-déposer d'une unité de programme côté serveur vers le client en utilisant l'explorateur d'objet.

Pour un code côté serveur on peut créer l'unité de programme en utilisant l'éditeur d'unités de programme stockés ou faire un glisser-déposer d'une unité de programme côté serveur vers le serveur en utilisant l'explorateur d'objet.

2.1.3 Le navigateur d'objets

L'explorateur d'objet est un navigateur qui permet de trouver et de travailler avec des unités de programmes client et serveurs ainsi que des librairies et des triggers. Il est possible de développer et réduire l'arborescence, copier et coller, chercher un objet et glisser-déposer des unités de programme PL/SQL entre les côtés client et serveur.

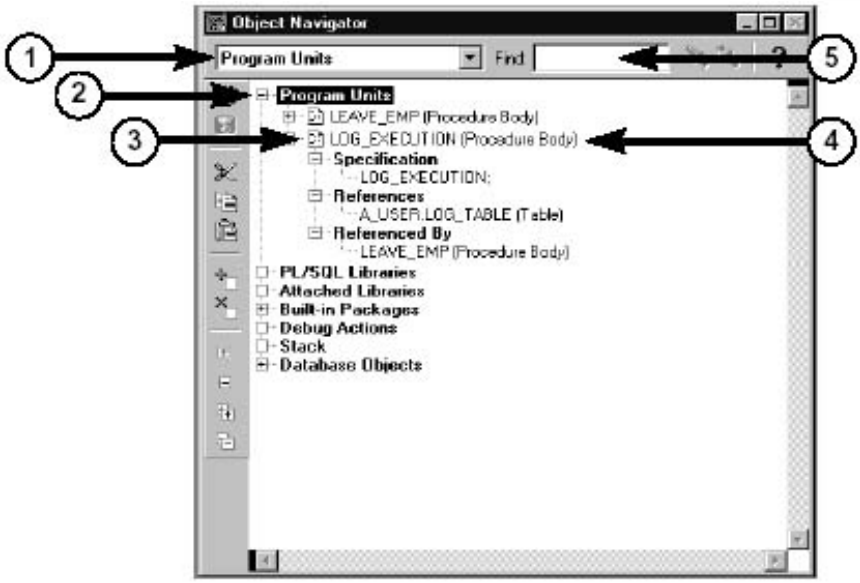

Figure 1 : Aperçu de l'explorateur d'objets

On peut distinguer plusieurs éléments important dans la fenêtre du navigateur :

L'indicateur d'emplacement (1) : il indique l'emplacement actuel dans la hiérarchie

L'indicateur de sous-objet (2) : permet de développer ou de réduire les nœuds pour voir ou cacher les informations sur les objets. Les différentes classes d'objets sont représentées par différentes icônes.

L'icône de type (3) : indique le type de l'objet, suivi du nom de l'objet. Dans l'exemple, l'icône indique que LOG_EXECUTION est un bloc PL/SQL. Si on double-clique sur l'icône, Procedure Builder ouvre l'éditeur d'unité de programme avec le code de cet objet.

Le nom de l'objet (4) : affiche le nom de l'objet

Le champs de recherche (5) : permet de chercher des objets

2.1.4 L'éditeur d'unités de programmes

L'éditeur d'unités de programme permet d'éditer, de compiler et de parcourir les warning et les erreurs pendant le développement de sous programmes PL/SQL clients.

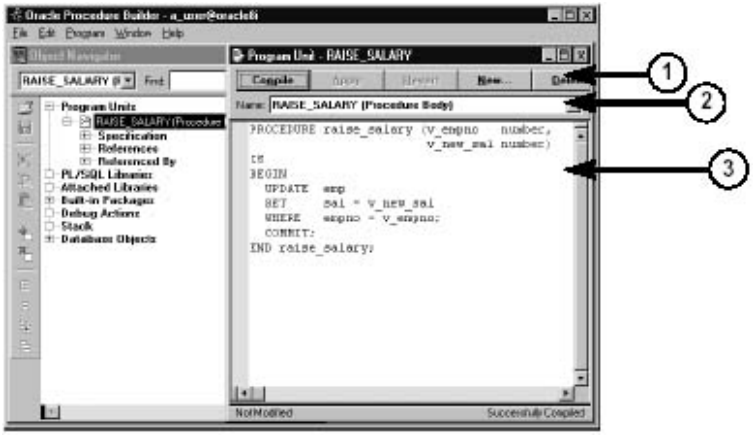

Figure 2 : Aperçu de l'éditeur d'unité de programme

La zone (1) contient tous les différents boutons utilisés pour créer et débugger des unités de programmes (*Compile, Apply, Revert, New, Delete, Close* et *Help*). Le nom du programme est affiché dans la zone (2) et le code source de la procédure est situé dans la zone (3).
Pour amener le code d'un sous programme dans le panneau de code source il faut choisir le nom dans la liste déroulante du champ Name.

2.1.5 L'éditeur d'unités de programme stockées

L'éditeur d'unités de programmes stockés se présente comme l'éditeur décrit précédemment à l'exception que dans ce cas l'opération de sauvegarde envoie le code source au compilateur PL/SQL du serveur.

2.2 Utilisation de Procedure Builder

2.2.1 Création d'une unité de programme cliente

Pour créer une unité de programme côté client il faut sélectionner l'objet *Program Unit* ou un sous objet dans l'explorateur d'objet. Ensuite clique sur *Create* pour faire apparaître la boîte de dialogue *New Program Unit*. Dans celle-ci, on choisit le nom du programme ainsi que son type puis on valide pour afficher l'éditeur d'unité de programme. L'éditeur contient le squelette du modèle PL/SQL. Le curseur est automatiquement positionné à la ligne suivant le BEGIN. Une fois le code écrit, on le compile en cliquant sur le bouton *Compile*. Les messages d'erreurs générés lors de la compilation sont affichés dans le panneau de messages de compilation. Lorsque l'on sélectionne une erreur, le curseur se déplace à l'endroit de l'erreur dans la fenêtre du programme. Lorsque le programme s'est bien compilé, le message *Successfully Compiled* est affiché dans la ligne de status de l'éditeur.
Les unités de programmes résidant dans l'arborescence sont perdus si l'on quitte Procedure Builder. Il faut les exporter vers un fichier, les enregistrer dans une librairie PL/SQL ou les stocker dans la base de données pour ne pas les perdre.

2.2.2 Création d'une unité de programme serveur

Pour créer une unité de programme côté serveur la procédure est la même que pour une unité de programme côté client mais cette fois ci il faut sélectionner l'objet *Stored Program Unit* dans le nœud *Database Objects* de l'arborescence.

2.2.3 Transférer les programmes entre le client et le serveur

Avec Procedure Builder il est également possible de copier des unités de programme créés sur le client en unités de programme stockées sur le serveur (et vice et versa). On peut le faire en déplaçant l'unité de programme vers l'unité de programme stockée de destination dans l'arborescence
Le code PL/SQL stocké sur le serveur est traité par le moteur PL/SQL du côté serveur, donc une requête SQL contenue dans une unité de programme n'a pas besoin d'être transférée entre une application client et le serveur.
Les unités de programmes stockées sur le serveur sont potentiellement accessibles à toutes les applications mais en fonction des privilèges de sécurité de l'utilisateur.

2.3 L'interpréteur PL/SQL

L'interpréteur PL/SQL est un outil de Procedure Builder permettant d'exécuter directement des unités de programme PL/SQL.

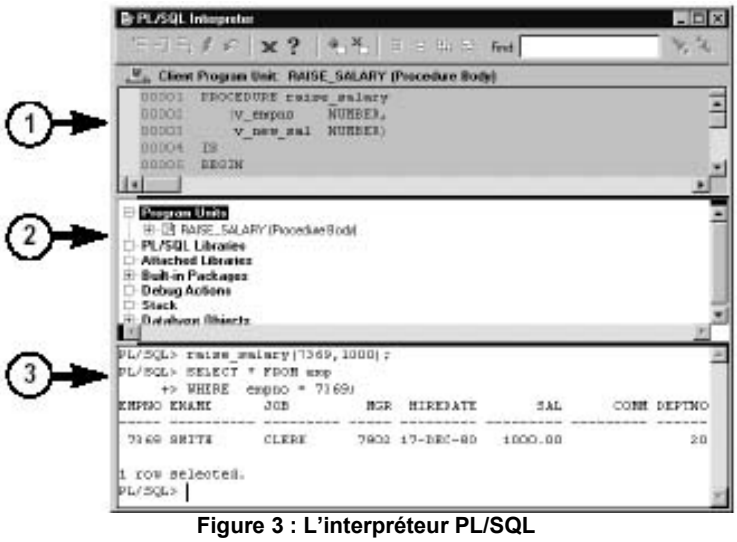

Figure 3 : L'interpréteur PL/SQL

L'interpréteur est composé de trois fenêtres : la première affiche le code source du programme, la seconde contient les mêmes informations que l'explorateur d'objets et enfin la dernière permet d'exécuter des sous programmes, des commandes de Procedure Builder et des requêtes SQL.
Pour exécuter des sous programmes il faut entrer le nom du programme au prompt PL/SQL, spécifier des paramètres si besoin, et ajouter un point virgule.

```
PL/SQL> nom_programme [paramètre1|paramètre2,…] ;
```

Pour exécuter un ordre SQL, il suffit d'entrer cet ordre et de placer un point-virgule à la fin.

2.4 Le package TEXT_IO

TEXT_IO est un package, faisant partie de Procedure Builder et ne pouvant être utilisé que par des fonctions et procédures côté client. Il permet de gérer les entrées/sorties de texte. Ce package inclus une procédure PUT_LINE qui écrit des informations dans la fenêtre de l'interpréteur PL/SQL. Cette procédure accepte un seul paramètre qui correspond au texte que l'on veut afficher.

Exemple :

```
PL/SQL> TEXT_IO.PUT_LINE(Ce texte sera affiché) :
Ce texte sera affiché
```

Ce package est très utile pour effectuer des débuggages de procédures clientes. En revanche, pour débugger un procédure du serveur il faut utiliser le package fournit par Oracle DBMS_OUTPUT, car TEXT_IO produira des erreurs de compilation sur le serveur. Le package DBMS_OUTPUT n'affiche pas de messages dans la fenêtre de l'interpréteur PL/SQL si l'on exécute une procédure depuis Procedure Builder.

3 CREATION DE PROCEDURES

3.1 Aperçu des procédures

Une procédure est un bloc PL/SQL nommé qui peut prendre des paramètres (qu'on appelle aussi arguments) et être invoqué.
Comme décrit précédemment une procédure est constituée d'un en-tête, une section déclarative, une section exécutable, et une section de gestion d'exception optionnelle.
Les procédures facilitent la réutilisation et la manipulation du code car une fois enregistrée, une procédure peut être utilisée par plusieurs autres applications. Si la définition change, seule la procédure est affectée ce qui simplifie la maintenance.

3.2 Créer des procédures

3.2.1 La syntaxe de création de procédures

Pour créer une procédure on utilise l'expression CREATE PROCEDURE qui définit les actions qui seront exécutées par le bloc PL/SQL. Cette expression peut définir une liste de paramètres.

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  (parameter1 [mode1] datatype1,
   parameter2 [mode2] datatype2,
   …)
IS|AS
Bloc PL/SQL
```

Le bloc PL/SQL commence par un BEGIN ou la déclaration des variables locales et se termine par END ou END *procedure_name.* Il est impossible de faire référence à des variables hôtes ou à des variables de substitution.
L'option REPLACE indique que si la procédure existe déjà, elle sera supprimée et remplacée par celle crée par la requête.

Définitions de la syntaxe :

Paramètre	**Description**
Procedure_name	Nom de la procédure
Parameter	Nom de la variable PL/SQL qui est passée, retournée à l'environnement appelant, ou les deux suivant le *mode* choisit
Mode	Type de l'argument : IN (par défaut) OUT IN OUT
Datatype	Type de donnée de l'argument
Bloc PL/SQL	Corps de la procédure définissant les actions à effectuer

3.2.2 Les modes de paramètres de la procédure

Les paramètres de la procédure permettent de transférer des valeurs du et vers l'environnement appelant. Les paramètres possèdent chacun trois modes : IN, OUT et IN OUT.

Type de paramètre	Description
IN (par défaut)	Une valeur constante est passée de l'environnement appelant vers la procédure
OUT	Une valeur est passée de la procédure vers l'environnement appelant
IN OUT	Une valeur constante est passée de l'environnement appelant vers la procédure et une valeur différente peut être renvoyée à l'environnement en utilisant le même paramètre.

3.2.3 Développer des procédures stockées

Pour développer une procédure stockée il faut tout d'abord choisir un environnement de développement tel que Procedure Builder ou SQL*Plus. Ensuite il faut saisir le code en utilisant la syntaxe définie précédemment. Sous Procedure Builder on utilise l'éditeur d'unité de programme et sous SQL*Plus on entre le texte dans un éditeur de texte puis on le sauvegarde en fichier script.*
Enfin il faut compiler le code en *p-code (pseudo-code)*. Avec Procedure Builder il faut juste cliquer sur Save et sous SQL*Plus il faut exécuter le fichier script.

3.2.4 Développer une procédure en utilisant SQL*Plus

Pour la création de procédure avec SQL*Plus il faut d'abord taper le texte de la requête CREATE PROCEDURE dans un éditeur de texte puis l'enregistrer en fichier script. Ensuite, pour le compiler en *p-code*, il suffit de l'exécuter depuis SQL*Plus. Si le terminal renvoie une ou plusieurs erreurs de compilation, la commande SHOW ERRORS permet de les afficher. Lorsque le script est compilé sans erreur il peut être exécuté depuis l'environnement du serveur Oracle.
Un fichier script avec la requête CREATE PROCEDURE (ou CREATE FUNCTION) permet de faire des changements directement dans le fichier si il y a des erreurs de compilation ou de faire des modifications ultérieures. Si une procédure a été compilée et qu'il retourne des erreurs de compilation, elle ne pourra pas être invoquée correctement. Il faut donc s'assurer du bon déroulement de la compilation avant de l'invoquer.
Lors de l'exécution, la commande CREATE PROCEDURE (ou CREATE FUNCTION) stocke le code source dans le dictionnaire de donnée même si la procédure contient des erreurs de compilation. Si l'on veut effectuer des changements, le mieux est donc d'utiliser l'option OR REPLACE ou bien il faut faire un DROP de la procédure.

3.2.5 Développer une procédure en utilisant procedure builder

3.2.5.1 Création d'une procédure client

Grâce au moteur PL/SQL intégré dans l'application Procedure Builder, il est possible de développer des procédures côté client. Pour cela il faut choisir le nœud *Program Units* dans l'arborescence de l'explorateur d'objet puis cliquer sur *Create* pour faire apparaître la boîte de dialogue de création d'une nouvelle unité de programme.
On entre le nom de la procédure en sélectionnant le type *Procedure* (qui est celui sélectionné par défaut). Après validation, la fenêtre de l'éditeur de programme apparaît avec le nom de la procédure et les mots clés IS, BEGIN et END. Le curseur est positionné automatiquement à la ligne suivant le BEGIN.
Une fois le code source entré on clique sur *Compile*. Les messages d'erreurs générés durant la compilation sont affichés dans la fenêtre de messages de compilation. Lorsque l'on sélectionne un message d'erreur dans la fenêtre, le curseur se positionne automatiquement à l'endroit de l'erreur dans le code source. Si la compilation s'effectue correctement, un message le précisant est affiché dans la fenêtre de l'éditeur de programme.
On peut ensuite sauvegarder le code dans un fichier en sélectionnant *Export* dans le menu *File*.
Dans Procedure Builder, les mots clés CREATE et CREATE OR REPLACE ne peuvent pas être utilisés.

3.2.5.2 Création d'une procédure serveur

On peut également utiliser le moteur PL/SQL du serveur pour développer des applications côté serveur. Pour cela il faut tout d'abord se connecter (*File → Connect)* à la base de donnée en utilisant son nom d'utilisateur et son mot de passe. Ensuite on développe le nœud *Database Objects* dans l'explorateur d'objets afin de faire apparaître le nom de notre schéma pour le développer à son tour. On sélectionne ensuite le nœud *Stored Program Units* de ce schéma et on clique sur *Create* pour pouvoir saisir le code source de cette procédure. Le reste de la création de procédures stockées se déroulent comme décrit précédemment pour les procédures. Une fois le code source compilé on clique sur *Save* pour sauvegarder la procédure sur le serveur.

3.2.5.3 Naviguer dans les erreurs de compilation

Procedure Builder affiche les erreurs de compilation dans un panneau séparé qui permet au développeur de débugger facilement son code. Lorsque l'on sélectionne une erreur dans ce panneau, le curseur se place automatiquement à l'endroit de l'erreur dans le code source. Une fois l'erreur résolue on recompile la procédure pour s'assurer de la réussite de la correction.

3.3 Procédures et paramètres

3.3.1 Création de procédures avec des paramètres

Les procédures peuvent prendre en compte des paramètres extérieurs lors de leur exécution. Ces paramètres peuvent être passés en entrée (la valeur est utilisée dans la procédure même), en sortie (la valeur est envoyée à l'environnement appelant) ou les deux.

IN	**OUT**	**IN OUT**
Par défaut	Doit être spécifié	Doit être spécifié
La valeur est passée dans le sous-programme	La valeur est renvoyée à l'environnement appelant	La valeur est passée au sous programme puis une valeur différente est retournée à l'environnement appelant
Le paramètre formel agit comme une constante	C'est une variable non initialisée	C'est une variable initialisée
Le paramètre actuel peut être une expression littérale, constante ou une variable initialisée	Doit être une variable	Doit être une variable

3.3.2 Le paramètre IN

Exemple:

```
SQL> CREATE OR REPLACE PROCEDURE raise_salary
2   (v_id in emp.empno%TYPE)
3   IS
4   BEGIN
5       UPDATE    emp
6       SET       sal = sal * 1.10
7       WHERE     empno = v_id;
8   END raise_salary;
9 /
Procedure created.

SQL> EXECUTE raise_salary (7369)
PL/SQL procedure successfully completed.
```

L'exemple montre une procédure utilisant un paramètre IN. Lorsque la procédure RAISE_SALARY est appelée, le paramètre est utilisé en tant que numéro d'employé pour exécuter l'ordre UPDATE.
Pour appeler une procédure avec un paramètre dans SQL*Plus on utilise la commande EXECUTE :

```
SQL> EXECUTE raise_salary (7569)
```

Pour appeler une procédure depuis Procedure Builder on utilise un appel direct. Pour cela on entre le nom de la procédure et le paramètre actuel au prompt de l'interpréteur de Procédure Builder :

```
PL/SQL> raise_salary (7369)
```

Les paramètres IN sont passés en tant que constantes donc si l'on essaye de modifier la valeur d'un paramètre IN, il se produira une erreur.

3.3.3 Le paramètre OUT

3.3.3.1 Utilisation du paramètre OUT

Le paramètre OUT permet de retourner des valeurs obtenues à l'intérieur de la procédure vers l'environnement appelant. Comme la valeur par défaut pour les paramètres est IN, il faut préciser explicitement OUT lorsque l'on veut retourner une valeur.

Exemple :

```
SQL> CREATE OR REPLACE PROCEDURE query_emp
1    (v_id       IN    emp.empno%TYPE,
2     v_name     OUT   emp.ename%TYPE,
3     v_salary   OUT   emp.sal%TYPE,
4     v_comm     OUT   emp.comm%TYPE)
5    IS
6    BEGIN
7      SELECT     ename, sal, comm
8      INTO       v_name, v_salary, v_comm
9      FROM       emp
10     WHERE      empno = v_id;
11   END query_emp;
12   /
```

→ Ce script crée une procedure acceptant un paramètre externe et renvoyant trois paramètres

Pour qu'une procédure avec un ou plusieurs paramètres OUT fonctionne il faut déclarer autant de variables hôtes que de valeurs retournées par la requête. Ces variables devront être du même type que les valeurs retournées. Ensuite ces variables précédées de deux points (:) seront passées en paramètres de la procédure.

3.3.3.2 Le paramètre OUT et SQL*Plus

Pour exécuter la procédure *query_emp* dans SQL*Plus, on crée d'abord trois variables en utilisant la commande VARIABLE. On appelle ensuite la procédure en indiquant une valeur en entrée et les trois variables précédées de deux points (:) pour les paramètres OUT. Pour voir les valeurs renvoyées dans les variables on utilise la commande PRINT.

```
SQL> VARIABLE g_name     VARCHAR2(15)
SQL> VARIABLE g_sal      NUMBER
SQL> VARIABLE g_comm     NUMBER

SQL> EXECUTE query_emp(7654,:g_name,:g_sal,:g_comm)
Procédure PL/SQL terminée avec succès.

SQL> PRINT g_name
G_NAME
---------------
MARTIN
```

→ Cet exemple affiche la valeur de la variable *g_name* renvoyée à l'environnement appelant par la procédure *query_emp*.

Pour afficher plusieurs variables en même temps il suffit de spécifier tous les noms à la suite dans la liste du PRINT. La commande PRINT ainsi que la commande VARIABLE sont des commandes spécifiques à SQL*Plus.
Lors de l'utilisation de la commande VARIABLE pour définir des variables hôtes, il n'est pas nécessaire de spécifier une taille pour les variables de type NUMBER. Une variable hôte de type CHAR ou VARCHAR2 a une taille par défaut de un, à moins qu'une valeur soit spécifiée entre parenthèses. Afin de ne pas créer d'erreurs il faut s'assurer que les variables peuvent contenir les valeurs renvoyées.

3.3.3.3 Le paramètre OUT et procedure builder

Avec Procedure Builder il faut également déclarer des variables. La commande qui permet de les déclarer est .CREATE. Pour cette commande il faut spécifier le type de donnée, le nom de la variable et la taille de celle-ci. On appelle ensuite la procédure comme dans SQL*Plus afin de stocker les valeurs retournées dans les variables définies. Pour ensuite afficher ces valeurs on utilise la procédure PUT_LINE du packageTEXT_IO.

Exemple :

```
PL/SQL> .CREATE CHAR g_name LENGTH 10
PL/SQL> .CREATE NUMBER g_sal PRECISION 4
PL/SQL> .CREATE NUMBER g_comm PRECISION 4
PL/SQL> QUERY_EMP (7654, :g_name, :g_sal,
     +> :g_comm);
PL/SQL> TEXT_IO.PUT_LINE (:g_name || ' gagne ' ||
     +> TO_CHAR(:g_sal) || ' et une commission de '
     +> || TO_CHAR(:g_comm));
MARTIN gagne 1250 et une commission de 1400
```

→ Cet exemple affiche les valeurs des variables modifiées par la procédure *query_emp*

3.3.4 Le paramètre IN OUT

3.3.4.1 Créer une procédure utilisant IN OUT

Le paramètre IN OUT permet de passer une valeur à la procédure et de retourner une valeur différente à l'environnement appelant. La valeur renvoyée peut être soit l'originale, si la valeur n'est pas modifiée par la procédure, soit une toute autre valeur définie dans la procédure. Un paramètre IN OUT se comporte comme une variable initialisée.

Exemple :

```
SQL> CREATE OR REPLACE PROCEDURE format_phone
2    (v_phone_no IN OUT VARCHAR2)
3    IS
4    BEGIN
5      v_phone_no := '(' || SUBSTR(v_phone_no,1,3) ||
6                    ')' || SUBSTR(v_phone_no,4,3) ||
7                    '-' || SUBSTR(v_phone_no,7);
8 END format_phone;
9 /
```

L'exemple montre la création de la procédure FORMAT_PHONE qui utilise un paramètre IN OUT pour faire une conversion de format sur ce paramètre.
Pour faire fonctionner une procédure avec un paramètre IN OUT il faut préalablement créer et initialiser une variable hôte.

3.3.4.2 Utilisation avec SQL*Plus

Pour invoquer la procédure FORMAT_PHONE, crée précédemment, dans SQL*Plus on va créer une variable hôte par la commande VARIABLE puis initialisée celle-ci grâce à un script PL/SQL.

Exemple :

```
SQL> VARIABLE g_phone_no VARCHAR2(15)
SQL> BEGIN :g_phone_no := '8006330575'; END;
  2  /

Procédure PL/SQL terminée avec succès.

SQL> PRINT g_phone_no

G_PHONE_NO
--------------------------------
8006330575

SQL> EXECUTE format_phone (:g_phone_no)

Procédure PL/SQL terminée avec succès.

SQL> PRINT g_phone_no

G_PHONE_NO
--------------------------------
(800)633-0575
```

→ La procédure FORMAT_PHONE effectue la modification de format sur la variable g_phone_no

3.3.4.3 Utilisation avec Procedure Builder

Dans Procedure Builder, la méthode pour invoquer la procédure FORMAT_PHONE est semblable à celle de SQL*Plus mais la syntaxe est légèrement différente.

Exemple :

```
PL/SQL> .CREATE CHAR g_phone_no LENGTH 15
PL/SQL> BEGIN
     +> :g_phone_no := '8006330575';
     +> END;
PL/SQL> FORMAT_PHONE (:g_phone_no);
PL/SQL> TEXT_IO.PUT_LINE (:g_phone_no);
(800)633-0575
```

3.3.5 Passer des paramètres

3.3.5.1 Méthodes pour passer des paramètres

Lorsqu'une procédure possède plusieurs arguments on dispose de plusieurs méthodes pour spécifier la valeurs des paramètres : par position, par association de nom et par combinaison.

Méthode	Description
Par position	Les valeurs sont listées dans l'ordre dans lequel sont déclarés les paramètres.
Par association de nom	Les valeurs sont listées dans un ordre arbitraire en associant chacune avec le nom de paramètre correspondant en utilisant une syntaxe spéciale (=>).
Par combinaison	C'est une combinaison des deux méthodes précédentes : les premières valeurs sont listées par position et le reste utilise la syntaxe spéciale de la méthode par association de nom.

3.3.5.2 L'option DEFAULT pour les paramètres

Lors de la déclaration des paramètres on peut spécifier une option DEFAULT à la suite du type de donnée.

```
Nom_variable [IN|OUT|IN OUT] type de donnée [DEFAULT valeur]
```

Cette option permet à l'utilisateur de ne pas spécifier de paramètres quand une procédure en réclame. Si les paramètres ne sont pas spécifiés, la procédure s'exécutera avec la valeur définie dans l'option DEFAULT.

Exemple :

```
SQL> CREATE OR REPLACE PROCEDURE add_dept
1    (v_name   IN   dept.dname%TYPE   DEFAULT 'unknown',
2     v_loc    IN   dept.loc%TYPE     DEFAULT 'unknown')
3    IS
4    BEGIN
5        INSERT INTO dept
6        VALUES (dept_deptno.NEXTVAL, v_name, v_loc);
7    END add_dept;
8 /
```

→ Cette requête crée une procédure ADD_DEPT qui permet d'ajouter un nouveau department à la table DEPT sans forcément préciser un nom et une localisation.

3.3.5.3 Exemple d'utilisation de paramètres

Exemple :

```
SQL> BEGIN
2    add_dept;
3    add_dept ( 'TRAINING', 'NEW YORK');
4    add_dept ( v_loc => 'DALLAS', v_name =>'EDUCATION');
5    add_dept ( v_loc => 'BOSTON') ;
6    END;
7    /
Procédure PL/SQL terminée avec succès.

