



ORACLE



Le langage PL/SQL



Ce document ne pourra être communiqué à un tiers ou reproduit, même partiellement, sans autorisation écrite préalable de l'auteur.



SOMMAIRE

1.	<i>Vue d'ensemble.....</i>	7
1.1.	Oracle Database	7
1.2.	Notion de schéma.....	8
1.3.	Règles de nommage	8
2.	<i>Le dictionnaire de données.....</i>	9
3.	<i>L'outil SQL*Plus.....</i>	11
3.1.1.	Environnement de travail.....	11
3.1.2.	Lancement de SQL*Plus sous Dos.....	12
4.	<i>Le langage SQL*Plus</i>	13
4.1.	Utilisation de paramètres	14
4.2.	Commandes SQL*Plus	14
4.2.1.	Mise en forme à l'affichage.....	14
4.2.2.	Ajouter des commentaires.....	15
4.2.3.	Exécuter le contenu d'un script.....	15
4.2.4.	Déclarer un éditeur	16
4.2.5.	Générer un fichier résultat	16
5.	<i>La base Exemple</i>	17
5.1.	Modèle Conceptuel de Données Tahiti.....	17
5.1.1.	Les contraintes d'intégrité	18
5.1.2.	Règles de passage du MCD au MPD.....	18
5.2.	Modèle Physique de données Tahiti :	19
5.3.	Script de création des tables	20
6.	<i>Rappel du langage SQL.....</i>	23
7.	<i>Présentation du langage LMD</i>	25
7.1.	Insérer des lignes dans une table	25
7.1.1.	La commande INSERT	25
7.1.2.	Insertion à partir d'une table existante.....	26
7.2.	Modifier les lignes d'une table	28
7.2.1.	La commande UPDATE	28



7.2.2.	Modifications de lignes à partir d'une table existante	29
7.3.	Spécifier la valeur par défaut d'une colonne.....	30
7.4.	Supprimer les lignes d'une table	31
7.4.1.	La commande DELETE	31
8.	Transactions et accès concurrents.....	33
8.1.	Découper une transaction	34
8.2.	Gestion des accès concurrents	35
8.3.	Les verrous.....	36
8.4.	Accès concurrents en mise à jours.....	37
9.	Le langage PL/SQL.....	39
9.1.	Structure d'un programme P/SQL	39
9.2.	Les variables et les constantes	40
9.3.	Les différents types de données.....	40
9.3.1.	Conversion implicite.....	41
9.3.2.	Conversion explicite	42
10.	Les instructions de bases.....	43
10.1.	Condition.....	43
10.2.	Itération	43
10.2.1.	Syntaxe du LOOP - END LOOP	43
10.2.2.	Syntaxe du While.....	44
10.2.3.	Syntaxe du FOR - IN.....	44
10.3.	Expression NULL.....	44
10.4.	Récupérer les valeurs sélectionnées dans un programme.....	45
10.5.	Gestion de l'affichage	45
11.	Les curseurs	47
11.1.	Opérations sur les curseurs	47
11.2.	Attributs sur les curseurs.....	49
11.3.	Exemple de curseur	49
12.	Les exceptions	51
12.1.	Implémenter des erreurs Oracle	52
12.1.1.	Exception ORACLE dont le nom est prédéfini	52
12.1.2.	Exception ORACLE dont le nom n'est pas prédéfini	53
12.2.	Fonctions pour la gestion des erreurs	54
12.3.	Implémenter des exceptions utilisateurs.....	55



13.	<i>Variables de substitution.....</i>	<i>57</i>
14.	<i>Exemples de programmes PL/SQL.....</i>	<i>58</i>
15.	<i>Variables et instructions avancées.....</i>	<i>60</i>
15.1.	Les structures	60
15.2.	Tables et tableaux.....	61
15.3.	L'expression CASE	62
15.4.	Expression GOTO	65
16.	<i>Complément sur les curseurs</i>	<i>67</i>
16.1.	Curseur BULK COLLECT.....	67
16.2.	Curseur FOR LOOP.....	68
16.3.	Variables curseur.....	69
17.	<i>Instruction "Execute Immediate"</i>	<i>71</i>
18.	<i>Transactions autonomes.....</i>	<i>72</i>
19.	<i>Compilation native du code PL/SQL.....</i>	<i>77</i>
20.	<i>Procédures, Fonctions et Packages.....</i>	<i>78</i>
20.1.	Procédures	80
20.1.1.	Créer une procédure.....	80
20.1.2.	Modifier une procédure	83
20.1.3.	Correction des erreurs.....	83
20.2.	Fonctions.....	86
20.2.1.	Créer une fonction.....	86
20.3.	Packages.....	87
21.	<i>Triggers</i>	<i>91</i>
21.1.	Créer un trigger.....	93
21.2.	Activer un trigger.....	96
21.3.	Supprimer un Trigger	96
21.4.	Triggers rattaché aux vues	97
21.5.	Triggers sur événements systèmes	97
22.	<i>Architecture du moteur PL/SQL</i>	<i>100</i>
22.1.	Procédures stockées JAVA	104
22.2.	Procédures externes.....	104
22.2.1.	Ordres partagés	104
22.3.	Vues du dictionnaire de données	105



23.	<i>Les dépendances.....</i>	<i>107</i>
23.1.	Dépendances des procédures et des fonctions	107
23.1.1.	Dépendances directes.....	107
23.1.2.	Dépendances indirectes	108
23.1.3.	Dépendances locales et distantes	108
23.1.4.	Impacte et gestion des dépendances.....	108
23.2.	Packages.....	109
24.	<i>Quelques packages intégrés.....</i>	<i>110</i>
24.1.	Le package DBMS_OUTPUT.....	110
24.2.	Le package UTL_FILE	110
24.3.	Le package DBMS_SQL.....	111



1. VUE D'ENSEMBLE

1.1. Oracle Database

C'est le produit phare d'Oracle : base de données relationnelle objet, il est disponible sur de nombreuses plates-formes.

Il offre un grand nombre de fonctionnalités : dont une machine virtuelle java intégrée et un serveur HTTP (*Apache*), qui permet de construire une application en architecture n' tiers (logique) avec un seul « produit ».

Les capacités XML ont été étendues, Oracle gère tous les types de données (relationnelles, email, documents, multimédia ou spatiales (géo localisation)).

Il intègre la notion de *Grid Computing* (réseau distribué d'ordinateurs hétérogènes en grille). Le but du *Grid* est de créer des pools de ressources :

- ⇒ de stockage,
- ⇒ de serveurs,

Le *Grid Computing* autorise un accès transparent et évolutif (en termes de capacité de traitement et de stockage) à un réseau distribué d'ordinateurs hétérogènes.

Les composants développés par Oracle pour le Grid Computing sont :

- ♦ **Real Application cluster (RAC)** : Supporte l'exécution d'Oracle sur un cluster d'ordinateurs qui utilisent un logiciel de cluster indépendant de la plate forme assurant la transparence de l'interconnexion.
- ♦ **Automatic Storage Management (ASM)** : Regroupe des disques de fabricants différents dans des groupes disponibles pour toute la grille. ASM simplifie l'administration car au lieu de devoir gérer de nombreux fichiers de bases de données, on ne gère que quelques groupes de disques.
- ♦ **Oracle Ressource Manager** : Permet de contrôler l'allocation des ressources des nœuds de la grille
- ♦ **Oracle Scheduler** : contrôle la distribution des jobs aux nœuds de la grille qui disposent de ressources non utilisées.
- ♦ **Oracle Streams** : Transfère des données entre les nœuds de la grille tout en assurant la synchronisation des copies. Représente la meilleure méthode de réplication.



1.2. Notion de schéma

Le terme schéma désigne l'ensemble des objets qui appartiennent à un utilisateur, ces objets sont préfixés par le nom de l'utilisateur qui les a créés.
En général on indique sous le terme de schéma, l'ensemble des tables et des index d'une même application.

Principaux types d'objets de schéma :

- ◆ Tables et index
- ◆ Vues et synonymes
- ◆ Programmes PL/SQL (procédures, fonctions, packages, triggers)

1.3. Règles de nommage

Un nom de structure Oracle doit respecter les règles suivantes

- ◆ 30 caractères maximums
- ◆ Doit commencer par une lettre
- ◆ Peut contenir des lettres, des chiffres et certains caractères spéciaux (_\$#)
- ◆ N'est pas sensible à la casse
- ◆ Ne doit pas être un mot réservé Oracle



2. LE DICTIONNAIRE DE DONNEES

C'est un ensemble de tables et de vues qui donnent des informations sur le contenu d'une base de données.

Il contient :

- ♦ Les structures de stockage
- ♦ Les utilisateurs et leurs droits
- ♦ Les objets (tables, vues, index, procédures, fonctions, ...)
- ♦ ...

Le dictionnaire de données chargé en mémoire est utilisé par Oracle pour traiter les requêtes.



Il appartient à l'utilisateur SYS et est stocké dans le tablespace SYSTEM.
Sauf exception, toutes les informations sont stockées en MAJUSCULE.
Il contient plus de 866 vues.

Il est créé lors de la création de la base de données, et mis à jour par Oracle lorsque des ordres DDL (*Data Définition Langage*) sont exécutés, par exemple CREATE, ALTER, DROP ...

Il est accessible en lecture par des ordres SQL (SELECT) et est composé de deux grands groupes de tables/vues :

Les tables et vues statiques

- ♦ Basées sur de vraies tables stockées dans le tablespace SYSTEM
- ♦ Accessible uniquement quand la base est ouverte « OPEN »
- ♦ Les tables et vues dynamiques de performance
- ♦ Ne sont en fait basées sur des informations en mémoire ou extraites du fichier de contrôle
- ♦ S'interrogent néanmoins comme de vraies tables/vues
- ♦ Donnent des informations sur le fonctionnement de la base, notamment sur les performances (d'où leur nom)
- ♦ Pour la plupart accessibles même lorsque la base n'est pas complètement ouverte (MOUNT)



Les **vues statiques** sont constituées de 3 catégories caractérisées par leur préfixe :

- ♦ **USER_*** : Informations sur les objets qui appartiennent à l'utilisateur
- ♦ **ALL_*** : Information sur les objets auxquels l'utilisateur a accès (les siens et ceux sur lesquels il a reçu des droits)
- ♦ **DBA_*** : Information sur tous les objets de la base

Derrière le préfixe, le reste du nom de la vue est représentatif de l'information accessible.

Les vues **DICTIONARY** et **DICT_COLUMNS** donnent la description de toutes les tables et vues du dictionnaire.

Oracle propose des synonymes sur certaines vues :

Synonyme	Vue correspondante
cols	User_tab_columns
dict	Dictionnary
ind	User_indexes
obj	User_objects
seq	User_sequences
syn	User_synonyms
tabs	User_tables

Les **vues dynamiques** de performance sont :

- ♦ Préfixées par « V\$ »
- ♦ Derrière le préfixe, le reste du nom de la vue est représentatif de l'information accessible
- ♦ Décrites dans les vues **DICTIONARY** et **DICT_COLUMNS**

Exemple de vues dynamiques

```
V$INSTANCE  
V$DATABASE  
V$SGA  
V$DATABASE  
V$PARAMETER
```



3. L'OUTIL SQL*PLUS

Outil ligne de commande nommé SQLPLUS.

Installé par défaut lors de l'installation des binaires Oracle.

```
SQLPLUS [ connexion ] [ @fichier_script [argument [,...]] ]
```

Il permet de saisir et d'exécuter des ordres SQL ou du code PL/SQL et dispose en plus d'un certain nombre de commandes.

```
-- sans connexion
C:\> SQLPLUS /NOLOG

-- avec connexion
C:\> SQLPLUS system/tahiti@tahiti

SQL> show user
USER est "SYSTEM"
SQL>

-- avec connexion et lancement d'un script sur la ligne de commande
C:\> SQLPLUS system/tahiti@tahiti @info.sql
```

3.1.1. *Environnement de travail*

SQL*PLUS est avant tout un interpréteur de commandes SQL. Il est également fortement interfacé avec le système d'exploitation. Par exemple, on pourra lancer des commandes UNIX ou windows sans quitter sa session SQL*PLUS.

Un SGBDR est une application qui fonctionne sur un système d'exploitation donné. Par conséquent, il faut se connecter au système avant d'ouvrir une session ORACLE. Cette connexion peut être implicite ou explicite.

Il est possible également d'utiliser SQL*Plus sur un client distant et d'accéder à la base de données.



3.1.2. *Lancement de SQL*Plus sous Dos*

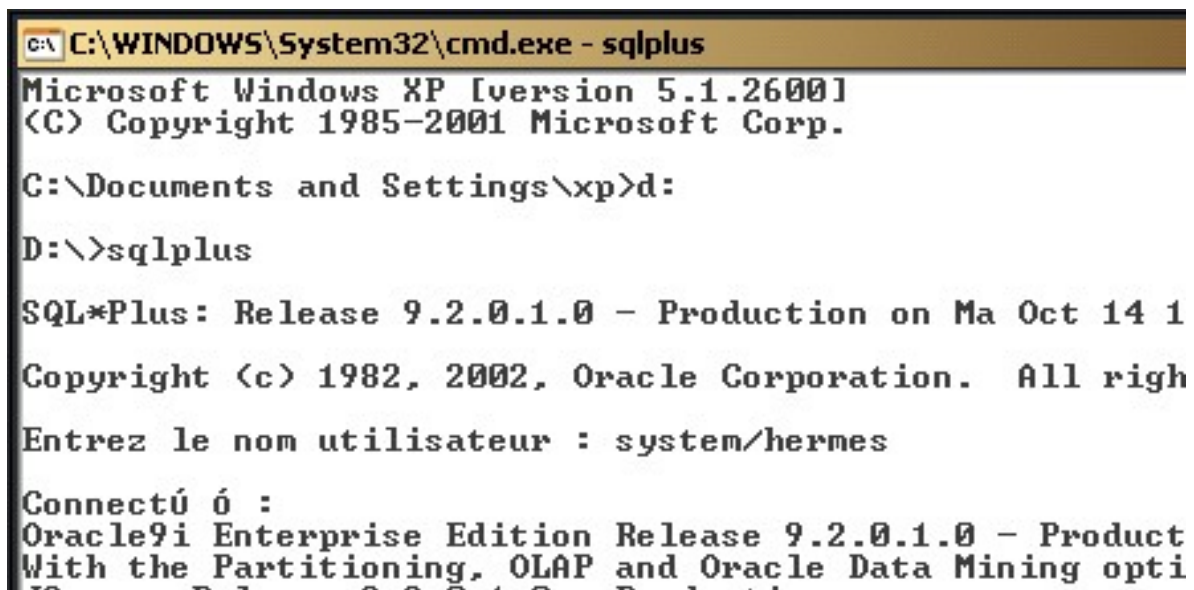
Positionnez la variable d'environnement ORACLE_SID au nom de l'instance sur laquelle vous voulez vous connecter puis exécutez la commande SQL PLUS présentée ci-dessous.

```
Set %ORACLE_SID%=ORCL
SQLPLUS / nolog
SQL> connect system/secret
SQL> connected
SQL>

-- avec connexion
C:\> SQLPLUS charly/secret@tahiti

SQL> show user
USER est "charly"
SQL>
```

Autre exemple :



```
C:\WINDOWS\System32\cmd.exe - sqlplus
Microsoft Windows XP [version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\xp>d:
D:\>sqlplus

SQL*Plus: Release 9.2.0.1.0 - Production on Ma Oct 14 1
Copyright (c) 1982, 2002, Oracle Corporation. All righ
Entrez le nom utilisateur : system/hermes

Connectú ó :
Oracle9i Enterprise Edition Release 9.2.0.1.0 - Product
With the Partitioning, OLAP and Oracle Data Mining opti
```



4. LE LANGAGE SQL*PLUS

Une fois une session SQL*PLUS débutée l'utilisateur peut travailler en interactif ou non. Dans le premier cas il saisira ses commandes sous le prompt SQL et devra les terminer par le caractère « ; » pour lancer l'interprétation.

Dans le second cas, il construit ses scripts (avec l'extension « .sql ») et les lance sous le prompt SQL en les faisant précéder de start ou @. Une session SQL*PLUS se termine par la commande exit. La transaction en cours est alors validée.

Une requête peut s'écrire sur plusieurs lignes. A chaque retour chariot l'interpréteur incrémente le numéro de ligne jusqu'au « ; » final qui marque la fin de la saisie.

```
SQL> select *
      2  from
      3  avion;

ID_AVION NOM_AVION
-----
1 Caravelle
2 Boeing
3 Planeur
4 A_Caravelle_2
```

Un script se lance par la commande start nom_script ou @ nom_script...

```
SQL> start all_avion;

ID_AVION NOM_AVION
-----
1 Caravelle
2 Boeing
3 Planeur
4 A_Caravelle_2
```

L'éditeur par défaut avec lequel s'interface SQL*PLUS est le « Bloc Notes » (c:\windows\notepad.exe). Les principes précédents restent les mêmes.



4.1. Utilisation de paramètres

L'instruction **ACCEPT** permet de saisir des valeurs de paramètres (ce ne sont pas des variables et à ce titre ne nécessitent aucune déclaration).

```
ACCEPT reference NUMBER PROMPT 'Entrez la référence d'un avion: '  
select * from avion where Id_avion=&reference;
```

```
SQL> @essai  
Entrez la référence d'un avion: 1  
  
  ID_AVION  NOM_AVION  
-----  
         1 Caravelle
```

4.2. Commandes SQL*Plus

Les commandes SQL*Plus sont des commandes de mise en forme pour la plupart.
A ne pas confondre avec des commandes SQL.

4.2.1. *Mise en forme à l'affichage*

Les principales commandes de mise en forme sont présentées ci-dessous :

- ♦ **SET LINESIZE 100**, reformater la taille de la ligne à 100 caractères
- ♦ **SET PAUSE ON**, afficher un résultat page par page
- ♦ **SET PAGESIZE 20** : affiche 20 lignes par page entre 2 entêtes de colonnes
- ♦ **COL Nomcol FORMAT A20**, formater l'affichage d'une colonne Nomcol sur 20 caractères
- ♦ **COL Nomcol FORMAT 99.99**, formater l'affichage d'une colonne Nomcol
- ♦ **COL Nomcol FORMAT 0999999999**, affiche le premier zéro
- ♦ **CLEAR COL**, ré-initialiser la taille des colonnes par défaut
- ♦ **SHOW USER**, visualiser le user sous lequel on est connecté
- ♦ **CONNECT [User/MotPass@adresseServeur](#)**, changer de session utilisateur
- ♦ **CLEAR SCREEN**, ré-initialiser l'écran
- ♦ **SET SQLPROMPT TEST>**, affiche le prompt SQL en : TEST>



- ♦ **DESC NomTable**, afficher la structure d'une table ou d'une vue
- ♦ **/**, ré-active la dernière commande
- ♦ **SAVE NomFichier.txt [append|create|replace]**, permet de sauvegarder le contenu du buffer courant dans un fichier « .sql ».
- ♦ **TI ON|OFF**, provoque l'affichage de l'heure avec l'invite
- ♦ **SQL }**, spécifie le caractère « } » comme étant le caractère de continuation d'une commande SQL*Plus.
- ♦ **SUFFIX txt**, spécifie l'extension par défaut des fichiers de commande SQL*Plus

L'option **ON** permet d'activer la production de la ligne d'informations, **OFF** permet de la désactiver. La ligne d'information est système-dépendant.

