



ORACLE



Le langage PL/SQL

Version du 25 Octobre 2009
Réalisée par Clotilde Attouche
Société Tellora



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>	8
1.1.	<i>Une solution globale.....</i>	8
1.2.	<i>Oracle Database</i>	9
1.3.	<i>La documentation Oracle</i>	13
1.4.	<i>Le support Oracle.....</i>	13
1.5.	<i>La base Exemple</i>	14
1.5.1.	Modèle Conceptuel de Données Tahiti	14
1.5.2.	Les contraintes d'intégrité.....	15
1.5.3.	Règles de passage du MCD au MPD.....	15
1.5.4.	Modèle Physique de données Tahiti :.....	16
1.6.	<i>Script de création des tables</i>	17
1.7.	<i>Les types de données utilisés dans les tables.....</i>	20
1.8.	<i>Notion de schéma.....</i>	20
1.9.	<i>Règles de nommage</i>	21
2.	<i>L'outil SQL*Plus.....</i>	22
2.1.1.	Environnement de travail.....	22
2.1.2.	Lancement de SQL*Plus sous Dos	23
3.	<i>Le langage SQL*Plus</i>	25
3.1.	<i>Utilisation de paramètres.....</i>	26
3.2.	<i>Commandes SQL*Plus</i>	26
3.2.1.	Mise en forme à l'affichage	26
3.2.2.	Ajouter des commentaires.....	27
3.2.3.	Exécuter le contenu d'un script.....	27
3.2.4.	Déclarer un éditeur	27
3.2.5.	Générer un fichier résultat	28
3.2.6.	Modifier l'affichage par défaut.....	28
3.2.7.	Mesurer les performances d'une requête.....	30
4.	<i>Le dictionnaire de données.....</i>	31
5.	<i>Rappel du langage SQL.....</i>	33



5.1. Requêtes avec comparaisons	36
5.1.1. La clause IN.....	36
5.1.2. La clause LIKE	37
5.1.3. La valeur NULL.....	38
5.1.4. La clause BETWEEN.....	38
5.1.5. Trier l’affichage d’une requête	39
5.1.6. Eliminer les doublons	40
5.2. Requêtes avec jointures.....	41
5.2.1. Equijointure.....	41
5.2.2. Inequijointure.....	42
5.2.3. Jointure multiple.....	43
5.2.4. Utiliser des ALIAS.....	44
5.3. Présentation du langage LMD.....	45
5.4. Insérer des lignes dans une table.....	45
5.4.1. La commande INSERT	45
5.4.2. Insertion à partir d’une table existante.....	46
5.5. Modifier les lignes d’une table	48
5.5.1. La commande UPDATE	48
5.5.2. Modifications de lignes à partir d’une table existante	49
5.5.3. Modifier une table par fusion : MERGE.....	50
5.5.4. Améliorations de la commande MERGE en version 10g.....	53
5.6. Spécifier la valeur par défaut d’une colonne.....	54
5.7. Supprimer les lignes d’une table	55
5.7.1. La commande DELETE	55
5.7.2. Vider une table.....	57
6. Les séquences.....	58
6.1. Créer une séquence.....	58
6.2. Utiliser une séquence	59
6.3. Modifier une séquence.....	59
6.4. Supprimer une séquence.....	60
7. Le langage PL/SQL.....	61
7.1. Structure d’un programme P/SQL	61
7.2. Les différents types de données	62
7.2.1. Conversion implicite.....	63
7.2.2. Conversion explicite.....	63
7.3. Les variables et les constantes	63
7.3.1. Les structures	64
7.4. Les instructions de bases.....	65
7.4.1. Condition.....	65
7.4.2. Itération.....	65



7.4.3.	L'expression CASE.....	65
7.4.4.	Expression GOTO.....	68
7.4.5.	Expression NULL.....	68
7.4.6.	Gestion de l'affichage.....	69
7.5.	Tables et tableaux	70
7.6.	Les curseurs	71
7.6.1.	Opérations sur les curseurs.....	72
7.6.2.	Attributs sur les curseurs	73
7.6.3.	Exemple de curseur.....	74
7.6.4.	Exemple de curseur avec BULK COLLECT.....	75
7.6.5.	Variables curseur.....	76
7.7.	Les exceptions.....	78
7.7.1.	Implémenter des exceptions utilisateurs.....	80
7.7.2.	Implémenter des erreurs Oracle.....	82
7.7.3.	Fonctions pour la gestion des erreurs.....	83
7.8.	Instruction "Execute Immediate"	84
7.9.	Exemples de programmes PL/SQL.....	84
7.10.	Compilation native du code PL/SQL	87
7.11.	Transactions autonomes	87
8.	<i>Procédures, Fonctions et Packages.....</i>	92
8.1.	Procédures.....	93
8.1.1.	Créer une procédure	93
8.1.2.	Modifier une procédure	95
8.1.3.	Correction des erreurs	95
8.2.	Fonctions	97
8.2.1.	Créer une fonction.....	97
8.3.	Packages	98
9.	<i>Triggers</i>	102
9.1.	Créer un trigger	103
9.2.	Activer un trigger	106
9.3.	Supprimer un Trigger.....	106
9.4.	Triggers rattaché aux vues	107
9.5.	Triggers sur événements systèmes.....	107
10.	<i>Architecture du moteur PL/SQL</i>	109
10.1.	Procédures stockées JAVA	111
10.2.	Procédures externes.....	112
10.2.1.	Ordres partagés	112
10.3.	Vues du dictionnaire de données	112



11. Les dépendances.....	114
11.1. Dépendances des procédures et des fonctions	114
11.1.1. Dépendances directes	114
11.1.2. Dépendances indirectes.....	115
11.1.3. Dépendances locales et distantes	115
11.1.4. Impacte et gestion des dépendances.....	115
11.2. Packages.....	116
12. Quelques packages intégrés.....	117
12.1. Le package DBMS_OUTPUT.....	117
12.2. Le package UTL_FILE.....	117
12.3. Le package DBMS_SQL.....	118
13. Transactions et accès concurrents.....	119
13.1. Découper une transaction.....	120
13.2. Gestion des accès concurrents.....	121
13.3. Les verrous	122
13.4. Accès concurrents en mise à jours	123
13.5. Les rollbacks segments ou segments d'annulation	124
14. Le Scheduler (<i>CJQ</i>)	126
14.1. Concepts du scheduler.....	128
14.2. Privilèges associés au Scheduler	128
14.2.1. Privilèges utilisateurs.....	129
14.2.2. Privilèges administrateurs	130
14.3. Créer et gérer un programme dans un schedule.....	130
14.4. Les JOBS.....	131
14.4.1. Créer un JOB	131
14.4.2. Spécifier des schedules pour un job.....	133
14.4.3. Créer et utiliser des schedules	134
14.5. Les JOB CLASS.....	135
14.5.1. Créer une JOB CLASS	135
14.6. Gestion des Logues de JOB	136
14.6.1. Le package DBMS_SCHEDULER	136
14.6.2. Les logs des JOBs.....	137
14.7. Les Windows.....	137
14.7.1. Créer une WINDOW	137
14.7.2. Attribuer des priorités aux JOBS dans les WINDOWS.....	139
14.8. Activer un composant du scheduler.....	140
14.9. Gérer des composants du scheduler.....	140



14.9.1.	Gérer un JOB.....	140
14.9.2.	Gérer un PROGRAM	141
14.9.3.	Gérer un schedule	141
14.9.4.	Gérer une fenêtre de temps : Window	142
14.9.5.	Priorité des Windows	143
14.9.6.	Gérer les attributs des composants du scheduler	144
14.10.	Vues du dictionnaire de données	145



1. VUE D'ENSEMBLE

1.1. Une solution globale

Trois produits sont regroupés sous la terminologie Oracle10G

- ⇒ Oracle10g Database (Oracle10g DB), la base de données
- ⇒ Oracle10g Application Server (Oracle 10g AS), le serveur d'application
- ⇒ Oracle10g Developer Suite (Oracle10g DS), l'ensemble des outils de développement
- ⇒ Oracle Collaboration Suite, regroupe vos applications de messagerie électronique, vocales, conférence Web, calendrier et administration de fichiers.
- ⇒ Oracle application ou e-Business suite, est un ensemble complet d'applications d'entreprise permettant de gérer les processus clients, produits, commandes, créances ...



1.2. Oracle Database

C'est le produit de base 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)).

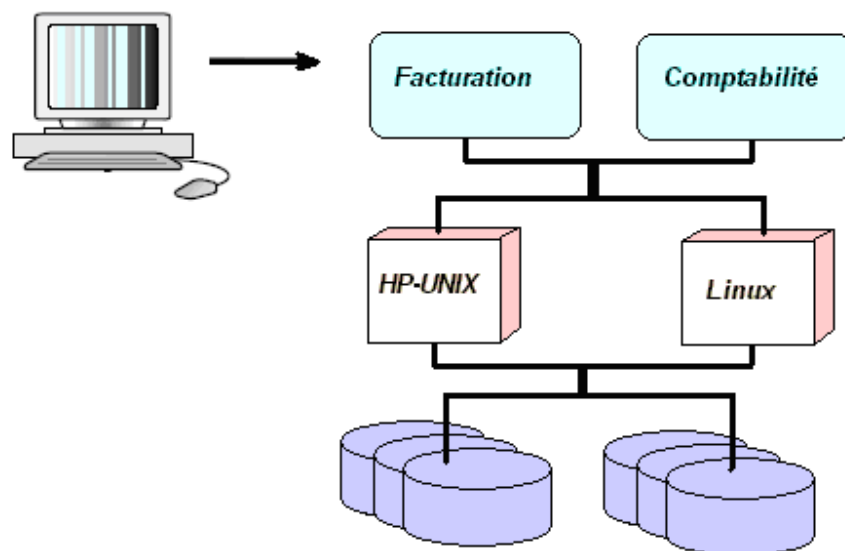
Il intègre la notion de *Grid Computing* (réseau distribué d'ordinateurs hétérogènes en grille). Le but du *Grid* est de créer des pools de ressources :

- ⇒ 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.

Oracle Database permet à ces machines d'interopérer ; l'ensemble étant considéré comme une seule ressource unifiée.

- ⇒ Chaque ressource est vue comme un service, (comme l'électricité par exemple).



Informatique en Grille





Par exemple :

Les deux applications Facturation et comptabilité se partagent des ressources de deux serveurs.

- ⇒ Chacune peut être hébergée sur n'importe lequel d'entre eux et ses fichiers de base de données sur n'importe quel disque.

Il est possible de mettre en place des réseaux grille national, voire mondiale.

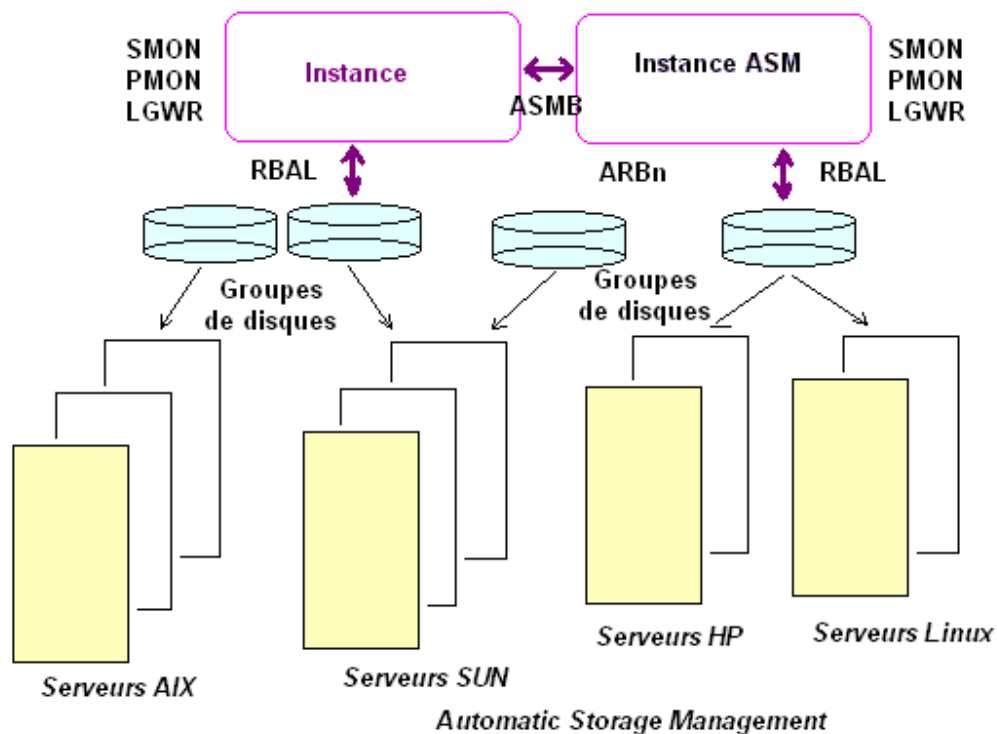
Ainsi chaque nouveau système peut être rapidement mis à disposition à partir du pool de composants.

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.



La nouvelle fonctionnalité *Automatic Storage Management* (ASM) permet à la base de données de gérer directement les disques bruts. Elle élimine le besoin pour un gestionnaire de fichier de gérer à la fois des fichiers de données et des fichiers de journaux.



L'ASM répartit automatiquement toutes les données de bases de données entre tous les disques, délivrant le débit le plus élevé sans aucun coût de gestion.

Au fur et à mesure de l'ajout et de l'abandon de disques, l'ASM actualise automatiquement la répartition des données.

Pour utiliser ASM vous devez démarrer une instance spéciale appelée « *ASM instance* » qui doit être démarrée avant de démarrer l'instance de votre propre base de données.

Les instances ASM ne montent pas de *databases* mais à la place gère les *metadatas* requises pour rendre les fichiers ASM disponibles à n'importe quelle instance de base de données.

Les deux, instance ASM et instance « *ordinaire* » ont accès au contenu des fichiers. Communiquant avec l'instance ASM seulement pour connaître le *layout* des fichiers utilisés.

Une instance ASM contient deux nouveaux *process* en tâche de fond.