SQL> SELECT * FROM dept;

DEPTNO DNAME          LOC
------ -------------- -------------
   ... ...            ...
    41 unknown        unknown
    42 TRAINING       NEW YORK
    43 EDUCATION      DALLAS
    44 unknown        BOSTON
```

→ Ce bloc anonyme illustre les différents moyens d'appeler une fonction contenant des paramètres. On voit que lorsque qu'un paramètre est manquant, celui-ci est remplacé par la valeur définie dans le DEFAULT : « unknown ».
Le premier appel de la procédure fonctionne car une valeur DEFAULT a été définie

3.4 Les sous programmes

3.4.1 Déclarer des sous-programmes

Un sous programme peut être déclaré dans n'importe quel bloc PL/SQL. C'est une alternative à la création de procédures à usage unique pour les appeler depuis d'autres procédures.
Le sous programme doit être déclaré dans la section déclarative du bloc et doit être le dernier élément de cette section, après tous les autres éléments du programme. **Si une variable est déclarée après la fin du sous programme cela créera une erreur de compilation.**

Exemple :

```
CREATE OR REPLACE PROCEDURE LEAVE_EMP2
  (v_id IN emp.empno%TYPE)
IS
  PROCEDURE log_exec
  IS
  BEGIN
    INSERT INTO log_table (user_id, log_date)
    VALUES (user,sysdate);
  END log_exec;
BEGIN
  DELETE FROM emp
  WHERE empno = v_id;
  log_exec;
END leave_emp2;
```

→ Cette requête crée une procédure acceptant un parametre. Cette procédure comporte un sous programme qui est appelé dans la section exécutable du bloc.

3.4.2 Invoquer une procédure depuis un bloc anonyme

Les procédures peuvent être appelées depuis n'importe quel outil ou langage prenant en compte le PL/SQL. Pour appeler une procédure depuis un bloc anonyme il faut indiquer son nom, en passant les paramètres éventuels entre parenthèses, suivi d'un point virgule.

Exemple :

```
DECLARE
   v_id NUMBER := 7900;
BEGIN
   raise_salary(v_id); --appel la procédure
   COMMIT;
...
END;
```

→ Ce bloc anonyme appelle la procédure RAISE_SALARY définie précédemment en lui donnant le paramètre *v_id*.initialisé dans la section déclarative.

3.4.3 Invoquer une procédure depuis une procédure stockée

Les procédures peuvent être également appelées à partir de procédures stockées. Pour ce faire, il faut utiliser le nom de la procédure comme pour l'appel à partir de blocs anonymes.

Exemple :

```
SQL> CREATE OR REPLACE PROCEDURE process_emps
2    IS
3      CURSOR emp_cursor IS
4         SELECT    empno
5         FROM      emp;
6    BEGIN
7       FOR emp_rec IN emp_cursor
8       LOOP
9          raise_salary(emp_rec.empno); --appel de la procédure
10      END LOOP;
11      COMMIT;
12   END process_emps;
13  /
```

→ La procédure PROCESS_EMPS utilise un curseur pour traiter toutes les données de la table EMP et passer le numéro de chaque employé à la procédure RAISE_SALARY, qui incrémente le salaire de 10 %.

3.5 Gestion des exceptions

3.5.1 Exceptions traitées

Lorsque l'on développe des procédures qui seront appelées depuis d'autres procédures, il faut prendre en compte les effets que peuvent avoir les exceptions traitées et non traitées sur la transaction et sur la procédure appelante.
Une transaction regroupe l'ensemble des ordres de manipulation de données effectués depuis le dernier COMMIT. Pour la contrôler on peut utiliser les commandes de contrôle de transaction, COMMIT, ROLLBACK et SAVEPOINT.
Dans une procédure appelant une autre procédure il faut faire attention à la manière dont les exceptions levées affectent la transaction et dont l'exception est propagée.
Quand une exception est levée dans un programme appelé, la section de gestion des exceptions prend automatiquement le contrôle du bloc. Si l'exception est traitée dans cette section, le bloc se termine correctement et le contrôle est rendu au programme appelant. Tout ordre DML effectué avant que l'exception ne soit levée reste dans la transaction

3.5.2 Exceptions non traitées

Lorsque l'exception n'est pas traitée par la section de gestion des exceptions, tout ordre DML effectué dans le bloc du programme appelé est implicitement annulé (ROLLBACK), le bloc se termine et le contrôle est rendu à la section de gestion des exceptions de du programme appelant.
Si l'exception est traitée par la procédure appelante, tous les ordres DML effectués dans ce bloc sont conservés dans la transaction.
Si l'exception n'est pas traitée par la procédure appelante, tous les ordres DML effectués dans ce bloc sont implicitement annulés (ROLLBACK), le bloc se termine et l'exception est propagée à l'environnement appelant.

3.6 Supprimer des procédures

Lorsqu'une procédure n'est plus utilisée, on peut utiliser un ordre SQL dans SQL*Plus ou dans l'interpréteur de Procedure Builder pour la supprimer (DROP). La méthode de suppression diffère légèrement selon le type de la procédure (cliente ou serveur).
Sous SQL*Plus on ne peut supprimer que des procédures serveurs alors que sous Procedure Builder les procédures serveurs et clientes peuvent être supprimées.

3.7 Supprimer des procédures serveur

3.7.1.1 Avec SQL*Plus

Sous SQL*Plus, on utilise la commande DROP PROCEDURE pour supprimer la procédure serveur souhaitée.

```
DROP PROCEDURE  procedure_name
```

Exemple :

```
SQL> DROP PROCEDURE  raise_salary;

Procédure supprimée.
```

→ Cet ordre permet de supprimer la procédure RAISE_SALARY créée précédemment.

Il est impossible d'effectuer un ROLLBACK après avoir supprimer une procédure car une commande de définition de donnée (DDL) ne peut pas être annulée.

3.7.1.2 Avec Procedure Builder

Pour supprimer une procédure serveur sous Procedure Builder, il faut tout d'abord se connecter à la base de données. Dans l'explorateur d'objet, on développe le nœud *Database Object* pour faire apparaître les différents schémas disponibles. On développe ensuite le schéma du propriétaire de la procédure puis le nœud *Stored Progam Units* (Unités de programme stockés) pour faire apparaître toutes les procédures existantes. On sélectionne ensuite la procédure à supprimer puis on clique sur *Delete* dans l'explorateur d'objet. Un message de confirmation apparaît alors, on clique ensuite sur *Yes* pour la supprimer définitivement.
Pour supprimer une procédure du serveur, on peut également cliquer sur *Drop* dans l'éditeur de programmes stockés.

3.8 Supprimer des procédures client

Pour supprimer une procédure client avec Procedure Builder, on développe le nœud *Program Units*, puis on sélectionne la procédure que l'on souhaite supprimer. On clique ensuite sur *Delete* dans l'explorateur d'objet et on clique sur *Yes* à l'apparition de la boîte de dialogue de confirmation.
Si le code de la procédure a été exporté dans un fichier texte et que l'on veut le supprimer, il faut passer par le système d'exploitation.

4 CREATION DE FONCTIONS

4.1 Les fonctions

4.1.1 Aperçu des fonctions stockées

Une fonction stockée est un bloc PL/SQL nommé pouvant accepter des paramètres et être appelée de la même façon que les procédures. Généralement, on utilise une fonction pour calculer une valeur. Les procédures et les fonctions possèdent une structure semblable sauf qu'une fonction doit retourner une valeur à l'environnement appelant.
Comme les procédures, les fonctions sont composées de quatre parties : un en-tête, une section déclarative, une partie exécutable et une partie optionnelle de gestion d'erreur. Une fonction doit avoir une clause RETURN dans l'en-tête et au moins un RETURN dans la partie exécutable.
Les fonctions facilitent la réutilisation et la maintenance. Une fois validée, les fonctions sont stockées dans la base de données en tant qu'objet de base de données et peuvent ainsi être réutilisées dans de nombreuses applications. Si la définition change, seule la fonction est affectée ce qui permet une maintenance simple.
Les fonctions peuvent être appelées dans une expression SQL ou dans une expression PL/SQL. Dans une expression SQL, la fonction doit obéir à certaines règles syntaxiques pour contrôler les effets secondaires.
Dans une expression PL/SQL, l'identifiant de la fonction se comporte comme une variable dont la valeur dépend du paramètre qu'on lui passe.

4.1.2 Syntaxe pour la création de fonctions

Pour créer une fonction, on utilise la commande CREATE FUNCTION, dans laquelle on peut spécifier une liste de paramètres. Dans cette commande on doit définir la valeur qui sera retournée à l'environnement appelant et définir les actions effectuées par le bloc PL/SQL standard.

```
CREATE [OR REPLACE] FUNCTION function_name
( parameter1 [ mode1] datatype1,
parameter2 [ mode2] datatype2,
. . .)
RETURN datatype
IS|AS
PL/SQL Block;
```

Paramètre	Description
Function_name	Nom de la fonction
Argument	Nom de la variable PL/SQL dont la valeur est passée à la fonction
Mode	Le type de paramètre ; seul le paramètre IN doit être déclaré
Datatype	Type de donnée du paramètre
RETURN *datatype*	Type de donnée de la valeur RETURN qui doit être renvoyée par la fonction
PL/SQL Block	Corps de la procédure définissant les actions effectuées par la fonction

L'option REPLACE indique que si la fonction existe déjà, elle sera remplacée par la nouvelle version crée par la requête.
Le type de donnée du RETURN ne doit avoir de taille spécifiée.
Le bloc PL/SQL commence soit par un BEGIN soit par une section de déclaration de variables locales et se tremine par END ou END *function_name.* Il doit obligatoirement y avoir au moins une expression RETURN (variable). Il est **impossible** de faire référence à des variables hôtes ou à des variables de substitution dans le bloc PL/SQL d'une fonction stockée.

4.1.3 Création de fonction

La méthode de création d'une fonction est semblable à celle de création de procédure. On choisit d'abord un environnement de développement (Procedure Builder ou SQL*Plus) dans lequel on entre la syntaxe de création. Ensuite on compile le code pour obtenir du *p-code*.
L'utilisation de plusieurs expressions RETURN dans un bloc PL/SQL est autorisée mais lors de la compilation une seule sera prise en compte et donc une seule valeur sera retournée lors de l'exécution. On utilise habituellement plusieurs RETURN lorsque le bloc contient une condition IF.

*4.2 Les fonctions dans SQL*Plus*

4.2.1 Création de fonctions stockées

Pour créer une fonction stockée à partir de SQL*Plus on tape l'expression CREATE FUNCTION dans un éditeur de texte, puis on sauvegarde ce fichier en tant que fichier script (*.sql). Ce fichier script doit ensuite être exécuté dans SQL*Plus afin de le compiler en *p-code.* Si la compilation produit des erreurs, on utilise la commande SHOW ERRORS pour les corriger. Une fois la compilation effectuée sans erreurs, on appelle la fonction depuis un environnement Oracle Server.
Un fichier script contenant l'expression CREATE FUNCTION nous permet de modifier la requête si il y a des erreurs de compilation ou d'exécution ou de faire des changements ultérieurs. Il est impossible d'appeler une fonction contenant des erreurs de compilation ou d'exécution.
L'exécution de la commande CREATE FUNCTION stocke le code source dans le dictionnaire de données même si la fonction contient des erreurs de compilation. Il faut donc supprimer la fonction (DROP) ou bien utiliser la syntaxe OR REPLACE si l'on veut effectuer des changements dans une fonction.

Exemple :

```
SQL> CREATE OR REPLACE FUNCTION get_sal
  2    (v_id  IN    emp.empno%TYPE)
  3    RETURN NUMBER
  4  IS
  5    v_salary     emp.sal%TYPE :=0;
  6  BEGIN
  7    SELECT   sal
  8    INTO     v_salary
  9    FROM     emp
 10    WHERE    empno = v_id;
 11    RETURN   (v_salary);
 12  END get_sal;
 13  /

Fonction créée.
```

→ Cette requête crée une fonction acceptant un paramètre IN et retournant une valeur de type NUMBER correspondant au salaire de l'employé dont le numéro a été fournit en paramètre.

4.2.2 Exécuter des fonctions

Les fonctions sont appelées à l'intérieur d'expression PL/SQL. Pour que la fonction s'exécute convenablement il faut créer une variable hôte pour y stocker la valeur retournée. Lors de l'exécution, la variable hôte est initialisée avec la valeur renvoyée par le RETURN.
Une fonction peut accepter plusieurs paramètres IN mais elle ne doit retourner qu'une seule valeur.

Exemple :

```
SQL> START get_salary.sql
Fonction créée.

SQL> VARIABLE g_salary number
SQL> EXECUTE :g_salary := get_sal(7934)
Procédure PL/SQL terminée avec succès.

SQL> PRINT g_salary

  G_SALARY
----------
      1300
```

→ Cet exemple crée une variable hôte dans laquelle est stockée la valeur renvoyée par la fonction utilisant le paramètre (numéro d'employé). La commande EXECUTE permet ici de faire exécuter à SQL*Plus une expression PL/SQL

4.3 Les fonctions dans Procedure Builder

4.3.1 Création de fonction

Comme il y a un moteur PL/SQL dans l'outil client de Procedure Builder, il est possible de développer des fonctions côté client. Avec Procedure Builder, on peut aussi utiliser le moteur PL/SQL du serveur pour développer des fonctions côté serveur.
La fonctionnalité glisser-déposer de Procedure Builder permet de déplacer facilement des fonctions entre le client et le serveur.

4.3.2 Exemple de création de fonction

Pour créer une fonction avec Procedure Builder on sélectionne le nœud *Program Units* dans l'explorateur d'objet puis on clique sur *Create*. On choisit un nom pour la fonction dans la boîte de dialogue de création d'une nouvelle unité de programme. Dans l'exemple on choisit Tax. On choisit ensuite le type *Function* puis on clique sur *OK* pour faire apparaître la fenêtre de l'éditeur de programme. On tape ensuite le code avant de cliquer sur *Compile*.

```
FUNCTION tax
      (v_value IN NUMBER)
      RETURN NUMBER
IS
BEGIN
      RETURN (v_value * .08);
END tax;
```

→ Ce code crée une fonction calculant la taxe sur un salaire donné en argument.

Si la compilation s'effectue sans erreurs le message *Successfully Compiled* est affiché. Sinon les erreurs sont signalées dans le panneau d'affichage des erreurs.
Il faut éviter d'utiliser les paramètres OUT et IN OUT avec les fonctions car elles sont conçues pour retourner une valeur unique.

4.3.3 Exécuter des fonctions

Pour exécuter une fonction dans Procedure Builder il faut tout d'abord créer une variable hôte pour y stocker la valeur retournée. Pour cela on utilise la syntaxe CREATE au prompt de l'interpréteur PL/SQL.
On crée ensuite une expression PL/SQL appelant la fonction TAX en lui passant en argument une valeur numérique. Les deux points (:) indiquent que l'on fait référence à une variable hôte.
On observe le résultat de la fonction en utilisant la procédure PUT_LINE du package TEXT_IO.

Exemple :

```
PL/SQL> .CREATE NUMBER x PRECISION 4
PL/SQL> :x := tax(1000);
PL/SQL> TEXT_IO.PUT_LINE (TO_CHAR(:x));
80
```

4.4 Les fonctions définies par l'utilisateur dans du SQL

4.4.1 Avantages des fonctions dans des expressions SQL

Les expressions SQL peuvent faire référence à des fonctions PL/SQL définies par l'utilisateur. Les fonctions définies par l'utilisateur peuvent être utilisées partout où une fonction SQL peut être utilisée.
Les fonctions définies par l'utilisateur permettent d'effectuer des calculs complexes, maladroit ou non disponibles dans SQL plus facilement.
Elles augmentent également l'indépendance des données en effectuant des analyses de données complexe au niveau du serveur Oracle plutôt que de la faire faire par les applications.
Elles augmentent l'efficacité des requêtes en exécutant les fonctions dans la requête plutôt que dans l'application.
Elles permettent également de manipuler des nouveaux types de données (par exemple des latitudes et des longitudes) en encodant les chaînes de caractères et en utilisant des fonctions pour agir sur ces chaînes.

Exemple :

```
SQL> SELECT   empno, ename, sal, tax(sal)
  2  FROM     emp;

     EMPNO ENAME             SAL   TAX(SAL)
---------- ---------- ---------- ----------
      7369 SMITH             800         64
      7499 ALLEN            1600        128
      7521 WARD             1250        100
…
      7902 FORD             3000        240
      7934 MILLER           1300        104

14 ligne(s) sélectionnée(s).
```

→ Cet ordre SQL utilise la fonction TAX dans sa liste de SELECT

4.4.2 Emplacements d'où appeler les fonctions

Les fonctions PL/SQL définies par l'utilisateur peuvent être appelées depuis toute expression SQL dans laquelle on peut utiliser une fonction prédéfinie.
C'est-à-dire que l'on peut utiliser une fonciton PL/SQL dans la liste d'un ordre SELECT, dans les conditions des clauses WHERE et HAVING, dans les clauses CONNECT BY, START WITH, ORDER BY et GROUP BY, dans la clause VALUES d'un ordre INSERT et dans la clause SET d'un ordre UPDATE.

4.4.3 Appel de fonctions : restrictions

Pour pouvoir être appelée depuis une expression SQL, une fonction définie par l'utilisateur doit respecter certaines conditions :

- Seules les fonctions stockées peuvent être utilisée. Les procédures stockées ne peuvent pas être appelées.
- Une fonction définie par l'utilisateur utilisée en SQL doit être une fonction SINGLE-ROW et pas une fonction de groupe.
- Ces fonctions n'acceptent que des paramètres IN, pas OUT ou IN OUT.

- Les types de données retournés doivent être des types de données SQL valides : CHAR, VARCHAR2, DATE ou NUMBER. Les types de données spécifiques au PL/SQL (BOOLEAN, RECORD, TABLE) ne peuvent pas être utilisés.
- La fonction ne doit pas modifier la base de donnée, par conséquent les ordres INSERT, UPDATE ou DELETE sont interdits dans une fonction appelée depuis du SQL.
- Les paramètres d'une fonction PL/SQL appelée depuis une expression SQL doit utiliser la notation par position. La position par nom n'est pas supportée dans le SQL.
- On doit posséder la fonction ou bien avoir le privilège EXECUTE sur celle-ci pour pouvoir l'appeler depuis un ordre SQL.
- Les fonctions PL/SQL stockées ne peuvent pas être appelées depuis la clause CHECK d'une commande CREATE ou ALTER TABLE ni être utilisée pour spécifier une valeur par défaut pour une colonne.
- Les fonctions appelées ne doivent pas faire appel à un autre sous programme ne respectant pas ces règles.

L'utilisation de fonction PL/SQL dans des ordres SQL est disponible depuis PL/SQL 2.1. Les outils utilisant une version plus ancienne de PL/SQL ne supportent pas cette fonctionnalité.

4.5 Supprimer des fonctions

4.5.1 Supprimer des fonctions serveur

Lorsqu'une fonction stockée n'est plus utilisée, on peut la supprimer en utilisant un ordre SQL dans SQL*Plus ou en utilisant le panneau de l'interpréteur Procedure Builder pour faire un DROP.
Pour supprimer une fonction côté serveur sous SQL*Plus on utilise la commande DROP FUNCTION.

```
DROP FUNCTION  function_name
```

Exemple :

```
SQL> DROP FUNCTION get_sal;
Fonction supprimée.
```

→ Cette requête supprime la function get_sal.

DROP FUNCTION est un ordre DDL, il est donc auto-commité et ne peut pas être annulé par un ROLLBACK.

Pour supprimer une fonction stockée avec Procedure Builder il faut tout d'abord se connecter à la base de donnée. On développe ensuite le nœud *Database Objects*, puis le schéma du propriétaire de la fonction. On sélectionne la fonction que l'on souhaite supprimer puis on clique sur *Drop* dans l'explorateur d'objet. On doit ensuite confirmer la suppression dans la fenêtre d'avertissement qui apparaît.

4.5.2 Supprimer des fonctions client

Pour supprimer une fonction côté client on utilise Procedure Builder. Pour cela on développe le nœud *Program Units* puis on sélectionne la fonction que l'on veut supprimer. On clique ensuite sur *Delete* dans l'explorateur d'objets en confirmant la suppression dans la boîte de dialogue qui apparaît.
Si le code source de la fonction a été exporté dans un fichier texte et que l'on veut l'effacer du client il faut utiliser les fonctionnalités du système d'exploitation.

4.6 Procédure ou fonction ?

4.6.1 Récapitulatif

Les procédures sont créées pour stocker une série d'actions à exécuter ultérieurement. Une procédure peut accepter ou non des paramètres, qui ne sont pas limités en nombre et peuvent être transférés du et vers l'environnement appelant. Une procédure ne retourne pas nécessairement une valeur.
Les fonctions sont créées pour calculer une valeur, elles doivent retourner une valeur à l'environnement appelant. Une fonction peut accepter ou non des paramètres qui sont transmis de l'environnement. Une fonction ne peut retourner qu'une seule valeur et ne peut pas accepter de paramètre OUT ou IN OUT.

Cependant si l'on en déclare il ne se produira pas d'erreurs de compilation mais il est conseillé de ne jamais en utiliser.

4.6.2 Comparaison entre procédures et fonctions

Procédure	Fonction
S'exécute comme une requête PL/SQL	Est appelé dans une expression
Pas de type de donnée RETURN	Doit avoir un type de donnée RETURN
Peut retourner une, aucune ou plusieurs valeurs	Ne retourne qu'une seule valeur

Une procédure contenant un seul paramètre OUT peut être réécrite en tant que fonction en utilisant le paramètre OUT pour la valeur retournée.

4.6.3 Les avantages des procédures et fonctions stockées

En plus d'apporter un développement modulaire des applications, les procédures et fonctions stockées ont d'autres avantages :

- Amélioration des performances
  Evite de reparser les lignes pour les différents utilisateurs en exploitant la zone SQL partagée.
  Evite le parsage PL/SQL à l'exécution en parsant lors de la compilation.
  Réduit le nombre d'appels à la base de données et diminue le traffic réseau en envoyant les commandes par paquets.
- Amélioration de la maintenance
  Possibilité de modifier les routines online sans interférer avec les autres utilisateurs
  Possibilité de modifier une routine pour affecter toutes les applications
  Possibilité de modifier une seule routine pour éviter les tests en double
- Amélioration de l'intégrité et de la sécurité des données
  Contrôle indirect de l'accès aux objets de la base de données des utilisateurs ne possédant pas de privilèges de sécurité
  Assurance que les actions liées sont exécutées ensemble, ou pas du tout, en centralisant l'activité des tables liées.

5 CREATION DE PACKAGES

5.1 Les packages

5.1.1 Aperçu des packages

Les packages sont des groupes comprenant des types, des éléments et des sous-programmes PL/SQL logiquement associés. Par exemple un package Ressources Humaines pourrait contenir les procédures embauche et licenciement, les fonctions commission et bonus et les variables d'exemption de taxe.
Généralement un package est constitué de deux parties stockées séparément dans la base de données : la spécification et le corps.
La spécification est l'interface pour les applications. Elle déclare les types, les variables, les constantes, les exceptions, les curseurs et les sous-programmes que l'on peut utiliser dans le package.
Le corps définit les curseurs et les sous-programmes et ce qui a été définit dans la spécification.
Le package lui-même ne peut pas être appelé, recevoir de paramètre ou être imbriqué. Cependant un package a le même format qu'un sous-programme. Une fois écrit et compilé, le contenu peut être partagé par plusieurs applications.
Quand on appelle un élément de package pour la première fois, tout le package est chargé en mémoire. Par conséquent les appels suivant à l'élément en question ne requièrent pas d'écriture ou de lecture sur le disque.

5.1.2 Les composantes d'un packages

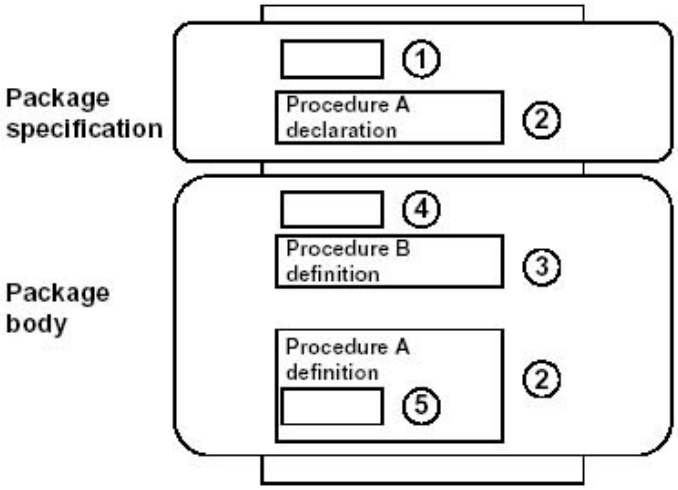

Figure 1 : Structure d'un package

Sur le schéma on distingue les zones de déclarations : pour les variables publiques (1), les procédures publiques (2), les procédures privées (3), les variables privées spécifiques au package (4) et les variables locales spécifiques à la procédure (5).
Le package se crée en deux parties : on définit d'abord la spécification et ensuite on crée le corps du package.
Les éléments publics d'un package sont ceux que l'on déclare dans la spécification et qui sont définit dans le corps. Les éléments privés d'un package sont ceux qui sont définis uniquement dans le corps.

Portée de l'élément	Description	Emplacement dans le package
Public	Peut être utilisé depuis n'importe quel environnement du Serveur	Déclaré dans la spécification du package et peut être définit dans le

	Oracle.	corps du package
Privé	Peut être utilisé par d'autres éléments faisait partie du même package uniquement	Déclaré et définit dans le corps du package

Le Serveur Oracle stocke la spécification et le corps du package séparément dans la base de données, ce qui permet de changer la définition d'un élément du corps du package sans que Oracle ait à désactiver d'autres objets du schéma qui font appel ou font référence à cet élément.

5.1.3 Référencer les objets d'un package

Les variables définies dans le package n'ont pas toutes la même visibilité c'est-à-dire que selon l'endroit où l'on se trouve on ne pourra pas utiliser certaines variables.

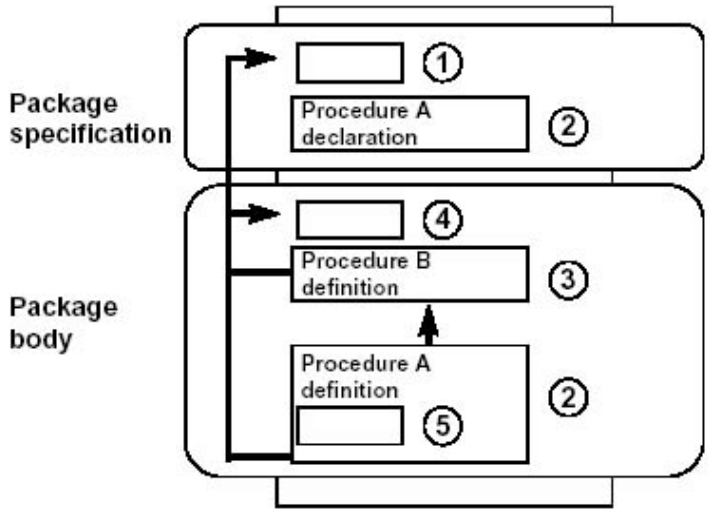

Figure 2 : Portée des éléments d'un package

Les éléments d'un package peuvent avoir une visibilité locale ou globale.
Une variable dont la visibilité est locale est une variable définie dans un sous-programme du package. Cette variable ne sera pas visible par les utilisateurs externes.
Les variables privées (4) sont des variables locales pour le package. Celles-ci sont définies dans le corps du package. Ces variables sont accessibles uniquement par d'autres objets du même package mais ne sont pas visible par des sous-programmes ou des objets extérieurs au package.
Les variables locales (5) sont des variables locales pour les procédures. Celles-ci sont définies au niveau de la fonction ou de la procédure. Ces variables sont accessibles uniquement par le sous-programme dans lequel elles sont déclarées. Elles ne sont visibles par aucun autre sous-programme, même ceux contenus dans le même package.
Une variable dont la visibilité est globale est une variable définie dans la spécification du package. Cette variable peut être utilisée et modifiée hors du package et est visible par les utilisateurs externes.

5.2 Créer des packages

5.2.1 Développement d'un package

Pour développer un package on choisit tout d'abord un environnement de développement : SQL*Plus ou Procedure Builder.
On tape ensuite la syntaxe. Sous Procedure Builder on utilise l'éditeur d'unités de programmes et sous SQL*Plus on utilise un éditeur de texte puis on enregistre le code en tant que fichier script.

Enfin on compile le code du package. Le code source est compilé en *p-code*. Pour compiler le package sous Procedure Builder il suffit de cliquer sur *Save*, sous SQL*Plus on lance le fichier script créé. Une fois le code source compilé, il est stocké dans le dictionnaire de données.
Pour faciliter le développement ultérieur, il est conseillé de sauvegarder dans des fichiers textes séparés le code du CREATE PACKAGE : un fichier pour la spécification et un pour le corps.
Une spécification de package peut exister sans qu'elle soit associée à un corps, mais le contraire n'est pas possible. Le corps d'un package doit toujours être associé à une spécification.
Si l'on a incorporé dans le package le code d'une procédure existante, il est conseillé de supprimer la procédure indépendante à l'aide de la commande DROP.

5.2.2 Création des spécifications du package

Pour créer la spécification d'un package on utilise la commande CREATE PACKAGE. Il faut spécifier toutes les structures publiques dans la spécification du package. On peut spécifier l'option REPLACE si la spécification du package existe déjà. Si nécessaire on initialise les variables avec une constante ou une formule dans la spécification, sinon la variable est initialisée implicitement à NULL.

```
CREATE [OR REPLACE] PACKAGE package_name
IS|AS
               public type and item declarations
               subprogram specifications
END package_name;
```

Paramètre	Description
Package_name	Nom du package
Public type and item declarations	Déclare les variables, les constantes, les curseurs, les exceptions
Subprogram specifications	Déclare les sous-programmes PL/SQL

5.2.3 Déclaration d'éléments publics

Dans la spécification du package on déclare les variables publiques, les procédures publiques et les fonctions publiques. Les fonctions et les procédures publiques sont des routines qui peuvent être appelées à plusieurs reprises par d'autres éléments du même package, ou d'en dehors du package.
Pour déclarer les éléments publics on spécifie le type de l'élément suivi de son nom, dans la section de spécification.

```
CREATE [OR REPLACE] PACKAGE package_name
IS|AS
     TYPE name [parameter]
END package_name;
```

5.2.4 Création de spécification de package : exemple

Exemple :

```
SQL> CREATE OR REPLACE PACKAGE comm_package IS
  2     PROCEDURE reset_comm
  3          (v_comm IN NUMBER);
  4  END comm_package;
  5  /
```

→ Cette requête crée un package déclarant

5.2.5 Création du corps du package

Pour créer le corps du package on utilise la commande CREATE PACKAGE BODY. Dans le corps du package on définit tous les éléments publics et privés du package. On peut spécifier l'option REPLACE pour supprimer et remplacer une version existant déjà.