4.2.2. *Ajouter des commentaires*

Le double tiret « -- » ou la commande **REM** permettent d'ajouter des commentaires à une commande sur une seule ligne.

On peut saisir un commentaire multi-ligne en utilisant « /* ... */ ».

```
-- mon commentaire  
DESC MaTable
```

4.2.3. *Exécuter le contenu d'un script*

Pour exécuter un ensemble de commandes dans un script SQL il suffit d'utiliser la commande suivante :

- ♦ **@ NomFichier.txt**, permet d'exécuter le contenu d'un fichier sql

```
-- mon commentaire  
@ MonFichier.txt
```



4.2.4. *Déclarer un éditeur*

Pour déclarer NotPad comme éditeur SQL*PLUS, et l'extension « .txt » pour exécuter un script il suffit de saisir ces deux lignes de commandes :

```
SET SUFFIX TXT
DEFINE _EDITOR = NOTPAD
```

Après avoir tapé ces 2 lignes de commandes taper :

- ♦ **ED** Pour afficher l'éditeur NotPad.

4.2.5. *Générer un fichier résultat*

La commande SPOOL permet de générer un fichier résultat contenant toutes les commandes passées à l'écran

- ♦ **SPOOL NomFichier.txt**, permet d'activer un fichier de format texte dans lequel on retrouvera les commandes et résultats affichés dans SQL Plus.
- ♦ **SPOOL OFF**, permet de désactiver le spool ouvert précédemment.

```
SPOOL MonFichier.txt
-- commandes SQL affichées
-- commandes SQL affichées
-- commandes SQL affichées

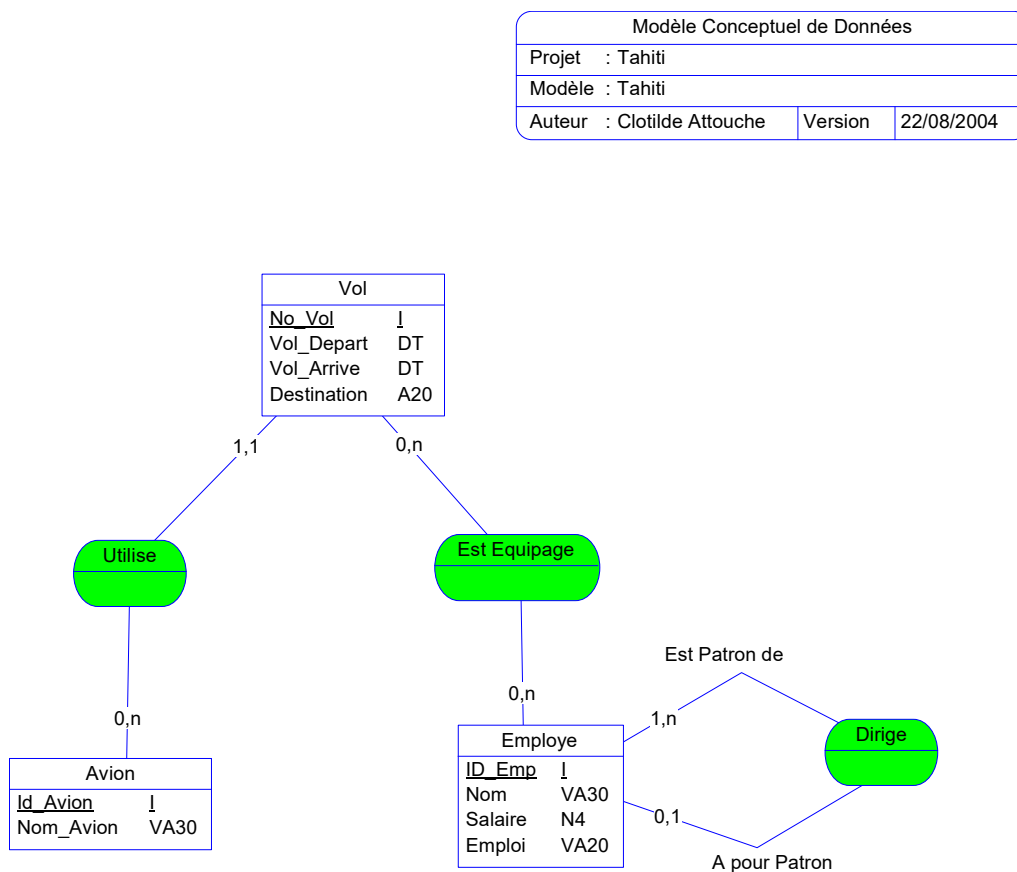
Spool OFF
```




5. LA BASE EXEMPLE

Nous vous présentons la base de données TAHITI qui servira de support aux exemples présentés dans le cours.

5.1. Modèle Conceptuel de Données Tahiti





5.1.1. *Les contraintes d'intégrité*

Les contraintes d'intégrité Oracle sont présentées ci-dessous :

- ♦ **UNIQUE** pour interdire les doublons sur le champ concerné,
- ♦ **NOT NULL** pour une valeur obligatoire du champ
- ♦ Clé primaire (**PRIMARY KEY**) pour l'identification des lignes (une valeur de clé primaire = une et une seule ligne),
- ♦ Clé étrangère (**FOREIGN KEY**) précisant qu'une colonne référence une colonne d'une autre table,
- ♦ **CHECK** pour préciser des domaines de valeurs.



Une clé primaire induit la création de deux contraintes (NOT NULL et UNIQUE).

5.1.2. *Règles de passage du MCD au MPD*

Le passage du Modèle Conceptuel de Données en Modèle Physique de données se fait en appliquant les règles citées ci-dessous :

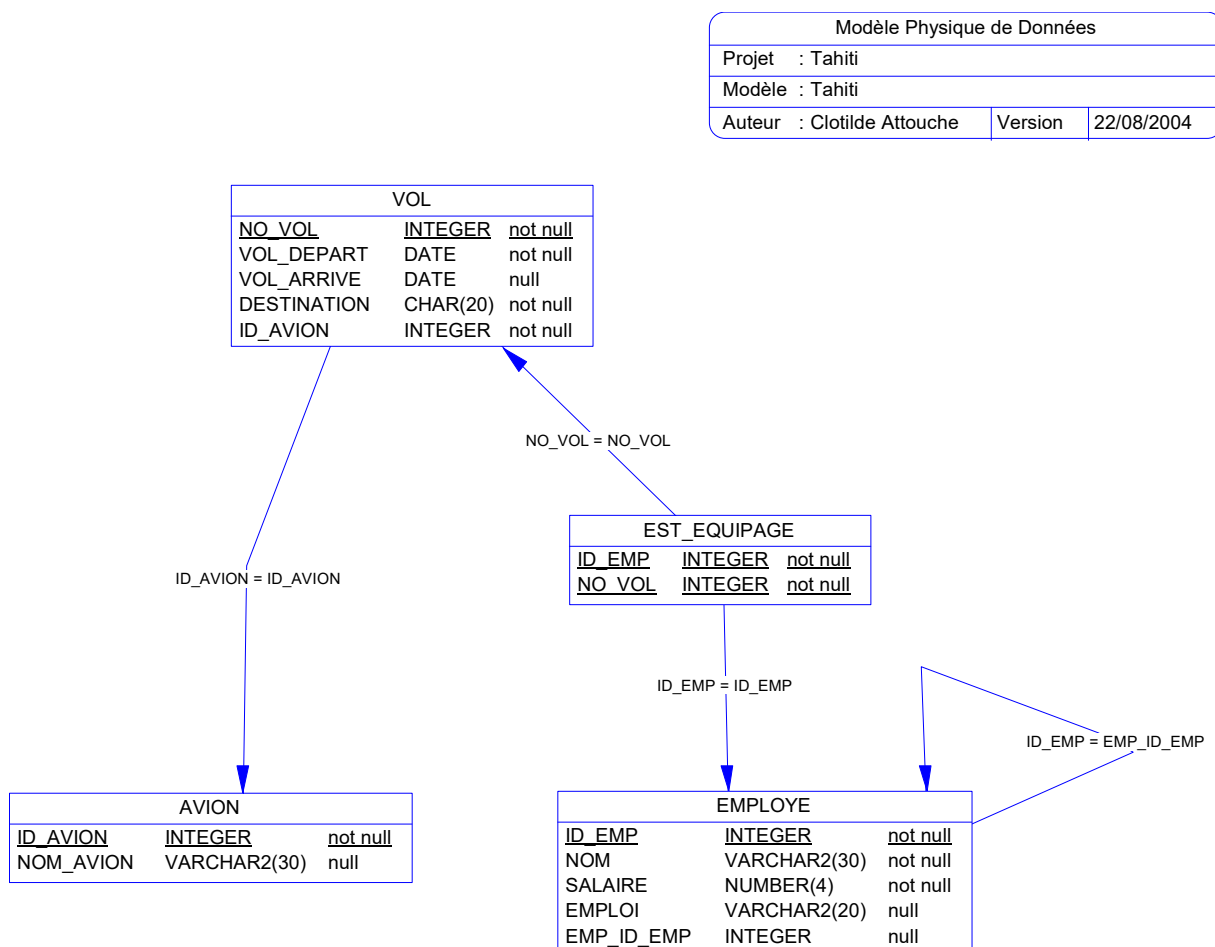
- ⇒ Les entités deviennent des tables
- ⇒ Les identifiants des entités deviennent les clés primaires de ces tables
- ⇒ Les relations dont toutes les cardinalités sont 0,N ou 1,N de chaque côté de la relation deviennent des tables
- ⇒ La concaténation des identifiants des entités qui concourent à la relation devient la clé primaire de la table issue de la relation ; chacun, pris séparément, devient clé étrangère
- ⇒ Pour les relations possédant des cardinalités 0,1 ou 1,1 d'un seul côté de la relation, on fait migrer l'identifiant coté 0,N dans l'entité coté 0,1 devenue table, l'identifiant devient alors clé étrangère ;
- ⇒ Pour les relations possédant des cardinalités 0,1 et 1,1 de chaque côté de la relation, il est préférable de créer une table, mais l'on peut également faire migrer l'identifiant dans l'une des deux entités ; celui ci devient alors clé étrangère (c'est ce que font des outils comme Power AMC)



5.2. Modèle Physique de données Tahiti :

Nous appliquons les règles de passage du MCD au MPD pour générer le modèle présenté ci-dessous avec Power AMC .

Le Modèle Physique de Données créé, une phase d'optimisation doit être effectuée avant de créer la base de données . Durant cette phase, des index sont positionnés sur les colonnes des tables les plus couramment utilisées dans des requêtes ; des informations seront dupliquées.





5.3. Script de création des tables

Nous présentons le script de création des tables de la base de données « Commande » généré avec Power AMC.

```
-- =====
--   Nom de la base   :   TAHITI
--   Nom de SGBD      :   ORACLE
--
-- =====

drop index EST_AFFECTE_PK
/

drop index EST_EQUIPAGE_FK
/

drop index EQUIPAGE_FK
/

drop table EST_EQUIPAGE cascade constraints
/

drop index VOL_PK
/

drop index UTILISE_FK
/

drop table VOL cascade constraints
/

drop index AVION_PK
/

drop table AVION cascade constraints
/

drop index EMPLOYE_PK
/

drop index A_POUR_PATRON_FK
/

drop table EMPLOYE cascade constraints
/

-- =====
--   Table : EMPLOYE
-- =====
create table EMPLOYE
(
    ID_EMP      INTEGER          not null,
    NOM         VARCHAR2(30)     not null,
```



```
        SALAIRE      NUMBER(4)                not null,
        EMPLOI       VARCHAR2(20)             null   ,
        EMP_ID_EMP   INTEGER                   null   ,
        constraint PK_EMPLOYE primary key (ID_EMP)
            using index
            tablespace INDX
    )
tablespace DATA
/

-- =====
--   Index : A_POUR_PATRON_FK
-- =====
create index A_POUR_PATRON_FK on EMPLOYE (EMP_ID_EMP asc)
tablespace INDX
/

-- =====
--   Table : AVION
-- =====
create table AVION
(
    ID_AVION      INTEGER                not null,
    NOM_AVION     VARCHAR2(30)           null   ,
    constraint PK_AVION primary key (ID_AVION)
        using index
        tablespace INDX
)
tablespace DATA
/

-- =====
--   Table : VOL
-- =====
create table VOL
(
    NO_VOL        INTEGER                not null,
    VOL_DEPART    DATE                   not null,
    VOL_ARRIVE    DATE                   null   ,
    DESTINATION   CHAR(20)               not null,
    ID_AVION      INTEGER                not null,
    constraint PK_VOL primary key (NO_VOL)
        using index
        tablespace INDX
)
tablespace DATA
/

-- =====
--   Index : UTILISE_FK
-- =====
create index UTILISE_FK on VOL (ID_AVION asc)
tablespace INDX
/

-- =====
--   Table : EST_EQUIPAGE
-- =====
create table EST_EQUIPAGE
(
    ID_EMP        INTEGER                not null,
```



```
        NO_VOL          INTEGER                not null,
        constraint PK_EST_EQUIPAGE primary key (ID_EMP, NO_VOL)
            using index
            tablespace INDX
    )
    tablespace DATA
    /

-- =====
--      Index : EST_EQUIPAGE_FK
-- =====
create index EST_EQUIPAGE_FK on EST_EQUIPAGE (ID_EMP asc)
tablespace INDX
/

-- =====
--      Index : EQUIPAGE_FK
-- =====
create index EQUIPAGE_FK on EST_EQUIPAGE (NO_VOL asc)
tablespace INDX
/

-- =====
--      Index : CLES ETRANGERES
-- =====
alter table EMPLOYE
    add constraint FK_EMPLOYE_A_POUR_PA_EMPLOYE foreign key  (EMP_ID_EMP)
        references EMPLOYE (ID_EMP)
    /

alter table VOL
    add constraint FK_VOL_UTILISE_AVION foreign key  (ID_AVION)
        references AVION (ID_AVION)
    /

alter table EST_EQUIPAGE
    add constraint FK_EST_EQUI_EST_EQUIP_EMPLOYE foreign key  (ID_EMP)
        references EMPLOYE (ID_EMP)
    /

alter table EST_EQUIPAGE
    add constraint FK_EST_EQUI_EQUIPAGE_VOL foreign key  (NO_VOL)
        references VOL (NO_VOL)
    /

alter table EMPLOYE
    add CONSTRAINT SALAIRE_CC
    CHECK (salaire >500);
    /
```



6. RAPPEL DU LANGAGE SQL


Le langage SQL (*Structured Query Language*) s'appuie sur les normes SQL ANSI en vigueur et est conforme à la norme SQL92 ou SQLV2 (ANSI X3.135-1889n, ISO Standard 9075, FIPS 127).

Il a été développé dans le milieu des années 1970 par IBM (*System R*). En 1979 Oracle Corporation est le premier à commercialiser un SGBD/R comprenant une incrémentation de SQL. Oracle comme acteur significatif intègre ses propres extensions aux ordres SQL.

Depuis l'arrivée d'internet et de l'objet Oracle fait évoluer la base de données et lui donne une orientation objet, on parle SGBDR/O : *System de Base de Données relationnel Objet*.

Les sous langages du SQL sont :

- ⇒ **LID** : Langage d'Interrogation des données, verbe **SELECT**
- ⇒ **LMD** : Langage de Manipulation des Données, utilisé pour la mise à jour des données, verbes **INSERT, UPDATE, DELETE, COMMIT, ROLLBACK**
- ⇒ **LDD** : Langage de définition des données, utilisé pour la définition et la manipulation d'objets tels que les tables, les vues, les index ..., verbe **CREATE, ALTER, DROP, RENAME, TRUNCATE**
- ⇒ **LCD** : Langage de Contrôle des Données, utilisé pour la gestion des autorisations et des privilèges, verbe **GRANT, REVOKE**

SELECT		
	SELECT	Liste des colonnes dans l'ordre d'affichage
	FROM	Liste des tables utilisées
	WHERE	Jointures
	AND	Conditions
	ORDER BY	Condition de Tri



```
SQL> connect charly/charly@tahiti
Connecté.
SQL> desc employe
  Nom                                     NULL ?    Type
-----
ID_EMP                                NOT NULL  NUMBER(38)
NOM                                   NOT NULL  VARCHAR2(30)
SALAIRE                              NOT NULL  NUMBER(4)
EMPLOI                                NULL      VARCHAR2(18)
EMP_ID_EMP                            NULL      NUMBER(38)


SQL> select nom, salaire, emploi
2  from   employe
3  where  salaire >=2000
4  order by nom ;

NOM                                SALAIRE EMPLOI
-----
Marilyne                          2000 Hotesse de l'Air
Spirou                             2000 Pilote
```

- Affiche le nom, le salaire et l'emploi des employés dont le salaire est supérieur ou égal à 2000 Euros.

SELECT nom, salaire, emploi

WHERE salaire >= 2000



EMPLOYEE				
ID_EMP	NOM	SALAIRE	EMPLOI	EMP_ID_EMP
1	Gaston	1700	Directeur	
2	Spirou	2000	Pilote	1
3	Titeuf	1800	Stewart	2
4	Marilyne	2000	Hotesse de l'air	1



7. PRESENTATION DU LANGAGE LMD

La mise à jour des données d'une base se fait par l'une des commandes suivantes :

- ⇒ **INSERT** Insertion d'une ligne
- ⇒ **UPDATE** Modification d'une ou plusieurs lignes
- ⇒ **DELETE** Suppression d'une ou plusieurs lignes

Les commandes de mise à jour de la base déclenchent éventuellement des triggers (cf. chapitre TRIGGERS) ou des contraintes d'intégrité. Elles n'accèdent donc pas directement aux données comme en témoigne le schéma suivant :

Nous allons présenter les points fondamentaux de la syntaxe de ces commandes (nous appuyons nos exemples sur le schéma de la base exemple précédente).

7.1. Insérer des lignes dans une table

7.1.1. La commande INSERT

La commande **INSERT** permet d'insérer une ligne dans une table.

```
INSERT INTO nom_table  
VALUES (liste de valeurs séparées par des virgules dans l'ordre des  
colonnes créées);
```

```
INSERT INTO nom_table (liste de colonnes séparées par des virgules dans l'ordre  
créées)  
VALUES (liste de valeurs séparées par des virgules dans l'ordre des  
colonnes citées);
```

Les CHAR et VARCHAR doivent être saisis entre apostrophes '....'