- ⇒ RBAL coordonne les activités de reprise (*Rebalance activities*) des groupes de disque.
- ⇒ ARBn effectue les opérations de reprises courantes. Il peut y en avoir plusieurs, appelées alors, ARB1, etc...



Une instance ASM a aussi les mêmes processus d'arrière plan que les instances « ordinaires » (SMON, PMON, LGWR)

Oracle intègre un outil d'administration graphique intranet : **Oracle Enterprise Manager**

Les différents produits d'Oracle sont proposés en trois gammes :

- ♦ **Enterprise Edition** - La gamme pour les grosses applications critiques de l'entreprise
- ♦ **Standard Edition** - La gamme pour les applications des groupes de travail ou des départements de l'entreprise, elle est destinées à des serveurs possédant 4 processeurs.
- ♦ **Standard Edition ONE** - la gamme destinées à un bi-processeur
- ♦ **Personal Edition** - La gamme pour l'utilisateur indépendant (développeur, consultant, ...), elle utilise un noyau Enterprise Edition.

Standard Edition, comporte toutes les fonctionnalités de base permettant de mettre en œuvre des applications client-serveur ou Internet/Intranet, et est limitée à des serveurs quadri-processeurs.

- ⇒ Elle est proposée avec l'option RAC (*Real application Cluster*) en version 10g.
- ⇒ Pour un total de 4 processeurs (2 serveurs de 2 processeurs ou 4 serveurs mono-processeur)

Enterprise Edition, propose des fonctionnalités supplémentaires, en standard ou en option, permettant d'améliorer la disponibilité et les capacités de montée en charge :

- ♦ Oracle Partitioning, permet de subdiviser le stockage des gros objets (tables et index) en plusieurs éléments appelés partitions.
- ♦ Advanced Security Option, offre des fonctionnalités avancées sur la gestion de la sécurité (cryptage, authentification, etc.).
- ♦ Oracle Tuning Pack, module d'administration permettant de faciliter l'optimisation des performances de la base de données.

Enterprise Edition intègre les options :

- ⇒ RAC qui permet d'utiliser Oracle sur des serveurs en cluster
- ⇒ *Data Guard* qui permet la mise en place de base de données de secours (*Standby*)



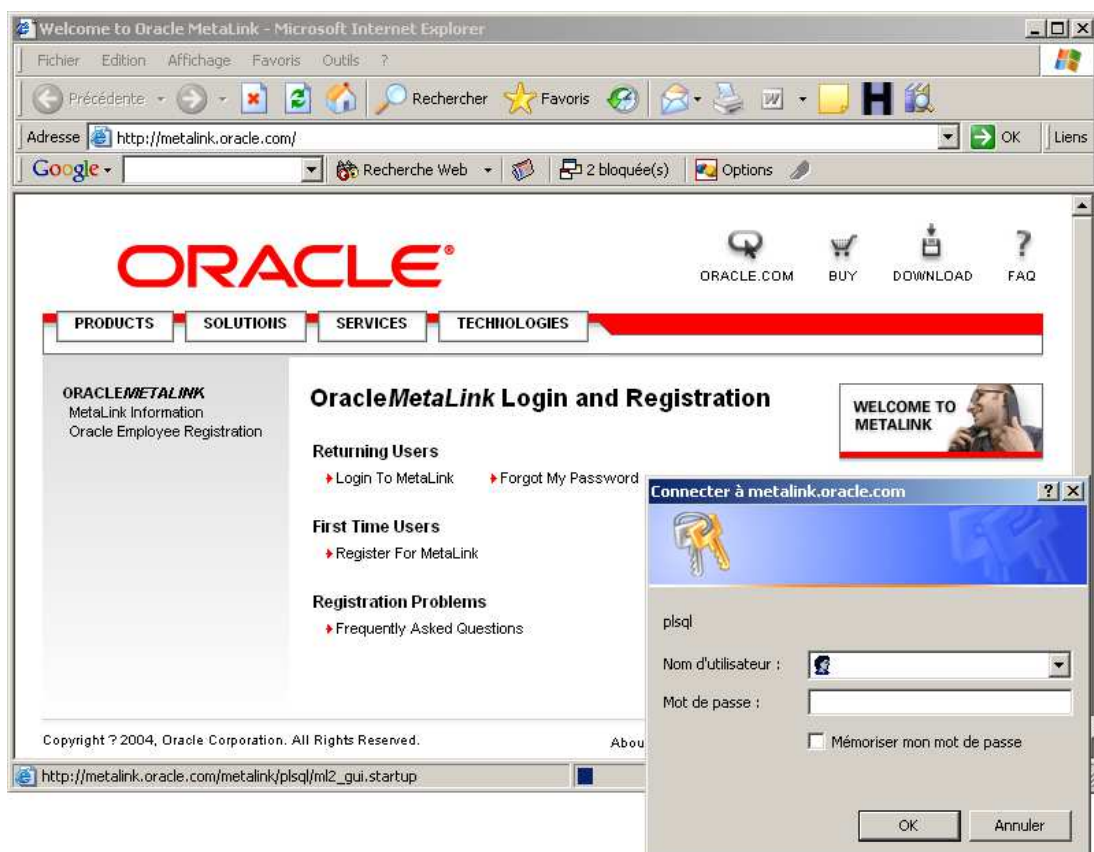
1.3. La documentation Oracle

La documentation Oracle peut s'obtenir à partir du serveur : <http://www.Oracle.com>

1.4. Le support Oracle

Autrefois le site Metalink etait le site de hotline en ligne : : <http://metalink.Oracle.com>
Aujourd'hui ce site est remplacé par : <http://support.oracle.com>, on y accede grace au numéro de CSI (numéro de licence Oracle).

On y trouve des résolutions d'erreurs référencées des patches et des scripts d'administration.

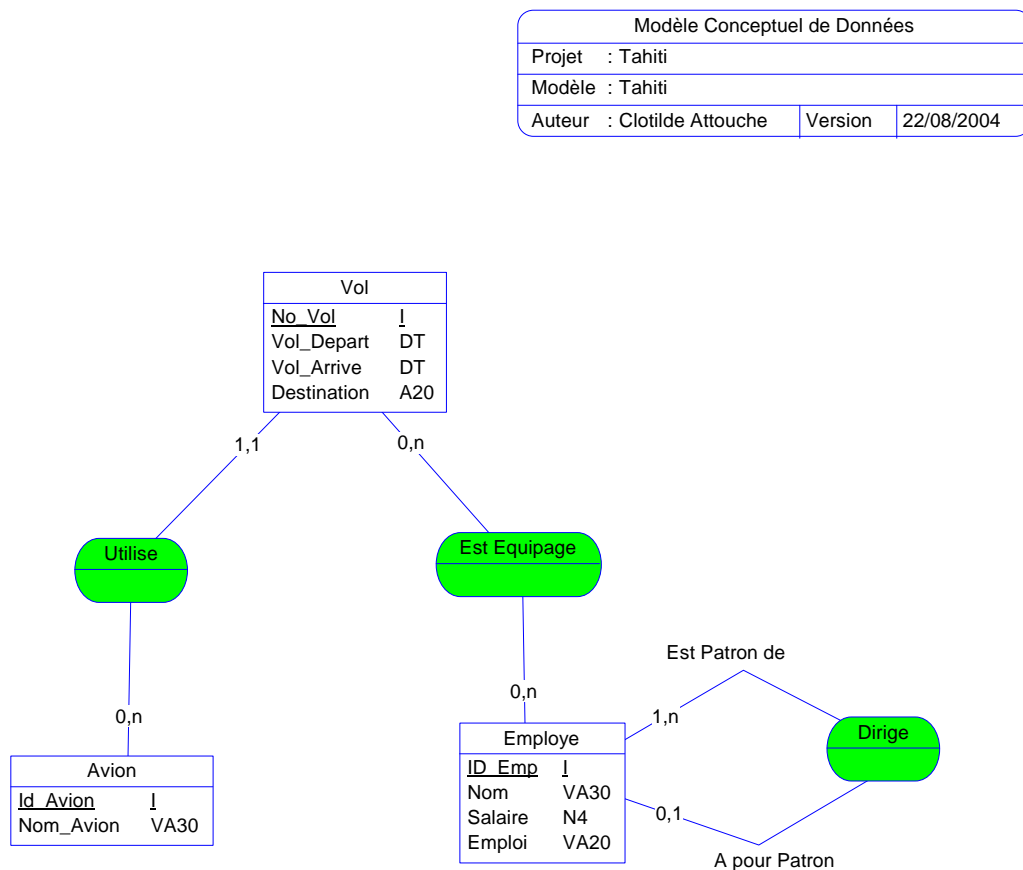




1.5. La base Exemple

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

1.5.1. *Modèle Conceptuel de Données Tahiti*





1.5.2. *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).

1.5.3. *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 :

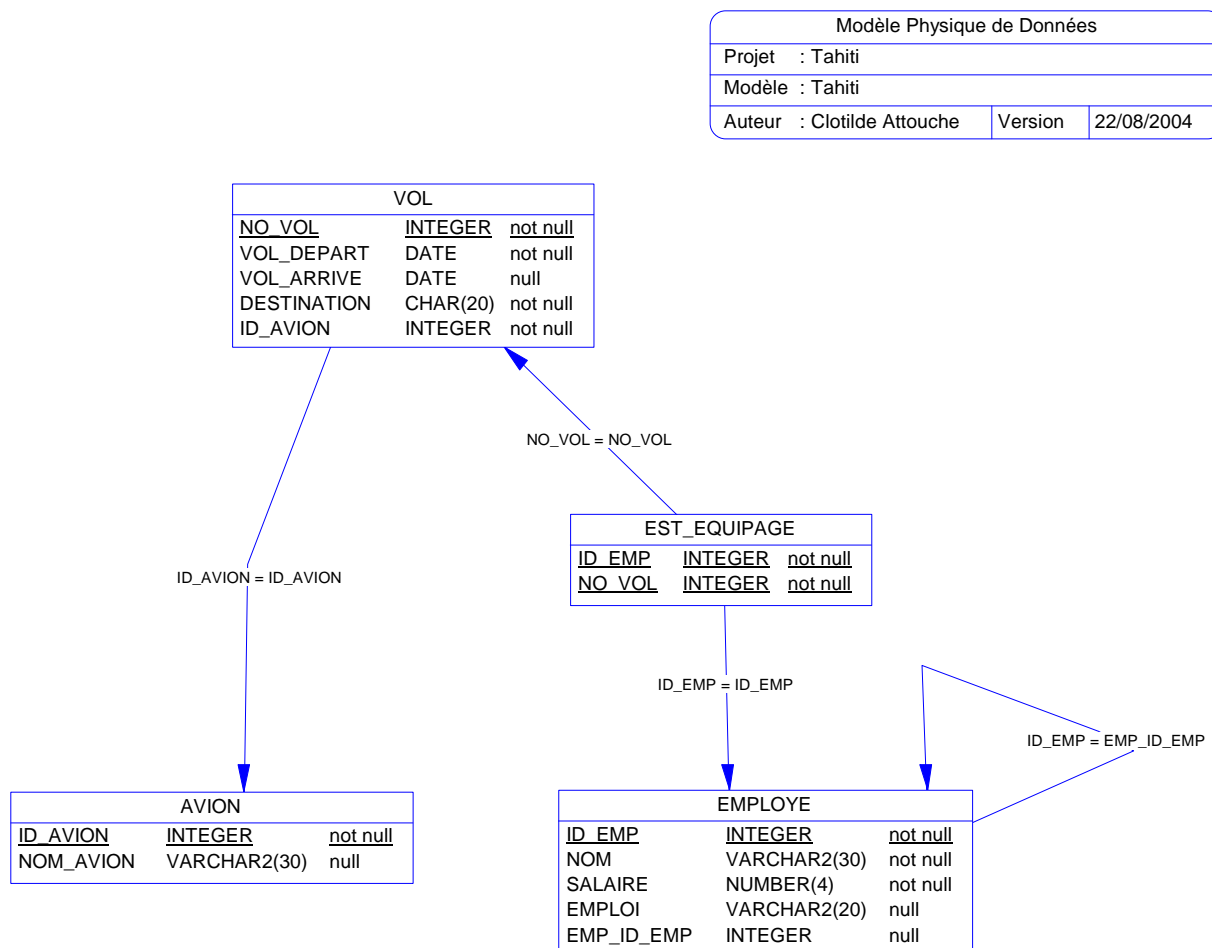
- ⇒ Les entités deviennent des tables
- ⇒ Les identifiants des entités deviennent les clés primaires de ces tables
- ⇒ Les relations dont toutes les cardinalités sont 0,N ou 1,N de chaque côté de la relation deviennent des tables
- ⇒ La concaténation des identifiants des entités qui concourent à la relation 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)



1.5.4. *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.





1.6. 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 Version 8
--   Date de cr,ation :  22/08/2004  17:09
-- =====

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);
/
```



1.7. Les types de données utilisés dans les tables

Les différents types utilisés pour les colonnes de tables 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 ...)
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.

Oracle vérifie la longueur maximum spécifiée lors de la déclaration des colonnes.

1.8. 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)

1.9. 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. 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
```

2.1.1. 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.



La boîte de dialogue suivante permet de saisir un compte et un mot de passe ORACLE ...



Connexion

Nom utilisateur : charly

Mot de passe : *****

Chaîne hôte : tahiti

OK Annuler

Le nom de la « Chaîne hôte » correspond au nom du service Oracle Net de la base de donnée à laquelle l'utilisateur veut se connecter. Celle-ci se trouve le plus souvent sur un serveur distant.
La session SQL*PLUS est ouverte ...

Pour se positionner dans le répertoire courant il suffit d'effectuer la manipulation suivante :

Fichier

Ouvrir (jusqu'à ce que l'on voit le contenu du répertoire de travail dans la boîte de dialogue)

OK pour sortir de la boîte de dialogue

Oracle mémorise le chemin du répertoire affiché.

2.1.2. 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.