```
CREATE [OR REPLACE] PACKAGE BODY package_name
IS|AS
           private type and item declarations
           subprogram bodies
END package_name;
```

Paramètre	Description
Package_name	Nom du package
Private type and item declarations	Déclaration des variables, constantes, curseurs, exceptions ou types
Subprogram bodies	Définition des sous-programmes PL/SQL publics et privés

L'ordre dans lequel les sous-programmes sont définis est très important. Il faut déclarer les variables avant qu'une autre variable ou un sous-programme y fasse référence. Il faut déclarer ou définir les sous-programmes privés avant de les appelés depuis d'autres sous-programmes. Le plus souvent, toutes les variables et sous-programmes privés sont déclarés en premier dans le corps du package et ensuite on définit les sous-programmes publics.

5.2.6 Eléments publics et privés

Pour clarifier et modulariser au maximum le code des procédures et fonctions publics on peut définir des fonctions et procédures privées dont la visibilité est limitée au package.
Pour créer des fonctions et procédures privées il faut entrer la syntaxe classique du type de ce sous-programme comme elle est expliquée précédemment dans ce cours.
Lorsque l'on code le corps du package, la définition d'un sous-programme privé doit être avant celle d'un sous-programme public.

5.2.7 Création de corps de package : exemple

Exemple :

```
SQL> CREATE OR REPLACE PACKAGE BODY comm_package
  2  IS
  3     FUNCTION   validate_comm
  4       (v_comm   IN   NUMBER)
  5       RETURN BOOLEAN
  6     IS
  7       v_max_comm    NUMBER;
  8     BEGIN
  9       SELECT   max(comm)
  10      INTO     v_max_comm
  11      FROM     emp;
  12      IF     v_comm > v_max_comm
  13      THEN     RETURN(FALSE);
  14      ELSE     RETURN(TRUE);
  15      END IF;
  16    END validate_comm;
  17    PROCEDURE   reset_comm
  18      (v_comm IN NUMBER)
  19    IS
  20      l_comm   NUMBER   := 10;
  21    BEGIN
  22      IF     validate_comm(v_comm)
  23      THEN   l_comm:=v_comm;
  24      ELSE
  25         RAISE_APPLICATION_ERROR
  26             (-20210,'Invalid commission');
  27      END IF;
  28    END reset_comm;
  29  END comm_package;
  30  /
```

→ Ce package définit une fonction qui valide la commission, celle ci ne doit pas être supérieure à la plus grande touchée par les employés. Il définit également une procédure qui appelle la fonction précédente pour réinitialiser et valider la valeur trouvée.

5.2.8 Directives pour développer des packages

Il faut essayer de faire des packages aussi généraux que possible afin de pouvoir les réutiliser dans des applications futures. Il faut aussi éviter de créer des packages qui dupliquent des fonctions fournies par Oracle.
La spécification du package définit la structure de l'application, donc il faut toujours définir la spécification, puis le corps du package.
La spécification du package ne doit contenir que les éléments devant être visible par les autres utilisateurs du package. Ainsi les autres développeurs ne pourront pas faire mauvaise utilisation en basant leur code sur des éléments non appropriés.
Pour réduire la nécessité de recompilation quand le code est changé il faut placé le plus petit nombre possible d'éléments dans la spécification du package. Les changements dans le corps du package ne nécessitent pas la recompilation des éléments dépendants, alors que les changements dans la spécification du package nécessitent la recompilation de tous les sous-programmes stockés faisant référence au package.

5.2.9 Les variables globales

On peut également créer des spécifications de packages qui ne nécessitent pas de corps de package. Ces packages particulier sont uniquement constitué de déclaration et d'initialisation de variables publiques. Les variables publiques (globales) sont des variables qui existeront toute la durée de la session de l'utilisateur.

Exemple 1 :

```
SQL> CREATE OR REPLACE PACKAGE global_vars IS
  2    mile_2_kilo   CONSTANT  NUMBER  := 1.6093;
  3    kilo_2_mile   CONSTANT  NUMBER  := 0.6214;
  4    yard_2_meter  CONSTANT  NUMBER  := 0.9144;
  5    meter_2_yard  CONSTANT  NUMBER  := 1.0936;
  6  END global_vars;
  7  /
```

→ Cet exemple montre un package ne contenant que des taux de conversion sous forme de variables globales.

Exemple 2 :

```
SQL> EXECUTE DBMS_OUTPUT.PUT_LINE -
> ('20 miles ='||20* global_vars.mile_2_kilo||' km')
20 miles =32.186 km

Procédure PL/SQL terminée avec succès.
```

→ Cet exemple montre qu'un corps de package n'est pas obligatoire pour que cette spécification de package puisse s'exécuter correctement.

5.3 Manipuler les packages

5.3.1 Exécuter une procédure publique d'un package

Une procédure publique est déclarée dans la spécification du package et est définie dans le corps du package. Par conséquent il est possible de l'appeler directement depuis l'environnement SQL*Plus. Pour cela on utilise la commande EXECUTE en précisant le nom du package, de la procédure et les arguments, s'il y a lieu d'en fournir.

```
SQL> EXECUTE  package_name. procedure_name( parameters)
```

Exemple :

```
SQL> EXECUTE comm_package.reset_comm(15)
```

→ Cet ordre SQL appelle la procédure RESET_COMM du package COMM_PACKAGE en donnant un argument. Cet ordre peut être exécuté à n'importe quel moment pendant la session de travail.

5.3.2 Invoquer des éléments de packages

Une fois le package stocké dans la base de données, il est possible de faire appel à un élément de package depuis le package même ou bien en dehors du package. Cette caractéristique dépend du type de l'élément : si il est privé seul le premier cas est possible, si il est public les deux cas sont possibles.
Lorsque l'on appelle une procédure ou une fonction à l'intérieur du même package, il n'est pas nécessaire de préciser le nom du package.

Exemple 1 :

```
CREATE OR REPLACE PACKAGE BODY comm_package IS
  . . .
 PROCEDURE reset_comm
  (v_comm IN NUMBER)
 IS
  l_comm NUMBER := 10;
 BEGIN
  IF validate_comm(v_comm)
  THEN l_comm := v_comm;
  ELSE
   RAISE_APPLICATION_ERROR
      (-20210, 'Invalid commission');
  END IF;
 END reset_comm;
END comm_package;
```

→ Cette procédure fait appel à la fonction VALIDATE_COMM qui est dans le même package COMM_PACKAGE donc son nom n'est pas précisé.

Lorsque l'on appelle une procédure ou une fonction depuis l'extérieur d'un package il faut spécifier le nom du sous-programme ainsi que le nom du package. Si le package se trouve dans un autre schéma, il faut préciser le nom du schéma avant le nom du package. On peut également appeler un package se trouvant sur une autre base de donnée. Pour cela on ajoute un arobase suivi du nom de la base de donnée.

Exemple 2 :

```
SQL> EXECUTE  comm_package.reset_comm(15)
```

→ Cet ordre exécute la procédure RESET_COMM située dans le package COMM_PACKAGE.

Exemple 3 :

```
SQL> EXECUTE  scott.comm_package.reset_comm(15)
```

→ Cet ordre exécute la procédure RESET_COMM située dans le package COMM_PACKAGE lui-même situé sur le schéma de SCOTT

Exemple 4 :

```
SQL> EXECUTE  comm_package.reset_comm@ny(15)
```

→ Cet ordre exécute la procédure RESET_COMM située dans le package COMM_PACKAGE qui se trouve sur la base de données NY

5.3.3 Référencer une variable publique à partir d'une procédure autonome

Les variables publiques (globales) sont visibles hors du package, donc il est possible d'y faire référence dans des procédures indépendantes. Pour cela on spécifie son nom ainsi que le nom du package dont elle est issue.

Exemple :

```
SQL> CREATE PROCEDURE meter_to_yard
  2    (v_meter IN NUMBER,
  3     v_yard OUT NUMBER)
  4  IS
  5  BEGIN
  6    v_yard := v_meter * global_vars.meter_2_yard;
  7  END meter_to_yard;
  8 /

SQL> VARIABLE yard NUMBER
SQL> EXECUTE meter_to_yard (1, :yard)
Procédure PL/SQL terminée avec succès.
SQL> PRINT yard
     YARD
---------
   1.0936
```

→ Cet exemple crée une procédure de conversion de mètres en yards en utilisant une variable globale du package GLOBAL_VARS. On appelle ensuite cette fonction pour convertir un mètre.

Les procédures indépendantes peuvent faire appel à tous les types de donnés pouvant être déclarés dans la spécification d'un package (variables, curseurs, constantes ou exceptions). A chaque fois il faut préciser le nom du package en plus du nom de l'élément.

5.3.4 Supprimer des packages

Quand un package n'est plus utilisé on peut le supprimer en utilisant un ordre SQL dans SQL*Plus ou dans Procedure Builder. Dans SQL*Plus on utilise la commande DROP PACKAGE BODY et DROP PACKAGE.

```
DROP PACKAGE  BODY package_name
DROP PACKAGE package_name
```

Dans Procedure Builder on clique sur DROP lorsque la procédure est sélectionnée dans l'explorateur d'objets. Comme un package est composé de deux parties distinctes on peut choisir de supprimer tous le package ou bien de supprimer seulement le corps et de garder la spécification du package. Il est impossible de supprimer uniquement la spécification puisque on ne peut pas avoir de corps de package non lié à une spécification.

5.3.5 Avantages des packages

L'utilisation de packages est une alternative à la création de fonctions et procédures indépendantes et offre de nombreux avantages.
Cela apporte une grande modularité car on encapsule des structures de programmes logiquement reliés dans un module nommé. Chaque package est facile à comprendre et la liaison entre les packages est simple, claire et bien définie.
On obtient également un modèle d'application plus simple. Tout ce dont on a besoin au début ce sont des informations sur l'interface dans la spécification du package. Il est possible de compiler et de coder une spécification de package sans corps. Ensuite les sous-programmes stockés faisant référence au package peuvent être compilé. Il n'est pas obligatoire d'avoir le corps du package entier lorsque l'on développe une application.
Les packages permettent de cacher des informations aux autres utilisateurs. On peut en effet décider si un élément sera public (visible et accessible par tous) ou privé (caché et inaccessible hors du package). Le package cache la définition des éléments privés et donc si la définition du package change, seul celui-ci est affecté, et les changements sont transparents pour l'application. De plus, l'intégrité du package est protégée en cachant les détails d'implémentation aux utilisateurs
Les packages ajoutent des fonctionnalités à la base de données : les variables et curseurs publiques sont conservés en mémoire pendant la durée de la session. Par conséquent ils peuvent être partagés par tous les

sous-programmes s'exécutant dans l'environnement. De plus ils permettent de maintenir les données entre les transactions sans avoir à les stocker dans la base de données.
La performance est améliorée car lorsque que l'on appelle un package pour la première fois, il est entièrement chargé en mémoire ce qui évite les accès disques ultérieurs lorsque que le package sera appelé de nouveau.
Les packages permettent de surcharger les procédures et les fonctions ce qui signifie que l'on peut créer de multiples sous-programmes avec le même nom dans le même package, chacun prenant des paramètres de types ou de nombre différents.

6 COMPLEMENTS SUR LES PACKAGES

6.1 La surcharge

La surcharge permet d'utiliser le même nom pour différents sous-programmes à l'intérieur d'un seul package. Les différents sous-programmes sont distingués grâce au nom, au nombre ou à l'ordre des arguments du sous-programme. Parfois le traitement dans deux sous-programmes est le même, donc dans ce cas il est logique de leur donner le même nom. Le moteur PL/SQL détermine de quelle procédure il s'agit en analysant les paramètres formels. Seuls les sous-programmes locaux et les sous-programmes de packages peuvent être surchargés.
La fonctionnalité de surcharge comporte quelques restrictions, on ne peut par surcharger :

Deux sous-programmes dont les paramètres initiaux diffèrent uniquement par le nom ou le mode de paramètre.

Deux sous-programmes dont les paramètres initiaux diffèrent uniquement par le type de donnée et ces types appartiennent à la même famille (les types nombres et décimal appartiennent à la même famille)

Deux sous-programmes dont les paramètres initiaux diffèrent uniquement par le sous-type et ces sous-types sont basés sur des types de la même famille (VARCHAR et STRING sont des sous-types de VARCHAR2)

Deux fonctions dont seul diffère le type de donnée retournée, même si ces types sont de famille différentes

Lorsque l'on appelle un sous-programme, le compilateur essaye de trouver une déclaration de package correspondant au sous-programme appelé. La recherche s'effectue d'abord dans le champs de vision actuel puis s'étend aux champs de vision imbriqués si nécessaire, elle s'arrête lorsque le compilateur a trouvé une ou plusieurs déclarations de sous-programme dans laquelle le nom correspond à celui de l'appel. Si il trouve plusieurs programmes ayant le même nom à un même niveau de visibilité, le compilateur analyse le nombre, l'ordre et le type de donnée pour trouver la correspondance entre les paramètres initiaux et les paramètres actuels.

Exemple 1 :

```
CREATE OR REPLACE PACKAGE over_pack
IS
  PROCEDURE add_dept
     (v_deptno IN  dept.deptno%TYPE,
      v_name   IN  dept.dname%TYPE   DEFAULT 'unknown',
      v_loc    IN  dept.loc%TYPE     DEFAULT 'unknown');
  PROCEDURE add_dept
     (v_name   IN  dept.dname%TYPE   DEFAULT 'unknown',
      v_loc    IN  dept.loc%TYPE     DEFAULT 'unknown');
END over_pack;
CREATE OR REPLACE PACKAGE BODY over_pack
IS
  PROCEDURE add_dept
     (v_deptno  IN   dept.deptno%TYPE,
      v_name    IN   dept.dname%TYPE   DEFAULT 'unknown',
      v_loc     IN   dept.loc%TYPE     DEFAULT 'unknown')
  IS
  BEGIN
    INSERT INTO dept
    VALUES (v_deptno,v_name,v_loc);
  END add_dept;
  PROCEDURE add_dept
     (v_name    IN   dept.dname%TYPE   DEFAULT 'unknown',
      v_loc     IN   dept.loc%TYPE     DEFAULT 'unknown')
  IS
  BEGIN
    INSERT INTO dept
    VALUES (dept_deptno.NEXTVAL,v_name,v_loc);
  END add_dept;
END over_pack;
```

→ On crée un package et sa spécification. Ce package est constitué de deux procédures portant le même nom mais ayant un nombre d'argument différent.

Si l'on appelle la procédure ADD_DEPT en précisant explicitement un numéro de département, le moteur PL/SQL utilise la première procédure. Si l'on ne spécifie pas de numéro de département, il utilisera la deuxième version.

Exemple 2 :

```
EXECUTE OVER_PACK.ADD_DEPT (76,'MARKETING','ATLANTA')
EXECUTE OVER_PACK.ADD_DEPT ('SUPPORT','ORLANDO')
```

→ La première commande exécutera la première procédure du package tandis que la deuxième lancera la deuxième version.

6.2 Les déclarations anticipées

Le PL/SQL n'autorise pas les références antérieures, c'est-à-dire qu'il faut déclarer un identifiant avant de pouvoir l'utiliser. Par conséquent un sous-programme doit être déclarer avant de pouvoir être appelé.

Exemple 1 :

```
CREATE OR REPLACE PACKAGE BODY forward_pack
IS
   PROCEDURE award_bonus(. . .)
   IS
   BEGIN
    calc_rating(. . .);         --référence illégale

   END;

   PROCEDURE calc_rating(. . .)
   IS
   BEGIN

   END;
   END forward_pack;
```

→ Dans cet exemple la référence est illégale car la procédure CALC_RATING n'as pas encore été déclarée.

Pour corriger un problème de référence illégale il suffit d'inverser l'ordre des deux références a la procédure. Cependant cette technique ne fonctionne pas toujours, par exemple si toutes les procédures s'appellent entre elles ou si l'on veut définir les procédures dans l'ordre alphabétique.
Pour résoudre ce problème il existe en PL/SQL une déclaration de sous-programme spéciale appelée déclaration anticipée. C'est en fait la spécification d'un sous-programme terminée par un point-virgule. On peut utiliser des déclarations en avance pour :

Définir des sous programmes en ordre alphabétique ou logique

Définir des programmes mutuellement récursifs (ce sont des programmes qui s'appellent les uns et les autres directement ou indirectement)

Regrouper des sous-programmes dans un package

Exemple 2 :

```
CREATE OR REPLACE PACKAGE BODY forward_pack
IS

  PROCEDURE calc_rating(. . .);             -- déclaration anticipée

  PROCEDURE award_bonus(. . .)
  IS                                  -- sous-programmes définis
  BEGIN                               -- par ordre alphabétique
    calc_rating(. . .);
  . . .
  END;

  PROCEDURE calc_rating(. . .)
  IS
  BEGIN
  . . .
  END;

END forward_pack;
```

→ La déclaration en anticipée permet d'utiliser la procédure CALC_RATING avant que celle-ci ne soit définie.

Lorsque l'on utilise des déclarations anticipées il faut veiller à ce que la liste de paramètres initiaux apparaisse dans la déclaration anticipée et dans le corps du sous-programme.
Le corps du sous-programme peut être placé n'importe où après la déclaration anticipée mais il doit se trouver dans la même unité de programme.
L'utilisation de déclarations anticipée permet de grouper plusieurs sous-programmes liés dans un package. Les spécifications du sous-programme se trouveront dans la spécification du package et les corps de sous-programmes se trouveront dans le corps du package, d'où ils seront invisibles pour les applications. De cette façon les packages permettent de masquer les détails de l'implémentation.

6.3 Création d'une procédure à usage unique

Une procédure à usage unique est une procédure qui n'est exécutée qu'une seule fois, quand le package est exécuté dans la session de l'utilisateur.

Exemple :

```
CREATE OR REPLACE PACKAGE taxes
IS
    tax NUMBER;
  ...           -- déclarations des fonctions/procédures publiques
END taxes;

CREATE OR REPLACE PACKAGE BODY taxes
IS
  ...           -- déclarations des variables privées
  ...           -- définition des procédures/fonctions privées/publiques
BEGIN
  SELECT   rate_value
  INTO     tax
  FROM     tax_rates
  WHERE    rate_name = 'TAX';
END taxes;
```

→ Dans cet exemple, la valeur de TAX est définie à la première exécution du package TAXES. La valeur prise par TAX correspond alors à la valeur contenue dans la table TAX_RATES

6.4 Restrictions sur les fonctions de package en SQL

Lorsque le serveur Oracle exécute un ordre SQL appelant une fonction stockée il doit savoir si la fonction possède des effets secondaires ou non. Les effets secondaires sont tous les changements effectués sur des tables de la base de données ou sur des variables de packages publics (celles déclarée dans la spécification du package). Les effets secondaires peuvent retarder l'exécution d'une requête en produisant des résultats dépendants de l'ordre (mais cependant indéterminés) ou en requerrant du package qu'il maintienne variables d'état du package au delà de la session de l'utilisateur (ce qui est interdit). Par conséquent les restrictions suivantes s'appliquent aux fonctions appelées dans des ordres SQL :

La fonction ne doit pas modifier les tables de la base de données, donc elle ne peut pas exécuter d'ordre INSERT, UPDATE, ou DELETE

Des fonctions lisant ou écrivant des valeurs de variables de package ne peuvent pas être exécutées à distance ou en parallèle

Seules les fonctions appelées depuis des clauses SELECT, VALUES ou SET peuvent écrire les valeurs des variables de package

La fonction ne peut pas appeler un sous-programme qui enfreint une des précédentes règles. De même une fonction ne peut pas faire référence à une vue qui enfreint une des ces règles.

6.5 Invoquer une fonction d'un package définit par l'utilisateur dans un ordre SQL

En SQL on appelle les fonctions PL/SQL de la même manière que l'on appelle les fonctions SQL prédéfinies : on précise le nom du package duquel la fonction est issue suivit du nom de la fonction et les arguments entre parenthèses si nécessaire.

Exemple :

```
SQL> SELECT  taxes_pack.tax(sal), sal,ename
  2  FROM    emp
  3  /
TAXES_PACK.TAX(SAL)        SAL ENAME
------------------- --------- ----------
                400      5000 KING
                228      2850 BLAKE
                196      2450 CLARK
                238      2975 JONES
                100      1250 MARTIN
                128      1600 ALLEN
                120      1500 TURNER
                 76       950 JAMES
                100      1250 WARD
                240      3000 FORD
                 64       800 SMITH
                240      3000 SCOTT
                 88      1100 ADAMS
                104      1300 MILLER
14 rows selected.
```

→ On appelle la fonction TAX du package TAXES_PACKAGE dans l'ordre SELECT.

6.6 L'état persistant

6.6.1 Les packages de variables

Il est possible de garder une trace de l'état d'une variable ou d'un curseur de package. Cet état est conservé pendant toute la durée de la session de l'utilisateur, du moment où il fait référence pour la première fois à la variable au moment où il se déconnecte.
Le serveur Oracle conserve une trace des valeurs successives des variables de package au cours de la session. A chaque initialisation ou modification de la valeur, il garde une trace de l'ancienne et de la nouvelle valeur. Quand l'utilisateur se déconnecte, la valeur de la variable est libérée.
Exemple :

```
CREATE OR REPLACE PACKAGE comm_package IS
```

```
 g_comm   NUMBER  := 10;    --initialisée à 10
 PROCEDURE reset_comm (v_comm   IN   NUMBER);
END comm_package;
CREATE OR REPLACE PACKAGE BODY comm_package IS
  FUNCTION validate_comm (v_comm  IN  NUMBER) RETURN BOOLEAN
  IS v_max_comm    NUMBER;
  BEGIN
   ... - valide la commission pour qu'elle ne soit pas supérieure
   ... - au salaire maximum actuel
  END validate_comm;
  PROCEDURE reset_comm (v_comm   IN  NUMBER)
  IS BEGIN
   ... - appelle VALIDATE_COMM. Si on essaye de réinitialiser la
   ... - commission à un montant supérieur au maximum,
   ... - une erreur est levée (RAISE_APPLICATION_ERROR)
  END reset_comm;
END comm_package;
```

→ Cet exemple montre les différentes valeurs des variables au cours de l'exécution du package

6.6.2 Les packages de curseurs

Il est possible de créer des packages de curseurs. La création de tel package est utile pour simplifier le FETCH d'un nombre de lignes définis grâce à des procédures publiques.
Exemple 1 :

```
SQL> CREATE OR REPLACE PACKAGE pack_cur
  2  IS
  3     CURSOR c1 IS   SELECT empno FROM emp
  4                    ORDER BY empno DESC;
  5     PROCEDURE proc1_3rows;
  6     PROCEDURE proc4_6rows;
  7  END pack_cur;
  8  /

SQL> CREATE OR REPLACE PACKAGE BODY pack_cur
  2  IS
  3    v_empno NUMBER;
  4    PROCEDURE proc1_3rows IS
  5    BEGIN
  6       OPEN c1;
  7      LOOP
  8         FETCH c1 INTO v_empno;
  9        DBMS_OUTPUT.PUT_LINE('Id :' ||(v_empno));
 10        EXIT WHEN c1%ROWCOUNT >= 3;
 11     END LOOP;
 12   END proc1_3rows;
 13  PROCEDURE proc4_6rows IS
 14  BEGIN
 15     LOOP
 16        FETCH c1 INTO v_empno;
 17       DBMS_OUTPUT.PUT_LINE('Id :' ||(v_empno));
 18       EXIT WHEN c1%ROWCOUNT >= 6;
 19     END LOOP;
 20     CLOSE c1;
 21    END proc4_6rows;
 22  END pack_cur;
 23  /
```

→ On crée un package constitué d'un curseur et de deux procédures. Ces procédures exécutent des FETCH un nombre précis de lignes.

Le curseur et les procédures étant définies en tant qu'éléments publiques il est possible d'appeler chaque procédure indépendamment depuis un ordre SQL.

Exemple 2 :

```
SQL> SET SERVEROUTPUT ON
SQL> EXECUTE pack_cur.proc1_3rows
       Id : 7934
       Id : 7902
       Id : 7900
SQL> EXECUTE pack_cur.proc4_6rows
       Id : 7876
       Id : 7844
       Id : 7839
```

→ On exécute ici les procédures du package précédent indépendamment

6.6.3 Les packages de tables et records PL/SQL

On peut également créer des packages composés de tables et de records PL/SQL.

Exemple :

```
CREATE OR REPLACE PACKAGE emp_package IS
  TYPE emp_table_type IS TABLE OF emp%ROWTYPE
     INDEX BY BINARY_INTEGER;
  PROCEDURE read_emp_table
   (emp_table  OUT  emp_table_type);
END emp_package;

CREATE OR REPLACE PACKAGE BODY emp_package IS
  PROCEDURE read_emp_table
    (emp_table  OUT  emp_table_type)
  IS
     I   BINARY_INTEGER := 0;
  BEGIN
    FOR emp_record IN (SELECT * FROM emp)
    LOOP
      emp_table(i) := emp_record;
      i:= i+1;
    END LOOP;
  END read_emp_table;
END emp_package;
```

→ Ce package crée une table de records. Cette table est remplie dans la procédure READ_EMP_TABLE.

Pour appeler la procédure READ_EMP_TABLE on utilise un bloc PL/SQL anonyme dans lequel on déclare une variable du type définit dans la spécification du package qui sera initialisé parla boucle de la procédure.

Exemple 2 :

```
SQL> DECLARE
  2    emp_table emp_package.emp_table_type;
  3  BEGIN
  4    emp_package.read_emp_table(emp_table);
  5    dbms_output.put_line('An example: '||emp_table(4).ename);
  6  END;
  7  /
```

→ Ce bloc PL/SQL affiche le nom du 4ème champs de la table de record. Cette table a été initialisée grâce à la procédure READ_EMP_TABLE du package EMP_PACKAGE.

7 PACKAGES FOURNIS PAR ORACLE

7.1 Les packages fournit par Oracle

Les packages fournis par Oracle sont inclus dans le serveur Oracle et permettent d'avoir accès à certaines fonction SQL dans un bloc PL/SQL ou d'étendre les fonctionnalités de la base de données.
On peut directement tirer avantage des fonctionnalités fournies par ces packages lorsque l'on crée des applications, ou bien s'en inspirer afin de créer nos propres procédures stockées.

7.2 Le package DBMS_PIPE

7.2.1 La composition de DBMS_PIPE

Le package DBMS_PIPE est constitué de plusieurs procédures et fonctions dont les principales sont ; PACK_MESSAGE, SEND_MESSAGE, RECEIVE_MESSAGE et UNPACK_MESSAGE.

Fonction ou Procédure	Description
PACK_MESSAGE (Procédure)	Compresse un élément dans le buffer de message local qui sera envoyé par la fonction SEND_MESSAGE. L'élément doit être de type VARCHAR2, NUMBER ou DATE
SEND_MESSAGE (Fonction)	Envoie un message contenu dans le buffer de message local vers le pipe désiré
RECEIVE_MESSAGE (Fonction)	Rapatrie un message du pipe choisit et le place dans le buffer de message local afin d'être décompressé par la procédure UNPACK_MESSAGE
UNPACK_MESSAGE (Procédure)	Décompresse un élément du buffer de message local. Cet élément doit être de type VARCHAR2, NUMBER ou DATE.

DBMS_PIPE possède beaucoup d'autres procédures et fonctions que celles énumérées ici. Pour voir toutes les fonctions et procédures du package on peut utiliser les informations du dictionnaire de données concernant DBMS_PIPE. En tant que sys :

```
SELECT  text
FROM    all_source
WHERE   name ='DBMS_PIPE'
ORDER BY LINE;
```

7.2.2 Les fonctions de DBMS_PIPE

Le package DBMS_PIPE possède plusieurs fonctions :

Il permet à deux sessions ou plus, connectées à la même instance, de communiquer par un pipe dont le fonctionnement est semblable aux pipes de UNIX.

Les pipes fonctionnent de manière asynchrone.

En fonction des paramètres de sécurité, les pipes peuvent être privés ou publics

Une fois l'information du buffer lue par un utilisateur, elle est supprimée du buffer et n'est pas disponibles pour les autres lecteurs du même pipe.

Pour envoyer un message il faut d'abord faire un appel ou plus à PACK_MESSAGE pour construire le message. Ensuite on appelle SEND_MESSAGE pour envoyer le message vers la pipe nommée.

Pour recevoir un message d'un pipe, il faut d'abord faire appel à RECEIVE_MESSAGE et ensuite appelé UNPACK_MESSAGE pour accéder aux différents éléments du message.

7.2.3 Exemple d'utilisation de DBMS_PIPE

```
CREATE OR REPLACE PROCEDURE send_message
    (v_message IN VARCHAR2)
IS  s    INTEGER;
BEGIN
  DBMS_PIPE.PACK_MESSAGE(v_message);
  s := DBMS_PIPE.SEND_MESSAGE('DEMO_PIPE');
  IF   s <> 0
  THEN RAISE_APPLICATION_ERROR (-20200,
            'Error '||TO_CHAR(s)||' sending on pipe.');
  END IF;
END send_message;