La valeur NULL permet de ne pas saisir un champ

La fonction to_date permet de traduire une date dans le format interne.



```
----- INSERT Avion -----
--
INSERT INTO Avion VALUES
(1, 'Caravelle' );
INSERT INTO Avion VALUES
(2, 'Boing' );
INSERT INTO Avion VALUES
(3, 'Planeur' );
insert into avion values
(4, 'A_Caravelle_2');

----- INSERT Vol -----
--
INSERT INTO VOL VALUES
(1, sysdate, sysdate+1, 'Tahiti', 1 );
INSERT INTO VOL VALUES
(2, NEXT_DAY(sysdate, 'JEUDI'), NEXT_DAY(sysdate, 'VENDREDI'), 'Marquises', 1 );
INSERT INTO VOL VALUES
(3, LAST_DAY(sysdate), NULL, 'Tokyo', 2 );
```

Vérification :

```
SQL> select * from avion;

  ID_AVION NOM_AVION
-----
         1 Caravelle
         2 Bo'ng
         3 Planeur
         4 A_Caravelle_2

SQL> select * from vol;

  NO_VOL VOL_DEPA VOL_ARRI DESTINATION ID_AVION
-----
         1 04/09/04 05/09/04 Tahiti          1
         2 09/09/04 10/09/04 Marquises     1
         3 30/09/04          Tokyo          2
```

7.1.2. Insertion à partir d'une table existante

Nous allons créer une table AVION_2, car pour notre exemple il faut travailler obligatoirement sur une autre table.

```
SQL> create table avion_2
2  (
3      ID_AVION      INTEGER          not null,
4      NOM_AVION     VARCHAR2(30)      null,
5      constraint PK_AVION_2 primary key (ID_AVION),
6      DESTINATION   VARCHAR2(30)      null
7  );

Table créée.
```



```
SQL> desc avion_2
Nom                                NULL ?   Type
-----
ID_AVION                          NOT NULL NUMBER(38)
NOM_AVION                          VARCHAR2(30)
DESTINATION                        VARCHAR2(30)

SQL> select * from avion_2;

aucune ligne sélectionnée
```

```
SQL> insert into avion_2
2  select a.id_avion, nom_avion, destination
3  from avion a, vol v
4  where v.id_avion = a.id_avion
5  and destination = 'Marquises' ;

1 ligne créée.

SQL> select * from avion_2;

ID_AVION NOM_AVION                                DESTINATION
-----
1 Caravelle                                Marquises

SQL> insert into avion_2 (id_avion, nom_avion)
2  select id_avion, nom_avion
3  from avion
4  where id_avion > 1 ;

3 ligne(s) créée(s).

SQL> select * from avion_2;

ID_AVION NOM_AVION                                DESTINATION
-----
1 Caravelle                                Marquises
2 Bo'ng
3 Planeur
4 A_Caravelle_2
```



7.2. Modifier les lignes d'une table

7.2.1. La commande UPDATE

La commande **UPDATE** permet de modifier une ou plusieurs lignes d'une table.

```
UPDATE nom_table SET liste d'affectations
WHERE conditions sur les lignes concernées;
```



Sans clause WHERE, toute la table est modifiée

```
SQL> select * from vol
2 ;

      NO_VOL VOL_DEPA VOL_ARRI DESTINATION      ID_AVION
-----
      1 04/09/04 05/09/04 Tahiti              1
      2 09/09/04 10/09/04 Marquises           1
      3 30/09/04              Tokyo             2

SQL> update vol set
2   vol_arrive = to_date('01/10/2004 03:30:00', 'DD/MM/YYYY HH24:MI:SS')
3   where no_vol = 3 ;

1 ligne mise à jour.
```

Vérification :

```
SQL> col depart for A20
SQL> col arrive for A20
SQL> select to_char(vol_depart, 'DD/MM/YYYY HH24:MI:SS') Depart,
2          to_char(vol_arrive, 'DD/MM/YYYY HH24:MI:SS') Arrive,
3          destination
4   from vol
5   where no_vol = 3 ;

DEPART      ARRIVE      DESTINATION
-----
30/09/2004 16:19:53 01/10/2004 03:30:00 Tokyo
```



7.2.2. Modifications de lignes à partir d'une table existante

Dans cet exemple nous allons modifier la table AVION_2 créée précédemment.

```
update Article_1
set (Id_article, designation)
  SELECT Id_article, designation
  FROM Article
 WHERE ....

SQL> select * from avion_2 ;

  ID_AVION NOM_AVION                                DESTINATION
-----
         1 Caravelle                                Marquises
         2 Boeing
         3 Planeur
         4 A_Caravelle_2

SQL> select * from vol ;

  NO_VOL VOL_DEPA VOL_ARRI DESTINATION                                ID_AVION
-----
         1 04/09/04 05/09/04 Tahiti                                1
         2 09/09/04 10/09/04 Marquises                             1
         3 30/09/04 01/10/04 Tokyo                                2
```

Modification de la table :

```
SQL> update avion_2
2   set (destination) = (select destination
3                           from vol
4                           where no_vol = 1)
5   where destination is null ;

3 ligne(s) mise(s) à jour.

SQL> select * from avion_2;

  ID_AVION NOM_AVION                                DESTINATION
-----
         1 Caravelle                                Marquises
         2 Boeing                                    Tahiti
         3 Planeur                                    Tahiti
         4 A_Caravelle_2                             Tahiti
```



7.3. Spécifier la valeur par défaut d'une colonne

Dans un ordre INSERT ou UPDATE, il est possible d'affecter explicitement à une colonne la valeur par défaut définie sur cette colonne

- ⇒ En mettant le mot clé **DEFAULT** comme valeur de la colonne
NULL est affecté si la colonne n'a pas de valeur par défaut

Lors d'un INSERT :

```
SQL> insert into avion_2
  2  values (5, 'Petit coucou', 'Canaries');

1 ligne cr  e.

SQL> insert into avion_2
  2  values (6, 'Petit coucou', default);

1 ligne cr  e.

SQL> select * from avion_2;
```

ID_AVION	NOM_AVION	DESTINATION
1	Caravelle	Marquises
2	Bo��ng	Tahiti
3	Planeur	Tahiti
4	A_Caravelle_2	Tahiti
5	Petit coucou	Canaries
6	Petit coucou	

```
6 ligne(s) s  lectionn  e(s).
```

Lors d'un UPDATE :

```
SQL> update avion_2
  2  set nom_avion = default
  3  where id_avion = 5 ;

1 ligne mise    jour.

SQL> select * from avion_2;
```

ID_AVION	NOM_AVION	DESTINATION
1	Caravelle	Marquises
2	Bo��ng	Tahiti
3	Planeur	Tahiti
4	A_Caravelle_2	Tahiti
5	Petit coucou	Canaries
6	Petit coucou	



6 ligne(s) sélectionnée(s).

```
SQL> update avion_2
2 set nom_avion = default
3 where destination like '%h%';
```

3 ligne(s) mise(s) à jour.

```
SQL> select * from avion_2 ;
```

ID_AVION	NOM_AVION	DESTINATION
1	Caravelle	Marquises
2		Tahiti
3		Tahiti
4		Tahiti
5		Canaries
6	Petit coucou	

7.4. Supprimer les lignes d'une table

7.4.1. La commande DELETE

La commande **DELETE** permet de supprimer une ou plusieurs lignes d'une table.

```
DELETE FROM nom_table
WHERE conditions sur les lignes concernées;
```



Sans la clause WHERE toute la table est vidée.

Suppression du vol numéro 10, et vérification.

```
SQL> select * from vol ;
```

NO_VOL	VOL_DEPA	VOL_ARRI	DESTINATION	ID_AVION
1	04/09/04	05/09/04	Tahiti	1
2	09/09/04	10/09/04	Marquises	1
3	30/09/04		Tokyo	2
10	11/09/04		Paris	



```
SQL> delete from vol where no_vol = 10;
```

1 ligne supprimée.

```
SQL> select * from vol ;
```

	NO_VOL	VOL_DEPA	VOL_ARRI	DESTINATION	ID_AVION
1	04/09/04	05/09/04		Tahiti	1
2	09/09/04	10/09/04		Marquises	1
3	30/09/04			Tokyo	2

Supprimer toutes les lignes de la table AVION_2 sans destination .:

```
SQL> select * from avion_2 ;
```

ID_AVION	NOM_AVION	DESTINATION
1	Caravelle	Marquises
2	Bo'ng	
3	Planeur	
4	A_Caravelle_2	

```
SQL> delete from avion_2  
2 where destination is null ;
```

3 ligne(s) supprimée(s).

```
SQL> select * from avion_2;
```

ID_AVION	NOM_AVION	DESTINATION
1	Caravelle	Marquises



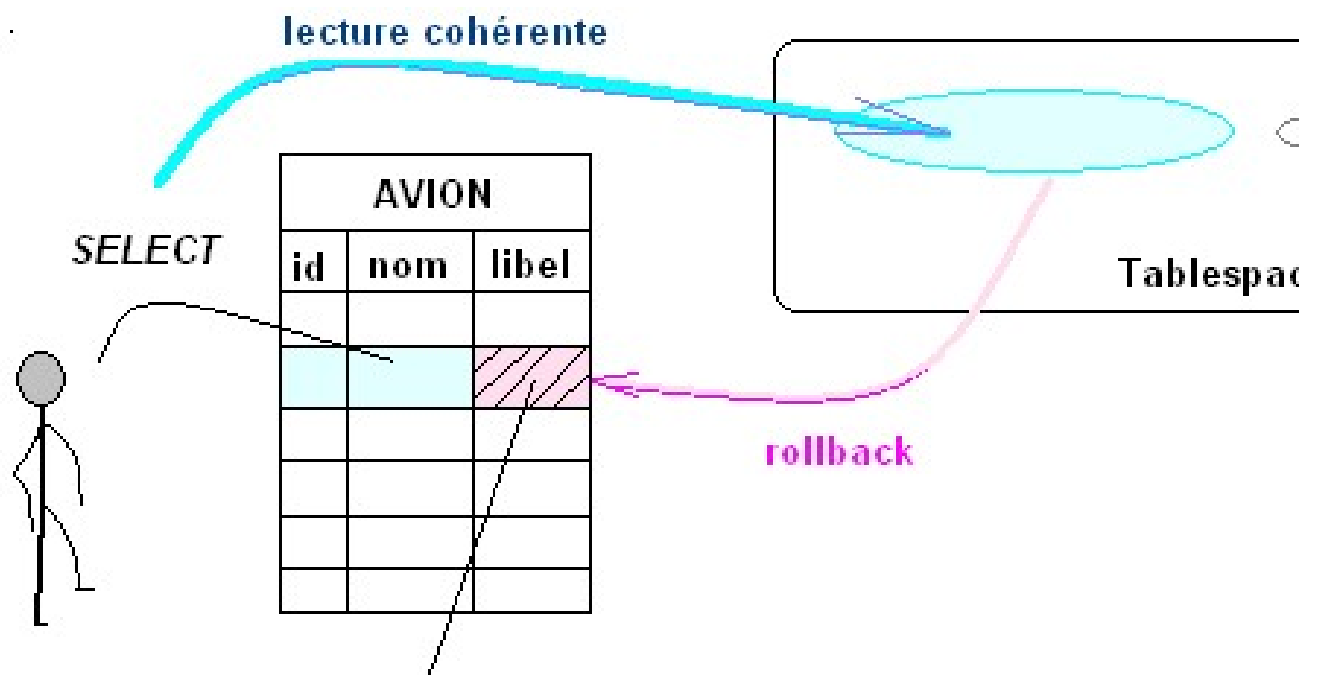
8. TRANSACTIONS ET ACCES CONCURENTS

La cohérence des données repose sur le principe des transactions et des accès concurrents. Une transaction correspond à un ensemble de commandes SQL que l'on appellera actions élémentaires. Cet ensemble forme un tout qui sera entièrement validé (mise à jour définitive de la base) ou pas du tout. ORACLE offre également un mécanisme de gestion des accès concurrents. Ce mécanisme repose sur la technique du verrouillage des données. Ce verrouillage peut être implicite (par ORACLE) ou explicite (par l'utilisateur).

Principe général :

- ⇒ ORACLE exécute une commande qui appartient à une transaction.
- ⇒ ORACLE valide une transaction dans sa globalité ou pas du tout.

La lecture cohérente garantie par Oracle est la possibilité de lire des données pendant la mise à jour de celles-ci tout en étant assuré que la version des données lues est la même.



Soit la transaction constituée des deux commandes :

```
INSERT INTO ligne_com VALUES (10,1,5,40);  
UPDATE article SET qtestock=qtestock - 40 WHERE Id_article=5;
```



La première commande insère une ligne de commande dans la table ligne_com (la commande numéro 10 concerne 40 articles numéro 5).

La seconde commande met à jour la quantité en stock de l'article 5 d'après la quantité commandée.

Ces deux commandes doivent être exécutées et validées toutes les deux. Si, pour une raison quelconque (panne, condition fausse, ...) l'une des commandes n'a pu être traitée, ORACLE doit annuler l'autre. Lorsque les deux commandes sont exécutées et deviennent effectives, la transaction est **valide**. Dans le cas contraire, elle est **annulée**.

⇒ La base revient dans l'état qu'elle avait avant la transaction.

L'exécution d'une commande (opération élémentaire) dépend de :

- ♦ syntaxe correcte,
- ♦ respect des contraintes,
- ♦ accessibilité physique ou logique des données (réseau, droits, ...)

Pour rendre définitive l'exécution des commandes il faut valider la transaction correspondante.

La **validation** d'une transaction est implicite ou explicite :

- ♦ La commande **commit** permet de **valider** l'ensemble des opérations élémentaires de la transaction en cours. La prochaine opération fera débiter une nouvelle transaction.
- ♦ La commande **rollback annule** l'exécution des opérations élémentaires de la transaction en cours. La prochaine opération fera débiter une nouvelle transaction.
- ♦ La fin normale d'une session (programme client ou session SQL*PLUS) entraîne la validation implicite de la transaction courante.
- ♦ La fin anormale d'une session entraîne l'annulation de la transaction courante.
- ♦ Les commandes de définition de données (**CREATE**, **ALTER**, **RENAME**, **DROP**) sont automatiquement validées.

8.1. Découper une transaction

Le début d'une application ou d'une session SQL constitue automatiquement le début d'une transaction. Chaque instruction commit ou rollback marque la fin de la transaction courante et le début d'une nouvelle transaction. Une transaction correspond donc à un ensemble de commandes comprises entre deux instructions commit ou rollback.



Il est cependant possible de définir plus finement une transaction en insérant des points de repères (*savepoints*).

L'instruction `SAVEPOINT` permet de préciser les points de repères jusqu'où l'annulation de la transaction pourra porter.

On crée donc ainsi des sous transactions.

```
INSERT INTO ligne_com VALUES (10,1,5,40);  
SAVEPOINT point1;  
UPDATE article SET qtestock=qtestock - 40 WHERE Id_article=5;
```

A ce niveau,

- l'instruction `commit` valide les deux commandes `INSERT` et `UPDATE`,
- l'instruction `rollback` annule les deux commandes `INSERT` et `UPDATE`
- l'instruction `ROLLBACK to point1` annule la commande `UPDATE`. La prochaine instruction `commit` ou `rollback` ne portera que sur la commande `INSERT`.

8.2. Gestion des accès concurrents

La gestion des accès concurrents consiste à assurer la sérialisation des transactions qui accèdent simultanément aux mêmes données. Cette fonctionnalité de base d'ORACLE est basée sur les concepts d'intégrité, de concurrence, et de consistance des données

Intégrité des données

L'intégrité des données est assurée par les différentes contraintes d'intégrité définies lors de la création de la base. Elle doit être maintenue lors de l'accès simultané aux mêmes données par plusieurs utilisateurs. La base de données doit passer d'un état cohérent à un autre état cohérent après chaque transaction.

Concurrence des données

La concurrence des données consiste à coordonner les accès concurrents de plusieurs utilisateurs aux mêmes données (deux `SELECT` doivent pouvoir s'exécuter en parallèle).

Consistance des données

La consistance des données repose sur la stabilité des données. Lorsqu'un utilisateur utilise des données en lecture ou en mise à jour, le système doit garantir que l'utilisateur manipule toujours les mêmes données.



Autrement dit, on ne doit pas débiter un traitement sur des données dont la liste ou les valeurs sont modifiées par d'autres transactions (un `SELECT` débutant avant un insert (même validé) ne doit pas afficher le nouveau *tuple* inséré)

8.3. Les verrous

Pour que l'exécution simultanée de plusieurs transactions donne le même résultat qu'une exécution séquentielle, la politique mise en œuvre consiste à verrouiller momentanément les données utilisées par une transaction. Dans ORACLE, le granule de verrouillage est la ligne. Tant qu'une transaction portant sur une ou plusieurs lignes n'est pas terminée (validée ou annulée), toutes les lignes sont inaccessibles en mise à jour pour les autres transactions. On parle de verrouillage. Il peut s'agir d'un verrouillage implicite ou explicite.

Verrouillage implicite

Toute commande insert ou update donne lieu à un verrouillage des lignes concernées tant que la transaction n'est pas terminée. Toute transaction portant sur ces mêmes lignes sera mise en attente.

Verrouillage explicite

Dans certains cas l'utilisateur peut souhaiter contrôler lui-même les mécanismes de verrouillage. En général, il utilise la commande :

```
| select * from vol for update
```

Tous les `VOLs` sont verrouillés mais une clause `WHERE` est possible. Le verrouillage ne porte alors que sur les lignes concernées.

Il existe différents modes de verrouillages d'une table (mode lignes partagées, équivalent au *select for update*, mode lignes exclusives, mode table partagée, mode partage exclusif, mode table exclusive).

En plus de la simple visibilité des données, on peut ainsi préciser les verrous autorisés par dessus les verrous que l'on pose. Par exemple, plusieurs *select for update* peuvent s'enchaîner (verrouillage en cascade).

Lorsque la première transaction sera terminée, le second *select for update* pose ses verrous et ainsi de suite. Par contre, un verrouillage en mode table exclusive empêche tout autre mode de verrouillage. A titre d'exemple, nous ne présenterons ici que les verrouillages standards (implicites suite à une commande insert, update, ou delete).



Verrouillage bloquant

ORACLE détecte les verrouillages bloquant (*deadlock*). Ces verrouillages correspondent à une attente mutuelle de libération de ressources.

Exemple

Transaction T1	Transaction T2
update article set qtestock=10 where Id_article=1;	update article set qtestock=30 where Id_article=2;
update article set qtestock=20 where Id_article=2;	update article set qtestock=40 where Id_article=1;
commit;	commit;

Si les deux transactions ne sont pas lancées « vraiment » en même temps, on ne parle pas de verrouillage bloquant. Les deux transactions s'exécutent normalement l'une à la suite de l'autre.



Dans tous les cas, après une instruction commit ou rollback :
Les verrous sont levés
Une nouvelle transaction commence à la prochaine instruction.

8.4. Accès concurrents en mise à jours

Si deux utilisateurs accèdent à des lignes différentes d'une table qui n'a pas fait l'objet d'un verrouillage particulier, les transactions s'effectuent normalement.

Si les deux utilisateurs accèdent aux mêmes lignes d'une table alors la transaction débutée le plus tard sera mise en attente. La validation de la première transaction « libérera » la seconde.

Les mécanismes internes de gestion des transactions et des accès concurrents sont gérés par ORACLE. Il reste à la charge du programmeur la gestion des verrous explicites et la maîtrise des verrous implicites. Les règles générales sont les suivantes :

Une transaction est constituée d'un ensemble d'opérations élémentaires (insert, update, ...),

- ⇒ ORACLE garantit qu'une transaction est entièrement validée ou défaite,
- ⇒ Toute session SQL (sous SQL*PLUS ou depuis un programme) démarre une transaction,
- ⇒ Toute fin normale de session déclenche un commit,
- ⇒ Toute fin anormale de session déclenche un rollback,
- ⇒ L'unité de verrouillage sous ORACLE est la ligne,



- ⇒ Une commande `INSERT`, `DELETE`, ou `UPDATE` entraîne un verrouillage implicite des lignes concernées,
- ⇒ La commande `SELECT FOR UPDATE` permet de verrouiller explicitement les lignes concernées. Elle peut utiliser la clause `WHERE` pour ne pas verrouiller toute la table,
- ⇒ Les verrous sont levés par les commandes `commit` ou `rollback`.