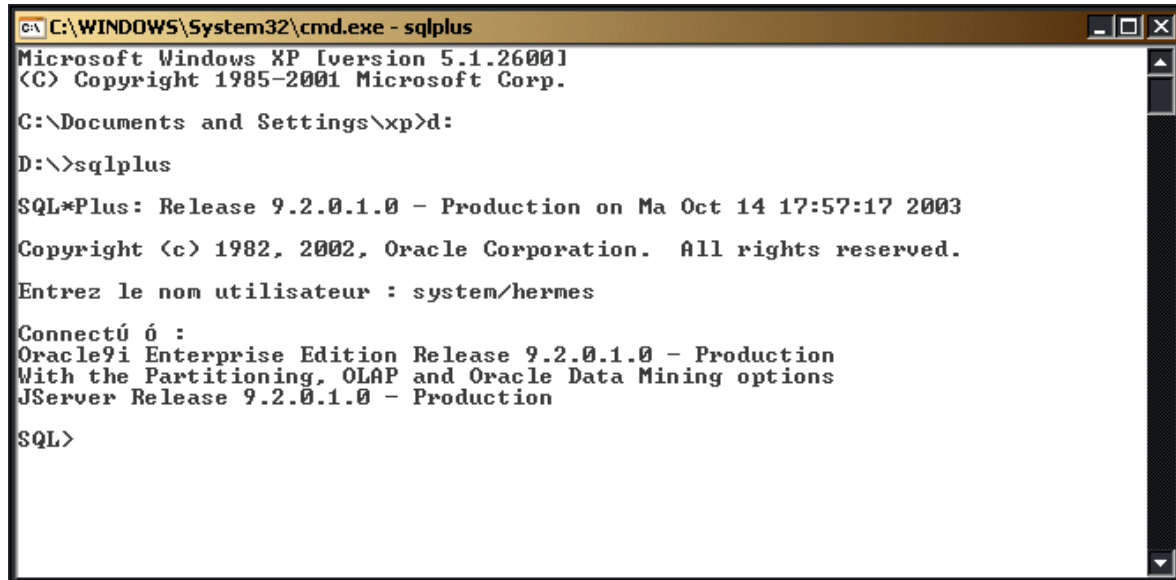
```
Set %ORACLE_SID%=ORCL
SQLPLUS /nolog
SQL> connect system/secret
SQL> connected
SQL>

-- 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 17:57:17 2003
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

Entrez le nom utilisateur : system/hermes

Connecté à :
Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production

SQL>
```




3. 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.



3.1. 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
```

3.2. 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.

3.2.1. 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.

3.2.2. *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
```

3.2.3. *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
```

3.2.4. *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.

3.2.5. *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
```

3.2.6. *Modifier l'affichage par défaut*

L'affichage des commandes SQL lors de l'exécution d'un script peut être modifié par un ensemble de commandes :

- ♦ **SET PAUSE ON**, afficher un résultat page par page
- ♦ **SET LINESIZE 100**, reformater la taille de la ligne à 100 caractères
- ♦ **SET ECHO ON/OFF**, affiche ou pas le texte de la requête ou de la commande à exécuter
- ♦ **SET TIMING ON|OFF**, provoque l'affichage d'informations sur le temps écoulé, le nombre d'E/S après chaque requête
- ♦ **TERM [ON|OFF]**, supprime tout l'affichage sur le terminal lors de l'exécution d'un fichier
- ♦ **VER [ON|OFF]**, provoque l'affichage des lignes de commandes avant et après chaque substitution de paramètre.

```
rem *-----*
rem * Nom audit2.sql
*
rem * Role : Procedure SQL d'audit d'une base ORACLE.
*
rem *      Liste: - des utilisateurs avec leurs privileges,
*
rem *      - des fichiers de la base avec leur taille et le cumul par
tablespace,      *
rem *      - des tables,index avec leur occupation par tablespace et par
proprietaire, *
```




3.2.7.

La variable SQL*Plus TIMING permet la production d'une ligne d'information à la fin de chaque requête. Cette ligne comporte les informations suivantes :

- ◆ **ELAPSED**, le temps écoulé
- ◆ **CPU**, le temps CPU consommé par la tâche SQL*Plus
- ◆ **BUFIO**, le nombre d'entrées/sorties physiques ou accès aux buffers
- ◆ **DIRIO**, les nombres d'entrées/sorties physiques ou accès disque
- ◆ **FAULTS**, le nombre de défauts de page

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. 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
```




5. 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                                VARCHAR2(18)
EMP_ID_EMP                             NUMBER(38)

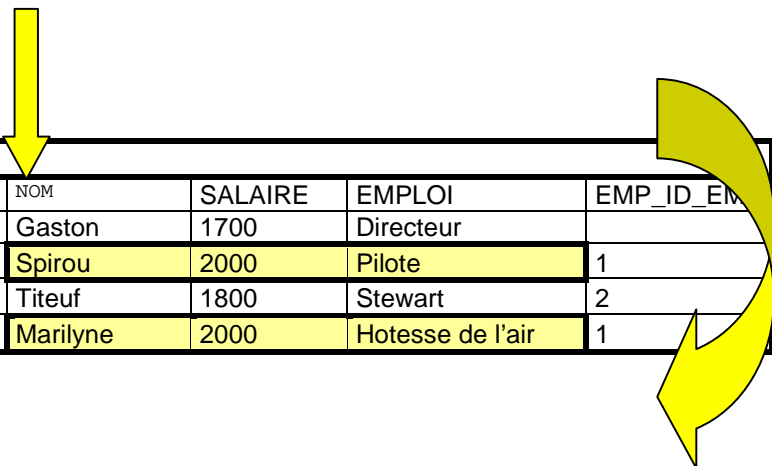
SQL> select nom, salaire, emploi
2   from   employe
3   where  salaire >=2000
4   order by nom ;

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_EM
1	Gaston	1700	Directeur	
2	Spirou	2000	Pilote	1
3	Titeuf	1800	Stewart	2
4	Marilyne	2000	Hotesse de l'air	1



Comme on peut s'en rendre compte, une requête `SELECT` est très intuitive car elle se rapproche du langage quotidien.

C'est une des raisons du succès du SQL. Cependant, le SQL est avant tout un langage de définition et d'extraction de données. Ses possibilités algorithmiques, de saisie, ou d'affichage sont limitées. Ce n'est d'ailleurs pas sa finalité.

Lorsqu'il ne suffit plus (impossibilité syntaxique ou requête trop lourde), on utilise un autre langage qui offre une plus grande puissance algorithmique ou graphique.

Le SQL se contente alors d'extraire les données nécessaires pour le langage hôte (PL/SQL, Pro*C, etc. ...). Beaucoup d'outils offrent en standard un interpréteur SQL pour consulter les données d'une base relationnelle (ORACLE ou autre).

Tous nos exemples vont s'appuyer sur la base exemple qui se présente dans l'état suivant :

```
SQL> select * from AVION;
```

ID_AVION	NOM_AVION
1	Caravelle
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		Tokyo	2

```
SQL> select * from EST_EQUIPAGE;
```

ID_EMP	NO_VOL
1	1
4	1
3	2
4	2

```
SQL> select * from EMPLOYE;
```

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



Nous avons classés les différents types de requêtes par thème :

- ⇒ Requêtes avec comparaisons
- ⇒ Requêtes avec jointures
- ⇒ Requêtes avec groupement
- ⇒ Requêtes ensemblistes
- ⇒ Sous requêtes
- ⇒ Balayage d'une arborescence

5.1. Requêtes avec comparaisons

La clause `WHERE` peut utiliser les opérateurs :

AND, OR, BETWEEN, NOT, IN, =, <>, !=, >, >=, <=

Ces opérateurs s'appliquent aux valeurs numériques, aux chaînes de caractères, et aux dates. Les chaînes de caractères et les dates doivent être encadrées par `'...'` contrairement aux nombres.

```
SQL> select * from avion
  2  where id_avion between 1 and 2
  3    and nom_avion like 'C%' ;

ID_AVION NOM_AVION
-----
      1 Caravelle
```

5.1.1. La clause IN

La clause **IN** permet d'éviter l'emploi de **OR** et simplifie la syntaxe ...

```
SQL> select nom, salaire
  2  from employe
  3  where salaire in (1700,1800) ;

NOM                                SALAIRE
-----
Gaston                             1700
Titeuf                              1800
```



```
SQL> select nom, salaire
  2   from employe
  3  where salaire < 2000
  4  and emp_id_emp in (null, 2, 4)
  5  ;
```

NOM	SALAIRE
Gaston	1700
Titeuf	1800

5.1.2. La clause LIKE

La clause **LIKE** permet de rechercher des chaînes de caractères :

% Toute chaîne de 0 à n caractères
_ 1 caractère

ESCAPE \ désigne \ pour inhiber les fonctions de « % » et « _ »

```
SQL> select nom, salaire, emploi
  2   from employe
  3  where nom like '%t_n';
```

NOM	SALAIRE	EMPLOI
Gaston	1700	Directeur

```
SQL> select nom_avion
  2   from avion
  3  where nom_avion like '_\_%' escape '\';
```

NOM_AVION
A_Caravelle_2

```
SQL> select nom_avion
  2   from avion
  3  where nom_avion like '*_%' escape '*';
```

NOM_AVION
A_Caravelle_2

- Cette requête affiche tous les avions dont le nom commence par n'importe quel caractère suivi d'un _ .
- Sans l'utilisation de % on se contenterait des noms sur 2 caractères qui respectent cette règle.



5.1.3. La valeur NULL

Pour manipuler une valeur non renseignée (en lecture ou mise à jour) on utilise le prédicat **NULL**



La valeur NULL pour un champ signifie non renseigné. Il ne faut pas confondre avec zéro ou blanc. Ce prédicat est utilisable dans toutes les commandes SQL (insert, select, ...).

```
SQL> select nom, emploi
       2  from employe
       3  where emp_id_emp is null ;

NOM                                EMPLOI
-----
Gaston                             Directeur
```

- Cette requête affiche le nom et le salaire des employés dont le salaire contient la valeur NULL

5.1.4. La clause BETWEEN

La clause **Between** permet de sélectionner des lignes à l'intérieure de bornes définies.

```
SQL> select * from avion
       2  where id_avion between 2 and 3 ;

ID_AVION  NOM_AVION
-----
         2  Boeing
         3  Planeur
```

Cette requête affiche l'identifiant et le nom des avion dont l'identifiant est compris entre 2 et 3 **bornes incluses** .



5.1.5. *Trier l'affichage d'une requête*

La clause **Order by** permet de trier le résultat affiché.

```
SQL> select id_avion, nom_avion
  2  from avion
  3  order by nom_avion ;

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

SQL> select id_avion, nom_avion
  2  from avion
  3  order by 2 ;

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

- Cette requête affiche l'identifiant et le nom des avions ordonnés par nom d'avion, sur un ordre croissant.

Pour afficher un ordre décroissant il suffit de préciser **Desc** derrière la colonne citée dans le tri. Le nom de colonne peut être remplacé par la position de la colonne derrière la clause SELECT.

```
SQL> select id_avion, nom_avion
  2  from avion
  3  order by nom_avion desc ;

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



5.1.6. *Eliminer les doublons*

Le mot clé **DISTINCT** permet d'éliminer les doublons lors de l'affichage. Il porte sur toutes les colonnes affichées sur une ligne.

```
SQL> select nom_avion Avion, nom
2   from employe, avion, est_equipage, vol
3   where est_equipage.id_emp = employe.id_emp
4         and est_equipage.no_vol = vol.no_vol
5         and vol.id_avion = avion.id_avion
6         and nom_avion = 'Caravelle'
7   order by nom ;
```

AVION	NOM
Caravelle	Gaston
Caravelle	Marilyne
Caravelle	Titeuf
Caravelle	Marilyne

```
SQL> select distinct nom_avion Avion, nom
2   from employe, avion, est_equipage, vol
3   where est_equipage.id_emp = employe.id_emp
4         and est_equipage.no_vol = vol.no_vol
5         and vol.id_avion = avion.id_avion
6         and nom_avion = 'Caravelle'
7   order by nom ;
```

AVION	NOM
Caravelle	Gaston
Caravelle	Marilyne
Caravelle	Titeuf

- Affichage des employés affectés à un équipage transportés par une caravelle.



DISTINCT provoque un tri,
a utiliser avec précautions.



5.2. Requêtes avec jointures

Principe de base

Les requêtes concernent souvent des informations qui sont ventilées dans plusieurs tables. La recherche de ces informations s'appuie sur le principe de jointure. Il s'agit de rapprocher une ou plusieurs tables qui ont des colonnes en commun. Ces liens se traduisent la plupart du temps par des clés étrangères.

Une jointure est donc un sous ensemble du produit cartésien de deux tables. Seules les lignes respectant les conditions de jointures sont conservées. La différence réside dans la condition de jointure (**WHERE**) et dans les arguments du **SELECT**.

5.2.1. *Equijointure*

Nous souhaitons afficher le nom de tous les AVIONS qui sont utilisés pour un VOL.

Nous devons donc utiliser la table VOL pour lister tous les vols prévus, et la table AVION pour trouver le nom des avions. Mais il ne faut pas afficher le nom de tous les avions. Seuls ceux dont l'identifiant est mentionné dans la table VOL ont forcément été prévus pour voler.

Cette requête s'écrira :

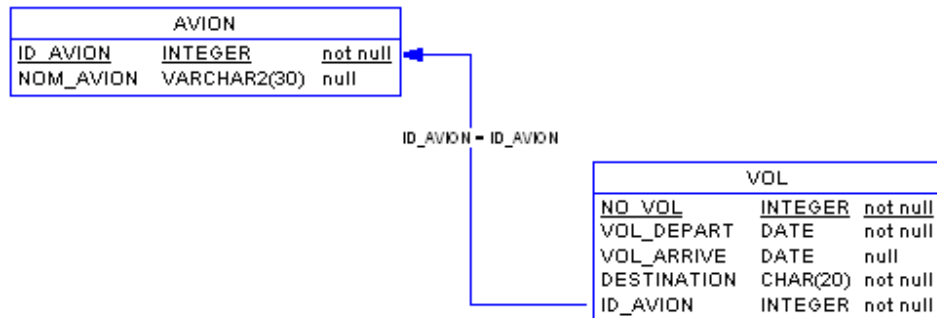
```
SQL> select nom_avion Avion, destination
2   from vol, avion
3   where vol.id_avion = avion.id_avion
4   order by destination ;
```

La clause **FROM** doit préciser les tables concernées par la jointure.

La clause **WHERE** doit préfixer la colonne **Id_avion** par le nom de la table concernée pour éviter les conflits. En effet Oracle effectue d'abord le produit cartésien entre les tables « VOL » et « AVION » avant d'extraire les données à afficher .

La colonne **Id_avion** existe deux fois dans le produit cartésien et Oracle ne sait pas quelle colonne afficher.

Nous allons détailler cette requête afin de bien nous imprégner de l'algorithme de base mis en œuvre pour rechercher les données.



Tout se passe comme si l'interpréteur construisait une table temporaire résultant de toutes les associations possibles entre les lignes des deux tables.

Le système n'est pas capable de « deviner » les liens entre les deux tables. Il doit construire l'association des données des deux tables en s'appuyant sur les valeurs communes des champs *Id_avion*.

Il suffit ensuite de ne garder que les lignes qui correspondent à la condition de jointure (ici égalité des champs *Id_avion*) et d'afficher les informations demandées.