CREATE OR REPLACE PROCEDURE receive_message
IS  s    INTEGER;
    v_message VARCHAR2(50);
BEGIN
   s := DBMS_PIPE.RECEIVE_MESSAGE('DEMO_PIPE');
   IF   s <> 0
   THEN RAISE_APPLICATION_ERROR (-20201,
          'Error '||TO_CHAR(s)||' reading pipe.');
   END IF;
   DBMS_PIPE.UNPACK_MESSAGE(v_message);
   DBMS_OUTPUT.PUT_LINE(v_message);
END receive_message;
```

→ On définit ici deux procédures : une pour envoyer un message et l'autre pour en recevoir un.

7.3 Le SQL dynamique

7.3.1 Définition

Oracle autorise l'écriture de blocs PL/SQL utilisant du SQL dynamique. Les ordres SQL dynamiques sont des ordres qui ne sont pas encastrés dans le code source du programme, ils sont stockés dans des chaînes de caractères qui peuvent être utilisées en entrée par les programmes ou être construites par ceux-ci. Par exemple on utilise un ordre SQL dynamique pour créer une procédure qui agit sur une table dont le nom n'est pas connu avant l'exécution, ou pour écrire et exécuter un ordre DDL dans du PL/SQL..
Dans Oracle8 et les versions antérieures il faut utiliser le package DBMS_SQL pour écrire des ordres SQL dynamiques. On peut également utiliser ce package dans Oracle8*i* ainsi que EXECUTE IMMEDIATE. Si l'expression est un SELECT portant sur plusieurs lignes, on peut utiliser DBMS_SQL ou les expressions OPEN-FOR, FETCH et CLOSE.
Le package DBMS_SQL permet d'écrire des procédures stockées et des blocs anonymes PL/SQL utilisant du SQL dynamique. Grâce à ce package il est possible d'exécuter des ordres DDL dans un bloc PL/SQL. Par exemple on pourra effectuer un DROP TABLE depuis une procédure stockée.
Les opérations fournies par ce package sont exécutées **sous le compte de l'utilisateur courant** et pas sous celui du propriétaire du package SYS. Donc si l'appelant est un bloc PL/SQL anonyme, les opérations sont exécutées en fonction des privilèges de l'utilisateur courant, si l'appelant est une procédure stockée les opérations sont exécutées suivant les privilèges du propriétaire de la fonction.
L'utilisation de ce package pour exécuter des ordres DDL peut aboutir à une impasse. La raison la plus probable est, par exemple, que l'on essaye de supprimer une procédure que l'on utilise toujours.

7.3.2 Le flux d'exécution

Les ordres SQL sont exécutés suivant un certain ordre : ils sont d'abord parser, puis lier, exécuter et enfin « fetcher ». Toutes ces étapes ne sont pas obligatoires.
Le parsage consiste en un parcours de l'intégralité de l'ordre SQL afin de vérifier la syntaxe de l'expression et de valider les ordres en s'assurant que toutes les références aux objets sont bonnes et que les privilèges appropriés existent pour ces objets.

Une fois que le serveur a parser les lignes il connaît le sens des expressions utilisées mais ne disposent pas d'assez d'informations pour exécuter la requête. Oracle peut avoir besoin de récupérer les valeurs des variables de la requête. La méthode pour les récupérer constitue la phase de liaison.
Ensuite, comme il possède assez d'informations, Oracle peut exécuter la requête.
Pendant la phase de « FETCH » les lignes sont sélectionnées, puis ordonnées (si la requête le demande), et chaque « FETCH » retourne une ligne de résultat différente jusqu'à ce que la dernière ligne ait été « fetchée ».

7.3.3 Le package DBMS_SQL

Le package DBMS_SQL utilise du SQL dynamique pour accéder à la base de données. Il est composés, entre autres, des fonction et procédures suivantes : OPEN_CURSOR, PARSE, BIND_VARIABLE, EXECUTE, FETCH_ROWS, CLOSE_CURSOR.

Fonction ou Procédure	Description
OPEN_CURSOR	Ouvre un nouveau curseur et lui assigne un numéro d'identifiant (ID)
PARSE	Parse les ordres DDL et DML : vérifie la syntaxe des expressions et l'associe au curseur ouvert (les expressions DDL sont exécutées lors du parsage)
BIND_VARIABLE	Lie la valeur donnée à la variable définie par son nom dans l'expression parsée du curseur ouvert.
EXECUTE	Exécute la requête SQL et retourne le nombre de lignes traitées
FETCH_ROWS	Retourne une ligne du curseur ouvert (si il y a plusieurs lignes il faut utiliser une boucle)
CLOSE_CURSOR	Ferme le curseur spécifié

Le package DBMS_SQL possèdent de nombreuses autres fonctions et procédures que celles listées ici, pour connaître toutes les fonctions et procédures du package on utilise les informations contenues dans le dictionnaire de données :

```
SELECT    text
FROM      all_source
WHERE     name ='DBMS_SQL'
ORDER BY  LINE;
```

7.3.4 Utilisation de DMBS_SQL

Pour traiter un ordre DML dynamique il faut tout d'abord utiliser OPEN_CURSOR pour établir un endroit en mémoire pour traiter l'ordre SQL. On utilise ensuite PARSE pour parcourir l'ordre et vérifier sa validité. La fonction EXECUTE exécute l'ordre SQL et retourne le nombre de lignes traitées. Enfin on ferme le curseur grâce à CLOSE_CURSOR

Exemple :

```
CREATE OR REPLACE PROCEDURE delete_all_rows
   (tab_name IN VARCHAR2,
    rows_del OUT NUMBER)
IS
  cursor_name INTEGER;
BEGIN
  cursor_name := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE ( cursor_name,'delete FROM '||tab_name,
      DBMS_SQL.NATIVE );
  rows_del := DBMS_SQL.EXECUTE (cursor_name);
  DBMS_SQL.CLOSE_CURSOR(cursor_name);
END;

SQL> VARIABLE deleted NUMBER
SQL> EXECUTE  delete_all_rows('emp',:deleted)
Procédure PL/SQL terminée avec succès.
SQL> PRINT deleted
DELETED
---------
14
```

→ Cette procédure supprime toutes les lignes de la table passée en paramètre et retourne le nombre de lignes effacées.

7.3.5 EXECUTE IMMEDIATE

Pour écrire des ordres SQL dynamiques avec Oracle8*i* on peut également utiliser l'expression EXECUTE IMMEDIATE. Cette expression parse et exécute immédiatement l'ordre SQL.

```
EXECUTE IMMEDIATE dynamic_string
  [INTO { define_variable
       [, define_variable] ... | record}]
[USING [IN|OUT|IN OUT] bind_argument
    [, [IN|OUT|IN OUT] bind_argument] ... ];
```

Paramètre	Description
Dynamic_string	C'est une chaîne de caractère représentant un ordre SQL ou un bloc PL/SQL
Define_variable	C'est une variable stockant la valeur retournée par le SELECT
Record	C'est un record définit par l'utilisateur ou par %ROWTYPE stockant le résultat du SELECT
Bind_argument	C'est une expression dont la valeur est passée à l'expression SQL ou au bloc PL/SQL dynamiques

La clause INTO ne peut être utilisée que pour une requête portant sur une seule ligne. Si la requête porte sur plusieurs lignes il faut utiliser OPEN-FOR, FETCH et CLOSE.

Commentaire : Merci du rappel des fois qu'on aurait oublié.

7.3.6

7.3.7 Utilisation de EXECUTE IMMEDIATE

```
CREATE PROCEDURE del_rows
   (v_table_name IN VARCHAR2,
    rows_deld OUT NUMBER)
IS
BEGIN
  EXECUTE IMMEDIATE 'delete from '||v_table_name;
  rows_deld := SQL%ROWCOUNT;
END;

SQL> VARIABLE deleted NUMBER
SQL> EXECUTE del_rows('emp',:deleted)
Procédure PL/SQL terminée avec succès.
SQL> PRINT deleted
DELETED
---------
14
```

→ On crée ici la même procédure que précedemment en utilisant EXECUTE IMMEDIATE à la place du package DBMS_SQL

7.4 Les autres packages disponibles

7.4.1 Le package DBMS_DDL

Le package DBMS_DDL donne accès à certains ordres DDL propres au SQL utilisable en PL/SQL. Ce package ne peut pas être utiliser dans des triggers, des procédures appelées depuis Form Builder ou dans des sessions distantes.
DBMS_DDL apporte la possibilité de recompiler les procédures, fonctions et packages modifiées en utilisant la procédure ALTER_COMPILE.

```
DBMS_DDL.ALTER_COMPILE(type_objet,propriétaire,nom_objet)
```

Il est également possible d'analyser un objet unique en utilisant la procédure ANALYZE_OBJECT. On peut également analyser plusieurs objets en même temps en utilisant DBMS_UTILITY.

```
DBMS_DDL.ANALYZE_OBJECT(_objet,propriétaire,nom_objet)
```

Grâce à ce package, les développeurs ont accès aux ordres ALTER et ANALYZE, qui sont des ordres SQL, dans un environnement PL/SQL.

7.4.2 Le package DBMS_JOB

Ce package permet de programmer des tâches, exécuter des procédures et modifier les jobs déjà programmés pour par exemple les forcer à s'exécuter immédiatement au lieu du moment prévu. On peut voir tous les jobs déjà définis grâce à la vue USER_JOBS du dictionnaire de données.
Pour ajouter un job en queue on utilise la procédure SUBMIT :

```
DBMS_JOB.SUBMIT(jobno,ordre,date)
```

JOBNO est un paramètre OUT, il contiendra le numéro donné au job crée pour que l'utilisateur puisse y faire référence ultérieurement.
ORDRE représente l'action a exécuter.
DATE correspond à la date à laquelle le job sera exécuté. Par défaut cette valeur est SYSDATE

Exemple :

```
SQL> VARIABLE jobno NUMBER
SQL> EXECUTE DBMS_JOB.SUBMIT (:jobno,                      -
>         'OVER_PACK.ADD_DEPT(''EDUCATION'',''ZURICH'');', -
>          SYSDATE)
SQL> COMMIT;
SQL> PRINT jobno
```

→ Cette requête ajoute un job qui s'exécute immédiatement (SYSDATE).

Pour forcer l'exécution d'un JOB qui se trouve en queue on utilise RUN en indiquant le numéro du JOB à exécuter.

```
DBMS_JOB.RUN (jobno)
```

7.4.3 Le package DBMS_OUTPUT.

Le package DBMS_OUTPUT permet de faire sortir des messages et des valeurs de blocs PL/SQL. Il est constitué entre autres des procédures suivantes : PUT, NEW_LINE, PUT_LINE, GET_LINE, GET_LINES et ENABLE/DISABLE.

Fonction ou Procédure	Description
PUT	Ajoute le texte de la procédure à la ligne courante du buffer de sortie
NEW_LINE	Place un marqueur End_Of_Line dans le buffer de sortie
PUT_LINE	Combine l'action de PUT et de NEW_LINE
GET_LINE	Récupère la ligne courante du buffer de sortie dans la procédure
GET_LINES	Récupère une série de lignes du buffer de sortie dans la procédure
ENABLE/DISABLE	Active ou désactive les appels à la procédure DBMS_OUTPUT

Ce package permet aux développeurs de suivre précisément le déroulement d'une fonction ou d'une procédure en envoyant des messages et valeurs au buffer de sortie. C'est une aide précieuse pour le débuggage car il permet de suivre les résultats intermédiaires lors de l'exécution.
Si on utilise cet package sous SQL*Plus, il faut s'assurer que l'affichage sur le terminal de sortie (SERVEROUTPUT) est bien définit à ON.

7.4.4 D'autres packages fournit par Oracle

Voici quelques autres packages Oracle dont le fonctionnement ne sera pas détaillé dans ce cours :

Package	Description
DBMS_ALERT	Fournit des notifications sur les événements de la base de données
DBMS_APPLICATION_INFO	Permet aux outils d'application et aux développeurs d'applications d'informer la base de données du niveau des actions qu'elle est en train d'exécuter
DBMS_DESCRIBE	Renvoie une description des arguments d'une procédure stockée
DBMS_LOCK	Demande, convertit et libères les USERLOCKS, qui sont gérés par les services de gestion des verrous RDBMS
DBMS_SESSION	Donne accès aux information de la session SQL actuelle
DBMS_SHARED_POOL	Garde les objets en mémoire partagée
DBMS_TRANSACTION	Contrôle les transaction logique et améliore les performances des transactions courtes et non distribuées
DBMS_UTILITY	Analyse les objets d'un schéma particulier, vérifie si le serveur tourne en mode parallèle ou non et renvoie l'heure
UTL_FILE	Ajoute des fonctionnalités d'écriture et de lecture de fichier

8 CREATION DE TRIGGERS DE BASE DE DONNEES

8.1 Les triggers de base de données

8.1.1 Aperçu des triggers

Les triggers sont des blocs PL/SQL s'exécutant implicitement chaque fois qu'un événement particulier a lieu.
Les triggers peuvent être mis en place soit sur une base de données soit sur une application.
Les triggers de base de données sont exécutés implicitement lorsque qu'un ordre INSERT, UPDATE ou DELETE (ordres déclenchants) est exécuté sur la table associée au trigger. Les triggers sont exécutés quelque soit l'utilisateur connecté ou l'application utilisée. Les triggers de base de données sont également exécutés implicitement pour les actions de l'utilisateur ou les actions systèmes de la base de données. Par exemple lorsque l'utilisateur se logue ou lorsque qu'un DBA arrête la base de données.
Les triggers d'application sont exécutés implicitement quand un événement particulier se produit dans une application. Un exemple d'applications utilisant des triggers : les applications développées avec *Form Builder*.
Les triggers de base de données peuvent être définis sur des tables ou sur des vues. Si une opération DML est effectuée sur une vue, le trigger INSTEAD OF définit les actions qui auront lieu. Si ces actions incluent des opérations DML sur des tables, tous les triggers sur la ou les tables de base sont déclenchés.

8.1.2 Directives de conception des triggers

L'utilisation de trigger garantit que lorsque qu'une opération spécifique est exécutée, les actions en rapport sont exécutées implicitement et de manière transparente pour l'utilisateur.
On utilise les triggers de base de données pour des opérations globales et centralisées qui doivent être déclenchées par des ordres indépendamment de l'utilisateur ou de l'application ayant émis l'ordre.
Il est préférable de ne pas définir de trigger pour reproduire ou remplacer les fonctionnalités déjà présentes dans la base de données Oracle. Par exemple il ne faut pas définir de trigger implémentant des règles d'intégrité qui peuvent être mises en place en utilisant des contraintes.
L'utilisation excessive de triggers peut aboutir à des interdépendances complexes, ce qui peut rendre difficile la maintenance des grosses applications. Il ne faut utiliser les triggers que si nécessaire et faire attention aux effets récursifs et en cascades qu'ils peuvent produire.
Un triggers récursif est un trigger contenant une opération DML modifiant la même table.
Un trigger en cascade est un trigger dont l'action entraîne le déclenchement d'un second trigger et ainsi de suite. Le serveur Oracle autorise l'exécution en cascade de 32 triggers maximum en même temps, mais on peut limiter le nombre de triggers en cascade en changeant la valeur du paramètre d'initialisation de la base de données OPEN_CURSORS, qui par défaut est initialisé à 50.

8.1.3 Exemple de trigger de base de données

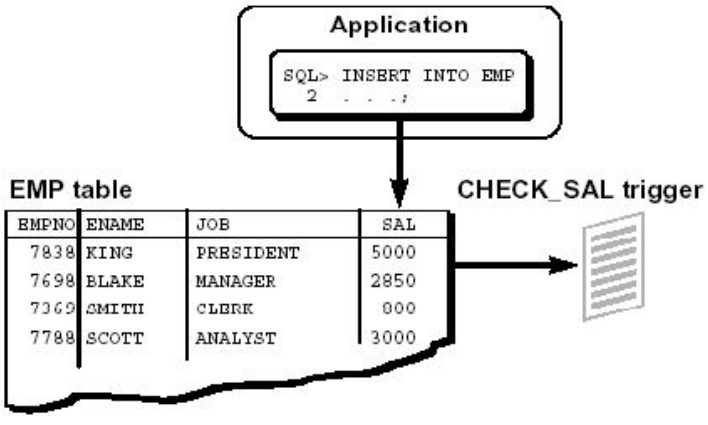

Figure 1 : Exemple de trigger de base de données.

Le schéma montre un trigger de base de données qui vérifie que le salaire inséré se situe bien dans la tranche définie. Les valeurs ne respectant pas le rang de salaire en fonction du type d'emploi peuvent être soit rejetées, soit être insérées et signalées dans une table d'audit.

8.1.4 Création de triggers

Avant de coder le corps du trigger il faut décider des paramètres de ses composantes : la synchronisation du trigger, l'évènement déclenchant et le type de trigger.

Composante	Description	Valeurs Possibles
Synchronisation du trigger	Définit à quel moment le trigger se déclenche par rapport à l'effet déclenchant	BEFORE AFTER INSTEAD OF
Evènement déclenchant	Définit quelle opération de manipulation de données sur la table ou la vue entraîne le déclenchement du trigger	INSERT UPDATE DELETE
Type de trigger	Définit combien de fois le corps du trigger est exécuté	Statement Row
Corps du trigger	Définit les actions qui seront exécutées par le trigger	Bloc PL/SQL complet

8.2 Les composantes d'un trigger

8.2.1 La synchronisation du trigger

La synchronisation du trigger définit le moment où le trigger sera déclenché. Elle peut prendre trois valeurs : BEFORE, AFTER, INSTEAD OF.
Un trigger BEFORE exécute le corps du trigger avant l'évènement DML déclenchant sur la table. Les triggers BEFORE sont utilisés quand l'action du trigger détermine si l'ordre déclenchant est autorisé à s'exécuter ou pas. Cette situation permet d'éliminer les exécutions inutiles de l'expression déclenchante et les éventuels ROLLBACK pour les cas où une expression est levée lors de l'action déclenchante. On l'utilise également pour modifier les valeurs des colonnes avant d'effectuer un ordre déclenchant de type INSERT ou UPDATE.
Un trigger AFTER exécute le corps du trigger après que l'évènement DML déclenchant se soit produit sur la table. Les triggers AFTER sont utilisés quand on veut que l'action déclenchante soit terminée avant d'exécuter l'action du trigger. Il est également utile lorsque l'on veut exécuter plusieurs actions sur un ordre déclenchant possédant déjà un trigger BEFORE.
Un trigger INSTEAD OF exécute le corps du trigger au lieu de l'ordre déclenchant. Ce trigger fournit un moyen de modifier des vues qui ne sont pas directement modifiables par des ordres DML parce que les vues ne sont pas modifiables par nature. Ainsi il est possible d'écrire des ordres INSERT, UPDATE et DELETE sur la vue et le trigger INSTEAD OF va agir en transparence pour exécuter l'action codée directement sur les tables sous-jacentes de la vue.

8.2.2 L'évènement déclenchant

Les évènements déclenchants les triggers peuvent être les ordres DML INSERT, UPDATE et DELETE.
Lorsque l'évènement déclenchant est un ordre UPDATE, il est possible de définir une liste de colonnes pour identifier celle(s) qui devra être modifiée pour déclencher le trigger. Il est **impossible** de spécifier une liste de colonnes pour les ordres INSERT et DELETE car ils portent sur toute une ligne.

```
. . .  UPDATE OF sal  . . .
```

L'évènement déclenchant peut être composé de plusieurs opérations DML :

```
. . .  INSERT or UPDATE or DELETE  . . .
. . .  INSERT or UPDATE OF job  . . .
```

8.2.3 Le type de trigger

Le type de trigger représente le nombre de fois que le corps du trigger sera exécuté lorsque l'évènement déclenchant se produit. Il peut prendre deux valeurs : Statement et Row.
Un trigger Statement ne s'exécute qu'une fois pour l'évènement déclenchant même si aucune ligne n'est concernée. Cette valeur est utilisée pour la valeur par défaut. Les triggers Statement sont utiles si l'action du trigger ne dépend pas des données des lignes concernées ou des données fournies par l'évènement déclenchant lui-même. Par exemple, un trigger qui exécute une vérification de sécurité complexe sur l'utilisateur courant.
Un trigger Row est un trigger dont le corps s'exécute une fois pour chaque ligne concernée par l'évènement déclenchant. Si l'évènement déclenchant n'affecte aucune ligne, le corps du trigger n'est pas exécuté. Les triggers Row sont utiles lorsque l'action du trigger dépend des données des lignes affectées ou des données fournies directement par l'évènement déclenchant.

8.2.4 Le corps du trigger

Le corps du trigger définit toutes les actions qui seront exécutées lors du déclenchement. Le corps est constitué d'un bloc PL/SQL ou d'un appel à une procédure. Le bloc PL/SQL peut être contenir des ordres SQL et PL/SQL, définir des éléments PL/SQL comme des variables, curseurs, exceptions et autres.

8.3 La séquence de déclenchement

La séquence de déclenchement correspond à l'ordre dans lequel sont exécutés la requête et le trigger. Cette séquence dépend de l'option du type de déclenchement choisit (BEFORE, AFTER, INSTEAD OF) et du type de trigger (Row ou Statement).
Lorsque le trigger est de type Statement, le trigger n'est exécuté qu'une seule fois. Si le déclenchement est de type BEFORE, le trigger s'exécutera d'abord pour ensuite laisser la requête DML s'effectuer. S'il s'agit d'un trigger AFTER la requête s'exécute et ensuite le trigger prend la main pour effectuer ses actions. Si le trigger est de type INSTEAD OF le corps du trigger est exécuté à la place de la requête DML.
Pour un trigger de type Row, le trigger s'exécute une fois par ligne traitée. Pour un trigger BEFORE, le corps du trigger s'exécute en premier puis la requête s'exécute sur la première ligne retournée, ensuite le trigger s'exécute une nouvelle fois, puis la requête s'exécute sur la ligne suivante et ainsi de suite jusqu'à ce qu'il n'y ait plus de ligne retournée. Pour un trigger AFTER la démarche est la même, sauf que le corps du trigger s'exécute après chaque ligne traitée. Lorsque la requête ne retourne qu'une ligne, la séquence de déclenchement est identique à celle d'un Statement trigger.

8.4 Création de Statement trigger

8.4.1 Syntaxe de création de Statement triggers

La syntaxe permettant de créer un Statement trigger est la suivante :

```
CREATE [OR REPLACE] TRIGGER trigger_name
   timing
      event1 [OR event2 OR event3]
         ON table_name
trigger_body
```

Trigger_name	Indique le nom du trigger qui sera créé
Timing	Indique l'instant auquel sera déclenché le trigger en relation avec l'événement déclenchant : BEFORE

	AFTER INSTEAD OF
Event	Identifie l'opération de manipulation de données qui déclenche le trigger : INSERT UPDATE [OF *column*] DELETE
Table/view_name	Indique le table ou vue qui sera associée au trigger
Trigger_body	Le corps du trigger définissant les actions qui seront exécutées par celui-ci. Le corps commence par DECLARE ou BEGIN et se termine par END ou est un appel à une procédure.

8.4.2 Création avec SQL*Plus

Un trigger BEFORE permet d'empêcher l'exécution de certaines requêtes si une condition n'est pas respectée. On va ainsi créer un trigger pour restreindre les ordres INSERT sur la table EMP aux heures ouvrées entre Lundi et Vendredi.

Exemple :

```
SQL> CREATE OR REPLACE TRIGGER secure_emp
  2  BEFORE INSERT ON emp
  3  BEGIN
  4   IF (TO_CHAR (sysdate,'DY') IN ('SAM','DIM')) OR
  5      (TO_CHAR(sysdate,'HH24') NOT BETWEEN
  6                              '08' AND '18')
  7   THEN RAISE_APPLICATION_ERROR (-20500,
  8   'Insertion dans EMP impossible hors des heures de travail.');
  9   END IF;
 10  END;
 11  /

Déclencheur créé.
```

→ Si un utilisateur essaye d'insérer une ligne dans la table EMP hors des heures ouvrées (samedi par exemple) il verra le message d'erreur, le trigger échoue et la requête déclenchante est annulée (ROLLBACK).

RAISE_APPLICATION_ERROR est une procédure serveur intégrée qui affiche un message sur le terminal et cause l'échec du trigger. Quand un trigger de base de données échoue, l'événement déclenchant est automatiquement annulée (ROOLBACK) par le serveur Oracle.

8.4.3 Création avec Procedure Builder

Pour créer un trigger de base de données en utilisant Procedure Builder il faut d'abord ce connecter à la base de données. Dans l'explorateur d'objets on clique sur le nœud *Database Objects* et on sélectionne ensuite l'éditeur de trigger de base de données dans le menu *Program* afin de faire apparaître la fenêtre suivante :

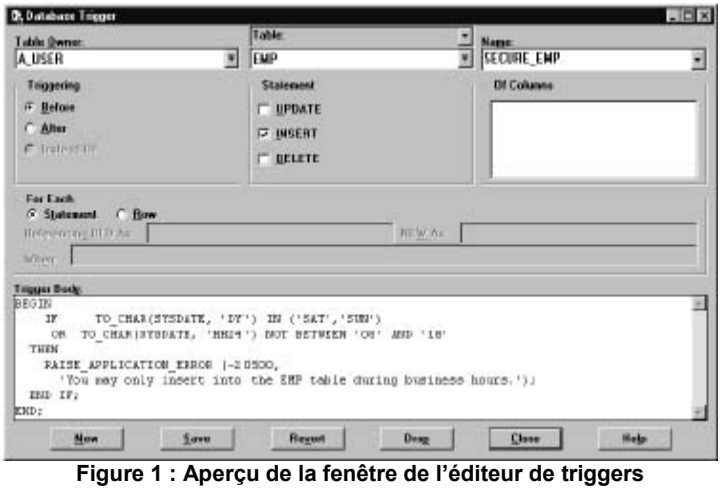

Figure 1 : Aperçu de la fenêtre de l'éditeur de triggers

Dans la partie supérieure de la fenêtre on sélectionne le propriétaire de la table affectée, le nom de cette table grâce au menu déroulant. On choisit ensuite le moment du déclenchement et le type d'ordre déclenchant en cochant les cases adéquates. On donne un nom à notre trigger ainsi que son type (Statement dans notre exemple). Dans la partie inférieure de la fenêtre, on entre le code du corps du trigger puis on clique sur *Save* pour le compiler grâce au moteur PL/SQL du serveur. Une fois la compilation effectuée correctement le code du trigger est stocké dans la base de donnée et activé automatiquement. Si la compilation retourne des erreurs, celles-ci sont affichées dans une fenêtre séparée.
Il est impossible de créer des trigger INSTEAD OF dans les versions plus anciennes de Procedure Builder.

8.4.4 Test de SECURE_EMP

Pour tester le trigger SECURE_EMP, on va essayer d'insérer une ligne dans la table EMP hors des heures de travail.

Exemple :

```
SQL>  INSERT INTO emp (empno, ename, deptno)
  2   VALUES          (7777, 'BAUWENS', 40);
 INSERT INTO emp (empno, ename, deptno)
             *
ERREUR à la ligne 1 :
ORA-20500: Insertion dans EMP impossible hors des heures de travail.
ORA-06512: à "SCOTT.SECURE_EMP", ligne 5
ORA-04088: erreur lors d'exécution du déclencheur 'SCOTT.SECURE_EMP'
```

→ Cet ordre DML INSERT produit une erreur due au trigger lorsque l'on essaye d'insérer des lignes hors des heures habituelles de travail.

8.4.5 Utilisation d'attributs conditionnels

Il est possible de combiner plusieurs évènements déclenchant une seule déclaration grâce aux attributs conditionnels INSERTING, UPDATING et DELETING à l'intérieur du corps du trigger. Ces attributs sont inclus dans des expressions conditionnelles de type IF-THEN-ELSE.
On peut par exemple créer un trigger qui restreindra toutes les opérations de manipulation de données sur la table EMP aux heures de travail entre Lundi et Vendredi.

Exemple :

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT OR UPDATE OR DELETE ON emp
BEGIN
 IF (TO_CHAR (sysdate,'DY') IN ('SAM','DIM')) OR
    (TO_CHAR (sysdate, 'HH24') NOT BETWEEN '08' AND '18')
   THEN
     IF    DELETING
       THEN RAISE_APPLICATION_ERROR (-20502,
         'Suppression de EMP impossible hors des heures de travail.');
     ELSIF    INSERTING
       THEN RAISE_APPLICATION_ERROR (-20500,
         'Insertion dans EMP impossible hors des heures de travail.');
     ELSIF    UPDATING ('SAL')
       THEN RAISE_APPLICATION_ERROR (-20503,
         'Mise à jour de SAL impossible hors des heures de travail.');
     ELSE
         RAISE_APPLICATION_ERROR (-20504,
         'Mise à jour de EMP impossible hors des heures de travail.');
     END IF;
 END IF;
END;
```

→ Ce trigger restreint les ordres DML sur la table EMP en utilisant des attributs conditionnels.

8.5 Création de Row Triggers

8.5.1 Syntaxe de création d'un Row Trigger

La syntaxe permettant de créer un Row Trigger est la suivante :

```
CREATE [OR REPLACE] TRIGGER trigger_name
  timing
    event1 [OR event2 OR event3]
      ON table_name
  [REFERENCING OLD AS old | NEW AS new]
FOR EACH ROW
  [WHEN condition]
trigger_body
```

Trigger_name	Indique le nom du trigger qui sera créé
Timing	Indique l'instant auquel sera déclenché le trigger en relation avec l'événement déclenchant : BEFORE AFTER INSTEAD OF
Event	Identifie l'opération de manipulation de données qui déclenche le trigger : INSERT UPDATE [OF *column*] DELETE
Table_name	Indique le table ou vue qui sera associée au trigger
REFERENCING	Spécifie les noms des liens réciproques pour l'ancienne et la nouvelle valeur de la ligne en cours (les valeurs par défauts sont OLD et NEW)
FOR EACH ROW	Indique que le trigger est un Row trigger
WHEN	Spécifie la restriction du trigger (l'attribut conditionnel est évalué pour chaque ligne afin de déterminer si le corps du trigger est exécuté ou non)
Trigger_body	Le corps du trigger définissant les actions qui seront exécutées par celui-ci. Le corps commence par DECLARE ou BEGIN et se termine par END ou est un appel à une procédure.