Ne jamais perdre de vue les scénarii d'activité des opérateurs afin d'éviter de mettre en place une gestion aussi fine qu'inutile de l'unité de verrouillage (ligne ?, table?). Concrètement, il faut se poser des questions de base comme « Combien d'accès concurrents sur telles données observe-t-on en moyenne ? ». Le code s'en trouvera considérablement simplifié.



9. LE LANGAGE PL/SQL

Le langage PL/SQL dérivé de l'ADA est un langage de programmation qui permet de grouper des commandes et de les soumettre au noyau comme un bloc unique de traitement.

Contrairement au langage SQL, qui est non procédural (on ne se soucie pas de comment les données sont traitées), le PL/SQL est un langage procédural qui s'appuie sur toutes les structures de programmations traditionnelles (variables, itérations, tests, séquences).

Le PL/SQL s'intègre dans les outils SQL*FORMS, SQL*PLUS, PRO*C, ...il sert à programmer des procédures, des fonctions, des triggers, et donc plus généralement, des packages.

9.1. Structure d'un programme P/SQL

Un programme PL/SQL se décompose en trois parties :

DECLARE

BEGIN

EXCEPTION

END ;

/

- ⇒ La zone **DECLARE** sert à la déclaration des variables, des constantes, ou des curseurs,
- ⇒ La zone **BEGIN** constitue le corps du programme,
- ⇒ La zone **EXCEPTION** permet de préciser les actions à entreprendre lorsque des erreurs sont rencontrées (pas de référence article trouvée pour une insertion, ...),
- ⇒ Le **END** répond au **BEGIN** précédent, il marque la fin du script.



9.2. Les variables et les constantes

La déclaration d'une variable ou d'une constante se fait dans la partie `DECLARE` d'un programme PL/SQL. On peut déclarer le type d'une variable d'une façon implicite ou explicite.

```
DECLARE
    ancien_vol          NUMBER ; -- type explicite
    autre_vol           NUMBER DEFAULT 0 ; -- initialisation à 0
    nouveau_vol article.prixunit%TYPE ; -- type implicite
```

L'utilisation de l'attribut `%ROWTYPE` permet à la variable d'hériter des caractéristiques d'une ligne de table.

```
DECLARE
    V_vol vol%ROWTYPE ;
```

Si une variable est déclarée avec l'option `CONSTANT`, elle doit être initialisée.

Si une variable est déclarée avec l'option `NOT NULL`, elle doit être initialisée et la valeur `NULL` ne pourra pas lui être affectée durant l'exécution du programme.

9.3. Les différents types de données

Les différents types utilisés pour les variables PL/SQL sont :

TYPE	VALEURS
BINARY-INTEGER	entiers allant de -2^{31} à 2^{31})
POSITIVE / NATURAL	entiers positifs allant jusqu'à $2^{31} - 1$
NUMBER	Numérique (entre -2^{418} à 2^{418})
INTEGER	Entier stocké en binaire (entre -2^{126} à 2^{126})
CHAR (n)	Chaîne fixe de 1 à 32767 caractères (différent pour une colonne de table)
VARCHAR2 (n)	Chaîne variable (1 à 32767 caractères)
LONG	idem VARCHAR2 (maximum 2 gigaoctets)
DATE	Date (ex. 01/01/1996 ou 01-01-1996 ou 01-JAN-96 ...)



RAW	Permet de stocker des types de données binaire relativement faibles(<= 32767 octets) idem VARCHAR2. Les données RAW ne subissent jamais de conversion de caractères lors de leur transfert entre le programme et la base de données.
LONG RAW	Idem LONG mais avec du binaire
CLOB	Grand objet caractère. Objets de type long stockés en binaire (maximum 4 giga octets) Déclare une variable gérant un pointeur sur un grand bloc de caractères, mono-octet et de longueur fixe, stocké en base de données.
BLOB	Grand objet binaire. Objets de type long (maximum 4 giga octets) Déclare une variable gérant un pointeur sur un grand objet binaire stocké dans la base de données (Son ou image).
NCLOB	Support en langage nationale (NLS) des grands objets caractères. Déclare une variable gérant un pointeur sur un grand bloc de caractères utilisant un jeu de caractères mono-octets, multi-octets de longueur fixe ou encore multi-octets de longueur variable et stocké en base de données.
ROWID	Composé de 6 octets binaires permettre d'identifier une ligne par son adresse physique dans la base de données.
UROWID	Le U de UROWID signifie Universel, une variable de ce type peut contenir n'importe quel type de ROWID de n'importe quel type de table.
BOOLEAN	Bouléen, prend les valeurs TRUE, FALSE, NULL

On y retrouve tous les types de données utilisés par les tables avec en plus le type booléen qui est propre au PL/SQL.

9.3.1. *Conversion implicite*

Le PL/SQL opère des conversions de type implicites en fonctions des opérandes en présence. Il est nécessaire de bien connaître ces règles comme pour tous les langages. Cependant, le typage implicite des variables (cf. chapitre sur les variables) permet de simplifier la déclaration des variables.



9.3.2. *Conversion explicite*

Le programmeur peut être maître d'oeuvre du typage des données en utilisant des fonctions standards comme `to_char`, `to_date`, etc. ..Des exemples seront donnés plus loin dans le support.



10. LES INSTRUCTIONS DE BASES

10.1. Condition

Exemple :

une société d'informatique augmente le salaire de ses employés d'un montant variant de 100 à 500 euros, en fonction de la demande de leur supérieur hiérarchique :

```
IF salaire < =1000 THEN
    nouveau_salaire := ancien_salaire + 100;
    ELSIF salaire > 1000 AND emp_id_emp = 1 THEN
        nouveau_salaire := ancien_salaire + 500;
    ELSE nouveau_salaire := ancien_salaire + 300;
END IF;
```

10.2. Itération

10.2.1. *Syntaxe du LOOP - END LOOP*

On dispose de l'instruction `LOOP` que l'on peut faire cohabiter avec les traditionnelles `WHILE` et `FOR`. La commande `EXIT` permet d'interrompre la boucle.

```
LOOP
    Select salaire into V_salaire
    from employe where id_emp=V_emp ;
    EXIT WHEN V_emp >= 10;

    IF V_salaire < =1000 THEN
        nouveau_salaire := ancien_salaire + 100;
        update employe set salaire = nouveau_salaire
        where id_emp = V_emp;
    END IF ;

    V_emp = V_emp + 1 ;
END LOOP;
```



10.2.2. *Syntaxe du While*

Equivalent du « Tant que ».

```
WHILE V_emp >= 10;
LOOP
    Select salaire into V_salaire
    from employe where id_emp=V_emp ;
    IF V_salaire < =1000 THEN
        nouveau_salaire := ancien_salaire + 100;
        update employe set salaire = nouveau_salaire
        where id_emp = V_emp;
    END IF ;

    V_emp = V_emp + 1 ;
END LOOP;
```

10.2.3. *Syntaxe du FOR - IN*

Utilisé principalement pour balayer une table indexée.

```
FOR compteur IN borne_inférieure..borne_supérieure
LOOP
    séquences
END LOOP;
```

10.3. Expression NULL

Lorsque vous voulez demander au PL/SQL de ne rien faire, l'instruction NULL devient très pratique. Le format d'une instruction NULL est le suivant :

```
NULL;
```

Le point virgule indique qu'il s'agit d'une commande et non pas de la valeur NULL.

```
--
IF V_etat_selection = 'detail'
Then
    Exec etat_detaille;
Else
    NULL ; -- ne rien faire
END IF ;
```



10.4. Récupérer les valeurs sélectionnées dans un programme

Pour récupérer les valeurs sélectionnées à partir d'une table dans un programme PL, il faut utiliser la commande INTO derrière le SELECT.

Exemple

```
Declare
V_destination      vol.destination%type ;
V_depart           vol.vol_depart%type ;

BEGIN
SELECT Destination, Vol_depart
INTO V_destination, V_depart
From Vol
Where destination = 'Tahiti' ;

-- suite des instructions

END;
/
```

Faire attention à l'ordre des variables derrière la commande INTO, qui doivent être situées dans le même ordre que l'ordre des colonnes derrière le SELECT.

⇒ ORACLE ne sait pas lire !

Dans notre exemple, la colonne destination alimente la variable V_destination car elle sont toutes les 2 situées en première position.

10.5. Gestion de l'affichage

Il est évidemment possible d'utiliser des fonctions prédéfinies d'entrée/sortie. Ces fonctions servent essentiellement durant la phase de test des procédures PL/SQL (l'affichage étant généralement géré du côté client).

Il faut d'abord activer le package DBMS_OUTPUT par la commande *set serveroutput on*. Ensuite on peut utiliser la fonction *put_line* de ce package.



affiche.sql

```
set serveroutput on
DECLARE
message      varchar2(100);
BEGIN
message := 'Essai d'affichage';
DBMS_OUTPUT.put_line ('Test : ' || message);
END;
/
```

Exécution

```
SQL> @affiche
Test : Essai d'affichage

PL/SQL procedure successfully completed.
```

Noter l'emploi de " pour pouvoir afficher le caractère ', et du double pipe || pour pouvoir concaténer plusieurs chaînes de caractères. La concaténation est souvent nécessaire car la fonction *put_line* n'accepte qu'une seule chaîne de caractères en argument.

Lorsque les données affichées « défilent » trop vite on pourra utiliser deux méthodes :

Utiliser la commande **set pause on** (une commande **set pause off** devra lui correspondre pour revenir à la normale). Il faut alors frapper la touche RETURN pour faire défiler les pages d'affichage.

Utiliser la commande **spool nom_fichier**. Elle permet de diriger l'affichage, en plus de la sortie standard qui est l'écran, vers un fichier nommé nom_fichier.lst. La commande **spool off** est nécessaire pour arrêter la redirection (attention aux fichiers de spool qui grossissent considérablement).



11. LES CURSEURS

Un curseur est une variable qui pointe vers le résultat d'une requête SQL. La déclaration du curseur est liée au texte de la requête.

Lorsqu'une requête n'extrait qu'une seule ligne l'utilisation d'un curseur n'est pas nécessaire. Par contre, si plusieurs lignes sont retournées, il faut pouvoir les traiter une à une à la manière d'un fichier.

Un curseur permet de lire séquentiellement le résultat d'une requête. On peut ainsi traiter les lignes résultantes une par une en déchargeant les données lues dans des variables hôtes.

Ce mécanisme est nécessaire car on ne peut écrire une instruction qui remplirait d'un coup toute notre structure `ma_struct` de l'exemple précédent. L'instruction suivante entraîne une erreur de type :

```
select no_vol into mes_vols from vol
;
PLS-00385: type mismatch found at 'mes_vols' in SELECT...INTO statement
```

11.1. Opérations sur les curseurs

Les seules opérations possibles sur un curseur sont :

⇒ DECLARE

Définition du curseur (donc de la requête associée).

```
DECLARE
  CURSOR mes_vols IS
  SELECT no_vol, destination
  FROM vol
  WHERE destination = 'Tahiti';
```

⇒ OPEN

Exécution de la requête, allocation mémoire pour y stocker le résultat, positionnement du curseur sur le premier enregistrement.

```
OPEN mes_vols;
```

⇒ FETCH

Lecture de la zone pointée par le curseur, affectation des variables hôtes, passage à l'enregistrement suivant.

```
FETCH mes_vols into v_vol, v_destination;
```

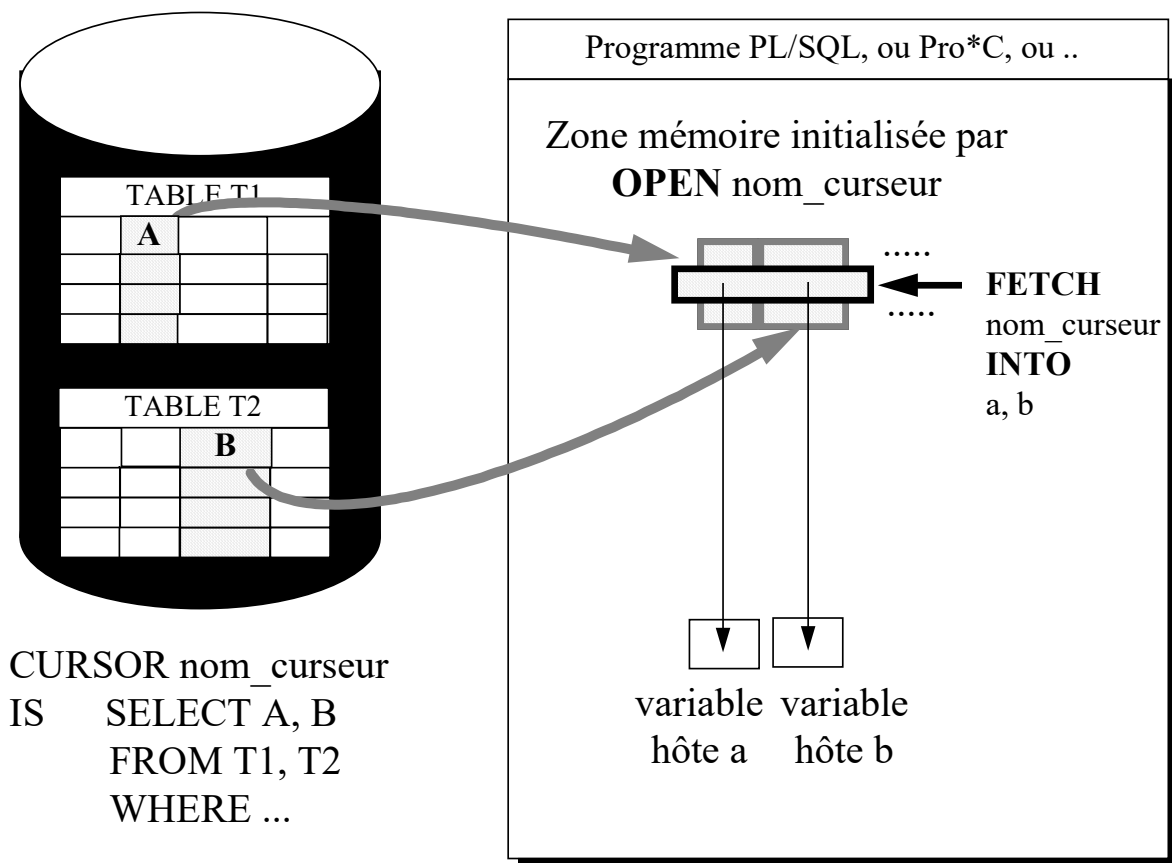


⇒ CLOSE

Fermeture du curseur, libération de la zone mémoire allouée pour le résultat de la requête.

```
CLOSE mes_vols;
```

Voici un schéma de synthèse sur le principe des curseurs :





11.2. Attributs sur les curseurs

Chaque curseur possède quatre attributs :

- ⇒ **%FOUND** Génère un booléen VRAI lorsque le FETCH réussi (données lues)
- ⇒ **%NOTFOUND** Inverse de %FOUND (généralement plus utilisé que %FOUND)
- ⇒ **%ISOPEN** Génère un booléen VRAI lorsque le curseur spécifié en argument est ouvert.
- ⇒ **%ROWCOUNT** Renvoie le nombre de lignes contenues.

Chaque attribut s'utilise en étant préfixé par le nom du curseur :

⇒ nom_curseur%ATTRIBUT

11.3. Exemple de curseur

Nous allons maintenant résumer, par un exemple, l'utilisation des curseurs.

Nous allons remplir une table « mail » destinée aux clients prévus pour un vol à destination des îles Marquises.

```
SQL> desc mail
Name                               Null?    Type
-----
NOM                                CHAR(20)
ADRESSE                            CHAR(80)
```

mailing.sql

```
DECLARE
  CURSOR curs_employes
  IS
  SELECT Nom_emp, adresse
  FROM employe
  WHERE adresse = v_adresse;

  v_nom_employe      client.Nom_cli%TYPE;
  v_adresse_employe  client.adresse%TYPE;
  v_adresse          varchar2(30) ;

BEGIN
  v_adresse := 'Marquises'
  OPEN curs_employes ;
  LOOP
    FETCH curs_employes into v_nom_employe, v_adresse_employe;
    EXIT WHEN curs_employes%NOTFOUND;
    INSERT INTO mail values (v_nom_employe,v_adresse_employe);
  END LOOP;
  CLOSE curs_employes;
END;
```



| /

Nous vous avons présenté l'utilisation d'un curseur paramétré dont la valeur de la variable `v_adresse` est égale à « Marquises » à l'ouverture de celui-ci.

Il est possible d'utiliser plusieurs variables dans la clause `WHERE` du curseur.



12. LES EXCEPTIONS

Le langage PL/SQL offre au développeur un mécanisme de gestion des exceptions. Il permet de préciser la logique du traitement des erreurs survenues dans un bloc PL/SQL. Il s'agit donc d'un point clé dans l'efficacité du langage qui permettra de protéger l'intégrité du système.

Il existe deux types d'exception :

- ⇒ interne,
- ⇒ externe.

Les exceptions internes sont générées par le moteur du système (division par zéro, connexion non établie, table inexistante, privilèges insuffisants, mémoire saturée, espace disque insuffisant, ...).

Les exceptions externes sont générées par l'utilisateur (stock à zéro, ...).

Le gestionnaire des exceptions du PL/SQL ne sait gérer que des erreurs nommées (noms d'exceptions). Par conséquent, toutes les exceptions doivent être nommées et manipulées par leur nom.

Les erreurs ORACLE générées par le noyau sont numérotées (ORA-xxxxx). Il a donc fallu établir une table de correspondance entre les erreurs ORACLE et des noms d'exceptions. Cette correspondance existe déjà pour les erreurs les plus fréquentes (*cf. tableau plus bas*). Pour les autres, il faudra créer une correspondance explicite.

Enfin, à chaque erreur ORACLE correspond un code SQL (`SQLCODE`) que l'on peut tester dans le langage hôte (PL/SQL, Pro*C, etc. ...).

- ⇒ Ce **code** est **nul** lorsque l'instruction se passe bien et **négatif** sinon.

Voici quelques exemples d'exceptions prédéfinis et des codes correspondants :

Nom d'exception	Erreur ORACLE	SQLCODE
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
STORAGE_ERROR	ORA-06500	-6500
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476



Signification des erreurs Oracle présentées ci-dessus :

- ♦ `CURSOR_ALREADY_OPEN` : tentative d'ouverture d'un curseur déjà ouvert..
- ♦ `DUP_VAL_ON_INDEX` : violation de l'unicité lors d'une mise à jour détectée au niveau de l'index unique.
- ♦ `INVALID_CURSOR` : opération incorrecte sur un curseur, comme par exemple la fermeture d'un curseur qui n'a pas été ouvert.
- ♦ `INVALID_NUMBER` : échec de la conversion d'une chaîne de caractères en numérique.
- ♦ `LOGIN_DENIED` : connexion à la base échouée car le nom utilisateur ou le mot de passe est invalide.
- ♦ `NO_DATA_FOUND` : déclenché si la commande `SELECT INTO` ne retourne aucune ligne ou si on fait référence à un enregistrement non initialisé d'un tableau PL/SQL.
- ♦ `NOT_LOGGED_ON` : tentative d'accès à la base sans être connecté.
- ♦ `PROGRAM_ERROR` : problème général dû au PL/SQL.
- ♦ `ROWTYPE_MISMATCH` : survient lorsque une variable curseur d'un programme hôte retourne une valeur dans une variable curseur d'un bloc PL/SQL qui n'a pas le même type.
- ♦ `STORAGE_ERROR` : problème de ressources mémoire dû à PL/SQL.
- ♦ `TIMEOUT_ON_RESOURCE` : dépassement du temps dans l'attente de libération des ressources (lié aux paramètres de la base).
- ♦ `TOO_MANY_ROWS` : la commande `SELECT INTO` retourne plus d'une ligne.
- ♦ `VALUE_ERROR` : erreur arithmétique, de conversion, de troncature, ou de contrainte de taille.
- ♦ `ZERO_DIVIDE` : tentative de division par zéro.