```
SQL> select nom_avion Avion, destination
2  from vol, avion
3  where vol.id_avion = avion.id_avion
4  order by destination ;
```

AVION	DESTINATION
-----	-----
Caravelle	Marquises
Caravelle	Tahiti
Boeing	Tokyo

Nous allons présenter maintenant d'autres types de jointures. Celui que nous venons de voir est une équi-jointure (la condition de jointure est une égalité sur deux colonnes de deux tables différentes).

5.2.2. Inequijointure

Une inéqui-jointure est une jointure sans condition d'égalité entre les deux colonnes.

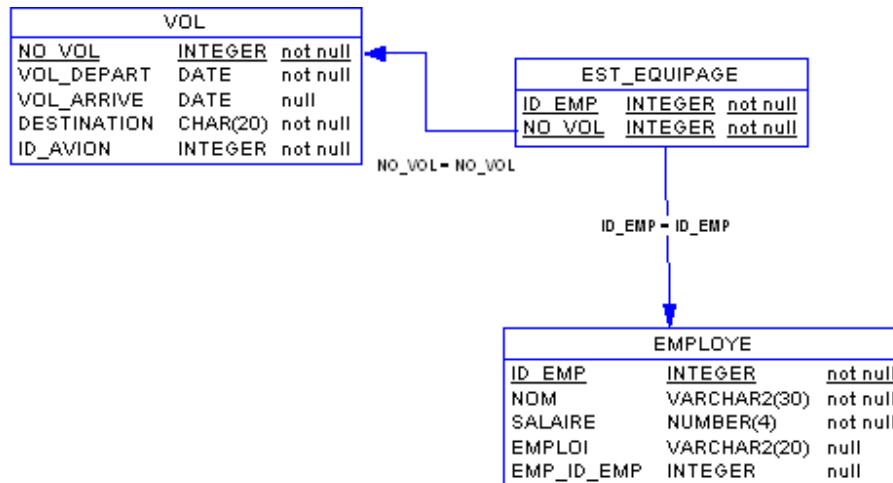
Elle utilise les opérateurs « <, >, <=, >=, <>, != » .



5.2.3. Jointure multiple

Une jointure multiple met en relation plusieurs colonnes de tables différentes, toujours en reliant ces tables par :

⇒ Clé étrangère vers clé primaire



Afficher les employés affectés à un vol pour Tahiti.

```
SQL> select nom, destination
2  from vol, employe, est_equipage
3  where est_equipage.id_emp = employe.id_emp
4    and est_equipage.no_vol = vol.no_vol
5    and destination = 'Tahiti' ;
```

NOM	DESTINATION
Gaston	Tahiti
Marilyne	Tahiti

Détail de la requête ...

Première jointure :

Sur la colonne *Id_emp* entre les tables EMPLOYE et EST_EQUIPAGE.

Seconde jointure :

Sur la colonne *No_vol* entre les tables EST_EQUIPAGE et VOL.



Condition :

La clause « and destination = 'Tahiti' » réduit la sélection concernant la destination.

Affichage :

La clause SELECT ne mentionnant que les colonnes « nom » et « destination », Oracle n'affichera que ces deux colonnes.

5.2.4. Utiliser des ALIAS

Un alias permet de remplacer le nom d'une table dans un ordre select par une lettre. Le nom de la table n'est plus reconnu que par la lettre concernée dans la totalité de la requête.

Afficher l'équipage à destination de Tahiti.

```
SQL> select nom, destination
  2  from vol, employe, est_equipage
  3  where est_equipage.id_emp = employe.id_emp
  4        and est_equipage.no_vol = vol.no_vol
  5        and destination = 'Tahiti' ;
```

NOM	DESTINATION
Gaston	Tahiti
Marilyne	Tahiti

```
SQL> select nom, destination
  2  from vol v, employe e, est_equipage eq
  3  where eq.id_emp = e.id_emp
  4        and eq.no_vol = v.no_vol
  5        and destination = 'Tahiti' ;
```

NOM	DESTINATION
Gaston	Tahiti
Marilyne	Tahiti

```
SQL> select nom, destination, v.no_vol
  2  from vol v, employe e, est_equipage eq
  3  where eq.id_emp = e.id_emp
  4        and eq.no_vol = v.no_vol
  5        and destination = 'Tahiti' ;
```

NOM	DESTINATION	NO_VOL
Gaston	Tahiti	1
Marilyne	Tahiti	1



5.3. Présentation 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).

5.4. Insérer des lignes dans une table

5.4.1. 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,'Boïng' );
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
```

5.4.2. 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                         NULL      VARCHAR2(30)
DESTINATION                       NULL      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	



5.5. Modifier les lignes d'une table

5.5.1. 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
```




5.5.2. 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



5.5.3. *Modifier une table par fusion : MERGE*

L'ordre SQL **MERGE** permet de sélectionner des lignes dans une table en vue de les insérer ou de les modifier dans une autre table

- ⇒ Le tout en une seule opération
- ⇒ L'ordre SQL **MERGE** peut être utilisé en PL/SQL

```
MERGE INTO table_cible [alias] USING source [alias] ON (condition)
WHEN MATCHED THEN clause_update
WHEN NOT MATCHED THEN clause_insert ;
```

```
SOURCEtable | vue | sous-requête
```

```
clause_updateUPDATE SET colonne = expression | DEFAULT [...]
```

```
clause_insertINSERT (colonne[,...]) VALUES (expression | DEFAULT [...])
```

- **INTO** table_cible [alias] : spécifie la table cible des insertions ou mises à jour
table_cible : nom de la table
alias : alias sur la table (optionnel)

USING source [alias] : spécifie la source des données
source peut être une table, une vue ou une sous-requête
alias : alias de la source (optionnel)

ON condition : définit la condition sur la table cible qui va déterminer la nature de l'opération effectuée sur chaque ligne de la table cible
Chaque ligne de la table cible telle que la condition est vraie est mise à jour avec les données correspondantes de la source
Si la condition n'est vérifiée pour aucune ligne de la table cible, Oracle insère une ligne dans la table cible avec les données correspondantes de la source

WHEN MATCHED THEN clause_update : spécifie l'ordre **UPDATE** qui est exécuté sur les lignes de la table cible lorsque la condition est vérifiée
UPDATE « normal » sans le nom de la table (déjà définie par la clause **INTO** de l'ordre **MERGE**)

WHEN NOT MATCHED THEN clause_insert : spécifie l'ordre **INSERT** qui est exécuté dans la table cible lorsque la condition n'est pas vérifiée
INSERT avec **VALUES** « normal » sans la clause **INTO** donnant le nom de la table (déjà définie par la clause **INTO** de l'ordre **MERGE**)

```
SQL> select v.id_avion, nom_avion, destination
       2         from avion a, vol v
       3         where v.id_avion = a.id_avion
```



```
4          and destination = 'Marquises'
5  ;

  ID_AVION NOM_AVION          DESTINATION
-----
      1 Caravelle            Marquises

SQL> select * from avion_2;

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

MERGE INTO avion_2 a                                -- cible
  USING (select v.id_avion, nom_avion, destination
        from avion a, vol v
        where v.id_avion = a.id_avion
        and destination = 'Marquises'
        ) d                                           -- source = requête => alias vm
  ON (a.id_avion = d.id_avion)                       -- en cas d'égalité
  WHEN matched then                                  -- correspondance
    update set a.nom_avion = 'Essai Merge'           -- mise à jour
  WHEN not matched then                              -- pas correspondance
    insert (a.nom_avion, a.destination)              -- insérer
    values (d.nom_avion, d.destination)
;

1 ligne fusionnée.

SQL> select * from avion_2;

  ID_AVION NOM_AVION          DESTINATION
-----
      1 Essai Merge            Marquises
      2                    Tahiti
      3                    Tahiti
      4                    Tahiti
      5                    Canaries
      6 Petit coucou
```



Exemple sur l'égalité des identifiant d'avion.

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

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

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

ID_AVION	NOM_AVION	DESTINATION
1	Caravelle	Marquises

```
MERGE INTO avion_2 a                                     -- cible
  USING (select v.id_avion, nom_avion, destination
        from avion a, vol v
        where v.id_avion = a.id_avion
        and destination = 'Marquises'
        ) d
  ON (a.id_avion != d.id_avion)                            -- source = requête => alias vm
  WHEN matched then                                       -- en cas d'égalité
    update set a.nom_avion = 'Petit Coucou'              -- correspondance
    -- mise à jour
  WHEN not matched then                                   -- pas correspondance
    insert (a.nom_avion, a.destination)                   -- insérer
    values (d.nom_avion, d.destination)
```

5 lignes fusionnées.

Vérification :

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

ID_AVION	NOM_AVION	DESTINATION
1	Essai Merge	Marquises
2	Petit Coucou	Tahiti
3	Petit Coucou	Tahiti
4	Petit Coucou	Tahiti
5	Petit Coucou	Canaries
6	Petit Coucou	

6 ligne(s) sélectionnée(s).



En version 9i, la condition doit se faire sur l'identifiant sinon Oracle affiche une erreur :

```
ON (a.nom_avion != d.nom_avion)          -- en cas d'égalité
*
ERREUR Ó la ligne 7 :
ORA-00904: "A"."NOM_AVION" : identificateur non valide
```

5.5.4. Améliorations de la commande **MERGE** en version 10g

En version 10g, il y a deux nouveautés majeures pour la commande **MERGE** :

- ⑩ De nouvelles clauses et extensions pour l'utilisation standard de la commande **MERGE**, facilitant et accélérant son utilisation.
- ⑩ Une clause optionnelle **DELETE** pour la commande **MERGE UPDATE**.

Commande **UPDATE** et **INSERT** conditionnels

Vous pouvez ajouter une clause conditionnelle **WHERE** à une clause **UPDATE** ou **INSERT** d'une commande **MERGE** pour conditionner les opérations **INSERT** ou **UPDATE**.

```
-- Cet exemple montre l'utilisation d'une clause WHERE qui permet
-- aux paramètres UPDATE ou INSERT de pointer vers des produits 'non-obsolètes'.

MERGE
  Into product_change PC  -- destination table1
  USING products P        -- source/delta table
  ON (P.prod_id = PC.prod_id) -- join condition
  WHEN MATCHED THEN
    UPDATE -- UPDATE IF JOIN
      SET PC.prod_new_price = P.prod_list_price
      WHERE P.prod_status <> 'obsolete'
  WHEN NOT MATCHED THEN
    INSERT (PC.prod_new_price)
    Values (P.prod_list_price)
    WHERE P.prod_status <> 'obsolete'
;
```

Clause optionnelle **DELETE**

Vous pouvez utiliser la clause **DELETE** dans une commande **MERGE UPDATE** pour nettoyer les tables en les mettant à jour.

Seules les lignes affectées par la clause **DELETE** seront mises à jour par l'opération **MERGE** dans la table de destination.

La condition **WHERE** du **DELETE** évalue la valeur mise à jour, et non la valeur originale qui a été évalué par la condition **UPDATE SET**. Ainsi, si une ligne de la table de destination correspond à la condition du **DELETE** mais n'est pas incluse dans la jointure définie par la clause **ON**, elle n'est pas effacée.



```
-- supprimer les lignes des produits dont le statut est devenu obsolète
-- en effectuant l'UPDATE.
-- elle supprime les produits obsolètes de la table de destination.

MERGE
  Into product_change PC  -- destination table 1
  USING products P       -- source/delta table
  ON (P.prod_id = PC.prod_id) -- join condition
  WHEN MATCHED THEN
    UPDATE
      SET PC.prod_new_price = P.prod_list_price ,
          PC.prod_new_status = P.prod_status
    DELETE WHERE (PC.prod_new_status = 'obsolete') -- Purge
  WHEN NOT MATCHED THEN -- INSERT IF NOT JOIN
    INSERT (PC.prod_id, PC.prod_new_price, PC.prod_new_status)
    Values (P.prod_id, P.prod_list_price, P.prod_status)
;
```

5.6. 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 créée.

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

1 ligne créée.

```
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) sélectionné(s).



Lors d'un UPDATE :

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

1 ligne mise à 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		Canaries
6	Petit coucou	

6 ligne(s) sélectionné(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	

5.7. Supprimer les lignes d'une table

5.7.1. 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



5.7.2. *Vider une table*

Le vidage d'une table supprime toutes les lignes de la table et libère l'espace utilisé. La table et ses index sont supprimés.

Une table référencée par une clé étrangère ne peut pas être supprimée.

Il se fait en utilisant la commande suivante :

```
truncate table [ shema. ] nom_table  
[ { drop | reuse } storage ]  
;
```

Si le paramètre DROP est utilisé, tous les extents sont supprimés.

Si le paramètre REUSE est spécifié, l'espace utilisé par la table est conservée.

```
SQL> truncate table est_equipage ;  
Table tronquée.  
  
SQL> truncate table avion;  
truncate table avion  
*  
ERREUR Ó la ligne 1 :  
ORA-02266: Les clés primaires/uniques de la table référencées par des clés  
étrangères
```



Lors du vidage d'une table, il faut inactiver les contraintes clé étrangères si nécessaire.
Ne pas oublier de les réactiver après !