8.5.2 Création avec SQL*Plus

On peut créer un Row trigger BEFORE afin d'empêcher l'opération déclenchante de s'exécuter sur chaque ligne concernée lorsqu'une certaine condition n'est pas respecter. On peut par exemple créer un Row trigger qui ne va autoriser que certains employés à gagner un salaire supérieur à 5000.

Exemple ;

```
SQL> CREATE OR REPLACE TRIGGER DERIVE_COMMISSION_PCT
  2    BEFORE INSERT OR UPDATE OF sal   ON emp
  3  FOR EACH ROW
  4  BEGIN
  5    IF NOT (:NEW.JOB IN ('MANAGER' , 'PRESIDENT'))
  6       AND :NEW.SAL > 5000
  7    THEN
  8       RAISE_APPLICATION_ERROR
  9         (-20202, 'L'employé ne peut pas gagner autant.');
 10  END IF;
 11  END;
```

→ Ce trigger empêche les utilisateurs d'insérer un salaire supérieur à 5000 pour tous les JOB sauf les MANAGER et PRESIDENT.

Si l'utilisateur essaye d'insérer un salaire supérieur à 5000 le trigger renvoie une erreur :

Exemple :

```
SQL> UPDATE EMP SET SAL = 6500 WHERE ENAME = 'MILLER'
2 /
UPDATE EMP SET SAL = 6500 WHERE ENAME = 'MILLER'
*
ERROR at line 1:
ORA-20202: L'employé ne peut pas gagner autant.
ORA-06512: à "SCOTT.DERIVE_COMMISSION_PCT", ligne 5
ORA-04088: erreur lors d'exécution du déclencheur
'SCOTT.DERIVE_COMMISSION_PCT'
```

→ Le trigger produit une erreur et ainsi la table EMP n'est pas modifiée.

8.5.3 Création avec Procedure Builder

On peut également créer un Row trigger BEFORE en utilisant Procedure Builder. Pour cela on se connecte à la base de données et on clique sur le nœud *Database Objects* dans l'explorateur d'objets. Puis on sélectionne l'éditeur de trigger de base de données dans le menu Program (cf § 8.4.3, Figure 1). On sélectionne ensuite le propriétaire et la table correspondante dans les listes déroulantes. On choisit le moment du déclenchement (BEFORE) et les événements déclenchant en cochant les cases appropriées. Dans la partie *For Each* on sélectionne Row pour indiquer à Procedure Builder que l'on développe un Row Trigger. On peut remplir la section *Referencing* afin de modifier les liens réciproques en spécifiant un nom différent pour le OLD et le NEW. On peut également indiquer un attribut conditionnel en remplissant le champ *When*. Ces deux options sont disponibles uniquement lorsque l'on crée un Row Trigger. Dans la partie inférieure de la fenêtre on entre le code du corps du trigger puis on clique sur *Save* afin de le faire compiler par le moteur PL/SQL du serveur. Une fois correctement compilé, le code est stocké dans la base de données et il est activé.

8.5.4 Utilisation des qualificatifs OLD et NEW

Lorsque l'on utilise un Row Trigger, celui-ci enregistre les valeurs de plusieurs colonnes avant et après la modification des données en utilisant les qualificatifs OLD et NEW associés au nom de colonne respectif
Pour l'exemple on va utiliser une table nommée AUDIT_EMP_TABLE. Cette table est constituée des colonnes user_name, timestamp, id, old_last_name, new_last_name, old_title, new_title, old_salary,_ new_salary). Cette table n'as pour but que de donner un exemple plus concret de l'utilisation des qualificatifs, elle est donc vide (NULL).

Exemple :

```
SQL> CREATE OR REPLACE TRIGGER audit_emp_values
  2  AFTER DELETE OR INSERT OR UPDATE ON emp
  3  FOR EACH ROW
  4  BEGIN
  5    INSERT INTO audit_emp_table (user_name,
  6     timestamp, id, old_last_name, new_last_name,
  7     old_title, new_title, old_salary, new_salary)
  8    VALUES (USER, SYSDATE, :OLD.empno, :OLD.ename,
  9       :NEW.ename, :OLD.job, :NEW.job,
 10       :OLD.sal, :NEW.sal );
 11  END;
 12  /
```

→ Cette requête crée un trigger qui, pour tout ordre DML sur la table EMP insérera une ligne d'audit en enregistrant l'ancienne et la nouvelle valeur du nom, du titre et du salaire.

8.5.5 Valeurs de OLD et NEW

Dans un Row Trigger il est possible de faire référence aux valeurs d'une colonne avant et après le changement dû à un ordre DML en préfixant le nom de celle-ci du qualificatif OLD ou NEW.
Selon l'opération DML effectuée les valeurs OLD et NEW ne seront pas les mêmes :

Opération	Valeur de OLD	Valeur de NEW
INSERT	NULL	Valeur Insérée
UPDATE	Valeur avant la mise à jour	Valeur après la mise à jour
DELETE	Valeur avant la suppression	NULL

Les qualificatifs OLD et NEW ne sont disponibles que pour les triggers de type Row.
Il faut préfixer ces qualificatifs de deux points (:) dans tout ordre SQL ou PL/SQL les utilisant.
Il ne faut **pas** utiliser de préfixe deux points (:) quand les qualificatifs **sont utilisés dans la condition** de restriction **WHEN**.

8.5.6 Restreindre un trigger de ligne

Grâce à la condition WHEN il est possible de restreindre l'action du trigger aux lignes conformes à certaines conditions.

Exemple :

```
SQL> CREATE OR REPLACE TRIGGER derive_commission_pct
  2  BEFORE INSERT OR UPDATE OF sal ON emp
  3  FOR EACH ROW
  4  WHEN (NEW.job = 'SALESMAN')
  5  BEGIN
  6    IF INSERTING
  7     THEN :NEW.comm := 0;
  8    ELSIF :OLD.comm IS NULL
  9     THEN :NEW.comm := 0;
 10    ELSE :NEW.comm := :OLD.comm * (:NEW.sal/:OLD.sal);
 11    END IF;
 12  END;
 13  /
```

→ Ce trigger est utilisé pour calculer la nouvelle commission quand une ligne est ajoutée ou modifiée dans la table EMP.

Si l'on veut assigner des valeurs aux colonnes en utilisant le qualificatif NEW il faut créer un Row trigger BEFORE. Si l'on essaye de compiler le code de l'exemple en utilisant un trigger AFTER il se produira une erreur

8.6 Trigger INSTEAD OF

8.6.1 Intérêt des triggers INSTEAD OF

On utilise les triggers INSTEAD OF pour modifier des données sur lesquelles un ordre DML a été effectué alors qu'elles font partie d'une vue non modifiable. Ces triggers exécutent les opérations INSERT, UPDATE et DELETE directement sur les tables sous-jacentes à la vue. Lorsque l'on écrit un ordre INSERT, UPDATE ou DELETE sur une vue, le trigger INSTEAD OF agit en transparence en tâche de fond pour exécuter les bonnes actions en remplacement.
On appelle ces triggers INSTEAD OF car ; contrairement aux autres triggers, le serveur Oracle déclenche le trigger au lieu d'exécuter l'expression déclenchante. Le type INSTEAD OF est un Row trigger.
Si une vue est constituée de plus d'une table, une insertion sur celle-ci peut entraîner une insertion dans une table et une mise à jour dans une autre. Donc on doit écrire un trigger INSTEAD OF qui se déclenche lors de l'exécution d'un ordre INSERT. A la place de l'ordre INSERT original, le corps du trigger est exécuté ce qui résulte en une insertion dans une table et une mise à jour dans l'autre.
Lorsqu'une vue est modifiable et qu'elle possède un trigger INSTEAD OF, le trigger aura la priorité d'exécution et donc exécutera le corps du trigger pour chaque ordre DML.
Les triggers INSTEAD OF ne sont disponibles que sur l'édition Entreprise de Oracle.

8.6.2 Création d'un trigger INSTEAD OF

La syntaxe pour la création de trigger INSTEAD OF est la suivante :

```
CREATE [OR REPLACE] TRIGGER trigger_name
  INSTEAD OF
    event1 [OR event2 OR event3]
      ON view_name
    [REFERENCING OLD AS old | NEW AS new]
[FOR EACH ROW]
trigger_body
```

Trigger_name Indique le nom du trigger

INSTEAD OF	Indique que le trigger portera sur une vue
Event	Précise le type d'opération de manipulation de données qui déclenchera le trigger. Cette valeur peut être : INSERT UPDATE [OF colonne] DELETE
View_name	Indique la vue associée au trigger
REFERENCING	Spécifie le nom des liens de corrélation pour les nouvelles et anciennes valeurs. (Par défaut ces valeurs sont OLD et NEW)
[FOR EACH ROW]	Désigne le trigger en tant que Row trigger. Les triggers INSTEAD OF ne peuvent être que des Row triggers donc cette clause est optionnelle.
Trigger_body	Le corps du trigger définit les actions qui seront exécutées par le trigger. Ce bloc peut être un bloc PL/SQL avec une clause DECLARE ou BEGIN et le mot clés END ou un appel à une procédure

Les triggers INSTEAD OF ne peuvent être écrit que pour des **vues**. Les options BEFORE et AFTER ne sont pas disponibles et on ne peut pas spécifier de clause WHEN avec un trigger INSTEAD OF.

8.7 Différence entre les triggers et les procédures stockées

Il existe plusieurs différences et points communs entre les triggers de base de données et les procédures stockées :

Triggers	**Procédures**
Utilise CREATE TRIGGER	Utilise CREATE PROCEDURE
Le dictionnaire de données contient la source et le *p-code*	Le dictionnaire de donnée contient la source et le *p-code*
Appelé implicitement	Appelé explicitement
COMMIT, SAVEPOINT et ROLLBACK ne sont pas autorisé	COMMIT, SAVEPOINT et ROLLBACK peuvent être utilisés

Les triggers sont compilés entièrement quand la commande CREATE TRIGGER est exécutée et la *p-code* est stocké dans le dictionnaire de données. En conséquence de quoi le déclenchement du trigger ne nécessite plus l'ouverture d'un curseur partagé pour effectuer l'action du trigger. A la place, le trigger est exécuté directement.
Si des erreurs se produisent durant la compilation d'un trigger, il sera quand même créé. Il est donc préférable d'utiliser la syntaxe OR REPLACE pour éviter d'avoir à supprimer l'ancien trigger quand celui-ci contient des erreurs.

8.8 Gestion des triggers

8.8.1 Activation des triggers

Lorsque l'on a créé un trigger, il existe des commandes permettant de les manipuler afin de les activer, les désactiver et les recompiler.
Lorsqu'un trigger est créé, il est automatiquement activé. Pour tous les triggers activés le serveur Oracle vérifie les contraintes d'intégrité et garanties qu'aucun trigger ne peut interférer avec un autre. De plus le serveur fournit des vues logiques pour les requêtes et le contraintes, gère les dépendances et fournit une validation (COMMIT) en deux phases si un trigger met à jour des tables distantes d'une base de données distribuée.
Cependant si l'on veut désactiver un trigger, pour effectuer une opération que celui-ci interdit par exemple, on peut utiliser la commande ALTER TRIGGER en précisant le nom du trigger. On peut également désactiver tous les triggers de la base de données en une seule commande en utilisant la syntaxe ALTER TABLE.

```
ALTER TRIGGER trigger_name DISABLE | ENABLE
ALTER TABLE   table_name   DISABLE | ENABLE   ALL TRIGGERS
```

La désactivation d'un trigger améliore les performances des ordres DML et permet d'éviter la vérification de l'intégrité des données lors de la manipulation d'une quantité importante de données. La désactivation est utile

lorsque celui-ci fait référence à un objet de la base de données momentanément indisponible à cause d'une connection réseau échouée, une panne de disque ou un fichier de donnée ou un tablespace offline.
Pour recompiler explicitement un trigger invalide on utilise la commande ALTER TRIGGER. Lorsque que l'on exécute un ordre ALTER TRIGGER avec l'option COMPILE, le trigger est recompilé sans perdre sa validité ou son invalidité.

```
ALTER TRIGGER trigger_name  COMPILE
```

8.8.2 Suppression de triggers

Lorsqu'un trigger n'est plus nécessaire, on peut utiliser une commande SQL DROP sous SQL*Plus ou dans l'interpréteur de commande de Procedure Builder pour la supprimer.

```
DROP TRIGGER  trigger_name
```

Exemple :

```
SQL> DROP TRIGGER secure_emp;
Déclencheur supprimé.
```

8.9 Test de triggers

8.9.1 Cas à tester

Pour s'assurer du bon fonctionnement d'un trigger, on doit vérifier les différents cas possibles.

- On teste toutes les opérations déclenchantes ainsi que les opérations non déclenchantes afin de s'assurer que le trigger ne s'exécute que pour les cas prévus.
- On teste également chaque cas possible pour la clause WHEN.
- On provoque le déclenchement du trigger directement depuis une opération basique de manipulation de données ainsi qu'indirectement depuis une procédure.
- On doit également tester l'effet qu'a ce trigger sur tous les autres présents dans la base de données ainsi que l'effet des autres triggers sur lui.

8.9.2 Modèle d'exécution de trigger et vérification de contrainte

Tous les triggers, lors de l'exécution suivent un modèle prédéterminé :

- Exécution de tous les Statement triggers de type BEFORE
- Une boucle pour chaque ligne affectée :
  - Exécution de tous les Row triggers de type BEFORE
  - Exécution de l'ordre DML et vérification des contraintes d'intégrité
  - Exécution de tous les Row triggers de type AFTER
- Vérification différée des contraintes d'intégrité
- Exécution de tous les Statement triggers de type AFTER

Un ordre DML peut donc à lui seul engendrer jusqu'à quatre types de triggers : des Statement et Row triggers de type BEFORE et AFTER. Un événement ou un ordre déclenchant dans un trigger peut entraîner la vérification d'une ou plusieurs contraintes d'intégrité. Les triggers peuvent également entraîner le déclenchement d'autres triggers (triggers en cascade).
Toutes les actions et vérifications faites à la suite d'un ordre SQL doivent réussir. Si une exception est levée dans le trigger et si cette exception n'est pas gérée explicitement, toutes les actions exécutées dans par la requête SQL originale sont annulées (ROLLBACK), y compris les actions exécutées par les triggers déclenchant. Cette procédure garantie que les contraintes d'intégrités ne seront jamais compromises par les triggers.
Quand un trigger se déclenche, les tables désignées dans l'action du trigger peuvent être en cours de modification par des transactions d'autres utilisateurs. Dans tous les cas une image valide pour les valeurs modifiées est utilisée par le trigger pour les opérations de lecture et d'écriture.

8.10 Interactions

8.10.1 Une démonstration type

Pour démontrer le fonctionnement de l'interaction entre les triggers, les procédures de package, les fonctions et les variables globales on va utiliser les triggers AUDIT_EMP_TRIG et AUDTI_EMP_TAB, les fonctions et procédures inclues dans le package VAR_PACK et les variables globales définies dans le package. La structure de ces différents éléments est détaillée plus loin dans ce chapitre.
La séquence d'événements pour la démonstration commence par l'exécution d'un ordre DML INSERT, UPDATE ou DELETE manipulant plusieurs lignes. Cet ordre déclenche le Row trigger de type AFTER, AUDIT_EMP_TRIG, qui appelle la procédure de package qui incrémente les variables globales du package VAR_PACK. Ce trigger étant un Row trigger, il s'exécute une fois par ligne retournée et donc les variables globales correspondent au nombre de ligne retournées.
Une fois la requête terminée, le Statement trigger de type AFTER ; AUDIT_EMP_TAB, appelle la procédure AUDIT_EMP qui assigne les valeurs des variables globales à des variables locales en utilisant les fonctions du package, puis met à jour la table AUDIT_TABLE et enfin réinitialise les variables globales.

Commentaire : Pas trouvé de description claire de cette table. Un schéma ou autre serait peut-être nécessaire ? Je rajoute le script de création des fois que !

8.10.2 La table d'audit

```
CREATE TABLE audit_table
(
  user_name   VARCHAR2(30),
  tablename   VARCHAR2(30),
  column_name VARCHAR2(30),
  del         NUMBER(4),
  ins         NUMBER(4),
  upd         NUMBER(4)
);
insert into audit_table (user_name, tablename)
values   ('SCOTT','EMP');
insert into audit_table (user_name, tablename, column_name)
values ('SCOTT','EMP','SAL');
```

8.10.3 Les triggers

La démonstration utilise deux triggers : AUDIT_EMP_TRIG et AUDIT_EMP_TAB.
AUDIT_EMP_TRIG se déclenchera après chaque ligne manipulée. Le trigger incrémente les variables globales qui représentent le nombre de lignes qui ont été affectées.
AUDIT_EMP_TAB se déclenche quand la requête s'est terminée. Il invoque la procédure AUDIT_EMP qui est décrite dans le paragraphe suivant.

```
CREATE OR REPLACE TRIGGER audit_emp_trig
  AFTER    UPDATE or INSERT or DELETE on EMP
  FOR EACH ROW
  BEGIN
    IF      DELETING    THEN  var_pack.set_g_del(1);
    ELSIF   INSERTING   THEN  var_pack.set_g_ins(1);
    ELSIF   UPDATING ('SAL')
                        THEN  var_pack.set_g_up_sal(1);
    ELSE                      var_pack.set_g_upd(1);
    END IF;
  END audit_emp_trig;

  CREATE OR REPLACE TRIGGER audit_emp_tab
  AFTER    UPDATE or INSERT or DELETE on EMP
  BEGIN
    audit_emp;
  END audit_emp_tab;
```

8.10.4 Spécifications du package VAR_PACK

Le package VAR_PACK définit toutes les fonctions et procédures permettant l'incrémentation des variables globales en fonction du nombre de lignes retournées.

```
  CREATE OR REPLACE PACKAGE var_pack
  IS
  -- these functions are used to return the
  -- values of package variables
    FUNCTION g_del RETURN NUMBER;
    FUNCTION g_ins RETURN NUMBER;
    FUNCTION g_upd RETURN NUMBER;
    FUNCTION g_up_sal RETURN NUMBER;
  -- these procedures are used to modify the
  -- values of the package variables
    PROCEDURE set_g_del    (p_val  IN  NUMBER);
    PROCEDURE set_g_ins    (p_val  IN  NUMBER);
    PROCEDURE set_g_upd    (p_val  IN  NUMBER);
    PROCEDURE set_g_up_sal (p_val  IN  NUMBER);
  END var_pack;
  /

  CREATE OR REPLACE PACKAGE BODY var_pack
  IS
    gv_del     NUMBER  := 0;
    gv_ins     NUMBER  := 0;
    gv_upd     NUMBER  := 0;
    gv_up_sal  NUMBER  := 0;
  FUNCTION  g_del RETURN NUMBER
  IS   BEGIN    RETURN gv_del;    END g_del;
  FUNCTION  g_ins RETURN NUMBER
  IS   BEGIN    RETURN gv_ins;    END g_ins;
  FUNCTION  g_upd RETURN NUMBER
  IS   BEGIN    RETURN gv_upd;    END g_upd;
  FUNCTION  g_up_sal RETURN NUMBER
  IS   BEGIN    RETURN gv_up_sal;  END g_up_sal;
  PROCEDURE set_g_del (p_val  IN NUMBER)
  IS   BEGIN
         IF p_val = 0   THEN  gv_del := p_val;
         ELSE                gv_del := gv_del +1;
         END IF;
       END set_g_del;
  PROCEDURE set_g_ins (p_val  IN NUMBER)
  IS   BEGIN
         IF p_val = 0   THEN  gv_ins := p_val;
         ELSE                gv_ins := gv_ins +1;
         END IF;
```

```
     END set_g_ins;
PROCEDURE set_g_upd (p_val  IN NUMBER)
IS   BEGIN
       IF p_val = 0   THEN  gv_upd := p_val;
       ELSE                 gv_upd := gv_upd +1;
       END IF;
     END set_g_upd;
PROCEDURE set_g_up_sal (p_val  IN NUMBER)
IS   BEGIN
       IF p_val = 0   THEN  gv_up_sal := p_val;
       ELSE          gv_up_sal := gv_up_sal +1;
       END IF;
     END set_g_up_sal;
END var_pack;
```

8.10.5 Procédure

La démonstration utilise la procédure AUDIT_EMP. Cette procédure met à jour la table AUDIT_TABLE et appelle les fonctions permettant de réinitialiser les variables globales pour pouvoir les réutiliser avec le prochain ordre DML.

```
CREATE OR REPLACE PROCEDURE audit_emp
IS
  v_del     NUMBER   := var_pack.g_del;
  v_ins     NUMBER   := var_pack.g_ins;
  v_upd     NUMBER   := var_pack.g_upd;
  v_up_sal  NUMBER   := var_pack.g_up_sal;
BEGIN
  IF  v_del + v_ins + v_upd != 0
  THEN
    UPDATE audit_table SET
      del = NVL(del,0) + v_del,
      ins = NVL(ins,0) + v_ins,
      upd = NVL(upd,0) + v_upd
    WHERE upper(user_name)   = user
    AND   upper(tablename)  = 'EMP'
    AND   column_name IS NULL;
  END IF;
  IF v_up_sal != 0
  THEN
    UPDATE audit_table SET
      upd = NVL(upd,0) + v_up_sal
    WHERE user_name   = user
    AND   tablename  = 'EMP'
    AND   column_name = 'SAL';
  END IF;
-- resetting global variables
-- in package VAR_PACK
  var_pack.set_g_del (0);
  var_pack.set_g_ins (0);
  var_pack.set_g_upd (0);
  var_pack.set_g_up_sal (0);
END audit_emp;
```

9 COMPLEMENTS SUR LES TRIGGERS

9.1 Création de triggers sur des événements utilisateur

En plus des triggers sur les ordres DML (INSERT, UPDATE et DELETE), il est possible de créer des triggers sur des évènements utilisateurs. Ces évènements peuvent être des ordres DDL (CREATE, ALTER et DROP) ou une connexion/ déconnexion d'un utilisateur. Contrairement aux triggers sur des ordres DML qui porte sur une table ou une vue, les triggers sur les événements de l'utilisateur sont définis au niveau de la base de données ou au niveau du schéma.
Ce type de trigger peut être définit au niveau de la base de données, il se déclenche alors pour tous les utilisateurs ; ou définit au niveau du schéma ou de la table il ne se déclenche que lorsqu'un événement impliquant la table ou le schéma est exécuté. Ces triggers ne seront déclenchés que lorsque l'objet modifié est un cluster, une fonction, un index, un package, une procédure, un rôle, une séquence, un synonyme, une table, un tablespace, un trigger, une vue ou un utilisateur.
Ce type de trigger peut être déclenché lorsqu'un ordre de définition de données est exécuté sur un objet de la base de données ou du schéma et une connexion ou déconnection d'un utilisateur particulier ou de tout utilisateur de la base de données.
La syntaxe pour la création de ce type de triggers est la suivante :

```
CREATE [OR REPLACE] TRIGGER trigger_name
   timing
      [ ddl_event1 [OR ddl_event2 OR ...]]
          ON {DATABASE|SCHEMA}
trigger_body
```

Le moment de déclenchement du trigger appelé *timing* dans la syntaxe peut être BEFORE ou AFTER.
Bien que les triggers puissent être définis au niveau de la base de données, ils seront toujours déclenchés dans la transaction actuelle de l'utilisateur.

9.2 Création de triggers sur des évènements systèmes

Il existe également des triggers sur les événements de la base de données et les événements systèmes. Ces triggers sur des événements systèmes peuvent être définit au niveau de la base de données ou au niveau du schéma. Par exemple un trigger d'extinction de base de données est définit au niveau de la base de données. Ce type de trigger est déclenché lorsque la base est arrêtée ou démarrée ou lorsqu'une erreur particulière ou n'importe quelle erreur survient.

```
CREATE [OR REPLACE] TRIGGER trigger_name
   timing
     [ database_event1 [OR database_event2 OR ...]]
         ON {DATABASE|SCHEMA}
trigger_body
```

Il existe cinq types de triggers sur les événements systèmes :
AFTER SERVERERROR : le serveur Oracle déclenche le trigger à chaque fois q'un message d'erreur est enregistré
AFTER LOGON : le serveur Oracle déclenche le trigger pour chaque connexion d'un utilisateur sur la base de données.
BEFORE LOGOFF : le serveur Oracle déclenche le trigger pour chaque déconnexion d'un utilisateur de la base de données
AFTER STARTUP : le serveur Oracle déclenche le trigger quand la base de données est démarrée
BEFORE SHUTDOWN : le serveur Oracle déclenche le trigger quand la base de données est arrêtée normalement
Pour tous ces événements on peut créer des triggers sur la base de données ou sur un schéma, excepté pour SHUTDOWN et STARTUP qui ne peuvent s'appliquer qu'à la base de données.
Pour chaque événement système déclenchant, le serveur Oracle ouvre un domaine de transaction autonome, déclenche le trigger et valide toute autre transaction en cours sans se soucier des transactions de l'utilisateur.

Un domaine de transaction autonome est une transaction indépendante qui peut être validée sans que cela affecte les autres transactions en cours.

9.3 Des exemples des trigger Log On et Log Off

Grâce aux triggers d'événements systèmes on peut créer un trigger pour surveiller la fréquence de connexions des utilisateurs ainsi que le temps qu'ils restent connectés.

Exemple :

```
SQL> CREATE OR REPLACE TRIGGER LOGON_TRIG
  2  AFTER logon  ON  SCHEMA
  3  BEGIN
  4    INSERT INTO log_trig_table
  5            (user_id, log_date, action)
  6      VALUES (user, sysdate, 'Logging on');
  7  END;
SQL> CREATE OR REPLACE TRIGGER LOGOFF_TRIG
  2  BEFORE logoff  ON  SCHEMA
  3  BEGIN
  4     INSERT INTO log_trig_table
  5              (user_id, log_date, action)
  6       VALUES (user, sysdate, 'Logging off');
  7  END;
```

→ Ces deux ordres insèrent dans la table LOG_TRIG_TABLE la date de connexion et celle de déconnection de l'utilisateur.

L'exemple utilise la table LOG_TRIG_TABLE qui possède la structure suivante :

```
SQL> describe log_trig_table
Name                           Null?    Type
------------------------------ -------- ----
USER_ID                                 VARCHAR2(30)
LOG_DATE                                DATE
ACTION                                  VARCHAR2(30)
```

9.4 La déclaration CALL

Un trigger est composé de quatre parties : la synchronisation, l'événement déclenchant, le type et le corps. Cette dernière partie définit les actions qui seront effectuées lors du déclenchement du trigger. Elle peut être constituée d'un bloc PL/SQL ou un appel à une procédure.
Lorsque l'on veut appeler une procédure depuis un trigger on utilise la commande CALL :

```
CREATE [OR REPLACE] TRIGGER trigger_name
  timing
    event1 [OR event2 OR event3]
      ON table_name
  [REFERENCING OLD AS old | NEW AS new]
[FOR EACH ROW]
  [WHEN condition]
CALL procedure_name
```

Exemple :

```
SQL> CREATE TRIGGER TEST3
  2  BEFORE INSERT ON EMP
  3  CALL LOG_EXECUTION
  4  /
```

→ Ce trigger appelle la procédure LOG_EXECUTION avant d'exécuter un ordre INSERT sur la table EMP.

9.5 Les tables en cours de modifications

9.5.1 Lecture de donnée dans une table en cours de modifications

Une table en cours de modifications est une table dans laquelle sont effectuées des modifications grâce à des ordres DML UPDATE, INSERT ou DELETE ou une table ayant besoin d'être mise à jour par les effets d'un DELETE CASCADE. Une table n'est pas considérée comme en cours de modification lorsque le trigger est de type Statement.
La lecture et l'écriture dans les tables en cours de modification sont soumises à certaines règles. Ces restrictions ne s'appliquent que pour les Row triggers ou pour les triggers déclenchés à la suite d'un ordre ON DELETE CASCADE.
Une table sur laquelle agit un trigger est considérée comme en cours de modification, ainsi que chaque table lui faisant référence en tant que FOREIGN KEY. Cette restriction empêche le Row trigger de voir un ensemble de données inconsistant.

9.5.2 Exemple de table en cours de modifications

Exemple :

```
SQL> CREATE OR REPLACE TRIGGER check_salary
  2  BEFORE INSERT OR UPDATE OF sal, job ON emp
  3  FOR EACH ROW
  4  WHEN (new.job <> 'PRESIDENT')
  5  DECLARE
  6    v_minsalary emp.sal%TYPE;
  7    v_maxsalary emp.sal%TYPE;
  8  BEGIN
  9    SELECT MIN(sal), MAX(sal)
 10    INTO v_minsalary, v_maxsalary
 11    FROM emp
 12    WHERE job = :new.job;
 13    IF :new.sal < v_minsalary OR
 14       :new.sal > v_maxsalary THEN
 15      RAISE_APPLICATION_ERROR(-20505,'Out of range');
 16    END IF;
 17  END;
 18  /

SQL> UPDATE emp
  2  SET sal = 1500
  3  WHERE ename = 'SMITH';
*
ERROR at line 2:
ORA-04091: table EMP is mutating, trigger/function
 may not see it
ORA-06512: at "CHECK_SALARY", line 5
ORA-04088: error during execution of trigger
 'CHECK_SALARY'
```

→ On crée un trigger vérifiant que le salaire se trouve dans la bonne fourchette en fonction du type de job lors de l'ajout d'un employé ou de la modification d'un employé existent.