12.1. Implémenter des erreurs Oracle

12.1.1. *Exception ORACLE dont le nom est prédéfini*

Il peut s'agir d'exception dont le nom est prédéfini ou non.

Il suffit de préciser le traitement dans le module `EXCEPTION`. Aucune déclaration n'est nécessaire.

Exemple de gestion de l'exception prédéfinie `NO_DATA_FOUND` :

```
DECLARE
...Variables
BEGIN
...Traitements
EXCEPTION
```



```
        WHEN NO_DATA_FOUND THEN
            traitement;
    END;
/
```

12.1.2. *Exception ORACLE dont le nom n'est pas prédéfini*

Il va falloir créer une correspondance entre le code erreur ORACLE et le nom de l'exception.

Ce nom est choisi librement par le programmeur.

Un nommage évocateur est largement préconisé et il faut éviter les noms comme erreur 1023, ...

Pour affecter un nom à un code erreur, on doit utiliser une directive compilée (*pragma*) nommée `EXCEPTION_INIT`.

Le nom de l'exception doit être déclaré comme pour les exceptions utilisateurs.

Prenons l'exemple du code erreur -1400 qui correspond à l'absence d'un champ obligatoire.

```
DECLARE
    champ_obligatoire EXCEPTION;
    PRAGMA EXCEPTION_INIT (champ_obligatoire, -1400);

BEGIN
    ...

    EXCEPTION

    WHEN champ_obligatoire THEN
        traitement;
END;
```

Contrairement aux exceptions utilisateurs on ne trouve pas de clause `RAISE`. Le code SQL -1400 est généré automatiquement, et il en est donc de même de l'exception `champ_obligatoire` qui lui est associée.

Le module `EXCEPTION` permet également de traiter un code erreur qui n'est traité par aucune des exceptions. Le nom générique de cette erreur (exception) est `OTHERS`.

Dans la clause `WHEN OTHERS THEN ...` on pourra encapsuler le code erreur `SQLCODE`.

```
EXCEPTION
    WHEN exception1 THEN
        traitement1;
    WHEN exception2 THEN
        traitement2;
    WHEN OTHERS THEN
        traitement3;
```



12.2. Fonctions pour la gestion des erreurs

- ⇒ **SQLCODE** Renvoie le code de l'erreur courante (0 si OK <0 sinon)
- ⇒ **SQLERRM** (code_erreur) Renvoie le libellé correspondant au code erreur passé en argument.

En pratique, SQLERRM est toujours utilisé avec SQLCODE comme argument.

```
excep3.sql
set serveroutput on
DECLARE
    Ex
    EXCEPTION;
    v_emp_Id_emp      employe.Id_emp%TYPE;
    v_emp_nom         employe.Nom%TYPE;
    v_emp_adresse     employe.adresse%TYPE;
    message_erreur    VARCHAR2(30);
BEGIN
    SELECT Id_emp, Nom, adresse
    into
        v_emp_Id_emp,
        v_emp_Nom,
        v_emp_adresse
    FROM emp
    WHERE Id_emp = 6;

    IF v_emp_adresse IS NULL THEN
        RAISE ex_emp_sans_adresse;
    END IF;

EXCEPTION
    When NO_data_found then traitement ;

    WHEN ex_emp_sans_adresse THEN
        insert into emp_sans_adresse
        values (      v_emp_Id_emp, v_emp_nom);

    WHEN OTHERS THEN
        message_erreur := SUBSTR (SQLERRM(SQLCODE),1,30);
        dbms_output.put_line(message_erreur);

END;
/
```

Résultats :

```
SQL> @excep3
ORA-01403: no data found
PL/SQL procedure successfully completed.
```



12.3. Implémenter des exceptions utilisateurs

Principe général :

Déclarer chaque exception dans la partie `DECLARE`,
Préciser le déclenchement de l'exception (dans un bloc `BEGIN ... END`),
Définir le traitement à effectuer lorsque l'exception survient dans la partie `EXCEPTION`.

```
DECLARE
...
nom_erreur EXCEPTION;
...
BEGIN
...
IF (anomalie) THEN
    RAISE nom_erreur;
END IF;
...
EXCEPTION
    WHEN nom_erreur THEN
        traitement;
END;
```

Exemple

Nous allons sélectionner un employé et déclencher une exception si cet employé n'a pas d'adresse. Le traitement de l'exception consiste à remplir la table des employés sans adresse (table créée par ailleurs).

```
SQL> desc employe_sans_adresse
Name                               Null?    Type
-----
ID_EMP                             NUMBER
NOM_EMP                             CHAR(20)
```

```
SQL> select * from employe;

ID_EMP  NOM        ADRESSE                                CODEPOST  VILLE        TEL
-----
1        MOREAU     28 rue Voltaire                        75016     Paris        47654534
2        DUPOND     12 rue Gambetta                        92700     Colombes     42421256
3        DURAND     3 rue de Paris                         92600     Asnières     47935489
4        BERNARD    10 avenue d'Argenteuil                 92600     Asnières     47931122
5        BRUN
```



Contenu du script :

```
excep.sql

DECLARE
    ex_emp_sans_adresse EXCEPTION;
    v_emp_id_emp      employe.id_emp%TYPE;
    v_emp_nom         employe.nom %TYPE;
    v_emp_adresse     employe.adresse%TYPE;

BEGIN
    SELECT      Id_emp, Nom, adresse
    INTO        v_emp_id_emp, v_emp_nom, v_emp_adresse
    FROM employe
    WHERE destination = 'Marquises';

    IF v_emp_adresse IS NULL THEN
        RAISE ex_emp_sans_adresse;
    END IF;

EXCEPTION
    WHEN ex_emp_sans_adresse THEN
        insert into employe_sans_adresse values (v_emp_id_emp, v_emp_nom);

END;
/
```

Compilation et Vérification :

```
SQL> @excep
PL/SQL procedure successfully completed.
SQL> select * from employe_sans_adresse;

 ID_EMP NOM_EMP
-----
      5 BRUN
```




13. VARIABLES DE SUBSTITUTION

SQL permet l'utilisation de variables de substitution afin que les valeurs soient saisies par l'utilisateur qui exécute le script.

```
SQL> Select * from employe where nom_emp = &nom ;
Enter value for no: 'Antoine'
Old      1: select * from employe where nom_emp = &nom ;
New      1: select * from employe where nom_emp = 'Antoine' ;
```

OU

```
SQL> Select * from employe where nom_emp = '&nom' ;
Enter value for no: Antoine
Old      1: select * from employe where nom_emp = '&nom' ;
New      1: select * from employe where nom_emp = 'Antoine' ;
```

NOM	PRENOM
Antoine	Charly
Antoine	Lyly



14. EXEMPLES DE PROGRAMMES PL/SQL

Script insérant des lignes dans la table EMPLOYE afin de simuler de l'activité sur la base de données.

```
-- redirection de la sortie dans un fichier journal
SPOOL SimulerActivite.log

create sequence seq_emp
increment by 1
start with 30
order ;

prompt Tapez une touche pour continuer !
pause

set serveroutput on

-- simulation de l'activité
DECLARE
erreur_Oracle      varchar2(30) ;
v_nombre           INTEGER := 0;

BEGIN
FOR i IN 1..10000 LOOP
    insert into opdef.employe
    values (seq_emp.nextval, 'TOURNESOL', 1500, 'Professeur',1);
    select Max(id_emp) INTO v_nombre from opdef.employe;
    update opdef.employe set nom='Martin' where id_emp=v_nombre;
    COMMIT;
END LOOP;
dbms_output.put_line ( '-- insertions effectuées --') ;
SELECT COUNT(id_emp) INTO v_nombre FROM opdef.employe;
IF v_nombre > 20000 THEN
    DELETE FROM opdef.employe WHERE id_emp > 20 ;
    COMMIT;
END IF;

EXCEPTION
When NO_DATA_FOUND
    dbms_output.put_line ('Problème !') ;
WHEN DUP_VAL_ON_INDEX THEN
    NULL;
WHEN OTHERS THEN
    erreur_Oracle := substr (sqlerrm(sqlcode),1,30) ;
    dbms_output.put_line (erreur_Oracle) ;
    ROLLBACK;

END;
/
```



Afficher l'identifiant et la destination d'un numéro de vol saisi.

```
-- appel du package Oracle qui gère l'affichage
set serveroutput on

prompt saisir un numéro de vol
accept no_vol

DECLARE
erreur_Oracle varchar2(30) ;

/* création du type enregistrement typ_vol*/
type typ_vol is record
    ( v_novol    vol.no_vol%type ,
      v_dest     vol.destination%type
    ) ;

/* affectation du type typ_vol à v_vol*/
v_vol          typ_vol;

BEGIN
select no_vol, destination
into   v_vol.v_novol, v_vol.v_dest
from   vol
where  no_vol = &no_vol ;

/* affichage et mise en forme du résultat */
dbms_output.put_line (' vol : ' || v_vol.v_novol || ' *-* ' ||
                      ' destination : ' || v_vol.v_dest
                      ) ;

EXCEPTION
WHEN no_data_found
then dbms_output.put_line ( '-- ce vol n'existe pas --') ;
WHEN others
then erreur_Oracle := substr (sqlerrm(sqlcode),1,30) ;
      dbms_output.put_line (erreur_Oracle) ;

END ;
/
```



15. VARIABLES ET INSTRUCTIONS AVANCEES

15.1. Les structures

Définition d'une structure art_qtecom (article et quantité commandée)

```
/* création du type enregistrement typ_vol*/  
type typ_vol is record  
    ( v_novol      vol.no_vol%type ,  
      v_dest       vol.destination%type  
    ) ;
```

Déclaration d'une variable de type v_vol :

```
/* affectation du type typ_vol à v_vol*/  
v_vol typ_vol;
```

Accès aux membres de la structure :

```
v_vol.v_dest := 'Tahiti';
```

Source complet

```
DECLARE  
    type typ_vol is record  
        ( v_novol      vol.no_vol%type ,  
          v_dest       vol.destination%type  
        ) ;  
    v_vol typ_vol;  
BEGIN  
    v_vol.v_dest := 'Tahiti';  
END;  
/
```



15.2. Tables et tableaux

Un tableau en PL/SQL est en fait une table qui ne comporte qu'une seule colonne.

Cette colonne ne peut correspondre qu'à un type scalaire et non à un type composé (table ou record). La structure de données nécessaire pour gérer une table ORACLE nécessitera donc autant de types tables que la table possède de colonnes. **On pourra éventuellement regrouper ces tables à l'intérieur d'un RECORD.**

Une table PL/SQL est indexée par une clé de type `BINARY_INTEGER`.

Définition de deux types table contenant respectivement des références articles et des désignations.

```
TYPE tab_destination is TABLE of vol.destination%TYPE  
index by binary_integer;
```

Déclaration d'une variable de type `tab_destination` :

```
ma_destination      tab_destination;
```

Déclaration d'une variable permettant de balayer la table :

```
j      integer := 1;
```

Accès au premier poste de la table :

```
ma_destination(j) := 'Tahiti';
```



Utilisation de tables à l'intérieur d'un RECORD :

Exemple de RECORD défini à l'aide de tables :

```
DECLARE

TYPE tab_vol is table of vol.no_vol%TYPE
index by binary_integer;

TYPE tab_destination is table of vol.destination%TYPE
index by binary_integer;

TYPE rec_vol is RECORD
    (v_vol          tab_vol,
     v_destination  tab_destination);

mon_vol          rec_vol;
ind              integer ;

BEGIN
ind := 1 ;
mon_vol.v_vol(ind) := 2048;
mon_vol.v_destination(ind) := 'Tahiti';

END;
/
```

15.3. L'expression CASE

L'expression CASE, introduite en version 8.1.6, permet d'implémenter en SQL une logique de type IF...THEN...ELSE.

```
CASE expression
  WHEN expression_comparaison THEN expression_résultat
  [ ... ]
  [ ELSE expression_résultat_par_défaut ]
END
```

Toutes les expressions doivent être de même type.

Oracle recherche la première clause WHEN . . THEN, telle que expression est égale à expression_comparaison et retourne : l'expression_résultat associée

Si aucune clause WHEN...THEN ne vérifie cette condition et qu'une clause ELSE existe, Oracle retourne l'expression_résultat_par_défaut associée

NULL sinon



```
-- instruction CASE : première syntaxe
BEGIN
  FOR l IN (SELECT nom, sexe FROM employe WHERE emploi = 'Aviateur')
  LOOP
    CASE l.sexe
      WHEN 'M' THEN
        DBMS_OUTPUT.PUT_LINE('Monsieur ' || l.nom);
      WHEN 'F' THEN
        DBMS_OUTPUT.PUT_LINE('Madame ' || l.nom);
      ELSE
        DBMS_OUTPUT.PUT_LINE(l.nom);
    END CASE;
  END LOOP;
END;
/
```

Deuxième syntaxe

```
CASE
  WHEN condition THEN expression_résultat
  [ ... ]
  [ ELSE expression_résultat_par_défaut ]
END CASE
```

⇒ Même principes, que la première syntaxe, mais avec recherche de la première clause WHEN...THEN telle que condition est vraie.

La deuxième syntaxe offre plus de liberté dans les conditions ; la première syntaxe est limitée à des conditions d'égalité.

Par contre, pour des conditions d'égalité, la première syntaxe est plus performante, l'expression étant évaluée une seule fois.

Bien noter que l'instruction CASE génère une exception si aucune condition n'est vérifiée et qu'il n'y a pas de ELSE. S'il est normal qu'aucune condition ne soit vérifiée, et qu'il n'y a rien à faire dans ce cas là, il est possible de mettre une clause ELSE NULL; (l'instruction NULL ne faisant justement rien).

Par contre, dans les mêmes circonstances, l'expression CASE ne génère pas d'exception mais retourne une valeur NULL.

Performance

```
CASE expression
  WHEN expression1 THEN résultat1
  WHEN expression2 THEN résultat2
  ...
```



Ce document ne pourra être communiqué à un tiers ou reproduit, même partiellement, sans autorisation écrite préalable de l'auteur.

est plus performant que



```
CASE
WHEN expression = expression1 THEN résultat1
WHEN expression = expression2 THEN résultat2
...
```

L'expression CASE ne supporte pas plus de 255 arguments, chaque clause WHEN...THEN comptant pour 2.

```
-- Première syntaxe
SELECT
    nom,
    CASE sexe
        WHEN 'M' THEN 'Monsieur'
        WHEN 'F' THEN CASE marie
            WHEN 'O' THEN 'Madame'
            WHEN 'N' THEN 'Mademoiselle'
        END
        ELSE 'Inconnu'
    END CASE
FROM
    employe
/

--Deuxième syntaxe
SELECT
    nom,
    CASE sexe
        WHEN sexe = 'M' THEN 'Monsieur'
        WHEN sexe = 'F' AND marie = 'O' THEN 'Madame'
        WHEN sexe = 'F' AND marie = 'N' THEN 'Mademoiselle'
        ELSE 'Inconnu'
    END CASE
FROM
    employe
/
```

Il est possible d'avoir des instructions CASE imbriquées.

15.4. Expression GOTO

L'instruction GOTO effectue un branchement sans condition sur une autre commande de la même section d'exécution d'un bloc PL/SQL.

Le format d'une instruction GOTO est le suivant :

```
GOTO Nom_Etiquette ;

<<Nom_Etiquette>>
instructions
```



Nom_Etiquette est le nom d'une étiquette identifiant l'instruction cible

```
BEGIN
  GOTO deuxieme_affichage
  Dbms_output.putline('Cette ligne ne sera jamais affichée') ;
  <<deuxieme_affichage>>
  dbms_output.putline('Cette ligne sera toujours affichée') ;
END ;
/
```

Limites de l'instruction GOTO

L'instruction GOTO comporte plusieurs limites :

- ♦ L'étiquette doit être suivie d'au moins un ordre exécutable
- ♦ L'étiquette cible doit être dans la même portée que l'instruction GOTO, chacune des structures suivantes maintient sa propre portée : fonctions, procédures blocs anonymes, instruction IF, boucles LOOP, gestionnaire d'exceptions, instruction CASE.
- ♦ L'étiquette cible doit être dans la même partie de bloc que le GOTO



16. COMPLEMENT SUR LES CURSEURS

16.1. Curseur BULK COLLECT

Nous allons maintenant résumer, par un exemple, l'utilisation des tables remplies par un curseur.

Nous allons remplir un record de trois table avec un curseur en une seule opération grâce à l'instruction
⇒ BULK COLLECT.

```
SET SERVEROUTPUT ON

PROMPT SAISIR UN NUMERO D'EMPLOYE
ACCEPT NO_EMP

DECLARE
TYPE TAB_NOM IS TABLE OF EMPLOYE.NOM%TYPE
INDEX BY BINARY_INTEGER;
TYPE TAB_SALAIRE IS TABLE OF EMPLOYE.SALAIRE%TYPE
INDEX BY BINARY_INTEGER;
TYPE TAB_EMPLOI IS TABLE OF EMPLOYE.EMPLOI%TYPE
INDEX BY BINARY_INTEGER;

TYPE TYP_EMP IS RECORD
    (E_NOM          TAB_NOM,
     E_SALAIRE      TAB_SALAIRE,
     E_EMPLOI       TAB_EMPLOI
    );

E_EMP          TYP_EMP;
NUM_EMP        INTEGER :=1;
NB_EMP         INTEGER;

BEGIN
SELECT          NOM, SALAIRE, EMPLOI
BULK COLLECT INTO E_EMP.E_NOM, E_EMP.E_SALAIRE, E_EMP.E_EMPLOI
FROM            EMPLOYE
WHERE ID_EMP > &NO_EMP;

NB_EMP := E_EMP.E_NOM.COUNT;

LOOP
    EXIT WHEN NUM_EMP > NB_EMP;
    /* AFFICHAGE ET MISE EN FORME DU RESULTAT */
    DBMS_OUTPUT.PUT_LINE (' NOM : ' || E_EMP.E_NOM(NUM_EMP) || ' *-* '
||
        ' SALAIRE : ' || E_EMP.E_SALAIRE(NUM_EMP) || ' *-* ' ||
        ' EMPLOI : ' || E_EMP.E_EMPLOI(NUM_EMP));
    NUM_EMP := NUM_EMP + 1;
END LOOP;
END;
/
```



16.2. Curseur FOR LOOP

Ce type de curseur permet de charger la totalité des lignes d'une table très facilement dans un curseur.

Syntaxe :

```
FOR record_index in cursor_name  
LOOP  
  {...statements...}  
END LOOP;
```

Exemple 1

```
Declare  
  
BEGIN  
  
FOR record_emp IN (select * from employes) ;  
LOOP  
  DBMS_OUTPUT.PUT_LINE(record_emp.nom_emp || ' gagne'  
                        || record_emp.salaire) ;  
  
End LOOP ;  
END ;  
/
```

Exemple 2

```
Declare  
  CURSOR curs_employes  
  IS  
  SELECT Nom_emp, salaire  
  FROM employe  
;  
  
BEGIN  
  
FOR record_emp IN curs_employes;  
LOOP  
  DBMS_OUTPUT.PUT_LINE(record_emp.nom_emp || ' gagne'  
                        || record_emp.salaire) ;  
  
End LOOP ;  
END ;  
/
```



|

16.3. Variables curseur

Contrairement à un curseur, une variable curseur est dynamique car elle n'est pas rattachée à une requête spécifique.