6. LES SEQUENCES

Les séquences sont des objets permettant de gérer les accès concurrents sur une colonne de table et d'éviter les inter-blocages.

Par exemple le calcul automatique d'une clé primaire contenant un numéro séquentiel.

L'appel de la séquence lors de l'insertion des données permet de récupérer un numéro calculé par Oracle à chaque accès base. Ce numéro est utilisé comme identifiant et est unique.

Une seule séquence doit être créée pour chaque table de la base de données.

Il est possible d'associer un synonyme à la séquence avant de donner les droits d'utilisation de celle-ci aux « USERS ».



Une séquence concerne obligatoirement une colonne numérique.

6.1. Créer une séquence

La création d'une séquence se fait avec la commande `CREATE SEQUENCE` :

```
create sequence Nom_Sequance
  increment by entier
  start with entier
  maxvalue entier | nomaxvalue
  minvalue entier | nominvalue
  cycle | nocycle
  cache entier | nocache
  order | noorder
```

INCREMENT BY : indique le pas d'incrément de la séquence

START WITH : permet de spécifier la valeur de la première valeur de séquence à générer. Par défaut cette valeur correspond à **MINVALUE** pour une séquence ascendante et à **MAXVALUE** pour une séquence descendante.

MAXVALUE : indique la valeur maximum de la séquence. Par défaut 10 puissance 27 pour l'ordre croissant et -1 pour l'ordre décroissant.

MINVALUE : indique la valeur minimum de la séquence. Par défaut 1 (**NOMINVALUE**) pour l'ordre croissant et -10 puissance 26 pour l'ordre décroissant.

CYCLE : permet de revenir à la valeur initiale en fin de limite. L'option **NOCYCLE** est prise par défaut.



CACHE : spécifie au système d'allouer plusieurs séquences en même temps. La valeur spécifiée doit être inférieure au nombre de valeur du cycle. Oracle alloue par défaut 20 valeurs.

ORDER : indique que les nombres doivent être générés dans l'ordre de la demande. **NOORDER** est l'option par défaut.

6.2. Utiliser une séquence

L'utilisation de **MA_SEQUENCE.NEXTVAL** permet de récupérer la valeur suivante attribuée par Oracle et de l'insérer dans la première colonne de la table client.

```
create sequence Ma_Sequence
  minvalue 100 ;

insert into client
  values ( Ma_Sequence.nextval, 'toto' ) ;
```

Pour rechercher la valeur courante il faut utiliser **CURRVAL** à la place de **NEXTVAL**.

```
select Ma_Sequence.currval
  from dual;
```

6.3. Modifier une séquence

La modification d'une séquence se fait en utilisant la commande **ALTER SEQUENCE**.

```
Alter sequence [schema.]sequence
  [increment by n]
  [start with n]
  [{maxvalue n | nomaxvalue}]
  [{minvalue n | nominvalue}]
  [{cycle | nocycle}]
  [{cache n | nocache}]
  [{order | noorder}] ;
```

Les paramètres sont les mêmes que pour la création d'une séquence.

INCREMENT BY : indique le pas d'incrément de la séquence

START WITH : permet de spécifier la valeur de la première valeur de séquence à générer. Par défaut cette valeur correspond à **MINVALUE** pour une séquence ascendante et à **MAXVALUE** pour une séquence descendante.

MAXVALUE : indique la valeur maximum de la séquence. Par défaut 10 puissance 27 pour l'ordre croissant et -1 pour l'ordre décroissant.



MINVALUE : indique la valeur minimum de la séquence. Par défaut 1 (**NOMINVALUE**) pour l'ordre croissant et -10 puissance 26 pour l'ordre décroissant.

CYCLE : permet de revenir à la valeur initiale en fin de limite. L'option **NOCYCLE** est prise par défaut.

CACHE : spécifie au système d'allouer plusieurs séquences en même temps. La valeur spécifiée doit être inférieure au nombre de valeur du cycle. Oracle alloue par défaut 20 valeurs.

ORDER : indique que les nombres doivent être générés dans l'ordre de la demande. **NOORDER** est l'option par défaut.

6.4. Supprimer une séquence

La suppression d'une séquence se fait en utilisant la commande **DROP SEQUENCE**.

```
Drop sequence [schema.]sequence ;
```



7. LE LANGAGE PL/SQL

Le langage PL/SQL est une extension procédurale du langage SQL. Il 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.

7.1. 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.



7.2. Les différentes 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

7.2.1. *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.

7.2.2. *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.

7.3. 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
nouveau_vol    article.prixunit%TYPE ; -- type implicite
ancien_vol      NUMBER ; -- type explicite
autre_vol       NUMBER DEFAULT 0 ; -- initialisation à 0
```

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.

7.3.1. 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;
/
```




7.4. Les instructions de bases

7.4.1. 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;
    ELSEIF salaire > 1000 AND emp_id_emp = 1 THEN
        nouveau_salaire := ancien_salaire + 500;
    ELSE nouveau_salaire := ancien_salaire + 300;
END IF;
```

7.4.2. Itération

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.

```
OPEN curs1;
LOOP
    FETCH curs1 into ancien_salaire;
    EXIT WHEN curs1%NOTFOUND;
    IF salaire < =1000 THEN
        nouveau_salaire := ancien_salaire + 100;
    ELSEIF salaire > 1000 AND emp_id_emp = 1 THEN
        nouveau_salaire := ancien_salaire + 500;
    ELSE nouveau_salaire := ancien_salaire + 300;
    end if;
    update employe set salaire = nouveau_salaire
        where current of curs1;
END LOOP;
```

Syntaxes générales

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

WHILE condition
LOOP
    séquences
END LOOP;
```

7.4.3. 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
...
```

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.



7.4.4. *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`

7.4.5. *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 :etat.selection = 'detail'  
Then  
    Exec etat_detaillie;  
Else  
    NULL ; -- ne rien faire  
END IF ;
```

7.4.6. 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 inconsidérément).

7.5. 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;
/
```

7.6. 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
```



7.6.1. *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.

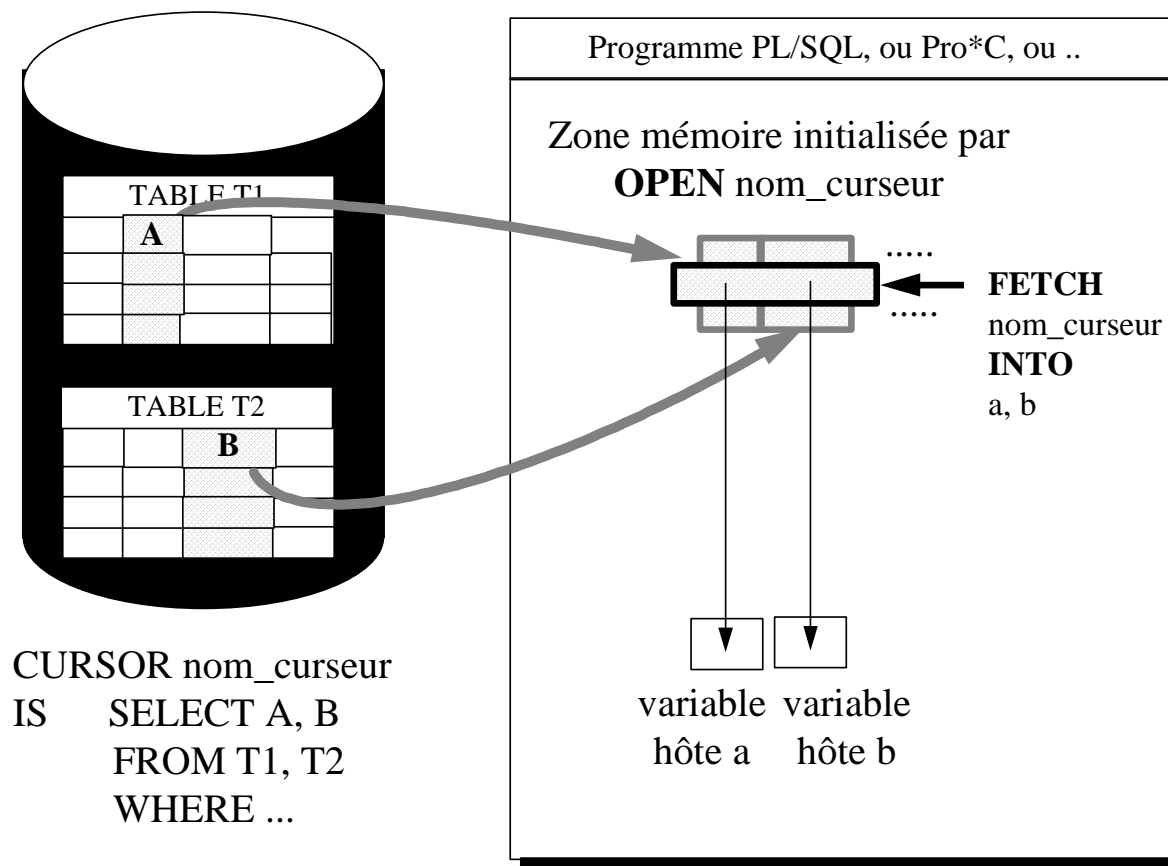
```
FETCH mes_vols into v_vol, v_destination;
```

⇒ 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 :



7.6.2. Attributs sur les curseurs

Chaque curseur possède quatre attributs :

- ⇒ **%FOUND** Génère un booléen VRAI lorsque le FETCH réussit (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



7.6.3. 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.



7.6.4. Exemple de curseur avec 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;
/
```



7.6.5. *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  
    ;  
record_emp    employe%rowtype ;  
  
BEGIN  
  
FOR record_emp IN curs_employes;  
LOOP  
    DBMS_OUTPUT.PUT_LINE(record_emp.nom_emp || ' gagne '  
                          || record_emp.salaire) ;  
  
End LOOP ;  
END ;  
/
```



7.6.6. 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;
/
```

7.7. 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 Oracles 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 initialise 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.

7.7.1. *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
```



7.7.2. *Implémenter des erreurs Oracle*

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

⇒ Exception ORACLE dont le nom est prédéfini

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
...
BEGIN
...
    EXCEPTION

    WHEN NO_DATA_FOUND THEN
        traitement;
END;
```

⇒ 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;
```

7.7.3. 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_emp_sans_adresse      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.
```

7.8. 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.

7.9. 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 ;
/
```



7.10. Compilation native du code PL/SQL

Avec Oracle9i, 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`.

7.11. 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.

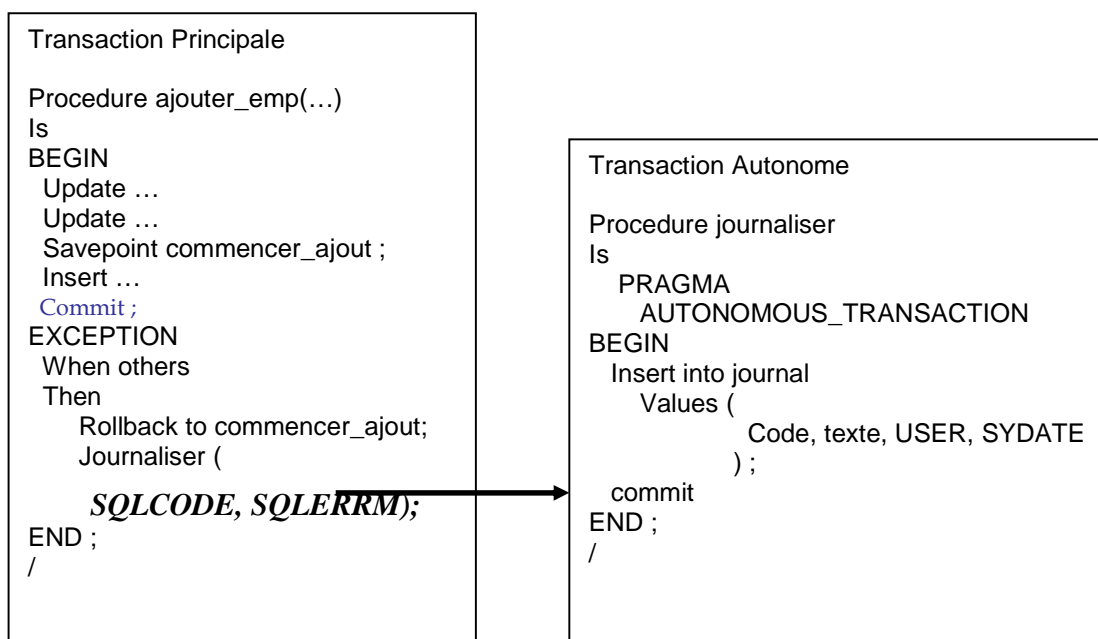
A partir de la version 8i d'Oracle, il est possible d'exécuter et de sauvegarder ou d'annuler certains ordres DML (`INSERT`, `UPDATE`, `DELETE`) sans affecter la totalité de la transaction.



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 « `PRAGAM 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.



8. 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 librairies par les programmes clients. Mais attention, il s'agit de librairies 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.



8.1. Procédures

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

8.1.1. *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
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;
/
```

Lancement du script demarre.sql :

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



8.1.2. *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.

8.1.3. *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.



8.2. 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)`.

8.2.1. *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

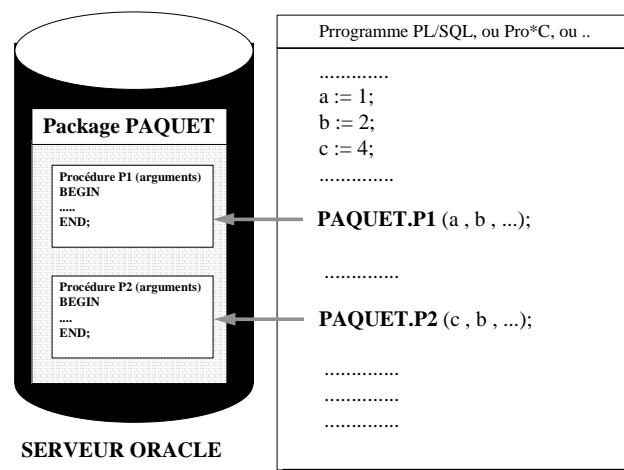
create or replace function moyenne_salaire (v_Id_employe IN NUMBER)
return NUMBER
is
valeur NUMBER;

begin

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

8.3. 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.

Notion de procédure privée :

On peut déclarer une fonction ou une procédure dans le corps de package sans qu'il soit déclaré dans l'entête de package. Dans ce cas la fonction ou la procédure reste privée et ne peut pas être utilisée via un appel extérieur au package. Elle sera appelée uniquement par une procédure ou une fonction interne au package.



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 EMPLOYE;
```

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.