Lorsque l'on essaye de restreindre le salaire à des valeurs comprises entre une fourchette constituée des valeurs minimales et maximales existantes, il se produit une erreur d'exécution. La table EMP est considérée comme en cours de modification du fait de l'utilisation des valeurs qu'elle contient donc il est impossible de lire et d'écrire dans celle-ci pour des raisons d'intégrité des données utilisées pour le trigger.

9.6 Fonctions des triggers

9.6.1 Sécuriser le serveur

9.6.1.1 Contrôler la sécurité du serveur

Le serveur Oracle autorise l'accès des tables à toute personne possédant un compte sur le serveur. Pour contrôler la sécurité sur le serveur on définit des schémas et des rôles afin de contrôler les opérations de données sur les tables en fonction du nom de l'utilisateur.
Les privilèges accordés sont basés sur le nom de l'utilisateur fournit lors de la connexion à la base de données. On peut déterminer l'accès aux tables, vues, synonymes et séquences ainsi que les privilèges concernant les requêtes, la manipulation et la définition des données.

Exemple :

```
SQL> GRANT SELECT, INSERT, UPDATE, DELETE
  2  ON emp
  3  TO CLERK;    -- database role
SQL> GRANT CLERK TO SCOTT;
```

9.6.1.2 Contrôler la sécurité avec un trigger

Le contrôle de l'accès aux tables grâce au trigger ne se base plus sur le nom de l'utilisateur mais sur les valeurs des données. Cela permet de mettre en œuvre des spécifications de sécurité plus complexes.
Les privilèges accordés sont basés sur des valeurs de la base de données telles que l'heure, le jour de la semaine et autres. La sécurité gérée par un trigger permet de déterminer uniquement l'accès aux tables et les privilèges de manipulation de données.

Exemple :

```
SQL> CREATE OR REPLACE TRIGGER secure_emp
  2  BEFORE INSERT OR UPDATE OR DELETE ON emp
  3  DECLARE
  4    v_dummy VARCHAR2(1);
  5  BEGIN
  6    IF TO_CHAR (sysdate, 'DY' IN ('SAT','SUN'))
  7    THEN RAISE_APPLICATION_ERROR (-20506,
  8      'Modification sur la table EMP possible uniquement durant
  9       les heures de travail.');
 10    END IF;
 11    SELECT COUNT(*) INTO v_dummy FROM holiday
 12    WHERE holiday_date = TRUNC (sysdate);
 13    IF v_dummy > 0 THEN RAISE_APPLICATION_ERROR (-20507,
 14       'Impossible de modifier les donnée pendant les vacances.');
 15    END IF;
 16  END;
 17  /
```

→ Ce trigger n'autorise la modification des données que lors des heures de travail et hors des vacances.

9.6.2 Audit

9.6.2.1 Audit grâce aux fonctions systèmes

Le serveur Oracle garde une trace des opérations de données effectuées sur les tables grâces à des fonctions prédéfinies. Les événements audités sont enregistrés dans la table du dictionnaire de données.

Il est possible d'auditer les ordres de récupération de données, de manipulation de données et de définition de données. Les traces de l'audit sont toutes écrites dans la table d'audit centralisée. On peut générer un rapport d'audit une fois par session ou une fois par tentative d'accès. L'audit peut se faire sur des tentatives d'accès réussies, des tentatives échouées ou les deux et il peut être activé et désactivé dynamiquement.
Pour activer un audit sur un objet on utilise la commande AUDIT :

```
AUDIT statement_opt| system_priv[, ...] [BY user[, ...]] [BY
[SESION|ACCESS]] [WHENEVER [NOT] SUCCESSFUL];
```

Exemples :

```
AUDIT ROLE;
AUDIT ROLE WHENEVER SUCCESSFUL;
AUDIT ROLE WHENEVER NOT SUCCESSFUL;
AUDIT SELECT TABLE, UPDATE TABLE;
AUDIT SELECT TABLE, UPDATE TABLE BY scott, blake;
AUDIT DELETE ANY TABLE;
```

9.6.2.2 Audit grâce à un trigger

Les triggers gardent une trace des valeurs pour les opérations de données sur les tables.
Les triggers ne peuvent auditer que les ordres de manipulation de données. Toutes les informations de l'audit seront enregistrées dans la table d'audit définie par l'utilisateur. Il est possible de générer un rapport d'audit une fois par ordre ou bien une fois pour chaque ligne. Seuls les tentatives réussies sont auditées. Comme pour l'audit à partir des fonctions systèmes, il est possible d'activer et de désactiver dynamiquement les triggers d'audit.

Exemple :

```
SQL> CREATE OR REPLACE TRIGGER audit_emp_values
  2  AFTER DELETE OR INSERT OR UPDATE ON emp
  3  FOR EACH ROW
  4  BEGIN
  5    IF audit_emp_package.g_reason IS NULL THEN
  6       RAISE_APPLICATION_ERROR (-20059, 'Spécifier une raison
  7       pour l''opération contenant la procédure
  8       SET_REASON avant de l''exécuter.');
  9    ELSE
 10       INSERT INTO audit_emp_table (user_name, timestamp, id,
 11        old_last_name, new_last_name, old_title, new_title,
 12        old_salary, new_salary, comments)
 13       VALUES (user, sysdate, :old.empno,:old.ename,
 14        :new.ename, :old.job, :new.job, :old.sal,
 15        :new.sal, :audit_emp_package.g_reason);
 16    END IF;
 17  END;
 18  /
SQL> CREATE TRIGGER cleanup_audit_emp
  2  AFTER INSERT OR UPDATE OR DELETE ON emp
  3  BEGIN
  4    audit_emp_package.g_reason := NULL;
5 END;
```

→ L'utilisateur fait appel à AUDIT_EMP_PACKAGE en définissant la variable G_REASON. Lorsqu'il effectue un ordre DML, le trigger vérifie la raison de cet ordre. Si G_REASON n'est pas définie une erreur est levée, et si elle est définie, l'ordre DML s'exécute correctement et l'évènement est enregistré dans la table AUDIT_EMP_TABLE. Le second trigger réinitialise la valeur de G_REASON.

9.6.3 Garantir l'intégrité des données

9.6.3.1 Surveiller l'intégrité des données sur le serveur

Le serveur Oracle permet de mettre en œuvre des contraintes d'intégrité afin de garantir l'intégrité des données. Les contraintes d'intégrités standards sont : NOT NULL, UNIQUE, PRIMARY KEY et FOREIGN KEY. Les contraintes permettent d'avoir des valeurs par défaut constantes. Il est également possible de mettre en place des contraintes statiques. Les contraintes peuvent être activées et désactivées dynamiquement.

Exemple :

```
SQL> ALTER TABLE emp ADD
  2   CONSTRAINT ck_salary CHECK (sal >= 500);
```

→ Cette contrainte assure que le salaire sera toujours supérieur a 500.

9.6.3.2 Protéger l'intégrité des données grâce à un trigger

L'utilisation de trigger pour protéger l'intégrité des données apporte un niveau plus élevé de complexité au niveau des règles d'intégrité.
L'utilisation de trigger pour protéger l'intégrité des données permet de spécifier des valeurs par défaut variables. Cela permet également de mettre en place des contraintes dynamiques ainsi que de les activer et de les désactiver dynamiquement.
Pour protéger l'intégrité des données il faut incorporer des contraintes déclaratives dans la déclaration de la table.

Exemple :

```
SQL> CREATE OR REPLACE TRIGGER check_salary
  2  BEFORE UPDATE OF sal ON emp
  3  FOR EACH ROW
  4  WHEN (new.sal < old.sal) OR
  5       (new.sal > old.sal * 1.1)
  6  BEGIN
  7     RAISE_APPLICATION_ERROR (-20508,
  8     'Le salaire ne doit pas être augmenter ou réduit de
  9      plus de 10%.');
 10  END;
 11  /
```

→ Permet de s'assurer que le salaire ne sera jamais augmenté ou réduit de plus de 10 % d'un coup.

9.6.4 Garantir l'intégrité référentielle

9.6.4.1 Appliquer l'intégrité référentielle sur le serveur

Le serveur Oracle permet de mettre en œuvre des règles standard d'intégrité référentielle, c'est-à-dire la mise en place de PRIMARY KEY et FOREIGN KEY.
L'intégrité référentielle au sein du serveur Oracle permet de restreindre l'utilisation de UPDATE et DELETE. Elle permet également de faire des DELETE en cascade. Il est possible de les mettre en place et de les désactiver dynamiquement.

Exemple :

```
SQL> ALTER TABLE emp
  2  ADD CONSTRAINT emp_deptno_fk
```

```
  3  FOREIGN KEY (deptno) REFERENCES dept(deptno)
  4  ON DELETE CASCADE;
```

→ Lorsqu'un departement est supprimé de la table parent DEPT, les lignes correspondantes à la clé sont supprimées de la table enfant EMP.

9.6.4.2 Protéger l'intégrité référentielle grâce à un trigger

L'utilisation des triggers pour gérer l'intégrité référentielle permet d'implémenter des fonctionnalités non standard sur cette intégrité.
Un trigger gérant l'intégrité référentielle permet d'effectuer des UPDATE en cascade ainsi que de définir une valeur par défaut et d'avoir une valeur NULL pour les UPDATE et les DELETE. Ce genre de trigger permet la mise en place d'une intégrité référentielle dans un système distribué.

Exemple :

```
SQL> CREATE OR REPLACE TRIGGER cascade_updates
  2  AFTER UPDATE OF deptno ON dept
  3  FOR EACH ROW
  4  BEGIN
  5    UPDATE emp
  6    SET    emp.deptno =  :new.deptno
  7    WHERE  emp.deptno =  :old.deptno;
  8  END;
  9  /
```

→ Ce trigger met en place l'intégrité référentielle. Lorsqu'une valeur DEPTNO est modifiée dans la table parent, ce changement est répercuté sur les lignes correspondantes dans la table enfant EMP par le biais d'un UPDATE. Cet exemple ne fonctionne que s'il n'existe pas d'intégrité référentielle entre les deux tables dans la definition des tables.

9.6.5 Dupliquer les tables

9.6.5.1 Dupliquer une table du serveur

Le serveur Oracle est capable de copier de manière asynchrone des tables dans des snapshots. Un snapshot est une copie en local de données d'une table provenant d'une ou plusieurs tables maîtres. Les données d'une table snapshot peuvent être lues, mais il est impossible d'effectuer des INSERT, UPDATE ou DELETE, les snapshots sont donc en lecture seule. Afin de conserver les données du snapshot à jour le serveur Oracle doit performer régulièrement des rafraîchissements de celui-ci avec les tables maîtres.
Il est possible de copier les tables de manière asynchrone, à des intervalles définis pas l'utilisateur. Les snapshots peuvent être basés sur plusieurs tables maîtres. On ne peut que lire à partir des snapshots. Les snapshots apportent une amélioration des performances lors de la manipulation de données sur la table maîtresse, particulièrement lorsque le réseau est hors service.

Exemple :

```
SQL> CREATE SNAPSHOT emp_copy AS
2  SELECT * FROM emp@ny;
```

→ Cet exemple crée un snapshot de la table EMP située à New York

9.6.5.2 Dupliquer une table grâce à un trigger

On peut également créer des copies de tables avec un trigger. Les tables dupliquées par cette méthode sont appelées des répliquas.
Ce trigger copiera les tables de manière synchrone en temps réel. Habituellement les répliquas sont basés sur une seule table maîtresse. Il est possible de lire à partir de ces répliquas mais contrairement aux snapshots on peut également y écrire. Les répliquas détérioreront les performances des manipulations de données sur la table maîtresse surtout si le réseau est hors service car il n'existe pas de synchronisation entre la table maîtresse et

le répliqua. Il est donc préférable de garder des copies mises à jour automatiquement avec la table maîtresse comme c'est le cas pour les snapshots.

Exemple :

```
SQL> CREATE OR REPLACE TRIGGER emp_replica
  2  BEFORE INSERT OR UPDATE ON emp
  3  FOR EACH ROW
  4  BEGIN
  5    IF INSERTING THEN
  6       IF :new.flag IS NULL THEN
  7          INSERT INTO emp@sf VALUES (:new.empno,
  8          :new.ename,...,'B');
  9          :new.flag = 'A';
 10        ELSE /* Updating. */
 11        IF :new.flag = :old.flag THEN
 12            UPDATE emp@sf SET ename = :new.ename, ...,
 13                              FLAG = :new.flag
 14            WHERE empno = :new.empno;
 15        END IF;
 16      IF :old.flag = 'A' THEN :new.flag := 'B';
 17      ELSE :new.flag := 'A';
 18      END IF;
 19    END IF;
 20  END;
 21  /
```

→ On duplique la table EMP située à New York

Dans ces deux exemples si le réseau est mis hors service et que l'on avait utilisé la méthode du snapshot, les fonctionnalités du serveur Oracle vont assurer que les utilisateurs du nœud principal ne seront pas affectés puisque le snapshot sera maintenu de manière asynchrone.
Si l'on avait utilisé la méthode du trigger, les utilisateurs ne pourront pas continuer à travailler puisque le trigger ne sera pas capable d'écrire sur la base de données distante.

9.6.6 Utiliser des données dérivées

9.6.6.1 Calculer des données dérivées sur le serveur

Le serveur Oracle calcule les valeurs dérivées des données manuellement.
Les valeurs dérivées des colonnes sont calculées de manière asynchrone, à des intervalles définis par l'utilisateur. Les valeurs dérivées ne peuvent être stockées que dans les tables de la base de données. La méthode de calcule des valeurs dérivées est la suivante : les données sont modifiées lors d'un premier parcours dans la base de données, puis les données dérivées sont calculées lors d'un second parcours.

Exemple :

```
SQL> UPDATE dept
  2  SET total_sal = (SELECT SUM(salary)
  3                   FROM emp
  4                   WHERE emp.deptno = dept.deptno);
```

→ Le total des salaires pour le department est conservé dans une colonne TOTAL_SALARY de la table DEPT

9.6.6.2 Calculer des valeurs dérivées grâce à un trigger

Un trigger calcule les valeurs dérivées automatiquement lors de l'exécution.
Les valeurs dérivées des colonnes sont calculées de manière synchrone, en temps réel. Les valeurs dérivées peuvent être stockées dans les tables de la base de données ou dans les variables globales d'un package. Les données sont modifiées puis les valeurs des valeurs dérivées sont calculées en un seul parcours de la base de données.

Exemple :

```
SQL> CREATE OR REPLACE PROCEDURE increment_salary
  2      (v_id IN dept.deptno%TYPE,
  3       v_salary IN dept.total_salary%TYPE)
  4  IS
  5  BEGIN
  6    UPDATE dept
  7    SET total_sal = NVL (total_sal,0)+ v_salary
  8    WHERE deptno = v_id;
  9  END increment_salary;
 10  /

SQL> CREATE OR REPLACE TRIGGER compute_salary
  2  AFTER INSERT OR UPDATE OF sal OR DELETE ON emp
  3  FOR EACH ROW
  4  BEGIN
  5  IF    DELETING
  6  THEN increment_salary(:old.deptno, -1 * :old.sal);
  7  ELSIF  UPDATING
  8  THEN increment_salary(:new.deptno, :new.sal-:old.sal);
  9  ELSE increment_salary(:new.deptno, :new.sal);--INSERT
 10  END IF;
 11  END;
 12  /
```

→ Ce trigger conserve un total courant des salaires pour chaque department dans la colonne TOTAL_SALARY de la table DEPT.

9.6.7 Gérer les logs d'événements avec les triggers

Le serveur Oracle gère les logs d'événements explicitement tandis que les triggers les gèrent de façon transparente.
Pour enregistrer un événement en utilisant les fonctionnalités du serveur on soumet une requête pour déterminer si l'opération est nécessaire. Ensuite on doit exécuter l'opération, qui peut être un envoi de message par exemple.
Lorsque l'on utilise un trigger, l'opération est exécutée implicitement sans que l'utilisateur le remarque. Les données sont modifiées et les opérations dépendantes sont exécutées en une seule étape. Avec les triggers les événements sont enregistrés dès que les données ont changé.

Exemple :

```
CREATE OR REPLACE TRIGGER notify_reorder_rep
BEFORE UPDATE OF amount_in_stock, reorder_point
  ON inventory FOR EACH ROW
  DECLARE
    v_descrip product.descrip%TYPE;
    v_msg_text VARCHAR2(2000);
    stat_send number(1);
  BEGIN
   IF :new.amount_in_stock <= :new.reorder_point THEN
     SELECT descrip INTO v_descrip
       FROM PRODUCT WHERE prodid = :new.product_id;
    v_msg_text := 'ALERT: INVENTORY LOW ORDER:'||CHR(10)||
    'It has come to my personal attention that, due to recent'
    ||CHR(10)||'transactions, our inventory for product # '||
    TO_CHAR(:new.product_id)||'-- '||v_descrip ||
    ' -- has fallen below acceptable levels.'||CHR(10) ||
    'Yours,' ||CHR(10) ||user || '.'|| CHR(10)|| CHR(10);
      ELSIF :old.amount_in_stock<:new.amount_in_stock THEN NULL;
      ELSE v_msg_text := 'Product #'|| TO_CHAR(:new.product_id)
    ||' ordered. '|| CHR(10)|| CHR(10); END IF;
      DBMS_PIPE.PACK_MESSAGE(v_msg_text);
     stat_send := DBMS_PIPE.SEND_MESSAGE('INV_PIPE');
END;
```

→ Dans le serveur les événements sont enregistrés en exécutant une requête sur des données puis en exécutant des actions manuellement, afin d'envoyer un message en utilisant un pipe quand l'inventaire pour un produit particulier atteint un certain seuil critique. Ce trigger utilise le package fournit par Oracle DBMS_PIPE pour envoyer le message.

9.6.8 Avantages des triggers de base de données

Les triggers de base de données apportent une sécurité des données accrue en effectuant des vérifications de sécurité basées sur les valeurs ainsi qu'en permettant l'audit basé sur les valeurs elles-mêmes et non sur les actions.
Les triggers de base de données apportent une meilleure intégrité des données en autorisant la mise en place dynamique de contraintes d'intégrité de données. Ils autorisent également l'utilisation de contraintes d'intégrité référentielle complexes et permettent de toujours exécuter les actions liées ensemble.
Les triggers sont utilisés en alternative à des fonctionnalités fournies par le serveur Oracle. Mais on les utilise le plus souvent pour obtenir des fonctionnalités plus simples ou plus complexes que celle fournies par le serveur, ou si la fonctionnalité n'existe pas sur le serveur.

10 GERER LES SOUS PROGRAMMES ET LES TRIGGERS

10.1 Privilèges

10.1.1 Privilèges systèmes

Les privilèges sont les privilèges permettant aux utilisateurs d'effectuer des modifications (création, suppression d'objet) sur la base de données directement.
Pour pouvoir créer les procédures et les triggers il faut les privilèges systèmes CREATE PROCEDURE et CREATE TRIGGER.
On peut modifier, supprimer ou exécuter (ALTER, DROP, EXECUTE) les sous-programmes et triggers que l'on a créés sans avoir besoin d'autres privilèges. Pour pouvoir modifier les sous-programmes et triggers des autres utilisateurs il faut spécifier le paramètre ANY lors du CREATE TRIGGER ou CREATE PROCEDURE. Il existe également des privilèges spécifiques permettant à l'utilisateur les possédant de supprimer, modifier ou exécuter n'importe quel sous-programme ou trigger : CREATE ANY [PROCEDURE | TRIGGER], DROP ANY [PROCEDURE | TRIGGER] ou EXECUTE ANY PROCEDURE.
Le mot clé PROCEDURE est utilisé pour désigner les droits sur les procédures stockées, les fonctions et les packages.

10.1.2 Privilèges objets

Les privilèges objets sont des privilèges que les utilisateurs peuvent accorder à d'autres utilisateurs sur des objets dont ils sont propriétaires.
Si un sous-programme PL/SQL ou le trigger fait référence à des objets situés sur d'autres schémas, il faut que l'on nous autorise explicitement à l'exécuter, cette autorisation n'est pas valable si elle est donnée par l'intermédiaire d'un rôle.
Par défaut les sous-programmes PL/SQL sont exécutés sous le domaine de sécurité du propriétaire donc pour exécuter des sous-programmes PL/SQL si l'on n'a pas le privilège système EXECUTE ANY, il nous faut le privilège objet EXECUTE. Ce privilège est accordé par le propriétaire de l'objet.
Comme les triggers sont exécutés à partir d'ordre DML, il n'est pas nécessaire d'avoir de privilèges pour les exécuter.

10.2 Accorder des accès aux données

Tous les utilisateurs de la base de données peuvent accorder des accès sur les objets dont ils sont propriétaires aux autres utilisateurs. Cet accès peut être un accès direct ou indirect.
Supposons une table EMP située sur le schéma du personnel et un développeur nommé Scott et un utilisateur final Green. On veut s'assurer que Green ne peut accéder à la EMP que par le biais de la procédure QUERY_EMP, dont Scott est propriétaire. Pour ce faire on peut donner un accès direct ou indirect à Green.
Accès Direct :
L'accès direct consiste à donner les privilèges objets sur l'objet concerné et a créer une procédure exploitant l'objet.

-depuis le schéma du personnel on donne les privilèges objet sur la table EMP à SCOTT

```
SQL> GRANT  select
  2  ON     emp
  3  TO     scott;
```

-Scott créé la procédure QUERY_EMP effectuant une requête sur la table EMP

Accès Indirect :
L'accès indirect consiste à donner le privilège objet EXECUTE sur la procédure nécessitant un accès à la table d'un autre utilisateur.

-SCOTT fournit le privilège objet EXECUTE à GREEN sur la procédure QUERY_EMP

```
SQL> GRANT  execute
  2  ON     query_emp
  3  TO     green:
```

Par défaut le sous-programme PL/SQL sera exécuté sous le domaine de sécurité du propriétaire. Comme Scott a les privilèges directs sur la table EMP et a créé la procédure QUERY_EMP, Green peut récupérer des informations de la table EMP en utilisant la procédure QUERY_EMP

Si Green possède une table EMP dans son schéma, la procédure QUERY_EMP ne fera pas référence à cette table, elle fera appel à la table EMP à laquelle le schéma de Scott peut accéder. Cela peut être une table EMP personnel ou bien une table EMP publique.

10.3 Spécifier les droits des utilisateurs

Si l'on veut être sur que la procédure s'exécute en utilisant le domaine de sécurité de l'utilisateur l'exécutant et pas celui du propriétaire on peut spécifier le paramètre AUTHID CURRENT_USER lors de la création de la procédure. Ainsi à chaque exécution de la procédure les droits de l'utilisateur seront utilisés en priorité.
Par défaut la procédure s'exécute en utilisant le domaine de sécurité du propriétaire mais on peut également explicitement forcer la procédure à toujours utiliser le domaine de sécurité du propriétaire en spécifiant le paramètre AUHTID.

Exemple :

```
CREATE OR REPLACE PROCEDURE
  query_emp
  (v_id IN emp.empno%TYPE,
   v_name OUT emp.ename%TYPE,
   v_salary OUT emp.sal%TYPE,
   v_comm OUT emp.comm%TYPE)
AUTHID CURRENT_USER
IS
BEGIN
   SELECT ename,sal,comm
   INTO v_name,v_salary,v_comm
   FROM emp
   WHERE empno=v_id;
END query_emp;
```

10.4 Gérer les objets PL/SQL stockés

Lorsqu'une procédure est créée, toutes les informations la concernant sont enregistrées dans le dictionnaire de données ou dans des variables afin que l'on puissent les consulter depuis un éditeur SQL.

Information stockée	Description	Méthode d'accès
Générale	Information sur l'objet	Vue USER_OBJECTS du dictionnaire de données Explorateur d'objets de Procedure Builder
Code source de procédure	Texte de la procédure	Vue USER_SOURCE du dictionnaire de données Editeur de programmes stockés de PB
Code source de trigger	Texte du trigger	Vue USER_TRIGGERS du dictionnaire de données Editeur de trigger de Procedure Builder
Paramètres	Mode : IN/OUT/IN OUT Et type de données	Commande DESCRIBE de SQL*Plus Commande .DESCRIBE de PB
p-code	Code de l'objet compilé	Non accessible
Erreurs de compilation	Erreurs de syntaxe PL/SQL	Vue USER_ERRORS du dictionnaire de données Commande SHOW_ERRORS de SQL*Plus Compilateur de Procedure Builder
Information de debug	Variables et messages de debug	Package fournit par Oracle DBMS_OUTPUT

Lors de l'exécution	Spécifiés par l'utilisateur	Interpréteur PL/SQL de Procedure Builder

10.5 Informations sur l'objet

10.5.1 USER_OBJECTS

Pour obtenir des informations sur les objets stockés dans un schéma de la base de données on utilise la vue USER_OBJECTS du dictionnaire de données. Cette vue contient quelques colonnes suivantes :

Colonne	Description de la colonne
OBJECT_NAME	Nom de l'objet
OBJECT_ID	Identifiant interne de l'objet
OBJECT_TYPE	Type de l'objet. Il peut être TABLE, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER
CREATED	Date de création de l'objet
LAST_DDL_TIME	Date de la dernière modification de l'objet
TIMESTAMP	Date et heure de la dernière recompilation
STATUS	VALID ou INVALIDE

On peut également regarder dans les vues ALL_OBJECTS et DBA_OBJECTS qui contiennent en plus le nom du propriétaire de l'objet dans la colonne OWNER.

10.5.2 Lister toutes les procédures et fonctions

Pour lister toutes les procédures et fonctions de notre schéma on fait un SELECT sur la vue en spécifiant le type PROCEDURE et FUNCTION dans la clause WHERE.

Exemple :

```
SQL> SELECT object_name, object_type
  2  FROM user_objects
  3  WHERE object_type in ('PROCEDURE', 'FUNCTION')
  4  ORDER BY object_name
  5  /

OBJECT_NAME          OBJECT_TYPE
-----------------    ---------------------
ADD_DEPT             PROCEDURE
LEAVE_EMP            PROCEDURE
LOG_EXECUTION        PROCEDURE
PROCESS_EMPS         PROCEDURE
QUERY_EMP            PROCEDURE
RECEIVE_MESSAGE      PROCEDURE
SEND_MESSAGE         PROCEDURE
TAX                  FUNCTION
```

→ Affiche le nom de toutes les procedures et fonctions que l'on a créées

10.6 Texte de la procédure

10.6.1 La vue du dictionnaire de données USER_SOURCE

Lorsque l'on compile une procédure, le code source est enregistré dans une vue du dictionnaire de données. Ainsi pour examiner le code source de la procédure on peut soit regarder dans le fichier script ayant servi pour sa création, soit interroger la vue USER_SOURCE.

Colonne	Description de la colonne
NAME	Nom de l'objet
TYPE	Type de l'objet, par exemple PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY
LINE	Numéro de ligne du code source
TEXT	Texte de la ligne de code

Cette vue du dictionnaire de données peut être utilisée sous SQL*Plus ou sous Procedure Builder pour régénérer un fichier script créant la procédure au cas ou le fichier source original soit manquant.
On peut également utiliser les vues ALL_SOURCE et DBA_SOURCE qui contiennent en plus le nom du propriétaire de la procédure dans la colonne OWNER.

10.6.2 Lister le code de toute les procédures et fonctions

Pour lister le code source d'une procédure on effectue un SELECT sur la vue USER_SOURCE en précisant le nom de cette procédure dans la clause WHERE.

Exemple :

```
SQL> SELECT text
  2  FROM user_source
  3  WHERE name = 'QUERY_EMP'
  4  ORDER BY line;

TEXT
--------------------------------------------------
PROCEDURE QUERY_EMP
  (v_id IN emp.empno%TYPE,
   v_name OUT emp.ename%TYPE,
   v_salary OUT emp.sal%TYPE,
   v_comm OUT emp.comm%TYPE)
IS
BEGIN
   SELECT  ename , sal, comm
   INTO    v_name, v_salary, v_comm
   FROM    emp
   WHERE   empno = v_id;
END query_emp;
```

→ Cette requête affiche le code source de la procedure QUERY_EMP.

10.6.3 Lister le code des procédures stockées avec procedure builder

Pour lister le code d'une procédure avec Procedure Builder on utilise l'explorateur d'objets.
On se connecte à la base de données, on sélectionne les objets de la base de données et on clique le bouton *Expand*. On choisit ensuite le schéma du propriétaire de la procédure et on clique de nouveau sur le bouton *Expand* On sélectionne le nœud des programmes stockés et on le développe comme précédemment. Enfin on double-clique sur la procédure stockée pour faire apparaître l'éditeur de programmes stockés contenant le code source de la procédure.
Une fois le code affiché il est possible de l'exporter vers un fichier texte en choisissant *Export* dans le menu de l'éditeur. Le code source sera stocké dans un fichier *.pls*.