Pour déclarer une variable curseur faire :

Etape 1 : définir un type REF CURSOR

```
DECLARE
type ref_type_nom is ref cursor return type_return ;
```

(type_return représente soit une ligne de table basée soit un record.

Etape 2 : déclarer une variable de type REF CURSOR

```
DECLARE
type emptytype is ref cursor return e_emp%rowtype ;
Emp_cs emptytype
```

Etape 3 : gérer une variable curseur

On utilise les commandes open-for, fetch, close pour contrôler les variables curseur.

```
BEGIN
OPEN {nom_var_curseur| :host_nom_var_curseur}
FOR ordre_select ;

FETCH nom_var_curseur
INTO var_hôte ;

CLOSE nom_curseur ;
```

Exemple

Utilisation d'un curseur référencé pour retourner des enregistrements choisis par l'utilisateur.

```
DECLARE
TYPE rec_type IS RECORD (client.nocli%TYPE ;
Enreg rec_type;
```



```
TYPE refcur IS REF CURSOR RETURN enreg%TYPE ;
Curref refcur ;

Enreg_emp      curref%ROWTYPE
V_choix NUMBER(1) := &choix

BEGIN
  If v_choix = 1 THEN
    OPEN curref FOR SELECT no, nom FROM e_client ;
  ELSIF v_choix = 2 THEN
    OPEN curref FOR SELECT no, nom FROM e_emp;
  ELSIF v_choix = 3 THEN
    OPEN curref FOR SELECT no, nom FROM e_service ;
  ELSIF v_choix = 4 THEN
    OPEN curref FOR SELECT no, nom FROM e_continet ;
  ELSIF v_choix = 5 THEN
    OPEN curref FOR SELECT no, nom FROM e_produit ;
  END IF ;
  LOOP
    EXIT WHEN v_choix NOT BETWEEN 1 AND 5;
    FETCH curref INTO enreg ;
    EXIT WHEN curref%NOTFOUND ;
    Dbms_output.put_line ('No : ' || enreg.no ||
                          'Nom' || enreg.nom) ;
  END LOOP ;
END;
/
```



17. INSTRUCTION “EXECUTE IMMEDIATE”

Cette commande permet de vérifier la syntaxe et d'exécuter de façon dynamique un ordre SQL ou un bloc anonyme PL/SQL.

```
Execute immediate chaine_dynamique  
[into {variables, ... | enregistrement}]  
[using [IN|OUT|IN OUT] argument ...]  
[{returning|return} INTO argument, ...]
```

En utilisant la clause USING, il n'est pas nécessaire de préciser le mode d'utilisation pour les paramètres entrants défini à IN par défaut.

Avec la clause RETURNING INTO, le mode d'utilisation des paramètres par défaut est OUT.



18. TRANSACTIONS AUTONOMES

Avant la version 8i du PL/SQL chaque session Oracle pouvait avoir au plus une transaction active à un instant donnée.

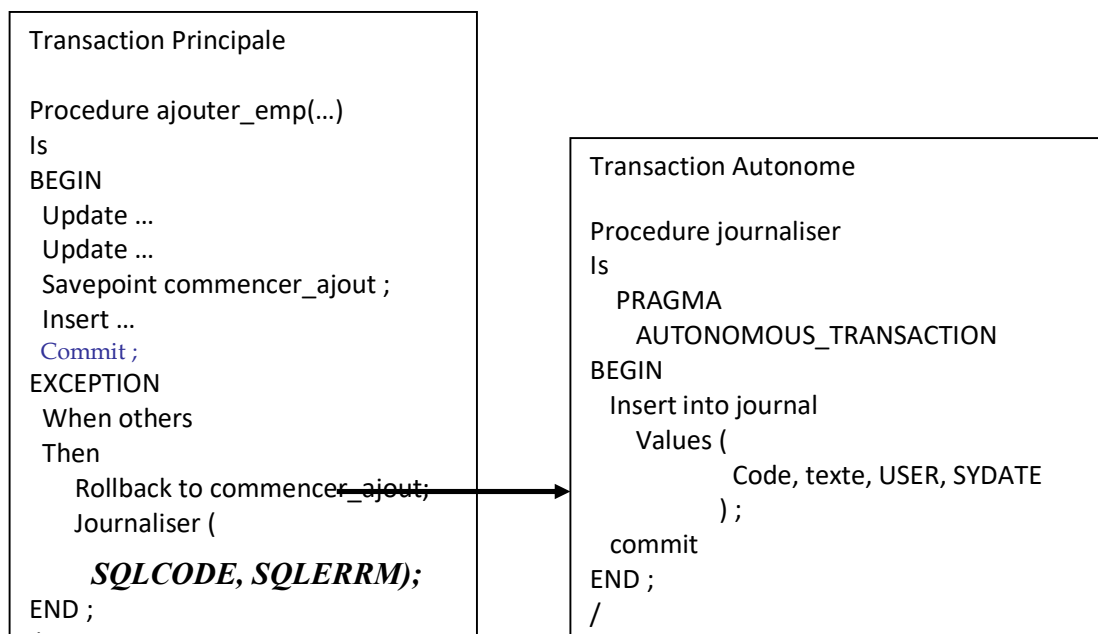
En d'autres termes toutes les modifications effectuées dans la session étaient soit totalement sauvegardées, soit totalement annulées.

A partir de la version 8i d'Oracle, il est possible d'exécuter et de sauvegarder ou d'annuler certains ordres DML (INSERT, UPDATE, DELETE) sans affecter la totalité de la transaction.

Lorsque vous définissez un bloc PL/SQL (bloc anonyme, fonction, procédure, procédure packagée, fonction packagée, trigger) comme une transaction autonome, vous isolez la DML de ce bloc du contexte de la transaction appelante.

Ce bloc devient une transaction indépendante démarrée par une autre transaction appelée transaction principale.

A l'intérieur du bloc de la transaction autonome, la transaction principale est suspendue. Vous pouvez effectuer vos opérations SQL, les sauvegarder ou les annuler, puis revenir à la transaction principale.





La transaction principale est suspendue pendant le déroulement de la transaction autonome.

La déclaration d'une transaction autonome se fait en utilisant le mot clé :

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

Cette directive demande au compilateur PL/SQL de définir un bloc PL/SQL comme autonome ou indépendant. Tous types de blocs cités ci-dessous peuvent être des transaction autonomes :

- ♦ Blocs PL/SQL de haut niveau (mais non imbriqués).
- ♦ Fonctions et procédures, définies dans un package ou autonomes
- ♦ Méthodes (fonctions et procédures) de type objet
- ♦ Triggers de base de données



Lors de l'exécution d'une transaction autonome il vous faut inclure dans le bloc PL/SQL une instruction `ROLLBACK` ou `COMMIT` afin de gérer la fin de transaction.

Cette directive doit apparaître dans la section de déclaration des variables.
En générale ces directives sont placées en début de section de déclaration.

Les modifications des transactions autonomes sont visibles par la transaction principale même si la transaction principale qui a appelé la transaction autonome n'est pas terminée.
Pour que la transaction principale ne puisse pas connaître les modifications apportées par la transaction autonome, il faut préciser un niveau d'isolation de la transaction à l'aide de la commande :

⇒ `Set transaction isolation level serializable`

L'instruction « `PRAGMA AUTONOMOUS_TRANSACTION` » permet de « commiter » une requête avant ou sans que la procédure appelante ne soit « commitée » également.

- ♦ La notion de transaction autonome n'a rien à voir avec la simultanéité ou la notion de parallélisme.



Lorsque des fonctions « autonomes » sont appelées, de nouvelles transactions sont ouvertes lors de chaque appel, concernant le code de la fonction.

Il n'y aura pas de changement concernant le parallélisme ou la simultanéité. Ajouter ou non ce « pragma » ne change rien à cela, par contre cela changera la visibilité des informations vues par les transactions autonomes.

En fait, il n'y a pas d'intérêt à utiliser ce « pragma » dans une fonction appelée dans un select. Il sert principalement à faire des routines de logs au sein d'un traitement transactionnel.

Si des fonctions sont en « autonomous transaction », cela veut simplement dire qu'elles ne verront pas les données modifiées (non commitées) de la transaction en cours, comme si c'était une autre session qui lisait les données.

Exemple

```
SQL> CREATE TABLE test      AS SELECT 1 a FROM dual;
TABLE created.

SQL> CREATE FUNCTION testf RETURN number IS
2          PRAGMA AUTONOMOUS_TRANSACTION;
3          r number;
4          begin
5              SELECT a INTO r FROM test WHERE rownum=1;
6              commit;
7          RETURN r;
8          end;
9  /

FUNCTION created.

SQL> SELECT a,testf FROM test;

   A    TESTF
-----
   1         1

SQL> UPDATE test SET a=2;
1 row updated.

SQL> SELECT a,testf FROM test;

   A    TESTF
-----
   2         1
```

La fonction ne voit pas le résultat de l'update de la transaction courante.



Vous définirez une transaction autonome dès que vous voudrez isoler les modifications faites du contexte de la transaction appelante.

Voici quelques idées d'utilisation :

⇒ Mécanisme de journalisation

D'une part vous voulez stocker une erreur dans une table de journalisation, d'autre part suite à cette erreur vous voulez annuler la transaction principale. Et vous ne voulez pas annuler les autres entrées de journal.

⇒ Commit et Rollback dans vos triggers de base de données

Si vous définissez un trigger comme une transaction autonome, alors vous pouvez exécuter un commit et un rollback dans ce code.

⇒ Compteur de tentatives de connexion.

Supposons que vous vouliez laisser un utilisateur essayer d'accéder à une ressource N fois avant de lui refuser l'accès ; vous voulez également tracer les tentatives effectuées entre les différentes connexions à la base. Cette persistance nécessite un `COMMIT`, mais qui resterait indépendant de la transaction.

⇒ Fréquence d'utilisation d'un programme

Vous voulez tracer le nombre d'appels à un même programme durant une session d'application. Cette information ne dépend pas de la transaction exécutée par l'application et ne l'affecte pas.

⇒ Composants d'applications réutilisables

Cette utilisation démontre tout l'intérêt des transactions autonomes. A mesure que nous avançons de le monde d'internet , il devient plus important de pouvoir disposer d'unités de travail autonomes qui exécutent leur tâches sans effet de bord sur l'environnement appelant.

Règles et limites des transactions autonomes

Un ensemble de règles sont à suivre lors de l'utilisation des transactions autonomes :

- ◆ Seul un bloc anonyme de haut niveau peut être transformé en transaction autonome.
- ◆ Si une transaction autonome essaie d'accéder à une ressource gérée par la transaction principale, un `DEADLOCK` peut survenir (la transaction principale ayant été suspendue).
- ◆ Vous ne pouvez pas marquer tous les modules d'un package comme autonomes avec une seule déclaration `PRAGMA`.
- ◆ Pour sortir sans erreur d'une transaction autonome, vous devez exécuter un commit ou un rollback explicite. En cas d'oubli vous déclencherez l'exception :
- ◆ `ORA-06519` : transaction autonome active détectée et annulée.
- ◆ Dans une transaction autonome vous ne pouvez pas effectuer un `ROLLBACK` vers un point de sauvegarde (`SAVEPOINT`) défini dans la transaction principale. Si vous essayez de le faire vous déclencherez l'exception suivante :



- ♦ ORA-01086 : le point de sauvegarde 'votre point de sauvegarde' n'a jamais été établi
- ♦ Le paramètre `TRANSACTION` du fichier d'initialisation, précise le nombre maximum de transactions concurrentes autorisées dans une session . En cas de dépassement de cette limite vous recevrez l'exception suivante :
- ♦ ORA-01574 : nombre maximum de transactions concurrentes dépassé.



19. COMPILEATION NATIVE DU CODE PL/SQL

Avec Oracle, les unités de programmes écrites en PL/SQL peuvent être compilées en code natif stocké dans des bibliothèques partagées :

- ⇒ Les procédures sont traduites en C, compilées avec un compilateur C standard et liées avec Oracle

Surtout intéressant pour du code PL/SQL qui fait du calcul intensif :

- ⇒ Pas pour du code qui exécute beaucoup de requêtes SQL

Mise en oeuvre

Modifier le fichier *make file* fourni en standard pour spécifier les chemins et autres variables du système. S'assurer qu'un certain nombre de paramètres sont correctement positionnés (au niveau de l'instance ou de la session) :

- ♦ `PLSQL_COMPILER_FLAGS` : indicateurs de compilation ; mettre la valeur `NATIVE` (`INTERPRETED` par défaut)
- ♦ `PLSQL_NATIVE_LIBRARY_DIR` : répertoire de stockage des bibliothèques générées lors de la compilation native
- ♦ `PLSQL_NATIVE_MAKE_UTILITY` : chemin d'accès complet vers l'utilitaire de *make*
- ♦ `PLSQL_NATIVE_MAKE_FILE_NAME` : chemin d'accès complet vers le fichier *make file*

Compiler le ou les programme(s) désirés.

Vérifier le résultat dans la vue `DBA_STORED_SETTING` (ou consœurs `USER_` et `ALL_`) du dictionnaire.

Le make file par défaut est :

- ⇒ `$ORACLE_HOME/plsql/spnc_makefile.mk`.



20. PROCEDURES, FONCTIONS ET PACKAGES

- ⇒ Une **procédure** est une unité de traitement qui contient des commandes SQL relatives au langage de manipulation des données, des instructions PL/SQL, des variables, des constantes, et un gestionnaire d'erreurs.
- ⇒ Une **fonction** est une procédure qui retourne une valeur.
- ⇒ Un **package** est un agrégat de procédures et de fonctions.

Les packages, procédures, ou fonctions peuvent être appelés depuis toutes les applications qui possèdent une interface avec ORACLE (SQL*PLUS, Pro*C, SQL*Forms, ou un outil client particulier comme NSDK par exemple).

Les procédures (fonctions) permettent de :

- ♦ Réduire le trafic sur le réseau (les procédures sont locales sur le serveur)
- ♦ Mettre en oeuvre une architecture client/serveur de procédures et rendre indépendant le code client de celui des procédures (à l'API près)
- ♦ Masquer la complexité du code SQL (simple appel de procédure avec passage d'arguments)
- ♦ Mieux garantir l'intégrité des données (encapsulation des données par les procédures)
- ♦ Sécuriser l'accès aux données (accès à certaines tables seulement à travers les procédures)
- ♦ Optimiser le code (les procédures sont compilées avant l'exécution du programme et elles sont exécutées immédiatement si elles se trouvent dans la SGA (zone mémoire gérée par ORACLE). De plus une procédure peut être exécutée par plusieurs utilisateurs.

Les packages permettent de regrouper des procédures ou des fonctions (ou les deux). On évite ainsi d'avoir autant de sources que de procédures. Le travail en équipes et l'architecture applicative peuvent donc plus facilement s'organiser du côté serveur, où les packages regrouperont des procédures à forte cohésion intra (Sélection de tous les articles, Sélection d'un article, Mise à jour d'un article, Suppression d'un article, Ajout d'un article). Les packages sont ensuite utilisés comme de simples bibliothèques par les programmes clients. Mais attention, il s'agit de bibliothèques distantes qui seront *processées* sur le serveur et non en locale (client/serveur de procédures).

Dans ce contexte, les équipes de développement doivent prendre garde à ne pas travailler chacune dans « leur coin ». Les développeurs ne doivent pas perdre de vue la logique globale de l'application et les scénarios d'activité des opérateurs de saisie. A l'extrême, on peut finir par coder une procédure extrêmement sophistiquée qui n'est sollicitée qu'une fois par an pendant une seconde. Ou encore, une gestion complexe de verrous pour des accès concurrent qui n'ont quasiment jamais lieu.



Ce document ne pourra être communiqué à un tiers ou reproduit, même partiellement, sans autorisation écrite préalable de l'auteur.



20.1. Procédures

Les procédures ont un ensemble de paramètres modifiables en entrée et en sortie.

20.1.1. *Créer une procédure*

Syntaxe générale :

```
procedure_general.sql
```

```
create or replace procedure nom_procedure  
(liste d'arguments en INPUT ou OUTPUT) is
```

déclaration de variables, de constantes, ou de curseurs

```
begin  
    ...  
    ...  
  
exception  
    ...  
    ...  
  
end;  
/
```

Au niveau de la liste d'arguments :

- les arguments sont précédés des mots réservés **IN**, **OUT**, ou **IN OUT**,
- ils sont séparés par des virgules,
- le mot clé **IN** indique que le paramètre est passé en entrée,
- le mot clé **OUT** indique que le paramètre est modifié par la procédure,
- on peut cumuler les modes **IN** et **OUT**
(cas du nombre de lignes demandées à une procédure d'extraction).

Contrairement au PL/SQL la zone de déclaration n'a pas besoin d'être précédé du mot réservé **DECLARE**. Elle se trouve entre le **IS** et le **BEGIN**.

La dernière ligne de chaque procédure doit être composée du seul caractère **/** pour spécifier au moteur le déclenchement de son exécution.



Exemple de procédure qui modifie le salaire d'un employé.

Arguments : Identifiant de l'employée, Taux

modifie_salaire.sql

```
create procedure modifie_salaire
(id in number, taux in number) is
V_var number;
begin
    update employe set salaire=salaire*(1+taux)
    where Id_emp= id;
exception
    when no_data_found then
        raise_application_error (-20010,'Employé inconnu :'||to_char(id));
end;
/
```

Compilation de la procédure modifie_salaire

Il faut compiler le fichier sql qui s'appelle ici modifie_salaire.sql (attention dans cet exemple le nom du script correspond à celui de la procédure, c'est bien le nom du script sql qu'il faut passer en argument à la commande start).

```
SQL> start modifie_salaire
Procedure created.
```

Appel de la procédure modifie_salaire

```
SQL> begin
2  modifie_salaire (15,-0.5);
3  end;
4  /
PL/SQL procedure successfully completed.
```

L'utilisation d'un script qui contient les 4 lignes précédentes est bien sûr également possible :

demarre.sql

```
begin
modifie_salaire (15,-0.5);
end;
```



Ce document ne pourra être communiqué à un tiers ou reproduit, même partiellement, sans autorisation écrite préalable de l'auteur.

|/



Lancement du script demarre.sql :

```
SQL> start demarre
PL/SQL procedure successfully completed.
```

20.1.2. *Modifier une procédure*

La **clause REPLACE** permet de remplacer la procédure (fonction) si elle existe déjà dans la base :

```
create or replace procedure modifie_salaire
...
```

Par défaut on peut donc toujours la spécifier.

20.1.3. *Correction des erreurs*

Si le script contient des erreurs, la commande **show err** permet de visualiser les erreurs.