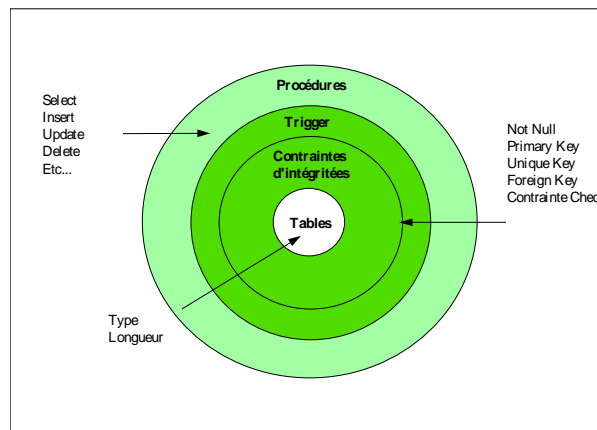


9. TRIGGERS

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 supérieure 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.

9.1. 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

9.2. 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.

9.3. 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.



9.4. 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.

9.5. 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 logiff
- ⇒ {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



```
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;
/
```



10. 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 généré par le compilateur. Il effectue des analyses sémantiques et syntaxiques. inclus 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.

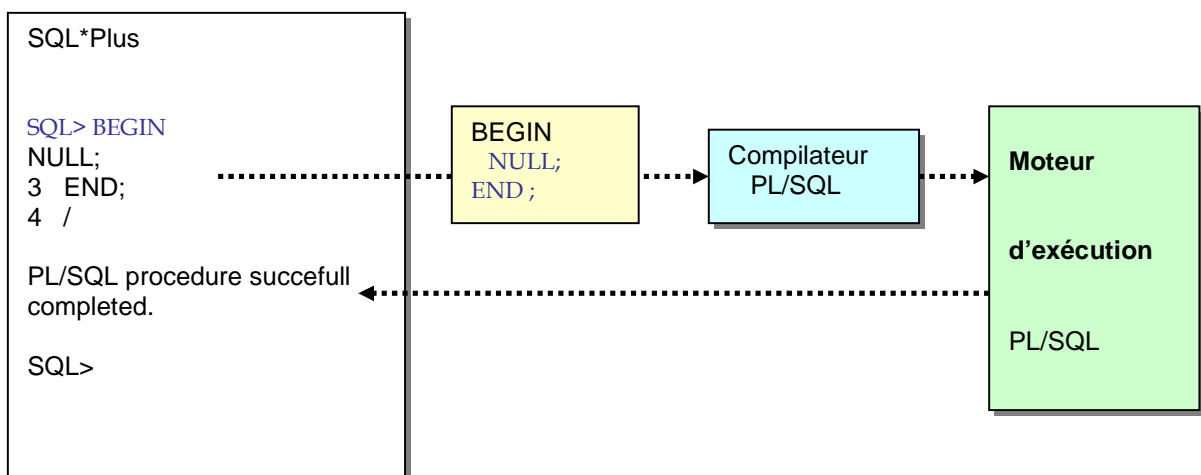
⇒ Moteur d'exécution PL/SQL

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 :

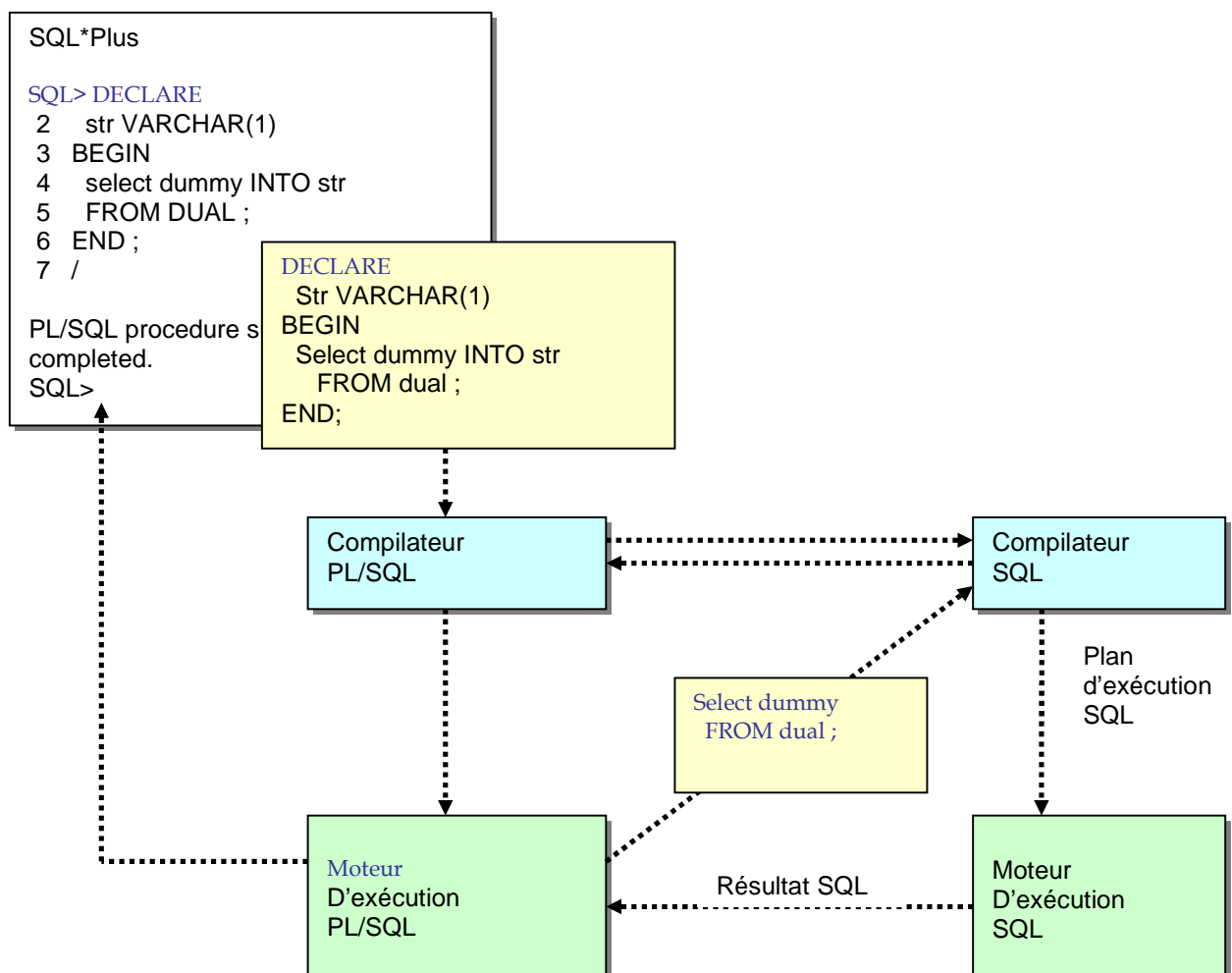




Déroulement de l'exécution :

- ⇒ 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.

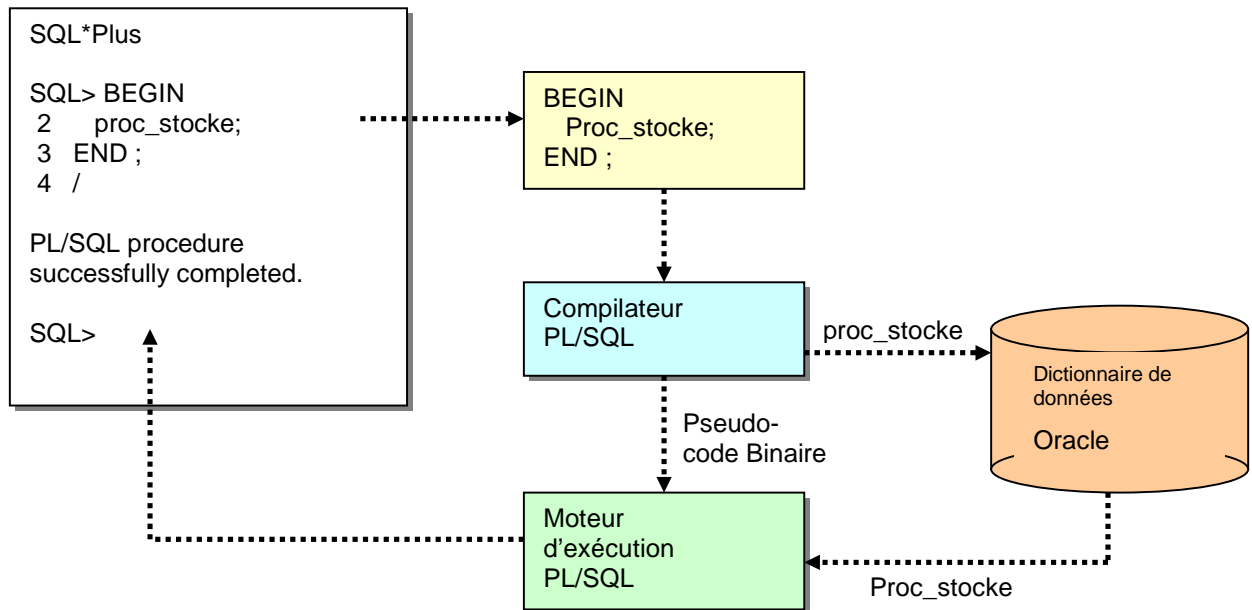


Dans Oracle 9i le PL/SQL et le SQL partagent le même analyseur syntaxique SQL.



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.

10.1. 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.



10.2. 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.

10.2.1. 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
```

10.3. 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.



11. 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

11.1. Dépendances des procédures et des fonctions

Il existe deux type de dépendance :

- ◆ Dépendance directe
- ◆ Dépendance indirecte

11.1.1. *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



11.1.2. *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.

11.1.3. *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.

11.1.4. *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 **invalidé**. 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  
;
```