10.7 Texte d'un trigger

10.7.1 USER_TRIGGERS

Comme pour les procédures, lorsque l'on compile un trigger le code source est enregistré dans la vue USER_TRIGGERS du dictionnaire de données. Cette vue est composée, entre autres, des colonnes suivantes :

Colonne	Description de la colonne
TRIGGER_NAME	Nom du trigger
TRIGGER_TYPE	Type : BEFORE, AFTER, INSTEAD OF
TRIGGERING_EVENT	Opération DML déclenchant le trigger
TABLE_NAME	Nom de la table concernée par le trigger
REFERENCING_NAMES	Nom utilisée par :OLD et :NEW
WHEN_CLAUSE	Clause WHEN utilisée pour le déclenchement
STATUS	Statut du trigger
TRIGGER_BODY	Actions effectuées lors du déclenchement

On peut également utiliser cette vue pour régénérer le fichier script d'un trigger par le biais de SQL*Plus ou de l'éditeur de trigger de base de données de Procedure Builder.
Les vues ALL_TRIGGERS et DBA_TRIGGERS permettent également d'obtenir des informations sur les triggers avec en plus le nom du propriétaire dans la colonne OWNER.

10.7.2 Lister le code des triggers

Pour afficher les informations d'un trigger spécifique on effectue un SELECT dans la vue en spécifiant le nom de trigger dans la clause WHERE :

Exemple :

```
SQL> SELECT trigger_name, trigger_type,
  2  triggering_event, table_name, referencing_names,
  3  when_clause, status, trigger_body
  4  FROM user_triggers
  5  WHERE trigger_name = 'DERIVE_COMMISSION_PCT';

TRIGGER_NAME              TRIGGER_TYPE     TRIGGERING_EVENT
------------------------- ---------------- ----------------
TABLE_NAME           REFERENCING_NAMES
-------------------- -----------------------------------
WHEN_CLAUSE                          STATUS
------------------------------------ --------
TRIGGER_BODY
--------------------------------------------------------------
DERIVE_COMMISSION_PCT     BEFORE EACH ROW INSERT OR UPDATE
EMP                   REFERENCING NEW AS NEW OLD AS OLD
ENABLED
BEGIN
  IF NOT (:NEW.JOB IN ('MANAGER' , 'PRESIDENT'))
     AND :NEW.SAL > 5000
  THEN
     RAISE_APPLICATION_ERROR
        (-20202, 'EMPLOYEE CANNOT EARN THIS AMOUNT');
  END IF;
END;
```

→ Affiche les informations sur le trigger DERIVE_COMMISSION_PCT en utilisant la vue USER_TRIGGERS

10.8 Les paramètres

10.8.1 DESCRIBE dans SQL*Plus

Pour afficher les informations concernant les paramètres d'une fonction on utilise la commande DESCRIBE de SQL*Plus. Cette commande affichera le nom de arguments, leur type, le type de paramètre ainsi que la présence d'une valeur par défaut s'il y en a une.

Exemple :

```
SQL> DESCRIBE QUERY_EMP
PROCEDURE QUERY_EMP
Argument Name      Type              In/Out Default?
------------------ ----------------- ------ --------
V_ID               NUMBER(4)         IN
V_NAME             VARCHAR2(10)      OUT
V_SALARY           NUMBER(7,2)       OUT
V_COMM             NUMBER(7,2)       OUT

SQL> DESCRIBE ADD_DEPT
PROCEDURE ADD_DEPT
Argument Name      Type               In/Out Default?
------------------ ------------------ ------ --------
V_NAME             VARCHAR2(14)       IN     DEFAULT
V_LOC              VARCHAR2(13)       IN     DEFAULT

SQL> DESCRIBE TAX
FUNCTION TAX RETURNS NUMBER
Argument Name      Type              In/Out Default?
------------------ ----------------- ------ --------
V_SALARY           NUMBER            IN
```

→ Ces exemples affichent les informations concernant différentes procedures et fonctions.

10.8.2 La commande .DESCRIBE sous procedure builder

Sous Procedure Builder, pour afficher les informations sur les paramètres d'une fonction on utilise la commande .DESCRIBE dans l'interpréteur de commandes PL/SQL. Le résultat sera affiché sous forme d'un rapport et pas sous forme d'un tableau comme sous SQL*Plus.

Exemple :

```
PL/SQL> .DESCRIBE PROCEDURE FORMAT_PHONE
Procedure Body; FORMAT_PHONE
  Paramters:
    v_phone_no IN OUT VARCHAR2
  Compiled: YES
  Open: NO
  References:
    Package Spec STANDARD
  Referenced by:
PL/SQL>
```

→ Informations sur la procédure FORMAT_PHONE par le biais de Procedure Builder

10.9 Erreurs de compilation

10.9.1 Détecter les erreurs de compilation avec l'éditeur de programme stocké

Lorsque l'on enregistre une procédure dans Procedure Builder, celle-ci est compilée. Si des erreurs se produisent durant cette compilation, elles seront affichées dans le panneau des erreurs de compilation, dans la base de la fenêtre de l'éditeur de programmes et le curseur se déplace automatiquement à l'endroit de la première erreur dans le code source.
Si aucune erreur n'est détectée, « Successfully Compiled » apparaît dans le panneau de compilation.

10.9.2 USER_ERRORS

Pour visualiser le texte des erreurs de compilation, on utilise la vue USER_ERRORS du dictionnaire de données ou la commande SQL*Plus SHOW ERRORS. La vue USER_ERRORS contient les colonnes suivantes :

Colonne	Description de la colonne
NAME	Nom de l'objet
TYPE	Type de l'objet, par exemple PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER
SEQUENCE	Numéro de séquence, pour le tri
LINE	Numéro de ligne du code source à laquelle l'erreur s'est produite
POSITION	Position dans la ligne où l'erreur s'est produite
TEXT	Texte du message d'erreur

On peut également obtenir des informations sur les erreurs de compilation grâce aux vues ALL_ERRORS et DBA_ERRORS, qui contiennent en plus le nom des propriétaires de l'objet.

10.9.3 Détecter les erreurs de compilation : exemple

Exemple 1 :

```
SQL> CREATE OR REPLACE PROCEDURE log_execution
  2  IS
  3  BEGIN
  4  INPUT INTO LOG_TABLE (user_id, log_date)    -- erreur
  5  VALUES (user, sysdate);
  6  END;
  7  /

Procédure créée avec erreurs de compilations.
```

→ Ce code produit une erreur mais ne précisent pas leurs natures.

Pour obtenir les informations concernant les erreurs on peut utiliser la vue USER_ERRORS dans un ordre SELECT afin d'afficher les informations dont on a besoin pour corriger le code.

Exemple 2 :

```
SQL> SELECT   line|| ''|| position POS,text
  2  FROM     user_errors
  3  WHERE    name = 'LOG_EXECUTION'
  4  ORDER BY line;
POS        TEXT
---------- ------------------------------------------------
4/7        PLS-00103: Encountered the symbol "INTO" when
           expecting one of the following:
           := . ( @ % ;
5/1        PLS-00103: Encountered the symbol "VALUES" when
           expecting one of the following:
           . ( , % from
           The symbol "VALUES" was ignored.
5/23       PLS-00103: Encountered the symbol ";" when
           expecting one of the following:
           . ( , % from
```

→ Cet ordre fournit des inforamtions completes sur les erreurs survenues lors de la compilation de la procedure LOG_EXECUTION.

Il est également possible de visualiser les erreurs de compilation grâce à la commande SHOW ERRORS au prompt SQL de SQL*Plus

Exemple 3 :

```
SQL> SHOW ERRORS PROCEDURE log_execution

Errors for PROCEDURE LOG_EXECUTION:
LINE/COL ERROR
-------- -------------------------------------------------
4/7      PLS-00103: Encountered the symbol "INTO" when
         expecting one of the following:
         := . ( @ % ;
5/1      PLS-00103: Encountered the symbol "VALUES" when
         expecting one of the following:
         . ( , % from
         The symbol "VALUES" was ignored.
5/23     PLS-00103: Encountered the symbol ";" when
         expecting one of the following:
         . ( , % from
```

→ Les erreurs lors de la compilation de la procedure LOG_EXECUTION sont affichées grace à la commande SHOW ERRORS.

L'utilisation de la commande SHOW ERRORS sans préciser d'arguments permet d'afficher les erreurs de compilation du dernier objet compilé.

10.10 Informations sur le débuggage

10.10.1 Débugger en utilisant le package DBMS_OUTPUT

Il est possible d'utiliser les procédures du package inclus au serveur Oracle DBMS_OUTPUT pour afficher des valeurs et des messages depuis un bloc PL/SQL. Cette affichage est fait par accumulation de données dans un buffer puis en autorisant le renvoie des données du buffer vers le terminal d'affichage. Pour faire appel aux procédures de ce package on les préfixe toujours du nom du package DBMS_OUTPUT.
Ce package permet aux développeurs de suivre très précisément le processus d'exécution d'une fonction ou d'une procédure en envoyant des messages et des valeurs. Dans SQL*Plus il est préférable d'utiliser SET SERVEROUTPUT ON ou OFF au lieu des procédures ENABLE ou DISABLE.
Il est conseillé d'effectuer certaines actions pour faciliter le debuggage :

-afficher un message lorsqu'une procédure commence, qu'elle s'arrête ou un message indiquant qu'une opération s'est terminée avec succès.

-afficher les valeurs d'un compteur de boucle

-afficher les valeurs d'une variable avant et après l'initialisation
Le buffer n'est pas vidé tant que le bloc ne s'est pas terminé correctement.

10.10.2 Débugger les sous programmes en utilisant procedure builder

On peut également exécuter des actions de débuggage sur des sous-programmes clients ou serveur dans Procedure Builder. Pour ce faire on doit charger le sous-programme :

1. Depuis l'explorateur d'objet on choisit *Program → PL/SQL Interpreter* dans le menu
2. Dans le menu on sélectionne *View → Navigator Pane*
3. Depuis le panneau de navigation, on développe le nœud *Program Units* ou *Database objects*
4. On clique ensuite sur le sous-programme que l'on veut débugger.

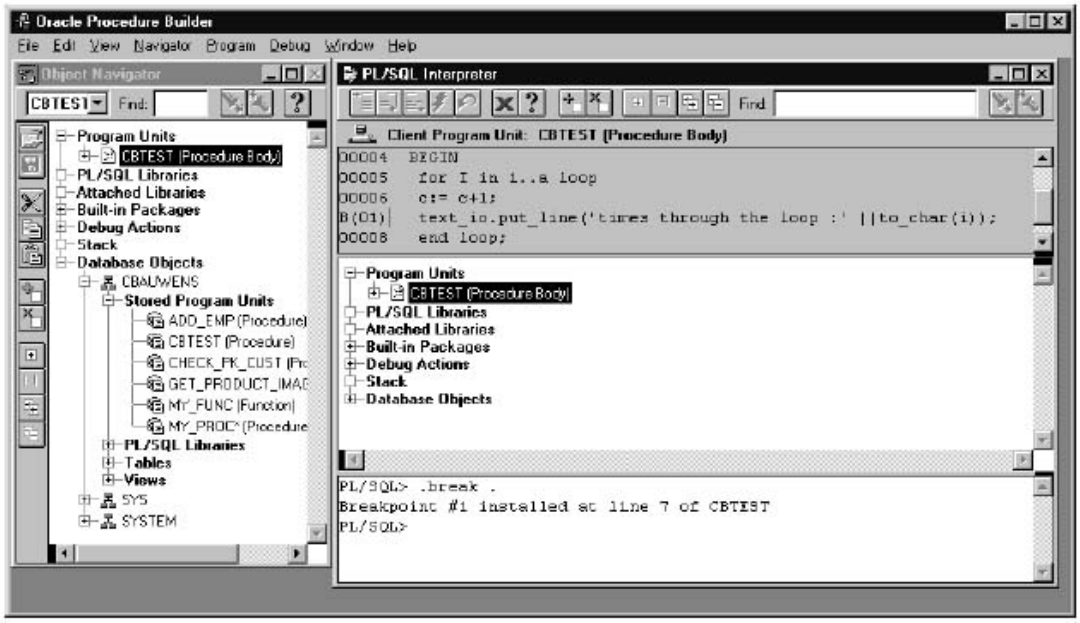

Figure n°1 : Aperçu de la fenêtre de debuggage dans Procedure Builder

10.10.3 Créer des breakpoints

Dans Procedure Builder il est possible de créer des breakpoints dans un programme pour interrompre l'exécution d'un programme à un endroit précis du code source. Quand le moteur PL/SQL rencontre un breakpoint dans un programme, il suspend l'exécution à la ligne précédant le breakpoint et donne le contrôle à l'interpréteur PL/SQL de Procedure Builder.
Pour créer un breakpoint dans un programme il faut simplement double cliquer sur la ligne à laquelle on veut le mettre en place.
Dans la figure précédente on peut voir que la ligne 00007 a été définie en tant que breakpoint. Les lignes où sont situées les breakpoints sont appelées B(XX), où XX représente le numéro du breakpoint dans le programme en commençant par 01.
Les breakpoints ne peuvent être créés que sur les lignes contenant une expression exécutable, il est impossible de placer un breakpoint sur une ligne de déclaration de variable dans la section DECLARE, à moins que cette variable ne soit en même temps initialisée.

10.10.4 Utiliser les niveaux de débuggages

Pour démarrer le débuggeur, on exécute le programme que l'on débugger.
Lorsque qu'une action de débuggage interrompt l'exécution du programme, l'interpréteur reprend le contrôle et établit ce que l'on appelle des niveaux de débuggage. A un niveau de débuggage il est possible d'entrer des commandes et des ordres PL/SQL pour inspecter et modifier l'état du programme interrompu.

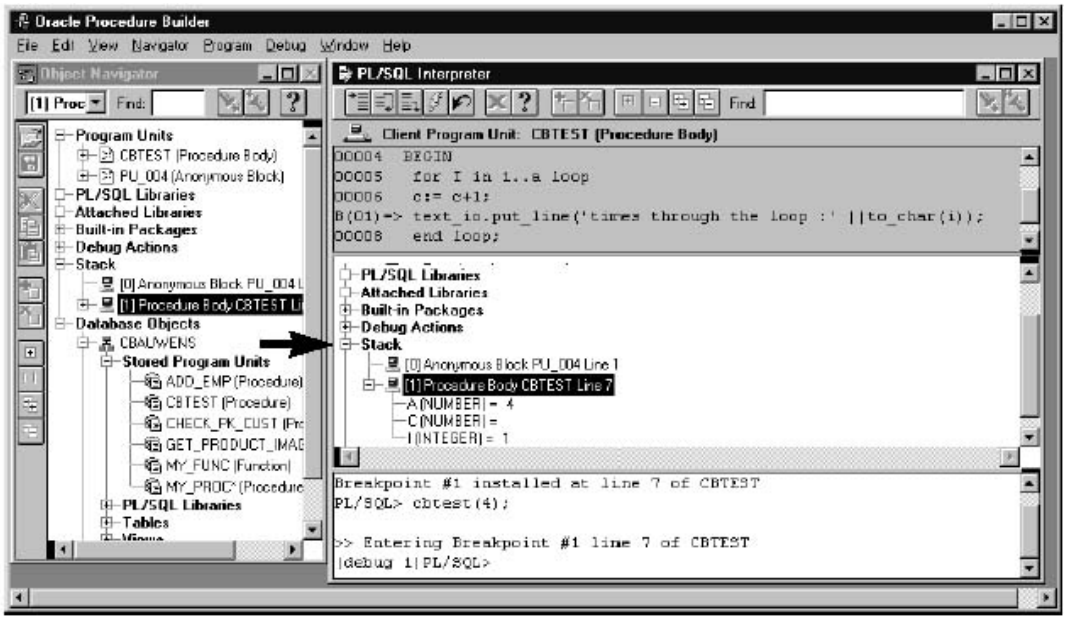

Figure n°2 : Etat des variables au breakpoint.

Dans la figure précédente, le nœud Stack est développé pour voir les valeurs des variables utilisées dans le programme au moment du breakpoint.

10.10.5 Contrôler l'exécution des programmes

Dans l'interpréteur de Procedure Builder il existe des boutons permettant de contrôler l'exécution du débuggeur. Ces boutons sont disponibles lorsque l'on crée des breakpoints dans un programme. Lorsque l'on exécute le programme, il s'arrête au premier breakpoint rencontré et l'on peut alors utiliser le navigateur d'objet pour visualiser les valeurs des variables. On peut alors choisir l'action à effectuer grâce à ces boutons. On peut soit continuer jusqu'au breakpoint suivant, soit continuer l'exécution sans prendre en compte les breakpoints suivants.

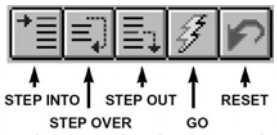

Figure n°3 : Aperçu des boutons de contrôle.

On utilise les boutons STEP INTO/OVER/OUT pour gérer l'exécution d'un programme arrêté à un breakpoint. STEP INTO exécute le programme en s'arrêtant à tous les breakpoints, STEP OVER exécute le programme sans se soucier des breakpoints et STEP OUT exécute le programme jusqu'à la fin lorsqu'il est arrêter à un breakpoint. Une fois l'opération exécutée, le contrôle est rendu à l'interpréteur.
Le bouton GO sert à exécuter le programme jusqu'à ce qu'il se termine correctement ou qu'il rencontre un breakpoint.

Le bouton RESET permet de rendre le contrôle à un niveau supérieur de debug sans continuer jusqu'au prochain breakpoint.

11 GERER LES DEPENDANCES

11.1 Objets dépendants et référencés

11.1.1 Comprendre les dépendances

Certains objets font référence à d'autres objets dans leurs définitions et ces objets sont dits dépendants. Par exemple une procédure stockée peut contenir un ordre SELECT retournant une colonne d'une table. Par conséquent une procédure stockée est appelée un objet dépendant alors que la table à laquelle elle fait référence est appelée un objet référencé.
Parmi les objets dépendants on trouve les vues, les procédures, les fonctions, les spécifications de package, les corps de package et les triggers de base de données.
Les objets référencés peuvent être une table, une vue, une séquence, un synonyme, une procédure, une fonction ou une spécification de package.
Si l'on modifie la définition d'un objet référencé, les objets dépendants peuvent ne pas continuer à fonctionner correctement. Par exemple si la définition de la table est modifiée il est possible que la procédure dépendante ne puisse plus fonctionner correctement.
Le serveur Oracle enregistre automatiquement les dépendances entre les objets. Pour gérer les dépendances, tous les objets de schéma on un *Statut,* qui peut être Valid ou Invalid, enregistré dans le dictionnaire de données. Ce statut peut être visualiser en utilisant la vue USER_OBJECTS du dictionnaire de données.
Lorsque le statut d'un objet apparaît comme valide, cela signifie qu'il a été compilé et qu'il est utilisable immédiatement. Si son statut apparaît comme invalide, l'objet doit être compilé avant d'être utilisé.

11.1.2 Les dépendances directes et indirectes

Les dépendances directes sont des dépendances pour lesquelles l'objet dépendant fait directement référence à l'objet référencé, par exemple dans le cas d'une procédure dépendante d'une table.
Les dépendances indirectes sont des dépendances pour lesquelles l'objet dépendant fait référence à un objet par le biais d'un autre objet. Par exemple dans la cas d'une procédure dépendante d'une vue et qui par conséquent est dépendante de la table sous-jacente.

11.1.3 Les dépendances locales

Les dépendances locales sont des dépendances pour lesquelles l'objet dépendant et l'objet référencé se trouvent sur le même nœud de la même base de données. Ces dépendances sont gérées automatiquement par le serveur Oracle en utilisant la table interne à la base de données « depends-on ».
Dans le cas des dépendances locales, le serveur Oracle recompile implicitement tout objet dont le statut est invalide lors de son prochain appel. Donc si un objet référencé vient à changer impliquant l'invalidation de l'objet dépendant, la recompilation sera automatique.

11.1.4 Les dépendances distantes

Les dépendances distantes sont des dépendances pour lesquelles l'objet dépendant et l'objet référencé sont situés sur des nœuds séparés. Le serveur Oracle ne gèrent pas les dépendances parmi les objets de schéma distants a part si il s'agit de dépendances entre une procédure locale et une procédure distante.
Si l'objet référencé est amené à changer, les procédures locales stockées et tous leurs objets dépendants seront invalidés mais ne seront pas recompilés automatiquement lors du prochain appel.

11.1.5 Un scénario de dépendances locales

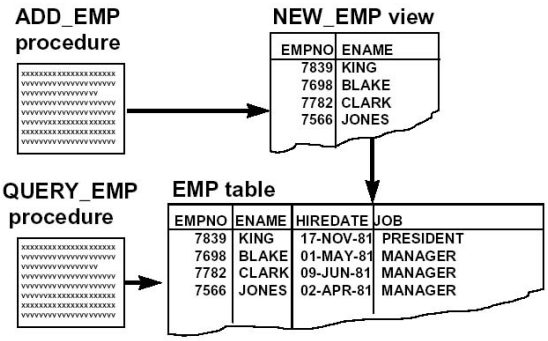

Figure n 1 : Schéma des dépendances locales

La procédure QUERY_EMP fait directement référence à la table EMP, alors que la procédure ADD_EMP met à jour la table EMP indirectement par le biais de la vue NEW_EMP.
Selon les modifications effectuées, la procédure ADD_EMP pourra ou non être invalidée.
Les modifications sur des procédures n'étant pas en rapport avec ADD_EMP n'auront pas de conséquences sur la validité de celle-ci.
Si l'on ajoute une colonne dans la table EMP la procédure ADD_EMP sera invalidée et sera recompilée avec succès à condition qu'une liste de colonnes soit donnée dans l'ordre INSERT et que la colonne ajoutée n'a pas une contrainte NOT NULL.

11.2 Visualiser les dépendances

11.2.1 Afficher les dépendances directes avec USER_DEPENDENCIES

Pour déterminer quels objets de la base de données il est nécessaire de recompiler manuellement on affiche les dépendances directes de la vue USER_DEPENDENCIES du dictionnaire de données. Cette vue est composée des colonnes suivantes :

Colonne	Description de la colonne
NAME	Nom de l'objet dépendant
TYPE	Type de l'objet dépendant (PROCEDURE, FUNCTION, PACKAGE, PACKAGE_BODY, TRIGGER ou VIEW)
REFERENCED_OWNER	Schéma où ce situe l'objet référencé
REFERENCED_NAME	Nom de l'objet référencé
REFERENCED_TYPE	Type de l'objet référencé
REFERENCED_LINK_NAME	Lien de la base de données utilisé pour accéder à l'objet référencé

On peut également visualiser les dépendances directes en examinant les vues du dictionnaire de données ALL_DEPENDENCIES et DBA_DEPENDENCIES qui contiennent en plus le nom du propriétaire dans la colonne OWNER.
Exemple :

```
SQL> SELECT  name, type, referenced_name, referenced_type
  2  FROM    user_dependencies
  3  WHERE   referenced_name IN ('EMP' , 'NEW_EMP' );

NAME       TYPE       REFERENCED_NAME   REFERENCED_TYPE
---------- ---------- ----------------- ---------------
QUERY_EMP  PROCEDURE  EMP               TABLE
ADD_EMP    PROCEDURE  NEW_EMP           VIEW
NEW_EMP    VIEW       EMP               TABLE
```

11.2.2 Afficher les dépendances directes et indirectes

Pour afficher les dépendances directes et indirectes on va exécuter un script créant les objets nous permettant de visualiser les dépendances. Ce script crée une table DEPTREE_TEMPTAB qui va contenir des informations sur un objet référencé spécifique. Cette table sera remplit grâce à la procédure DEPTREE_FILL. Cette procédure accepte trois paramètres : *object_type,* qui est le type de l'objet référencé, *object_owner,* qui est le schéma de l'objet référencé et *object_name,* qui est le nom de l'objet référencé.
Les informations concernant les dépendances seront affichées en utilisant les vues utilisateurs supplémentaires DEPTREE et IDEPTRE.

Exemple :

```
SQL> @UTDLTREE

SQL> EXECUTE deptree_fill ('TABLE', 'SCOTT', 'EMP')
Procédure PL/SQL terminée avec success.
```

→ On exécute le script UTLDTREE, puis on remplit la table DEPTREE_TEMPTAB grace au script DEPTREE_FILL en utilisant les informations de la table EMP de SCOTT.

11.2.3 Afficher les dépendances

On affiche une représentation des dépendances en effectuant une requête sur la vue DEPTREE.

Exemple :

```
SQL> SELECT nested_level, type, name
2 FROM deptree
3 ORDER BY seq#;

NESTED_LEVEL TYPE      NAME
------------ --------- ----
           0 TABLE     EMP
           1 VIEW      NEW_EMP
           2 PROCEDURE ADD_EMP
           1 PROCEDURE QUERY_EMP
```

On peut également afficher une représentation indentée des mêmes informations en effectuant une requête sur la vue IDEPTREE, contenant une colonne unique, DEPENDENCIES.

Exemple :

```
SQL> SELECT *
  2  FROM ideptree;

DEPENDENCIES
--------------------------------------------------
TABLE SCOTT.EMP
    VIEW SCOTT.NEW_EMP
         PROCEDURE SCOTT.ADD_EMP
    PROCEDURE SCOTT.QUERY_EMP
```

11.3 Un scénario de dépendances de nom locales

Il faut faire attention aux cas où la création d'une table, d'une vue ou d'un synonyme entraîne l'invalidation d'un objet dépendant parce qu'il interfère avec la hiérarchie du serveur Oracle pour résoudre les références aux noms.
Il faut toujours prévoir l'effet qu'un nouveau nom d'objet aura sur une procédure dépendante.
Par exemple supposons que la procédure QUERY_EMP fasse référence au synonyme public EMP. Cependant on vient juste de créer une nouvelle table nommée EMP dans notre propre schéma. Si la table que l'on a créer

est une copie de la vue (même colonnes), la procédure sera recompilée correctement et sera par conséquent toujours valide. En revanche si la table comporte des colonnes différentes, la recompilation de QUERY_EMP produira des erreurs et par conséquent restera INVALID.
Maintenant si l'on supprime la table EMP, les objets dépendants deviennent invalides. Si l'on a accès à un objet public nommé EMP par le biais d'un synonyme public et que cet objet possède la même structure, l'objet recompilé fera dorénavant référence à l'objet EMP public.
On peut surveiller les dépendances de sécurité dans la vue USER_TAB_PRIVS du dictionnaire de données.

11.4 Dépendances distantes

11.4.1 Recompilation des dépendances

Lorsqu'un objet référé vient à changer, l'objet dépendant peut devenir invalide. Si l'invalidité concerne une dépendance locale, l'objet est recompilé implicitement. En revanche lorsqu'il s'agit d'une dépendance distante la recompilation ne se fait pas de manière automatique, il faut les recompiler explicitement.
On vérifie le succès de la recompilation explicite des procédures dépendantes distantes et la recompilation implicite des procédures dépendantes locales en vérifiant le statut de ces procédures dans la vue USER_OBJECTS.
Si une recompilation automatique d'une procédure locale dépendante échoue, le statut de celle-ci reste INVALID et le serveur Oracle produit une erreur d'exécution. Donc, pour éviter d'interrompre le fonctionnement de la procédure, il est fortement recommandé de recompiler les objets dépendants locaux manuellement plutôt que de se reposer sur la compilation automatique.

11.4.2 Concept de dépendances distantes

Le comportement des dépendances est dirigé par le mode choisit par l'utilisateur : la vérification de TIMESTAMP (ou marqueur de temps) ou la vérification de SIGNATURE.
Vérification de TIMESTAMP :
Lorsqu'une procédure est compilée ou recompilée, un marqueur de temps lui est attribué et est enregistré dans le dictionnaire de données. Le marqueur de temps est un enregistrement de l'heure à laquelle la procédure a été créée, modifiée ou remplacée. En plus la version compilée de la procédure contient des informations sur chaque procédure distante à laquelle elle fait référence, et notamment le schéma de la procédure distante, le nom du package, le nom de la procédure et le marqueur de temps.
Lorsqu'une procédure dépendante est utilisée, Oracle compare les marqueurs de temps distant enregistré lors de la compilation avec les marqueurs de temps courant des procédures distantes référencées. En fonction du résultat de cette comparaison, deux situations peuvent se produire :

Si les marqueurs de temps concordent, les procédures locales et distantes s'exécutent sans compilation.

Si un marqueur de temps des procédures référencées distantes ne concorde pas, la procédure locale est invalidée et une erreur est retournée à l'environnement appelant. De plus, toutes les autres procédures locales qui dépendent de la procédure distante avec le nouveau marqueur de temps sont invalidées. Par exemple, si plusieurs procédures locales appellent une procédure distante et que la procédure distante a été recompilée, lorsqu'une procédure locale s'exécute et remarque que le marqueur de temps a changé, toutes les procédures locales dépendant de la procédure distante sont invalidées.
Les marqueurs de temps sont comparés uniquement lorsqu'un ordre du corps d'une procédure locale exécute une procédure distante.
Vérification de SIGNATURE :
Le serveur Oracle fournit la possibilité de vérifier les dépendances distantes en utilisant les signatures. Les procédures locales ne sont pas affectées par la vérification de signature puisque dans ce cas la recompilation se fait toujours automatiquement.
La signature d'une procédure contient le nom du package, de la procédure ou de la fonction, les types de paramètres de base et le mode des paramètres (IN, OUT, IN OUT). Seuls le type et le mode des paramètres sont importants, le nom du paramètre n'a aucune incidence sur la signature.
Le marqueur de temps enregistré dans le programme appelant est comparé avec le marqueur de temps actuel du programme distant appelé. Si les marqueurs correspondent, l'appel s'effectue normalement. S'ils ne concordent pas, la couche Remote Procedure Calls (RPC) effectue un test simple pour comparer la signature pour déterminer si l'appel est sécurisé ou non. Si la signature n'a pas été changée d'une manière incompatible l'exécution continue, sinon un statut d'erreur est renvoyé.