Pour visualiser le script global : commande **l** (lettre l)

pour visualiser la ligne 4 : commande **l4**

pour modifier le script sous **VI** sans sortir de sa session SQL*PLUS commande **!vi modifie_salaire.sql**

```
SQL> start modifie_salaire

Warning: Procedure created with compilation errors.

SQL> show err
Errors for PROCEDURE MODIFIE_SALAIRE:

LINE/COL ERROR
-----
4/8      PLS-00103: Encountered the symbol "VOYAGE" when expecting one of the
following:
         := . ( @ % ;
         Resuming parse at line 5, column 21.

SQL> 14

4*      update employe set salaire=salaire*(1+taux)
```



Autre exemple de procédure qui modifie le salaire de chaque employé en fonction de la valeur courante du salaire :

Arguments : aucun



Cette procédure utilise deux variables locales (ancien_salaire, nouveau_salaire) et un curseur (curs1).

```
maj_salaire.sql
create or replace procedure salaire is
    CURSOR curs1 is
        select salaire
        from employe
        for update;
    ancien_salaire number;
    nouveau_salaire number;

BEGIN
    OPEN curs1;
    LOOP
        FETCH curs1 into ancien_salaire;
        EXIT WHEN curs1%NOTFOUND;
        if ancien_salaire >=0 and ancien_salaire <= 50
        then nouveau_salaire := 4*ancien_salaire;
        elsif ancien_salaire >50 and ancien_salaire <= 100
            then nouveau_salaire := 3*ancien_salaire;
            else  nouveau_salaire = :ancien_salaire;
        end if;
        update employe set salaire = nouveau_salaire
            where current of curs1;
    END LOOP;
    CLOSE curs1;
END;
/
```

Si l'on ne désire pas faire de procédure mais créer une commande SQL (procédure anonyme) qui réalise l'action précédente, il suffit de commencer le script par :

```
DECLARE
    CURSOR curs1 is
```

....

Dès la compilation le script est exécuté.



20.2. Fonctions

Une fonction est une procédure qui retourne une valeur. La seule différence syntaxique par rapport à une procédure se traduit par la présence du mot clé `RETURN`.

Une fonction précise le type de donnée qu'elle retourne dans son prototype (signature de la fonction).

Le retour d'une valeur se traduit par l'instruction `RETURN (valeur)`.

20.2.1. *Créer une fonction*

Syntaxe générale :

```
fonction_general.sql
create or replace function nom_fonction
(liste d'arguments en INPUT ou OUTPUT)
return type_valeur_retour
is
déclaration de variables, de constantes, ou de curseurs
begin
    ...
    return (valeur);
exception
    ...
end;
/
```



Exemple de fonction qui retourne la moyenne des salaires des employés par bureau (regroupé par responsable du bureau) .

- Argument : Identifiant de l'employé
- Retour : moyenne du bureau

```
ca.sql

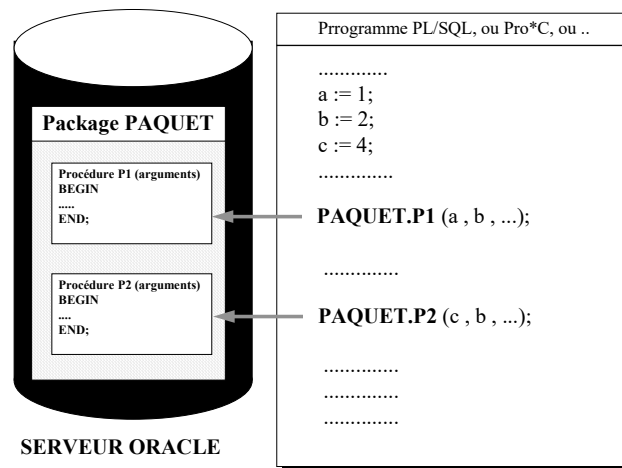
create or replace function moyenne_salaire (v_Id_employe IN NUMBER)
return NUMBER
is
valeur NUMBER;

begin

select avg(salaire)
into valeur
from employe
groupe by emp_id_emp
having emp_id_emp=v_id_emp;
return (valeur);
end;
/
```

20.3. Packages

Comme nous l'avons vu un package est un ensemble de procédures et de fonctions. Ces procédures pourront être appelées depuis n'importe quel langage qui possède une interface avec ORACLE (Pro*C, L4G, ...).



Principe général d'utilisation d'un package



La structure générale d'un package est la suivante :

```
package_general.sql
```

```
CREATE OR REPLACE PACKAGE nom_package IS  
  
définitions des types utilisés dans le package;  
prototypes de toutes les procédures et fonctions du package;  
  
END nom_package;  
/  
  
CREATE OR REPLACE PACKAGE BODY nom_package IS  
  
déclaration de variables globales;  
  
définition de la première procédure;  
  
définition de la deuxième procédure;  
  
etc. ...  
  
END nom_package;  
/
```

Un package est composé d'un en tête et d'un corps :

- ⇒ L'en tête comporte les types de données définis et les prototypes de toutes les procédures (fonctions) du package.
- ⇒ Le corps correspond à l'implémentation (définition) des procédures (fonctions).

Le premier **END** marque la fin de l'en tête du package. Cette partie doit se terminer par **/** pour que l'en tête soit compilé.

Ensuite le corps (body) du package consiste à implémenter (définir) l'ensemble des procédures ou fonctions qui le constitue. Chaque procédure est définie normalement avec ses clauses **BEGIN ... END**.

Le package se termine par **/** sur la dernière ligne.



Exemple

Package comportant deux procédures (augmentation de salaire et suppression de vendeur) :

paquet1.sql

```
create or replace package ges_emp is

procedure augmente_salaire (v_Id_emp in number,
                           v_taux_salaire in number);

function moyenne_salaire (v_Id_employe IN NUMBER)
return NUMBER;

end ges_emp;
/

create or replace package body ges_emp is

procedure augmente_salaire (v_Id_emp in number,
                           v_taux_salaire in number)
is
begin
    update employe set salaire= salaire * v_taux_salaire
    where Id_emp= v_Id_emp;
    commit;

end augmente_salaire;

function moyenne_salaire (v_Id_employe IN NUMBER)
return NUMBER
is
valeur NUMBER;

begin

select avg(salaire)
into valeur
from employe
groupe by emp_id_emp;
return (valeur);
end moyenne_salaire;

end ges_emp;
/
```



Compilation

```
SQL> @paquet1  
  
Package created.  
  
Package body created.
```

Exécution

Exécution de la procédure augmente_salaire du package ges_emp

```
SQL> begin  
2  ges_emp.augmente_salaire(4,50);  
3  end;  
4  /  
  
PL/SQL procedure successfully completed.
```

Tests

```
SQL> select * from EMPLOYEE;
```

ID_EMP	NOM	SALAIRE	EMPLOI	EMP_ID_EMP
1	Gaston	1700	Directeur	
2	Spirou	2000	Pilote	1
3	Titeuf	1800	Stewart	2
4	Marilyne	4000	Hotesse de l'Air	1

Nous constatons que Marilyne a un salaire de 4000 euros alors qu'il était de 2000 euros.



21. TRIGGERS

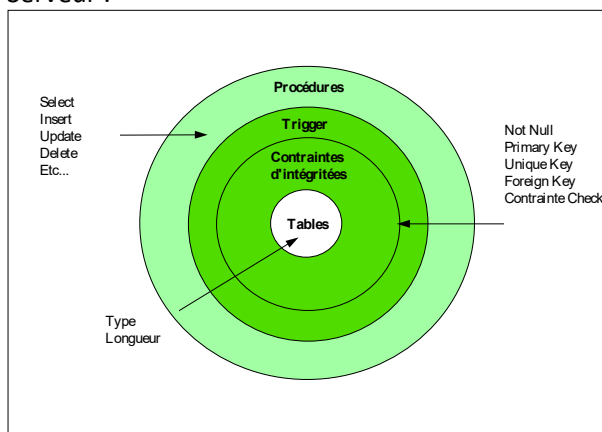
Un trigger permet de spécifier les réactions du système d'information lorsque l'on « touche » à ses données. Concrètement il s'agit de définir un traitement (un bloc PL/SQL) à réaliser lorsqu'un événement survient.

Les événements sont de six types (dont trois de base) et ils peuvent porter sur des tables ou des colonnes :

- ⇒ BEFORE INSERT
- ⇒ AFTER INSERT
- ⇒ BEFORE UPDATE
- ⇒ AFTER UPDATE
- ⇒ BEFORE DELETE
- ⇒ AFTER DELETE

Pour bien situer le rôle et l'intérêt des TRIGGERS, nous présentons ici une vue générale des contraintes sur le

Serveur :



Vue générale des contraintes

Les TRIGGERS permettent de :

- ⇒ renforcer la cohérence des données d'une façon transparente pour le développeur,
- ⇒ mettre à jour automatiquement et d'une façon cohérente les tables (éventuellement en déclenchant d'autres TRIGGERS).



Rappelons que les contraintes d'intégrité sont garantes de la cohérence des données (pas de ligne de commande qui pointe sur une commande inexistante, pas de code postal avec une valeur supérieur à 10000, pas de client sans nom, etc. ...).



Les TRIGGERS et les contraintes d'intégrité ne sont pas de même nature même si les deux concepts sont liés à des déclenchements implicites.

Un trigger s'attache à définir un traitement sur un événement de base comme « Si INSERTION dans telle table alors faire TRAITEMENT ». L'intérêt du TRIGGER est double. Il s'agit d'une part de permettre l'encapsulation de l'ordre effectif (ici INSERTION) de mise à jour de la base, en vérifiant la cohérence de l'ordre. D'autre part, c'est la possibilité d'automatiser certains traitements de mise à jour en cascade.

Les traitements d'un TRIGGER (insert, update, delete) peuvent déclencher d'autres TRIGGERS ou solliciter les contraintes d'intégrité de la base qui sont les « derniers gardiens » de l'accès effectif aux données.

21.1. Créer un trigger

Principes généraux

Lorsque l'on crée un TRIGGER on doit préciser les événements que l'on désire gérer.

On peut préciser si l'on désire exécuter les actions du TRIGGER pour chaque ligne mise à jour par la commande de l'événement ou non (option **FOR EACH ROW**). Par exemple, dans le cas d'une table que l'on vide entièrement (delete from table), l'option **FOR EACH ROW** n'est pas forcément nécessaire.

Lorsque l'on désire mémoriser les données avant (la valeur d'une ligne avant l'ordre de mise à jour par exemple) et les données après (les arguments de la commande) on utilisera les mots réservés :

⇒ OLD et NEW.

Exemple

Nous allons prendre l'exemple d'un trigger qui met automatiquement à jour les voyages (colonne qte de la table voyage) lorsque l'on modifie (création, modification, suppression) les commandes de nos clients (table ligne_com).



trigger.sql

```
CREATE OR REPLACE TRIGGER modif_voyage
AFTER DELETE OR UPDATE OR INSERT ON ligne_com
FOR EACH ROW

DECLARE
quantite_voyage  voyage.qte%TYPE;

BEGIN
IF DELETING THEN
/* on met à jour la nouvelle quantite de voyages annuel par employe */
UPDATE voyage SET
qte = qte + :OLD.qte
WHERE Id_voyage = :OLD.Id_voyage;
END IF;

IF INSERTING THEN
UPDATE voyage SET
qte = qte - :NEW.qte
WHERE Id_voyage = :NEW.Id_voyage;
END IF;

IF UPDATING THEN
UPDATE voyage SET
Qte = qte + (:OLD.qte - :NEW.qte)
WHERE Id_voyage = :NEW.Id_voyage;
END IF;

END;
/
```

Nous allons tester ce trigger dans l'environnement suivant :

Etat des voyages avant mise à jour :

```
SQL> select * from article where Id_voyage between 1 and 4;
```

ID_VOYAGE	DESIGNATION	PRIXUNIT	QTE
2	Tahiti	1330	10
4	Marquises	1350	10

Etat des commandes avant mise à jour :

```
SQL> select * from ligne_com where Id_voyage between 2 and 4;
```

NO_COMM	NUMLIGNE	ID_VOYAGE	QTECOM
1	1	2	10
1	2	4	10
2	1	2	4



Test 1

```
SQL> delete from ligne_com where No_comm=2 and No_ligne=1;
1 row deleted.
```

Vérifions que le traitement associé au trigger s'est bien effectué :

```
SQL> select * from voyage where Id_voyage between 1 and 4;

ID_VOYAGE  DESIGNATION          PRIXUNIT  QTE
-----
          2  Tahiti              1330      14
          4  Marquises           1350      10
```

Le voyage 2 a augmenté de la quantité libérée par la ligne de commande supprimée.

Test2

```
SQL> update ligne_com set qtecom=2 where No_comm=1 and No_ligne=2;
1 row updated.
```

```
SQL> select * from voyage where Id_voyage between 1 and 4;

ID_VOYAGE  DESIGNATION          PRIXUNIT  QTE
-----
          2  Tahiti              1330      14
          4  Marquises           1350      18
```

Le voyage N° 4 a augmenté de la différence (10 - 2 = 8) entre la nouvelle quantité commandée et l'ancienne.

La commande rollback scrute toutes les transactions non encore commitées. Par conséquent, elle porte à la fois sur les commandes explicites que nous avons tapées mais également sur les traitements du trigger.

```
SQL> rollback;
Rollback complete.
```

Etat des voyages après le rollback :

```
SQL> select * from voyage where Id_voyage between 2 and 4;

ID_VOYAGE  DESIGNATION          PRIXUNIT  QTE
-----
          2  Tahiti              1330      10
          4  Marquises           1350      10
```



Etat des commandes après le rollback :

```
SQL> select * from ligne_com where Id_voyage between 2 and 4;
```

NO_COMM	NUMLIGNE	ID_VOYAGE	QTECOM
1	1	2	10
1	2	4	10
2	1	2	4

21.2. Activer un trigger

Un trigger peut être activé ou désactivé respectivement par les clauses `ENABLE` et `DISABLE`.

```
SQL> alter trigger modif_voyage disable;
```

Trigger altered.

21.3. Supprimer un Trigger

Comme pour tout objet ORACLE :

```
SQL> drop trigger modif_voyage;
```

Trigger dropped.



Une instruction commit ou rollback valide ou annule toutes les transactions implicites créées par un trigger.
A partir de la version 10g il est possible de coder une instruction commit ou rollback dans un trigger.



21.4. Triggers rattaché aux vues

Jusqu'à la version 7 d'Oracle, les triggers ne pouvaient être rattachés qu'aux tables.

A partir de la version 8 d'Oracle il est possible de créer des triggers sur les vues en utilisant le mot clé : `INSTEAD`.

```
create or replace trigger tr_voyage
instead of delete on v_voyage
for each row
begin
delete from voyage
where id_voyage = :old.id_voyage;
dbms_output.put_line('le trigger fonctionne !!!');
end;
/
```

Comme nous le voyons ce trigger ne s'écrit pas tout à fait de la même manière que les précédents, sont comportement par contre reste le même.

Lorsque le trigger est déclenché par événement, la table à partir de laquelle est construite la vue est modifiée tout comme la vue.

21.5. Triggers sur événements systèmes

A partir de la version 8 d'Oracle, il est possible d'écrire des Triggers rattachés à des événements systèmes d'administration de base de données.

Ils se déclenchent sur un événement tels que :

- ⇒ After startup
- ⇒ Before shutdown
- ⇒ After servererror
- ⇒ After logon
- ⇒ Befor logoff
- ⇒ {before|after} alter
- ⇒ {before|after} analyse
- ⇒ {before|after} {audit|noaudit}
- ⇒ {before|after} {comment|DDL|drop|rename|truncate}
- ⇒ {before|after} create
- ⇒ {before|after} grant
- ⇒ {before|after} revoke



Ce document ne pourra être communiqué à un tiers ou reproduit, même partiellement, sans autorisation écrite préalable de l'auteur.



```
create table log_actions
(timestamp date,sysevent varchar2(20),
 login_user varchar2(30), instance_num number,
 database_name varchar2(50), dictionary_obj_type
 varchar2(20),
 dictionary_obj_name varchar2(30),
 dictionary_obj_owner varchar2(30),
 server_error number );

create or replace trigger on_shutdown
befor shutdown on database
begin
insert into log_actions
(timestamp,sysevent,login_user,server_error,database_name)
values
(sysdate,ora_sysevent,ora_login_user,ora_server_error(1),ora_database_name)
;
end;
/
```



22. ARCHITECTURE DU MOTEUR PL/SQL

Dans ce chapitre sont expliqués quelques termes utilisés représentant la façon dont Oracle traite le code source PL/SQL.

⇒ **Compilateur PL/SQL**

Composant Oracle qui analyse le code source P/SQL en vérifiant la syntaxe, résoud les noms, vérifie la sémantique et génère 2 formes binaires de code stocké.

⇒ **DIANA**

DIANA (Distributed Intermediate Annotated Notation pour Ada), c'est une forme intermédiaire du PL/SQL et de ses dépendances générée par le compilateur. Il effectue des analyses sémantiques et syntaxiques. inclut une spécification d'appel de vos objets stockés (données d'entête du programme, noms de paramètres, séquences, position et type des données).

⇒ **Pseudo-code Binaire**

Forme exécutable du PL/SQL compilé désigné parfois sous le terme de « *mcode* » ou « *Pcode* ». Ce pseudo code est équivalent à un fichier objet.

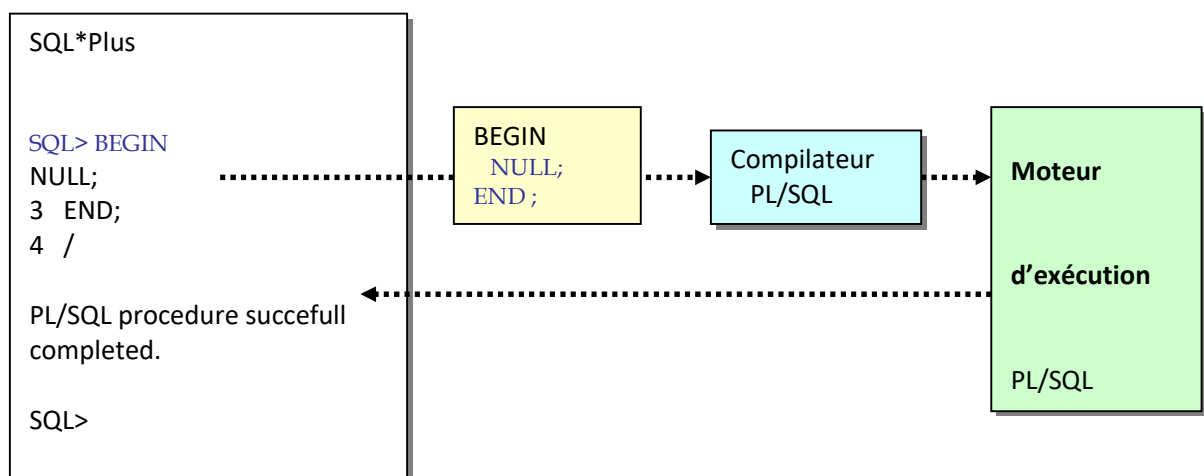
⇒ **Moteur d'exécution PL/SQL**

Machine virtuelle PL/SQL, qui exécute le pseudo code binaire d'un programme PL/SQL. Il effectue les appels nécessaires au moteur SQL du serveur et renvoie les résultats à l'environnement d'appel.

⇒ **Session Oracle**

Coté serveur c'est l'ensemble des processus et l'espace mémoire partagé associé à un utilisateur. Celui-ci est authentifié via une connexion réseau ou inter-processus. Chaque session a sa propre zone mémoire dans laquelle elle peut gérer les données du programme s'exécutant.