11.2. 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  
;
```



12. QUELQUES PACKAGES INTEGRES

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

12.1. 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|constante 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.

12.2. 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.

12.3. 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.



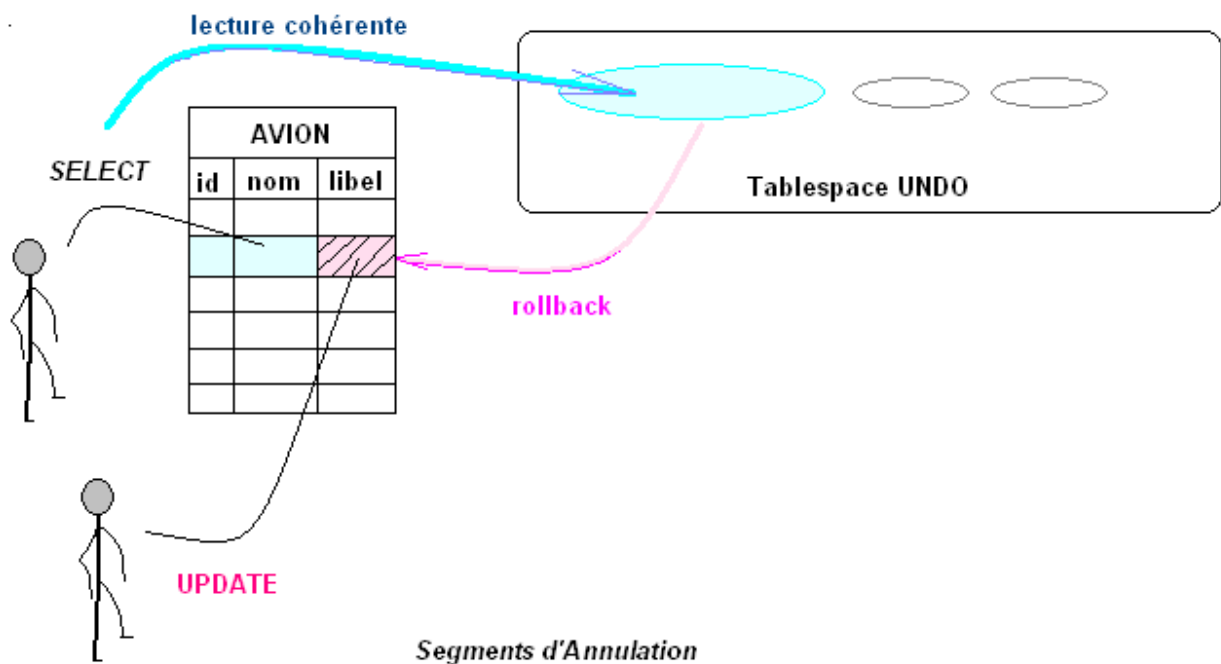
13. TRANSACTIONS ET ACCES CONCURENTS

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.

13.1. 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.

13.2. 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)



13.3. 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.

13.4. 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 gestions 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

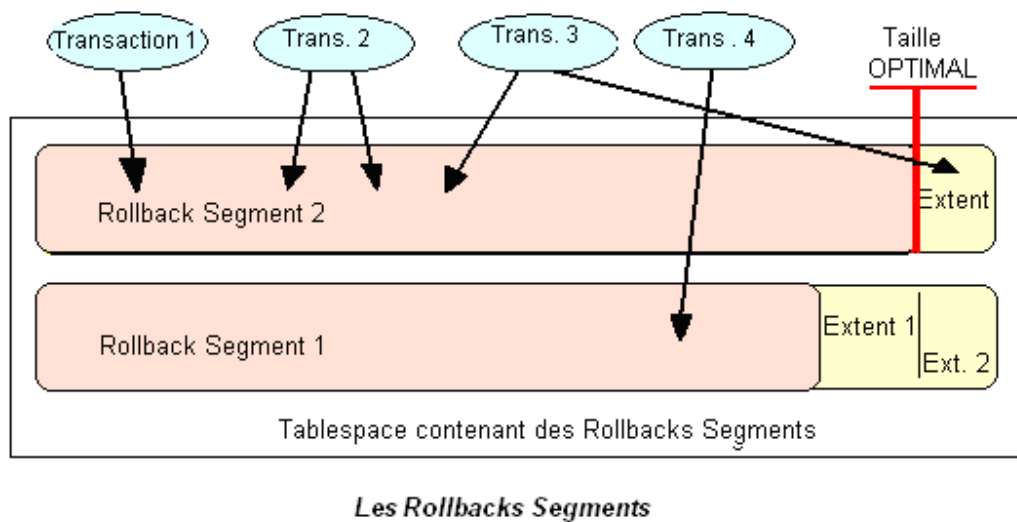


de base comme « Combien d'accès concurrents sur telles données observe-t-on en moyenne ? ». Le code s'en trouvera considérablement simplifié.

13.5. Les rollbacks segments ou segments d'annulation

Les rollbacks segments sont des segments permettant à Oracle de stocker l'image avant les modifications effectuées durant une transaction.

C'est Oracle qui alloue les transactions aux rollback segments.



Lorsque la transaction se termine, elle libère le rollback segment mais les informations de rollback ne sont pas supprimées immédiatement

⇒ Ces informations peuvent encore être utiles pour une lecture cohérente

Par défaut c'est Oracle qui alloue les rollback segment aux transactions en cherchant à répartir les transactions concurrentes sur les différents rollback segment. Dans certain cas il est possible d'allouer un rollback segment à une transaction en utilisant l'ordre SQL : `SET TRANSACTION USE ROLLBACK SEGMENT`.

Lorsqu'un rollback segment est plein et que la transaction a besoin d'espace, une erreur se produit et la transaction est arrêtée. Le rollback segment grossit dans la limite de la taille du tablespace qui le contient. En cas d'erreur, il faut alors retailler le rollback segment puis relancer la transaction en lui affectant le rollback segment agrandi.



L'erreur « *snapshot to hold* » correspond à problème de lecture cohérente. Une requête (`SELECT`) dans un segment peut être écrasée par une transaction, lors de la lecture cohérente si il y a besoin de cette requête (`SELECT`) cela provoque l'erreur « *snapshot to hold* ».



14. LE SCHEDULER (CJQ)

Job Queue Coordinator (CJQ), utilisé par le *Scheduler*, génère les processus pour exécuter les jobs planifiés qui se trouvent dans la file d'attente interne d'Oracle.

Les utilisateurs peuvent créer des jobs et les soumettre à ce coordinateur.



Quand le paramètre `JOB_QUEUE_PROCESSES` est supérieur à 0, il permet de définir le nombre de job soumis en simultanés.

Ce processus permet l'automatisation de tâches dans la base de données.

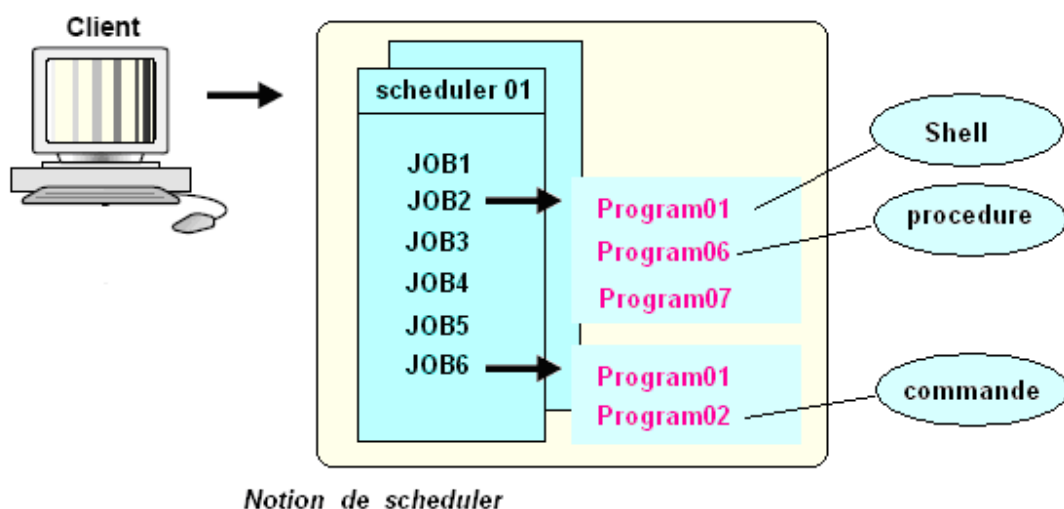
En effet, il est intéressant d'avoir la possibilité d'effectuer des travaux en différés, tels que :

- ♦ La génération de statistiques sur des tables et des index
- ♦ La réplication de données
- ♦ La gestion des sauvegardes
- ♦ Lister les événements en attente depuis un certain temps
- ♦ Exécuter un Batch
- ♦ ...

Aujourd'hui, des applications BtoB (*Business to Business*) demandent un suivi régulier et rigoureux. Il peut s'avérer nécessaire d'exécuter des *Batches* afin de répartir l'information.

A terme, le scheduler doit remplacer `DBMS_JOB` utilisé dans la réplication de données. Ainsi on pourra faire cohabiter des jobs anciens et nouveaux, ce qui n'est pas encore le cas aujourd'hui.

Le package `DBMS_SCHEDULER` permet d'avantage de possibilités que le package `DBMS_JOB` qui était une première version d'exécuteur de travaux.





Le *Scheduler* permet de spécifier :

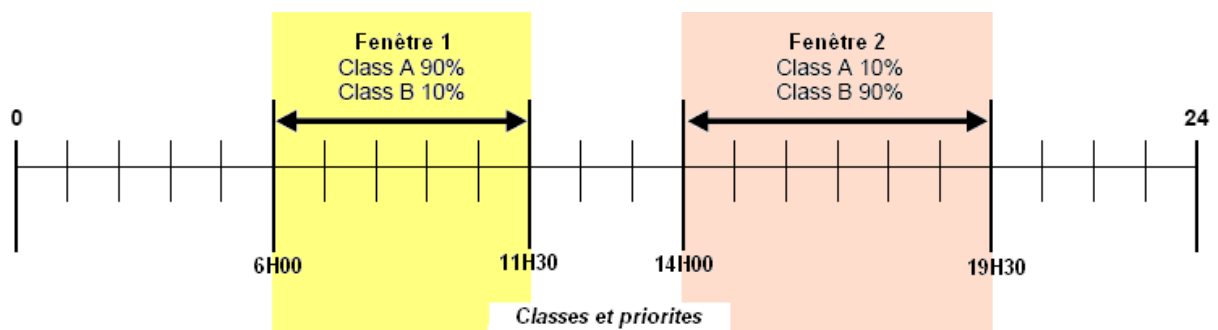
- ♦ Un **JOB** défini, ce qui doit être exécuté : le **QUOI** + **QUAND**
Par exemple s'il s'agit d'un programme Java, shell ou pl/sql.
Il est possible de spécifier le programme et de le planifier comme faisant partie de la définition du **JOB** ou vous pouvez utiliser un **SCHEDULE** existant ou un **PROGRAM** existant.
Il est possible de passer des arguments à un **JOB** afin de personnaliser son comportement.
- ♦ Un **SCHEDULE** défini **QUAND** et **COMBIEN** de fois l'action doit être exécutée, il peut s'appliquer à plusieurs **JOBs**
Spécifie quand et combien de fois un **JOB** doit être exécuté.
Vous pouvez stocker le **SCHEDULE** pour un **JOB** séparément et ainsi utiliser le même *Schedule* pour plusieurs **JOB**.
- ♦ Un **PROGRAM** défini ce que le **JOB** exécute
C'est une collection de méta-données d'un exécutable, d'un script ou d'une procédure. Un **JOB** automatisé exécute certaines tâches. En utilisant un **PROGRAM**, cela vous permet de modifier la tâche du **JOB** sans modifier le **JOB** lui-même.
Vous pouvez définir les arguments pour un **PROGRAM**, autorisant les utilisateurs à modifier le comportement de la tâche.



Chaque composant de *Scheduler* est un objet de schéma et doit avoir un nom unique.

Le scheduler permet de définir des fenêtres ouvertes dans le temps, associées à des heures bien précises. Dans ces fenêtres il est possible de définir des *classes de priorités de Jobs*. A ces classes sont associées des pourcentages de consommation de :

- ⇒ Ressource machine.
- ⇒ Plan d'exécution
- ⇒ Fenêtre de travail



Lorsque le *Job* est créé, il est placé dans une table du dictionnaire de données.

Le *Scheduler* utilise une table de *Jobs* par Database et un *Job Coordinateur* (JCQ) par instance.

Le JCQ est un processus d'arrière plan qui se réveille lorsque des *Jobs* doivent être exécutés, ou lorsqu'une fenêtre doit être ouverte. Ainsi lorsqu'un *Job* doit s'exécuter, le JCQ notifie automatiquement à un processus esclave d'effectuer le *Job*.



Le *scheduler* permet aux DBA de contrôler différents aspects des planifications tels qu'attribuer des priorités aux jobs.

Ils sont utiles pour assurer la limitation des ressources pendant que des *jobs* s'exécutent.

14.1. Concepts du scheduler

Les concepts avancés du scheduler permettent des contrôles plus poussés des éléments de planification comme, par exemple, la priorité des jobs.

Les composants sont :

- ♦ Un **JOB CLASS** qui définit une catégorie de JOBS qui partagent des ressources communes de la même façon. Un **JOB CLASS** regroupe des JOBS dans des entités plus globales.
- ♦ Un **RESSOURCE CONSUMER GROUP** associé aux **JOB CLASS** et qui désigne les ressources qui sont allouées pour les JOBS dans ces **JOB CLASS**.
- ♦ Un **RESSOURCE PLAN** qui permet aux utilisateurs de définir les priorités des ressources (notamment la CPU) parmi le **RESSOURCES CONSUMER GROUP**.
- ♦ Une **WINDOWS** est représentée par un intervalle de temps avec un début et une fin bien déterminée et est utilisée pour activer différents **RESSOURCE PLANS** à divers moments. Ceci permet de changer l'allocation de la ressource pendant une période de temps, comme l'heure dans la journée ou la période dans l'année.
- ♦ Un **WINDOW GROUP** représente une liste de **WINDOWS** et permet la gestion plus facile des **WINDOWS**. Une **WINDOW** ou un **WINDOW GROUP** peut être utilisé comme un **SCHEDULE** pour un **JOB** afin d'assurer que le job tourne seulement quand des **WINDOWS** et leurs **RESSOURCE PLANS** associés sont actifs.

Ainsi un groupe de ressources offre un moyen de rassembler les utilisateurs qui partagent les mêmes besoins en termes de ressources machine.

Le package **DATABASE_RESSOURCE_MANAGER** peut être utilisé pour allouer une quantité maximale de CPU utilisable par session, définir le degré de parallélisme maximal, et aussi pour spécifier le nombre de sessions qui peuvent être actives pour un groupe.

Un plan de ressources est élaboré pour des groupes de destinataires et fournit un moyen de définir la façon dont les ressources seront allouées.

Un utilisateur peut être déplacé vers un groupe de priorité moindre pour mettre les ressources à la disposition d'autres sessions. Vous pouvez aussi planifier l'activation ou la désactivation des plans de ressources.

14.2. Privilèges associés au Scheduler

Afin de pouvoir utiliser le *Scheduler* il est indispensable d'avoir un ensemble de privilèges.

Les privilèges system et objets associés à l'utilisation du scheduler sont :

- ⇒ **CREATE [ANY] JOB**
- ⇒ **EXECUTE ANY PROGRAM**
- ⇒ **EXECUTE ANY CLASS (Java)**
- ⇒ **MANAGE SCHEDULER**



- ⇒ EXECUTE ON <program or class>
- ⇒ ALTER ON <Job, program or schedule>
- ⇒ ALL ON >Job, program, class or schedule>

Un ensemble de privilèges, permettant d'utiliser le *Scheduler*, est disponible dans le rôle :

- ⑩ SCHEDULER_ADMIN
 - ◆ CREATE JOB
 - ◆ CREATE ANY JOB
 - ◆ EXECUTE ANY PROGRAM
 - ◆ EXECUTE ANY CLASS
 - ◆ MANAGE SCHEDULER



Le rôle SCHEDULER_ADMIN est attribué par défaut au rôle DBA, disponible à l'installation d'une base Oracle.
Il est préférable d'utiliser ce rôle pour l'administration.

Le rôle CREATE JOB permet de créer un *job*, un *Scheduler* ou un *program* dans son propre schéma.

Le rôle MANAGE SCHEDULER permet de créer des fenêtres de temps (*Windows*), des classes ou des groupes de fenêtres.

Le privilège EXECUTE permet à des utilisateurs d'avoir le droit d'utiliser un composant du *Scheduler*. Si l'option WITH GRANT OPTION est spécifiée, cet utilisateur pourra à son tour attribuer le privilège. Il s'agit d'un privilège objet.

Pour permettre à un utilisateur d'utiliser tous les *programs* d'un *job* il doit avoir le privilège GRANT ANY PROGRAM.