11.4.3 Scénario de dépendance distante en mode Timestamp

11.4.3.1 Recompilation automatique

Lorsqu'un objet distant a été modifié, il est fortement conseillé de le recompiler manuellement les objets locaux dépendants plutôt que de se reposer sur le mécanisme automatique de dépendances distantes afin d'éviter les perturbations de production.
Le mécanisme de dépendance distante est différent du mécanisme automatique des dépendances locales déjà évoqué. La première fois qu'un sous-programme distant recompilé est appelé par un sous-programme local, il se produit une erreur d'exécution et le sous-programme local est invalidé. Au second appel du sous-programme une recompilation implicite a lieu.

11.4.3.2 Exemple

Une procédure locale faisant référence à une procédure distante est invalidée par le serveur Oracle si la procédure distante est recompilée après que la procédure ait été compilée. Cela vient du fait que lors de la compilation de la procédure locale, la procédure enregistre le marqueur de temps correspondant à tous les objets dépendants dans le *p-code*. Donc si la procédure distante est recompilée, la comparaison des marqueurs de temps ne concorde pas ce qui par conséquent entraîne l'invalidation de la procédure.
Par exemple on, compile une procédure distante B, dont dépend une procédure A, à 8 heures.
Si la procédure A est recompilée à 9 heures elle sera toujours considérée comme valide par le serveur Oracle puisque le marqueur de temps enregistré dans le *p-code* de la procédure stockée correspondra au marqueur de temps de la procédure distante lors de la compilation à 8 heures.
SI l'on exécute la procédure A sans la recompiler, le marqueur de temps correspondra à une compilation de la procédure B antérieure à la dernière compilation et donc résultera en une invalidation de la procédure A puisque la comparaison de marqueur de temps ne concorde pas. Le mécanisme de recompilation des procédures locales recompilera alors la procédure A en prenant le marqueur de temps de la compilation de la procédure B à 8 heures et ainsi lors de la deuxième exécution de celle-ci la comparaison des marqueurs de temps sera concordante.

11.4.4 Le mode de signature

Pour résoudre certains problèmes posés par le modèle de dépendance de marqueur de temps on peut utiliser le modèle de signature. Cela permet à la procédure distante d'être recompilée sans que la procédure locale dépendante soit affectée.
La signature d'un sous-programme est constituée du nom du sous-programme, le type de données des paramètres, le mode des paramètres, le nombre des paramètres et le type de données de la valeur retournée par une fonction.
Si un programme distant est changé puis recompilé mais que la signature ne change pas, alors la procédure locale peut exécuter la procédure distante. Avec la méthode des marqueurs de temps la procédure locale aurait été invalidée puisque ces marqueurs ne correspondent plus.

11.5 Recompilation manuelle

11.5.1 Recompiler un programme PL/SQL

La recompilation d'un programme PL/SQL peut se faire de manière automatique par une recompilation lors de l'exécution ou bien elle peut être effectuée de manière explicite avec un ordre ALTER avec le paramètre COMPILE.

```
ALTER PROCEDURE [SCHEMA.] procedure_name COMPILE
```

```
ALTER FUNCTION [SCHEMA.] function_name COMPILE

ALTER PACKAGE [SCHEMA.] package_name COMPILE [PACKAGE]
ALTER PACKAGE [SCHEMA.] package_name COMPILE BODY

ALTER TRIGGER trigger_name [COMPILE[DEBUG]]
```

Si une recompilation est réalisée avec succès, l'objet devient valide. Sinon le serveur Oracle retourne une erreur et le statut de l'objet reste invalide.
Lorsque l'on recompile un objet PL/SQL, le serveur Oracle recompile d'abord tous les objets invalides dont il dépend.
L'option COMPILE PACKAGE recompile le corps du package ainsi que la spécification sans se soucier de son invalidité. L'option COMPILE BODY ne recompile que le corps du package. La recompilation de la spécification d'un package entraîne l'invalidité de tous les objets locaux dépendants de cette spécification, comme les procédures appelant des procédures ou fonctions du package. Le corps du package dépend également de la spécification.

11.5.2 Echec de recompilation

La recompilation de procédures et fonctions dépendantes échouera si :

- L'objet référencé est supprimé ou renommé
- Le type de donnée de la colonne référencée a changé
- La colonne référencée est supprimée
- Une vue référencée est remplacée par une vue basée sur des colonnes différentes
- La liste de paramètre d'une procédure référencée est modifiée

Le succès d'une recompilation est basé sur la dépendance exacte. Si une vue référencée est recrée, tout objet étant dépendant de la vue doit être recompilé. La réussite d'une recompilation dépend des colonnes que la vue actuelle contient et des colonnes dont l'objet référencé a besoin pour s'exécuter. Si les colonnes requises ne font pas parties de la nouvelle vue, l'objet restera invalide.

11.5.3 Recompilation réussie

La recompilation de procédures et fonctions dépendantes sera réussie si :

- Des colonnes ont été ajoutées à la table et qu'aucune des nouvelles colonne n'est définie comme NOT NULL
- Le type de données des colonnes référencées n'a pas changé
- Une table privée a été supprimée, mais une table publique possédant le même nom et la même structure existe. Le programme sera dorénavant dépendant de la table publique
- Le corps d'une procédure PL/SQL référencée a été modifié et recompilé sans erreurs

11.5.4 Recompiler les procédures

Lors de la recompilation de procédures, plusieurs facteurs peuvent entraîner l'échec de celle-ci. Pour minimiser les échecs liés aux dépendances il faut :

- Déclarer les records en utilisant l'attribut %ROWTYPE
- Déclarer les variables en utilisant l'attribut %TYPE
- Effectuer des requêtes en utilisant la notation SELECT *
- Spécifier une liste de colonnes dans les ordres INSERT

11.6 Packages et dépendances

Pour simplifier la gestion des dépendances avec les packages il est préférable de faire référence à une procédure ou fonction d'un package depuis une procédure ou fonction indépendante.
Ainsi si le corps du package est modifié alors que la spécification reste la même, la procédure indépendante faisant référence à la spécification du package restera valide.
En revanche si la spécification change, la procédure externe y faisant référence sera invalide tout comme le corps du package.
La référence à une procédure externe depuis le corps d'un package n'apporte pas d'amélioration de gestion des dépendances car le corps du package entier dépendra alors de la validité de la procédure. Il est fortement conseillé d'inclure cette procédure dans le package.
Si la procédure externe référencée dans le package change alors le corps du package sera invalidé alors que la spécification du package restera valide.

12MANIPULER LES LARGES OBJECTS

12.1 Les LOBs

12.1.1 Qu'est-ce qu'un LOB

Un LOB est un objet volumineux stocké directement dans la base de données. Il existe quatre types de LOBs (BLOB, CLOB, NCLOB, BFILE). Les BLOB sont des Large Objects de type binaire, les CLOB sont des Large Objects de type caractère, les NCLOB sont des Large Objects de type caractère de taille fixe, les BFILE sont des fichiers binaires stockés hors de la base de données.
Les LOBs sont caractérisés de deux manières : leur interprétation par le serveur Oracle (binaire ou caractère) et leur méthode de stockage. Les LOBs peuvent être stockés dans la base de données ou dans des fichiers hôtes. Il existe deux catégories de LOBs :

Les LOBs internes (CLOB, NCLOB, BLOB) sont stockés directement dans la base de donnée
Les LOBs externes (BFILE) sont stockés hors de la base de données

Le serveur Oracle ne convertira pas les données entre les types. Par exemple un utilisateur crée une table X avec une colonne CLOB et une table Y avec une colonne BLOB, les données ne seront pas directement transférables entre les deux colonnes.
Les BFILES sont accessibles en lecture seule depuis le serveur Oracle.

12.1.2 Opposition entre les types de données LONG et LOB

Les types de données LONG et LONG RAW étaient utilisés précédemment pour les données non structurées telles que les images binaires, les documents ou les informations géographiques. Ces types de données sont maintenant remplacés par les types de données LOB. Ces types de données sont différents des LONG et LONG RAW car les LOBs ne sont pas interchangeables. Les LOBs ne sont pas supportés par l'interface de programmation d'application LONG et vice versa.
Voici une liste des principales différences entre les LONG et les LOB :

LONG, LONG RAW	LOB
Une seule colonne par table	Plusieurs colonnes par table
Jusqu'à 2 Giga	Jusqu'à 4 Giga
Un SELECT retourne les données	Un SELECT retourne le localisateur
Les données sont stockées dans la base	Les données sont stockées dans la base ou en dehors
Accès séquentiel aux données	Accès aléatoire aux données

12.1.3 Anatomie d'un LOB

Un LOB est composé de deux parties biens distinctes : le localisateur et la valeur. Le localisateur est un indicateur de l'endroit où la valeur du Lob est située dans la base de donnée. La valeur représente les données constituant la vraie valeur de l'objet stocké.
De plus, un programme accédant et manipulant des LOBs nécessite la déclaration d'un pointeur ou d'un localisateur de LOB.
Lorsque qu'un utilisateur créé un LOB interne, la valeur est stockée dans le segment LOB et le localisateur de la valeur hors-ligne du LOB est placé dans la colonne de la ligne correspondante de la table.Les LOBs externe stockent les données hors de la base de données donc la table contient juste le localisateur de la valeur du LOB.
Pour un LOB de type interne, la valeur du LOB est directement stockée en ligne avec les autres lignes si la taille du LOB est inférieure à 4000 bits. Quand la valeur du LOB est supérieure à 4000, elle est automatiquement déplacée hors de la ligne.
Lorsque l'on crée une table contenant une colonne LOB, le stockage par défaut sera ENABLE STORAGE IN ROW. Si l'on ne veut pas que les valeurs des LOBs soient stockées dans les lignes, même si leur taille est inférieure à 4000 bits on spécifie l'option DISABLE STORAGE IN ROW dans le clause de stockage.

12.2 LOBs internes et externes

12.2.1 Les LOBs internes

Les LOBs internes sont des LOBs dont la valeur est stockée dans le serveur Oracle. Les LOBs internes sont les BLOB, les CLOB et les NCLOB. Ceux-ci peuvent apparaître en tant qu'attribut d'un type définit par l'utilisateur, en tant que colonne d'une table, en tant qu'une variable de substitution ou hôte ou en tant que résultat, paramètre ou variable PL/SQL.
Les LOBs de type internes peuvent tirer avantage des fonctionnalités du serveur Oracle telles que les mécanismes de coopération et de restauration.
Le type de donnée BLOB est interprété par le serveur Oracle comme un flot de bits, similaire à un type de données LONG RAW.
Le type CLOB est interprété comme un flot de caractères à bit unique.
Le type NCLOB est interprété comme un flot de caractères à bit multiple et à taille fixée, basé sur la taille de bit définit par le caractère national de la base de données.

12.2.2 Gérer les LOBs internes

Pour interagir pleinement avec les LOBs, des interfaces sont fournies dans le package PL/SQL DBMS_LOB et dans Oracle Call Interface.
Le serveur Oracle fournit également un support pour la gestion de LOB grâce au SQL.
La méthode générale pour gérer les LOB interne est la suivante :

- On crée et on remplit une table contenant le type de données LOB
- On déclare et on initialise le localisateur de LOB dans le programme
- On utilise SELECT FOR UPDATE pour verrouiller la ligne contenant le LOB
- On manipule le LOB en utilisant les procédures du package DBMS_LOB ou des appel OCI en utilisant le localisateur LOB en tant que référence à la valeur du LOB
- On utilise l'ordre COMMIT pour valider les changements

12.2.3 Les LOBs externes

Les LOBs externes sont des LOBs dont la valeur est stockée en dehors de la base de données, la base de données contiendra uniquement le localisateur du fichier sous forme d'un objet DIRECTORY contenant le chemin d'accès au répertoire. Le type de données BFILE est fournit par le serveur Oracle afin de donner un accès à des fichiers externes aux utilisateurs de la base de données.
Le SQL Oracle permet de définir des objets BFILE, d'associer des objets BFILE avec les fichiers externes correspondant, d'accéder à la sécurité des BFILE.
Un objet de type DIRECTORY spécifie un alias pour un répertoire situé sur le serveur. En donnant le privilège READ sur cet objet, on peut garantir un accès sécurisé, en fonction de l'utilisateur, aux fichiers du répertoire (certains répertoires peuvent ainsi être en lecture seule ou inaccessible).

12.2.4 Définir les BFILEs

Pour définir les BFILEs il faut tout d'abord créer un répertoire dans le système d'exploitation en tant qu'utilisateur Oracle en définissant les permissions de sorte que Oracle puisse lire le contenu du répertoire. On place ensuite les fichiers souhaités dans le répertoire. Sur le serveur Oracle on crée une table contenant le type de données BFILE et on crée également un objet DIRECTORY auquel on donne le privilège READ. On insère ensuite les lignes dans la table en utilisant la fonction BFILENAME en associant les fichiers du système d'exploitation avec le champ correspondant. Puis on déclare et initialise le localisateur LOB dans le programme, le localisateur sera initialisé avec la ligne et la colonne contenant le BFILE. On peut ensuite lire le BFILE avec OCI ou la fonction DBMS_LOB en utilisant le localisateur comme référence au fichier.

12.3 Le package DBMS_LOB

12.3.1 Utilisation de DBMS_LOB

Pour manipuler les LOBs, on utilise le package fournit par Oracle DBMS_LOB. Ce package fournit des routines pour accéder et manipuler les LOBs internes et externes.
Pour pouvoir charger le package DBMS_LOB, le DBA doit se connecter en tant que SYS et exécuter les scripts *dbmslob.sql* et *prvtlob.plb* ou exécuter le script *catproc.sql.* Les utilisateurs peuvent ensuite se voir accorder des privilèges pour utiliser le package.
Les routines DBMS_LOB ne verrouillent pas implicitement les lignes contenant le LOB, donc l'utilisateur doit verrouiller lui-même les lignes contenant le LOB interne avant d'appeler un sous-programme nécessitant une écriture dans la valeur du LOB.

12.3.2 Fonctions et procédures de DBMS_LOB

Les fonctions et procédures incluses dans le package DBMS_LOB peuvent être classées dans deux catégories : les modificateurs et les observateurs. Les modificateurs peuvent modifier les valeurs du LOB alors que les observateurs n'accèdent aux LOBs que en lecture.

Type	Nom	Description
Modificateur	APPEND	Ajoute le contenu du LOB source au LOB de destination
Modificateur	COPY	Copie tout ou une partie du Lob source dans le LOB de destination
Modificateur	ERASE	Efface tout ou une partie du LOB
Observateur	LOADFROMFILE	Charge les données d'une BFILE dans un LOB interne
Modificateur	TRIM	Réduit la valeur LOB à une taille spécifiée
Modificateur	WRITE	Ecrit des données dans les LOB à un endroit spécifique
Observateur	GETLENGTH	Récupère la longueur d'une valeur d'un LOB
Observateur	INSTR	Renvoie la position dans le LOB de la *n*ième occurrence du modèle
Observateur	READ	Lit les données dans un LOB, à partir de la position spécifiée
Observateur	SUBSTR	Renvoie une partie du LOB en commençant à la position spécifiée
Modificateur	FILECLOSE	Ferme le fichier
Modificateur	FILECLOSEALL	Ferme tous les fichiers ouverts précédemment
Observateur	FILEEXISTS	Vérifie si le fichier existe sur le serveur
Observateur	FILEGETNAME	Récupère l'alias du répertoire et le nom du fichier
Observateur	FILEISOPEN	Vérifie si un fichier a été ouvert en utilisant le localisateur BFILE
Modificateur	FILEOPEN	Ouvre un fichier

Les fonctions FILECLOSE, FILECLOSEALL, FILEEXISTS, FILEGETNAME, FILEISOPEN et FILEOPEN sont spécifiques aux BFILEs.
Toutes les fonctions du package DBMS_LOB retourne une valeur NULL si un des paramètres passés est NULL. Toutes les procédures modificatrices du package lèvent une erreur si la destination pour le LOB est spécifiée comme NULL.
Les positions utilisées dans les fonctions WRITE, READ, INSTR et SUBSTR doivent toujours être positives. Elles représentent le nombre de bits/caractères à partir du début du LOB où l'opération va s'effectuer. Par défaut la valeur est 1, ce qui signifie que toute opération ne spécifiant pas de valeur débutera au début du LOB. Pour les BLOB et les BFILE, la position est mesurée en bits, alors que pour les CLOB et les NCLOB la mesure s'effectue en caractères.

12.3.3 DBMS_LOB READ et WRITE

DBMS_LOB.READ
La procédure READ est utilisée pour lire et retourner tout ou partie (en fonction du paramètre AMOUNT) d'un LOB en commençant à la position spécifiée.

```
PROCEDURE READ (
  lobsrc IN BFILE|BLOB|CLOB ,
  amount IN OUT BINARY_INTEGER,
  offset IN INTEGER,
  buffer OUT RAW|VARCHAR2 )
```

Si la fin du LOB est atteinte avant que le nombre de bits/caractères spécifié ait été lue, la valeur retournée par AMOUNT sera inférieure à celle spécifiée.
Le PL/SQL supporte une valeur maximale de 3267 pour les RAW et VARCHAR2. Il faut s'assurer que l'on a alloué suffisamment de ressources systèmes pour supporter ces tailles de buffers par rapport au nombre de sessions utilisateurs, sinon le serveur Oracle renverra des erreurs de mémoire.
Les BLOB et les BFILES retournent des RAW, les autres retournent VARCHAR2.

DBMS_LOB.WRITE

La procédure WRITE est utilisée pour écrire tout ou partie (en fonction du paramètre AMOUNT) de données dans un LOB depuis un BUFFER définit par l'utilisateur en commençant à la position spécifiée ou depuis le début du LOB.

```
PROCEDURE WRITE (
  lobdst IN OUT BLOB|CLOB,
  amount IN OUT BINARY_INTEGER,
  offset IN INTEGER := 1,
  buffer IN RAW|VARCHAR2 ) -- RAW for BLOB
```

On doit s'assurer que la taille en bits correspond à la taille des données dans le buffer. WRITE n'a aucun moyen de vérifier si ces tailles correspondent et écrira un nombre de bit équivalent à la valeur de AMOUNT du buffer dans le LOB.

12.3.4 Exemple de création de table avec des LOBs

Les colonnes LOB sont définies au moyen d'ordre SQL de définition de données (DDL) tel que CREATE TABLE. Les contenu d'un colonne LOB est stocké dans le segment LOB de la base de données alors que la colonne de la table contient simplement une référence à cette zone de stockage spécifique, cette référence est appelée localisateur. Il est possible, en PL/SQL, de définir des variables de type LOB qui contiendront, comme pour les tables, seulement la valeur du localisateur du LOB.

```
SQL> CREATE TABLE employee
  2  (emp_id     NUMBER,
  3   emp_name   VARCHAR2(35),
  4   resume     CLOB,
  5   picture    BLOB);
```

12.4 Manipuler des LOBs en SQL

12.4.1 Insertion en utilisant SQL

Il est possible d'insérer directement une valeur dans une colonne LOB en utilisant des variables en SQL, PL/SQL, 3GL-SQL ou OCI

```
INSERT INTO employee(emp_id, emp_name,
   resume,                          picture)
VALUES(             7899,   'James Dean',
   'Date of Birth = 8 February 1931',NULL);
```

Il est également possible d'initialiser un LOB mais sans lui assigner de donnée en utilisant la fonction EMPTY_CLOB() ou EMPTY_BLOB(). Pour ensuite entrer une valeur dans ce LOB on pourra utiliser un ordre UPDATE.

```
INSERT INTO employee
      (emp_id, emp_name,         resume,      picture)
VALUES(7898,   'Marilyn Monroe', EMPTY_CLOB(),NULL);
```

Si l'on utilise NULL comme valeur pour une colonne LOB, le Lob n'est pas initialisé. Par conséquent, il ne pourra pas être peuplé en utilisant un ordre UPDATE. Il faudra utiliser un ordre INSERT pour insérer une nouvelle ligne à la colonne LOB et y assigner une valeur en même temps.
Lorsque l'on créer une instance LOB, le serveur Oracle crée et place un localisateur vers la valeur hors-ligne du LOB dans la colonne LOB. SQL, OCI et d'autres interfaces de programmation agissent sur les LOBs en utilisant les localisateurs.
La fonction EMPTY_C/B/NCLOB() peut être utiliser en tant que contrainte DEFAULT d'une colonne. Ceci permet d'initialiser la colonne avec des localisateurs.

Exemple :

```
CREATE TABLE employee
 (emp_id     NUMBER,
  emp_name   VARCHAR2(35),
  resume     CLOB  default  EMPTY_CLOB(),
  picture    BLOB  default  EMPTY_BLOB());
```

→ Cet ordre crée une table avec deux colonnes LOBs qui ont une valeur par défaut définie par la fonction EMPTY_CLOB/BLOB.

12.4.2 Mise à jour des LOBs en utilisant SQL

On peut mettre à jour une colonne d'un LOB en l'initialisant à autre valeur LOB, à NULL ou en utilisant la fonction EMPTY_CLOB ou EMPTY_BLOB. On peut mettre à jour le Lob en utilisant une variable de substitution en SQL qui peut être NULL, vide ou peuplée.

Exemple :

```
UPDATE employee SET
   resume = 'Date of Birth = 1 June 1926'
WHERE  emp_id = 7898;
```

→ Cet ordre modifie la valeur de resume dans la table EMPLOYEE

Lorsque l'on initialise un LOB à une valeur égale à un autre, une nouvelle copie du LOB est créée. Ces actions ne requièrent pas un ordre SELECT FOR UPDATE, on doit verrouiller une ligne avant de faire un UPDATE uniquement lorsque l'on mets à jour une partie d'un LOB.

12.4.3 Mise à jour en utilisant DBMS_LOB

Pour mettre à jour un LOB on peut également utiliser les fonctions WRITE et WRITEAPPEND du package DBMS_LOB.

Exemple :

```
DECLARE
  lobloc CLOB;          -- servira de localisateur LOB
  text   VARCHAR2(32767) := 'Died = 5 August 1962';
  amount NUMBER ;       -- taille à écrire
  offset INTEGER;       -- où commencer l'écriture
BEGIN
  SELECT resume   INTO   lobloc
  FROM   employee WHERE  emp_id = 7898 FOR UPDATE;
  offset := DBMS_LOB.GETLENGTH(lobloc) + 2;
  amount := length(text);
  DBMS_LOB.WRITE(lobloc, amount, offset, text );
  text   := ' Died = 30 September 1955';
  SELECT resume   INTO   lobloc
  FROM   employee WHERE  emp_id = 7899 FOR UPDATE;
  amount := length(text);
  DBMS_LOB.WRITEAPPEND(lobloc, amount, text );
  COMMIT;
END;
```

Dans l'exemple ci-dessus, la variable LOBLOC est utilisée en tant que localisateur et la variable AMOUNT correspond à la taille du texte à ajouter. L'ordre SELECT FOR UPDATE verrouille la ligne et renvoie le localisateur du LOB pour la colonne LOB RESUME. Enfin la procédure WRITE du package est appelée pour écrire le texte dans la valeur du LOB à l'endroit spécifié. WRITEAPPEND ajoute le texte à la valeur du LOB. Pour écrire dans les LOB en utilisant DBMS_LOB.WRITE, il faut que le LOB soit initialisé. Si l'on essaye d'écrire une valeur dans un LOB non-initialisé, le serveur Oracle renverra une erreur.

Exemple :

```
INSERT INTO employee (emp_id, emp_name, resume)
VALUES               (7900, 'JOHN WAYNE',NULL);

SQL> DECLARE         -- assignation de valeur avec du PL/SQL
  3  text VARCHAR2(32767) := 'Date of Birth = 26 May 1907 ';
  9  SELECT . . . WHERE emp_id = 7900 FOR UPDATE;
 15  DBMS_LOB.WRITE(lobloc, amount, offset, text );
 17  END;

DECLARE      -- assignation de valeur avec du PL/SQL
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at "SYS.DBMS_LOB", line 708
ORA-06512: at line 15
```

→ On insert une ligne avec un CLOB non initialisée puis on exécute un script pour écrire mais une erreur est renvoyée car le LOB n'est pas initialisé.

12.4.4 Sélectionner les valeurs CLOB

Il est possible de voir les données d'une colonne CLOB en utilisant un ordre SELECT, mais on ne peut pas visualiser les données d'une colonne BLOB ou BFILE dans un ordre SELECT sous SQL*Plus. Pour cela il faut utiliser un outil pouvant afficher des informations binaires pour un BLOB et un logiciel approprié pour un BFILE.

Exemple :

```
SQL> SELECT emp_id, emp_name , resume -- CLOB
  2  FROM employee;
EMP_ID     EMP_NAME
---------- ---------------
RESUME
------------------------------------------------------------
      7899 James Dean
Date of Birth = 8 February 1931 Died = 30 September 1955

      7898 Marilyn Monroe
Date of Birth = 1 June 1926 Died = 5 August 1962
```

→ Cette requête affiche le contenu de la table EMPLOYEE et notamment la colonne de type CLOB.

On peut également n'afficher qu'une partie du LOB grâce à la fonction SUBSTR du package DBMS_LOB, son utilisation étant similaire à la fonction SUBSTR en SQL.
On peut aussi afficher la position d'un caractère dans un LOB grâce à la fonction DBMS_LOB.INSTR. Cette fonction est utile pour faire des recherches de caractères dans un LOB.

Exemple :

```
SQL> SELECT  DBMS_LOB.SUBSTR(resume,5,19),
  2          DBMS_LOB.INSTR(resume,'=')
  3  FROM    employee;

DBMS_LOB.SUBSTR(RESUME,5,19) DBMS_LOB.INSTR(RESUME,'=')
---------------------------- --------------------------
Febru                                                 15
June                                                  15
```

→ Cette requête ne garde que 5 caractères du CLOB et cherche la position dans le CLOB du caractère =.

Ces deux fonctions du package DBMS_LOB fonctionnent sous SQL*Plus lorsque les colonnes sont de type CLOB. Si les LOB avaient été de type BLOB ou BFILE.

12.4.5 Exemple de suppression de LOBs

Une instance LOB peut être supprimée en utilisant des ordres DML appropriés. L'ordre SQL DELETE une ligne et la valeur du LOB interne associée. Pour éviter que la ligne soit supprimée et que seul la référence au LOB soit supprimée, il faut mettre à jour la ligne en remplaçant la colonne LOB avec une valeur NULL ou une chaîne vide grâce à la fonction EMPTY_B/C/NCLOB(). Remplacer la valeur d'une colonne avec un NULL ou la fonction EMPTY_B/C/NCLOB() n'est pas la même chose. En utilisant NULL, la valeur de la colonne sera NULL alors que EMPTY_B/C/NCLOB s'assure qu'il n'y a rien dans la valeur de la colonne.
Un LOB est détruit quand la ligne contenant la colonne LOB est supprimée, que la table est droppée ou tronquée ou implicitement lorsque les données du LOB sont mises à jour.

Exemple :

```
SQL> DELETE FROM employee
  2  WHERE emp_name = 'Marilyn Monroe';
SQL> UPDATE employee SET
  3   resume = EMPTY_CLOB()
  4  WHERE emp_id =7899;
SQL> DELETE FROM lob_table_1 WHERE key = 21;
SQL> DROP TABLE lob_table_1;
SQL> TRUNCATE TABLE lob_table_1;
SQL> UPDATE lob_table_1 SET blob_col = EMPTY_BLOB()
  2  WHERE key = 12;
```

→ Plusieurs exemples de suppression d'un LOB.

Lorsque que l'on veut supprimer le fichier associé à un fichier BFILE il faut utiliser les commandes fournies par le système d'exploitation.
Si l'on veut supprimer une partie d'un LOB interne, on peut utiliser DBMS_LOB.ERASE.

12.5 Les LOBs temporaires

Les LOBs temporaires peuvent être de type BLOB, CLOB ou NCLOB.
Ces LOBs ne sont pas associés avec une table spécifique, les données sont stockées dans le tablespace temporaire de l'utilisateur. Par conséquent ils ne sont pas stockés de manière permanente dans la base de données. Tous les LOBs temporaires sont supprimés à la fin de la session dans laquelle ils ont été créés, mais ils peuvent également être supprimés à la fin de l'appel en le spécifiant à la création.
On crée des LOBs temporaires en utilisant DBMS_LOB.CREATETEMPORARY.
Les LOBs temporaires sont utiles lorsque l'on veut exécuter des opérations de transformation sur un LOB, par exemple pour changer un LOB d'un format à un autre et d'ensuite répercuter ces changements au LOB permanent. Un LOB temporaire sera vide lors de sa création et ne supporte pas les fonctions EMPTY_B/C/NCLOB.
Pour utiliser et manipuler les LOBs temporaires on utilise le package DBMS_LOB.