Regardons l'exécution d'un bloc anonyme qui ne fait rien présentée ci-dessous :





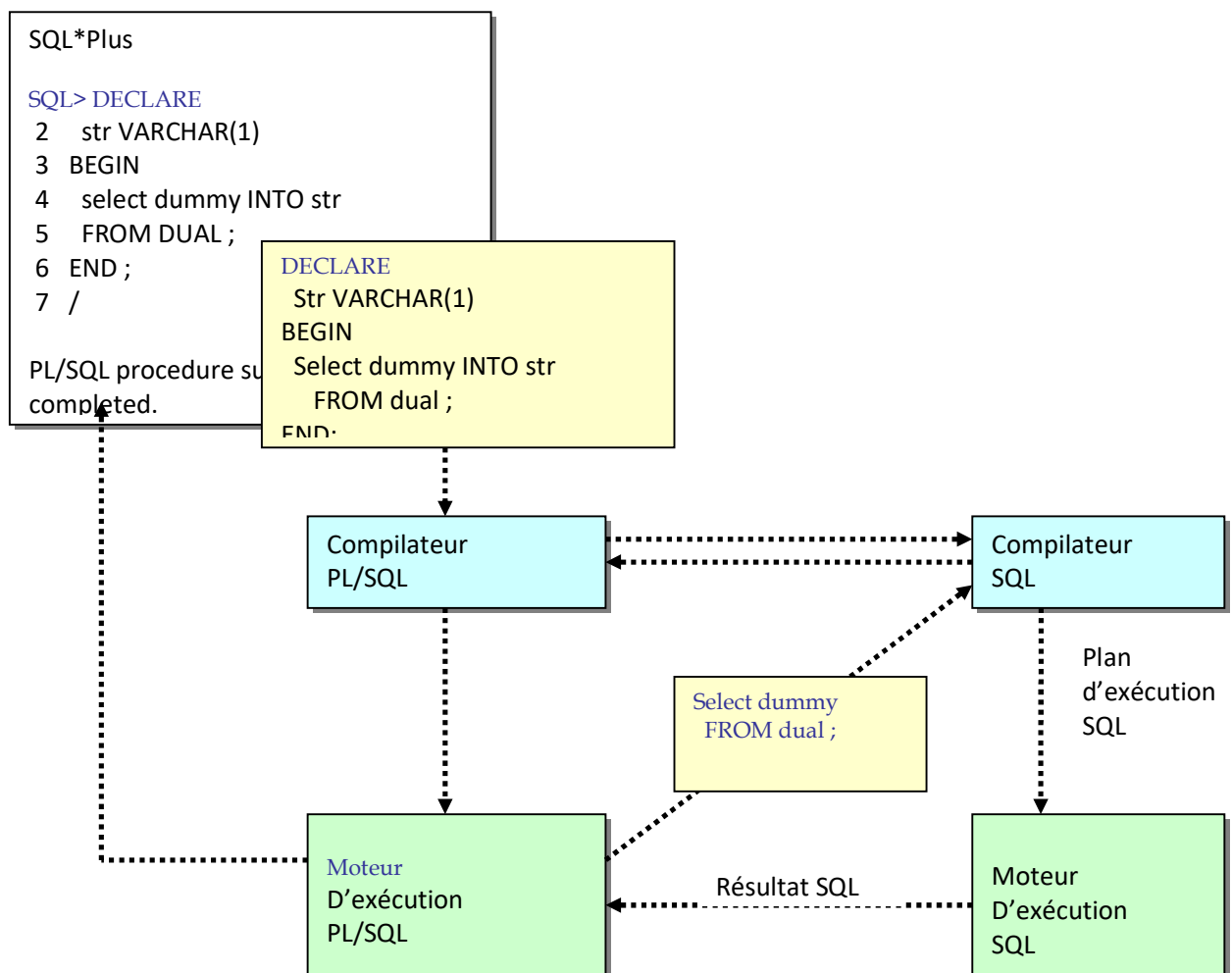
Ce document ne pourra être communiqué à un tiers ou reproduit, même partiellement, sans autorisation écrite préalable de l'auteur.



Déroulement de l'exécution :

- ⇒ L'utilisateur compose le bloc ligne et envoie à SQL*Plus une commande de départ.
- ⇒ SQL*Plus transmet l'intégralité du bloc à l'exception du « / » au compilateur PL/SQL.
- ⇒ Le compilateur PL/SQL essaie de compiler le bloc anonyme et crée des structures de données internes pour analyser le code et gérer du pseudo code binaire.
- ⇒ La première phase correspond au contrôle de syntaxe.
- ⇒ Si la compilation réussit, Oracle met le pseudo-code du bloc dans une zone mémoire partagée.
- ⇒ Sinon le compilateur renvoie un message d'erreur à la session SQL*Plus.
- ⇒ Pour finir le moteur d'exécution PL/SQL intègre le pseudo-code binaire et renvoie un code de succès ou d'échec à la session SQL*Plus.

Ajoutons une requête imbriquée au bloc PL/SQL et regardons les modifications qu'elle entraîne.

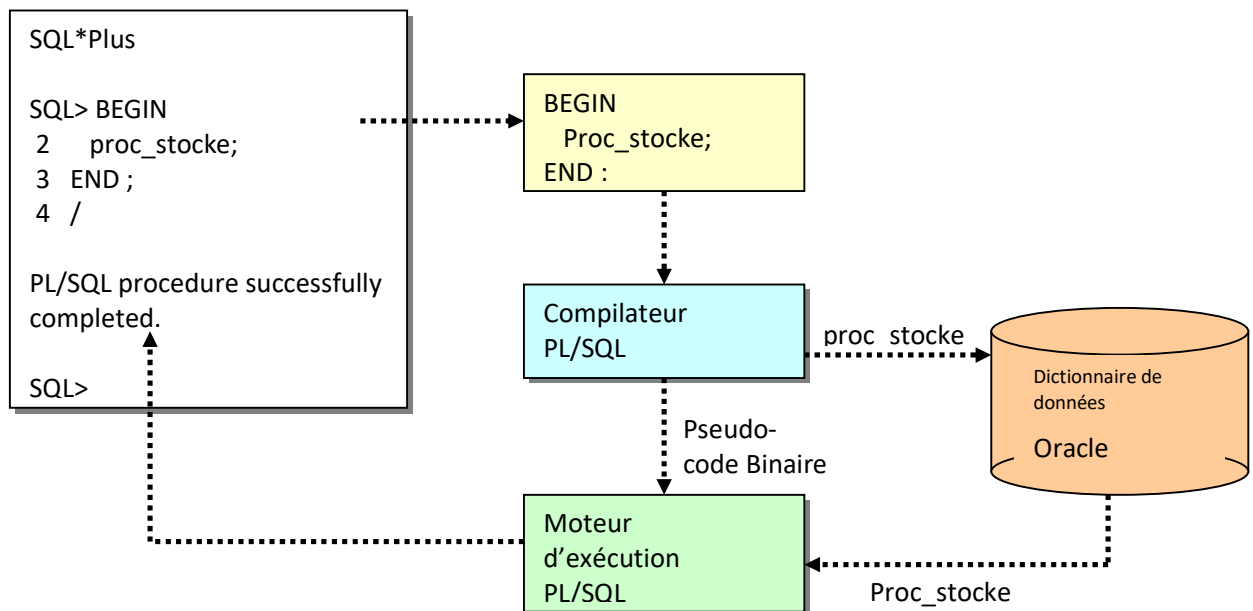




Dans Oracle le PL/SQL et le SQL partagent le même analyseur syntaxique SQL.

Dans les versions précédentes le PL/SQL possédait son propre analyseur syntaxique SQL ce qui générerait parfois des distorsions entre les ordres SQL saisis en ligne de commande et ceux exécutés dans du PL/SQL.

Appel d'une procédure stockée :



Déroulement de l'exécution :

- ⇒ Le compilateur doit résoudre la référence externe de « proc_stocke » pour savoir si elle se réfère à un programme pour lequel l'utilisateur possède un droit d'exécution.
- ⇒ Le DIANE de « proc_stocke » sera nécessaire pour savoir si le bloc anonyme effectue un appel légal au programme stocké.
- ⇒ L'avantage est que l'on dispose déjà du pseudo-code binaire et du DIANA de la procédure stockée dans le dictionnaire de données Oracle. Oracle ne perd pas de temps en recompilation.
- ⇒ Une fois le code lu sur le disque Oracle le stocke en mémoire dans une zone appelée « cache SQL » qui réduira et éliminera les entrées/sortie.



22.1. Procédures stockées JAVA

L'installation par défaut du serveur Oracle ne contient pas seulement une machine virtuelle PL/SQL mais aussi une machine virtuelle JAVA.

Vous pouvez écrire un prototype de programme PL/SQL dont la logique est implémentée dans une classe statique JAVA.

22.2. Procédures externes

Vous pouvez implémenter la partie exécutable d'un programme PL/SQL dans du code C personnalisé, et à l'exécution Oracle exécutera votre code dans un processus et dans un espace mémoire séparé.

22.2.1. *Ordres partagés*

Oracle peut partager le code source et les versions compilées d'ordres SQL et de blocs anonymes, même s'ils sont soumis à partir de différentes sessions, par différents utilisateurs.

Certaines conditions doivent être remplies :

- ⇒ La casse et les blancs des codes sources doivent correspondre exactement.
- ⇒ Les références externes doivent se référer au même objet sous-jacent afin que le programme puisse être partagé.
- ⇒ Les valeurs de données doivent être fournies via des variables de liaison plutôt que via des chaînes littérales (ou alors le paramètre système `CURSOR_SHARING` doit avoir la valeur appropriées).
- ⇒ Tous les paramètres de base influant l'optimiseur SQL doivent correspondre (Par exemple
- ⇒ Les sessions évocatrices doivent utiliser le même langage.

En PL/SQL toute variable utilisée dans un ordre SQL analysé de manière statique devient automatiquement une variable de liaison.

Si vous avez la bonne habitude de mettre les valeurs littérales dans des paramètres et des constantes et que dans vos ordres SQL vous faites référence à ces variables, plutôt qu'aux variables littérales, vous n'aurez pas de problèmes de performances.

Il est également possible d'utiliser des variables de liaison dans des blocs anonymes sous SQL*Plus.

```
VARIABLE combien NUMBER ;  
EXEC :combien := maxcats
```




22.3. Vues du dictionnaire de données

La recherche d'information sur le serveur se fait en utilisant le dictionnaire de données Oracle en utilisant les vues destinées au stockage des procédures stockées PL/SQL.

Liste des vues utiles :

⇒ SYS.OBJ\$ (USER_OBJECTS)

S'applique à tous les objets PL/SQL sauf les blocs anonymes.

Contient les noms, types d'objets, status de compilation (VALID ou INVALID)



- ⇒ **SYS.SOURCE\$ (USER_SOURCE)**
S'applique à tous les objets PL/SQL sauf les blocs anonymes.
Code source des procédures stockées.
- ⇒ **SYS.TRIGGER\$ (USER_TRIGGERS)**
S'applique aux triggers.
Code source et événement de l'élément déclencheur.
- ⇒ **SYS.ERROR\$ (USER_ERRORS)**
S'applique à tous les objets PL/SQL sauf les blocs anonymes.
Dernières erreurs pour la compilation la plus récente (y compris pour les triggers)
- ⇒ **SYS.DEPENDENCY\$ (USER_DEPENDENCIES)**
S'applique à tous les objets PL/SQL sauf les blocs anonymes.
Hiérarchie de dépendance d'objets.
- ⇒ **SYS.SETTING\$ (USER_STORED_SETTINGS)**
S'applique à tous les objets PL/SQL sauf les blocs anonymes.
Options du compilateur PL/SQL.
- ⇒ **SYS.IDLUB1\$, SYS.IDL_CHAR\$, SYS.IDL_UB2\$, SYS.IDL_SB4\$ (USER_OBJECT_SIZE)**
S'applique à tous les objets PL/SQL sauf les blocs anonymes.
Stockage interne du `DIANA`, du pseudo-code binaire.
Répertoire défini dans le paramètre `PLSQL_NATIVE_LIBRARY_DIR`
Tout ce qui a été compilé de manière native.
Fichiers objets partagés, compilés nativement en PL/SQL via du C.



23. LES DEPENDANCES

Il y a dépendance des objets (procédures, fonctions, packages et vues) lorsqu'ils font référence à des objets de la base tels qu'une vue (qui fait elle même référence à une ou plusieurs tables), une table, une autre procédure, etc...

Si l'objet en référence est modifié, il est nécessaire de recompiler les objets dépendants.

Le comportement d'un objet dépendant varie selon son type :

- ⇒ Procédure ou fonction
- ⇒ package

23.1. Dépendances des procédures et des fonctions

Il existe deux type de dépendance :

- ◆ Dépendance directe
- ◆ Dépendance indirecte

23.1.1. *Dépendances directes*

Le traitement de la procédure ou de la fonction fait explicitement référence à l'objet modifié qui peut être une table, une vue, une séquence, un synonyme, ou une autre procédure ou fonction.

Exemple

```
Une table TAB1 sur laquelle travaille une procédure PROC1  
Il y a dépendance directe de la procédure par rapport à la table.
```

Pour connaître les dépendances directes on peut consulter les tables :

- ⇒ USER|ALL|DBA_DEPENDENCIES



23.1.2. *Dépendances indirectes*

Il y a dépendance indirecte lorsque le traitement fait indirectement référence à un autre objet

- ⇒ Une vue liée à une autre table
- ⇒ Une vue liée à une autre vue
- ⇒ Un objet au travers d'un synonyme

Exemple

Une table TAB1 sur laquelle porte une vue VUE1
une procédure PROC1 travaille à partir de la vue VUE1
Il y a dépendance indirecte de la procédure par rapport à la table TAB1.

Pour connaître les dépendances indirectes on peut utiliser la procédure :

- ⇒ Oracle_home/rdbms/admin/DEPTREE_FILL liée aux vues DEPTREE et IDEPTREE.

23.1.3. *Dépendances locales et distantes*

Dans le cas de dépendances locales, les procédures et les fonctions sont sur la même base que les objets auxquels elles font référence.

Dans le cas de dépendances distantes, les procédures et les fonctions sont sur une base différente de celle des objets auxquels elles font référence.

23.1.4. *Impacte et gestion des dépendances*

Chaque fois qu'un objet référencé par une procédure ou une fonction est modifié, le statut du traitement dépendant passe à **invalid**. Il est alors nécessaire de le recompiler.

Pour consulter le statut d'un objet utiliser la vue USER | ALL | DBA_objects

Si l'objet est sur la même base locale, Oracle vérifie le statut des objets dépendants et les recompile automatiquement.

Pour les objets sur bases distantes, Oracle n'intervient pas, la re-compilation doit se faire manuellement.



Même si Oracle re-compile automatiquement les procédures invalides, il est conseillé de le faire manuellement afin d'éviter les contentions et d'optimiser les traitements.

Pour re-compiler un objet utiliser la commande :

Pour re-compiler un objet utiliser la commande :

```
Alter {procedure|function|view} nom_objet COMPILE  
;
```

23.2. Packages

La gestion des dépendances pour les packages est plus simple :

- ⇒ Si on modifie une procédure externe au package, il faut recompiler le corps du package.
- ⇒ Si on modifie un élément dans le corps du package, sans rien modifier dans la partie spécification, il n'y a pas besoin de re-compiler le corps du package.

```
Alter package body nom_package COMPILE  
;
```



24. QUELQUES PACKAGES INTEGRES

Oracle contient un certain nombre de packages utiles en développement et en administration. Nous vous en présentons ici quelques uns.

24.1. Le package DBMS_OUTPUT

Ce package permet de stocker de l'information dans un tampon avec les procédures `PUT` ou `PUT_LINE`. Il est possible de récupérer l'information grâce aux procédures `GET` et `GET_LINE`.

Liste des procédures :

- ♦ `Get_line` (ligne out varchar2, statut out integer) : extrait une ligne du tampon de sortie.
- ♦ `Get_lines` (lignes out varchar2, n in out integer) : extrait à partir du tampon de sortie un tableau de n lignes.
- ♦ `New_line` : place un marqueur de fin de ligne dans le tampon de sortie.
- ♦ `Put` (variable|conatante in {varchar2|number|date}) : combinaison de `put` et `new_line`.
- ♦ `Enable` (taille tampon in integer default 2000) : permet de mettre en route le mode trace dans une procédure ou une fonction.
- ♦ `Disable` : permet de désactiver le mode trace dans une procédure ou une fonction.

24.2. Le package UTL_FILE

Ce package permet à des programmes PL/SQL d'accéder à la fois en lecture et en écriture à des fichiers système.

Ce package est remplacé aujourd'hui par l'utilisation du scheduler.

On peut appeler `utl_file` à partir de procédures cataloguées sur le serveur ou à partir de modules résidents sur la partie cliente de l'application, comme Oracle forms.

Liste des procédures et fonctions :

- ♦ `Fopen` (location in varchar2, nom_fichier in varchar2, mode_ouverture in varchar2) return `utl_file.file_type` : cette fonction ouvre un fichier et renvoie un pointeur de type `utl_file.file_type` sur le fichier spécifié. Il faut avoir le droit d'ouvrir un fichier dans le répertoire spécifié. Pour cela il faut accéder au paramètre `utl_file_dir` dans le fichier `init.ora`.



- ♦ `Get_line` (pointeur_fichier in utl_file.file_type, ligne out varchar2) : cette procédure lit une ligne du fichier spécifié, s'il est ouvert dans la variable ligne. Lorsqu'elle atteint la fin du fichier l'exception `no_data_found` est déclenchée.
- ♦ `Put_line` (pointeur_fichier in utl_file.file_type, ligne out varchar2) : cette procédure insère des données dans un fichier et ajoute automatiquement une marque de fin de ligne. Lorsqu'elle atteint la fin du fichier, l'exception `no_data_found` est déclenchée.
- ♦ `Put` (pointeur_fichier utl_file.file_type, item in {varchar2|number|date}) : cette procédure permet d'ajouter des données dans le fichier spécifié.
- ♦ `New_line` (pointeur_fichier utl_file.type) : cette procédure permet d'ajouter une marque de fin de ligne à la fin de la ligne courante.
- ♦ `Putf` (pointeur_fichier utl_file.file_type, format in varchar2, item1 in varchar2, [item2 in varchar2, ...]) : cette procédure insère des données dans un fichier suivant un format.
- ♦ `Fclose` (pointeur_fichier in utl_file.file_type) : cette procédure permet de fermer un fichier.
- ♦ `Fclose_all` : cette procédure permet de fermer tous les fichiers ouverts.
- ♦ `Is_open` (pointeur_fichier in utl_file.file_type) return boolean : cette fonction renvoie TRUE si pointeur_fichier pointe sur un fichier ouvert.

24.3. Le package DBMS_SQL

Ce package permet d'accéder dynamiquement au SQL à partir du PL/SQL.

Les requêtes peuvent être construites sous la forme de chaînes de caractères au moment de l'exécution puis passées au moteur SQL.

Ce package offre notamment la possibilité d'exécuter des commandes DDL dans le corps du programme.

Liste des procédures et fonctions :

- ♦ `Open_cursor` return integer : cette fonction ouvre un curseur et renvoie un « integer ».
- ♦ `Parse` (pointeur in integer, requête_sql in varchar2, dbms.native) : cette procédure analyse la chaîne 'requête_sql' suivant la version sous laquelle l'utilisateur est connecté.
- ♦ `Execute` (pointeur in integer) return integer : cette fonction exécute l'ordre associé au curseur et renvoie le nombre de lignes traitées dans le cas d'un insert, delete ou update.
- ♦ `Close_cursor` (pointeur in out integer) : cette procédure ferme le curseur spécifié, met l'identifiant du curseur à NULL et libère la mémoire allouée au curseur.