```
|Grant execute on calc_stats to charly ;
```

14.2.1. Privilèges utilisateurs

Pour permettre à un utilisateur de modifier un composant d'un *Scheduler* d'un autre schéma il doit posséder le privilège objet ALTER nécessaire pour cet objet.

Il pourra alors modifier tous les attributs du *job* sauf les attributs suivants :

- ⇒ Program_name, program_type, program_action, number_of_arguments
- ⇒ Modifier le PL/SQL du jobs



Jobs, *programs* et *schedules* sont créés dans le schéma de l'utilisateur.
Jobs, *Class*, *Windows* et *Window Groups* sont créés dans le schéma SYS.



Ainsi, pour créer un *job*, il n'est pas nécessaire d'avoir tous les privilèges concernant le *schedule*, la fenêtre de temps ou le groupe de fenêtre ; le privilège `CREATE JOB` suffit. Par contre il faut préfixer le job par le schéma `SYS`.

Par exemple si vous créez un job qui utilise la fenêtre `APPL_USER_WINDOWS` dans votre *schedules* et le programme `UPDATE_REPORT_TABS` :

⇒ Vous devez avoir le privilège `CREATE JOB`

Si le programme ne réside pas dans votre schéma, vous devez avoir le privilège objet `EXECUTE` pour le programme `UPDATE_REPORT_TABS` ou le privilège système `EXECUTE ANY PROGRAM`.

Vous devez qualifier la fenêtre par : `schedule => 'SYS. APPL_USER_WINDOWS'`

Si vous créez un job et que vous désirez l'affecter à une classe spécifique, vous devez avoir le privilège objet `EXECUTE` pour la classe de jobs en question ou le privilège système `EXECUTE ANY CLASS`.



Attention le privilège système `EXECUTE ANY CLASS` est à manipuler avec précaution.

14.2.2. Privilèges administrateurs

Pour administrer un scheduler il faut avoir les privilèges suivants :

- ⇒ `CREATE, DROP, ALTER JOB CLASS, WINDOWS` et `WINDOW GROUPS`
- ⇒ `STOP ANY JOB` pour avoir la possibilité d'utiliser l'option `FORCE`
- ⇒ `START` ou `STOP WINDOWS` prématurément

Pour attribuer tous les privilèges nécessaires à un utilisateur, il faut avoir le privilège `MANAGE SCHEDULER`. Il est affecté au rôle `SCHEDULER_ADMIN`.

Le rôle `SCHEDULER_ADMIN` est attribué par défaut au rôle `DBA` et permet d'effectuer toutes les tâches d'administration.

14.3. Créer et gérer un programme dans un schedule

Si vous avez une procédure appelée `MAJ_SCHEMA_STATS` qui collecte les statistiques pour un schéma, vous pouvez créer un programme pour appeler cette procédure comme illustré ci-dessous.

```
BEGIN
  DBMS_SCHEDULER.CREATE_PROGRAM
  (
    PROGRAM_NAME      => 'prog_stats2' ,
    PROGRAM_ACTION     => 'clo.maj_schema_stats' ,
    PROGRAM_TYPE       => 'stored procedure' ,
    ENABLED            => TRUE
  ) ;
```



```
DBMS_SCHEDULER.CREATE_JOB
(
  JOB_NAME           => 'clo.job_stats2' ,
  PROGRAM_NAME       => 'clo.prog_stats2' ,
  START_DATE         => '20-DEC-05' 07.00.00 AM Greenwich' ,
  REPEAT_INTERVAL    => 'FREQ=HOURLY ; INTERVAL=2' ,
  END_DATE           => '20-DEC-06' 07.00.00 AM Greenwich' ,
  COMMENTS           => 'Toutes les 2 heures pendant 1 an'
) ;
END ;
/
```

Ce script crée un job qui fait tourner le programme PROG_STAT2 toutes les 2 heures pendant une année, la date de début étant le 20 décembre 2005 à 7H00.



Pour créer un job qui utilise un programme dans un autre schéma, l'utilisateur qui crée ce job doit avoir le privilège d'accéder au programme.

14.4. Les JOBS

14.4.1. Créer un JOB

Plusieurs étapes sont nécessaires pour créer un job :

Spécifier les composants en utilisant la procédure CREATE_JOB

Spécifier la tâche à sauvegarder

Le programme et le schedule existant appelés

Le programme et les spécificités du schedule existant appelés.

Exemple

Créer un job qui exécute des scripts de sauvegardes toutes les nuits à 23H00 et qui commence cette nuit.

```
set echo on

BEGIN
  DBMS_SCHEDULER.CREATE_JOB
  (
    job_name           => 'clo.sauv' ,
    job_type           => 'executable' ,
    job_action         => 'd:\admin10\schedul\sauv_23.bat' ,
    start_date         => 'trunc(sysdate)+23/24' ,           /*cette nuit à 23H00 */
    repeat_interval    => 'trunc(sysdate+1)+23/24' ,         /* prochaine nuit à 23H00 */
    comments           => 'sauvegardes toutes les nuits à 23H00'
  ) ;
END ;
/
```



Le job est créé avec le statut `DISABLED` par défaut.
Il devient actif et utilisable par un scheduler dès qu'il est explicitement `ENABLED`.
Son nom est de la forme `[SCHEMA].JOB`.

Par défaut un job est créé dans le schéma courant.

Il est possible de créer un job dans un autre schéma à condition de spécifier le schéma dans lequel il doit être créé.

Le schéma propriétaire est l'utilisateur du job mais le job est exécutable avec les privilèges du propriétaire du job.

L'environnement « NLS » du job (date et heure) est l'environnement **de référence** pour l'exécution du job.

Le paramètre **JOB_TYPE** indique le type de programme appelé, il peut prendre les valeurs :

- ♦ `PLSQL_BLOCK` : bloc PL/SQL anonyme
- ♦ `STORED_PROCEDURE` : procédure cataloguée PL/SQL, Java ou externe.
- ♦ `EXECUTABLE` : programme exécutable en ligne de commandes (dos, unix).

Le paramètre **JOB_ACTION** indique le nom du programme à exécuter. Il dépend du paramètre **JOB_TYPE**.

Le paramètre **REPEAT_INTERVAL** permet de spécifier l'intervalle de réveil du programme appelé en utilisant une expression calendaire ou une date du type `DATE` ou `TIMESTAMP WITH TIME ZONE`.

La méthode de base pour établir la périodicité d'un job se fait en mettant l'attribut **REPEAT_INTERVAL** à la valeur de l'expression calendaire qui possède 3 parties principales :

- ⇒ Fréquence (la spécification d'une fréquence est obligatoire)
- ⇒ intervalle de répétition
- ⇒ éléments spécifiques (qui fournissent de l'information détaillée sur la période pendant laquelle le job doit tourner)

Expression datetime :

`REPEAT_INTERVAL => 'SYSDATE + 36/24'`

`REPEAT_INTERVAL => 'SYSDATE + 1'`

`REPEAT_INTERVAL => 'SYSDATE + 15/(24*60)' /* 15 minutes */`

Expression calendaire :

`REPEAT_INTERVAL => 'FREQ=HOURLY ; INTERVAL=4'`

`REPEAT_INTERVAL => 'FREQ=DAILY ;'`

`REPEAT_INTERVAL => 'FREQ=MINUTELY ; INTERVAL=5'`

`REPEAT_INTERVAL => 'FREQ=YEARLY ;`

`BYMONTH=MAR,JUN,SEP,DEC ; /* mois */`

`BYMONTHDAY=15' /* N°jour */`



Si vous voulez créer un *job* qui tourne toutes les 2 semaines, toutes les 5 minutes ou bien toutes les secondes, vous utiliserez une combinaison entre la fréquence et l'intervalle.

```
toutes les 2 semaines :      freq=weekly ;interval=2
toutes les 5 minutes :      freq=minutely ;      interval=5
toutes les secondes :       freq=secondly ;
```

Si vous avez besoin de spécifier des intervalles de définition plus complexes comme le 15^{ème} jour du mois, toutes les 4 semaines le lundi, ou, à 6H23 tous les mardis, il faut utiliser les clauses BY* pour fournir cette information complémentaire.

```
15ème jour du mois :          freq=monthly ;      bymonthday=15
tous les 4 semaines le lundi : freq=yearly ;
                                byweekno=4,8,12,16,20,24,28,32,36,40,44,48,52 ;
                                byday=mon
à 6H23 tous les mardis :      freq=weekly ;      byday=tue ;      byhour=6 ;
byminute=23
```



Oracle ne garantit pas que le job se déroule à l'heure exacte !
L'exécution se déclenche en fonction de la disponibilité des ressources machine.

14.4.2. Spécifier des *schedules* pour un *job*

Le temps pendant lequel un *job* démarre et se termine sont spécifiés en utilisant le type de donnée `TIMESTAMP WITH TIME ZONE`.

La précision d'un *schedule* est la seconde (pas moins). Malgré que le type de donnée `TIMESTAMP WITH TIME ZONE` soit plus précis, le *scheduler* arrondi tout de même à la seconde supérieure.

La procédure `CREATE_SCHEDULE` du package `DBMS_SCHEDULER` permet de sauvegarder le *schedule* comme un objet de schéma. Ceci permet d'utiliser le même *schedule* par plusieurs *JOB* ou *WINDOWS*.

Le paramètre `REPEAT_INTERVAL` pour un *schedule* doit être créé en utilisant l'expression calendaire. Vous ne pouvez pas utiliser des expressions `DATETIME` pour spécifier l'intervalle de répétition pour un *schedule* sauvegardé.

Le paramètre `END_DATE` pour un *schedule* sauvegardé correspond à la date après laquelle le *schedule* n'est plus valable.

Le *Scheduler* supporte intégralement toutes les fonctionnalités NLS fournies par la base de données. Par exemple vous pouvez utiliser tous les paramètres de type `NLS_TIMESTAMP_TZ_FORMAT` ainsi que le type de donnée `TIMESTAMP WITH TIME ZONE`, par exemple :



```
1 :00 :00p.m.  
13 :00 :00 hrs  
2003-04-15 8 :00 :00 US/Pacific  
8 :00 :00 -8 :00  
2003-01-31 09 :26 :50.124
```

14.4.3. Créer et utiliser des *schedules*

L'utilisation d'un *schedule* permet de gérer l'exécution planifiée d'une multitude de JOBS sans avoir à mettre à jour les définitions de cet ensemble de job. En effet, on utilise un *schedule* pour spécifier le temps d'exécution d'un JOB au lieu de spécifier le temps d'exécution du job dans la définition de celui-ci.

Si un *schedule* est modifié alors chaque job qui utilise ce *schedule* utilisera automatiquement le nouveau *schedule*.

La procédure CREATE_SCHEDULE du package DBMS_SCHEDULER permet de créer un *schedule*.

Le paramètre START_DATE représente la date à laquelle le *schedule* devient actif. Le *schedule* ne peut pas faire référence à toute autre date avant cette date.

Le *schedule* n'est plus actif après la valeur du paramètre END_DATE.

Il est possible de planifier des exécutions répétées en fournissant une expression calendaire pour le paramètre REPEAT_INTERVAL. Cette expression calendaire est utilisée pour générer la prochaine date d'exécution.

Les dates après le paramètre END_DATE ne seront pas incluses dans le *schedule*.

```
BEGIN  
  DBMS_SCHEDULER.CREATE_SCHEDULE  
  (  
    SCHEDULE_NAME => 'schedule_stats' ,  
    START_DATE    => SYSTIMESTAMP ,  
    END_DATE      => SYSTIMESTAMP + 30  
    REPEAT_INTERVAL => 'FREQ=HOURLY ; INTERVAL=4' ,  
    COMMENTS      => 'Toutes les 4 heures'  
  ) ;  
  
  DBMS_SCHEDULER.CREATE_JOB  
  (  
    JOB_NAME       => 'clo.job_stats' ,  
    PROGRAM_NAME   => 'clo.calc_stats2' ,  
    SCHEDULE_NAME  => 'schedule_stats'  
  ) ;  
END ;  
/
```

Dans cet exemple un *schedule* appelé SCHEDULE_STATS est créé avec :

- un intervalle de répétition de 4 Heures
- il commence immédiatement
- il dure pendant 30 jours.

Le *schedule* sera utilisé par la suite quand le job JOB_STAT sera créé pour déterminer quand le job sera exécuté.



14.5. Les JOB CLASS

14.5.1. Créer une JOB CLASS

La procédure `CREATE_JOB_CLASS` du package `DBMS_SCHEDULER` permet la création d'un `JOB CLASS`.

```
Exemple
EXECUTE DBMS_SCHEDULER.CREATE_JOB_CLASS
(
  JOB_CLASS_NAME      => 'class_jobs' ,
  RESOURCE_CONSUMER_GROUP => 'group_jobs' ,
  LOGGING_LEVEL       => DBMS_SCHEDULER.LOGGING_OFF
);
```

Une fois le `JOB CLASS` créé, les `JOBS` qui sont membres de ce `JOB CLASS` peuvent être spécifiés à la création des `JOBS` ou après leur création en utilisant la procédure `SET_ATTRIBUTE` du package `DBMS_SCHEDULER`.

Le *Scheduler* utilise le concept de `JOB CLASS` pour gérer l'allocation des ressources pour les différents `JOBS`. La configuration de l'allocation de ressource est faite par l'attribution d'un `JOB CLASS` à un `CONSUMER GROUP`.

Le `CONSUMER GROUP` auquel un `JOB CLASS` a été attribué peut être spécifié au moment de la création d'un `JOB CLASS` ou bien ultérieurement en utilisant la procédure `SET_ATTRIBUTE`.



Il existe un `JOB CLASS` par défaut appelé `DEFAULT_JOB_CLASS`

Si un job n'est pas associé avec un `JOB CLASS` alors le `JOB` appartient au `JOB CLASS` par défaut.

Le paramètre `DEFAULT_JOB_CLASS` est associé avec le `RESOURCE CONSUMER GROUP` par défaut appelé `DEFAULT_CONSUMER_GROUP`. Ce qui arrive quand un `RESOURCE CONSUMER GROUP` n'est pas spécifié à la création d'un `JOB CLASS`.

Les `JOBS` dans le `JOB CLASS` par défaut ou dans un `JOB CLASS` associé au `DEFAULT_CONSUMER_GROUP` peuvent ne pas avoir alloués assez de ressources pour accomplir leur tâche quand le *Ressource Manager* est activé.



Un `JOB CLASS` appartient toujours au schéma `SYS`. La création d'un `JOB CLASS` nécessite le privilège `MANAGE_SCHEDULER`.



14.6. Gestion des Logues de JOB

Par défaut tous les JOBS sont logués.

- ⇒ A la création d'un nouveau JOB CLASS, il existe des paramètres qui vont contrôler que l'information sera loguée et la persistance de cette information pourra être spécifiée.

14.6.1. Le package DBMS_SCHEDULER

Dans le package DBMS_SCHEDULER le paramètre LOGGING_LEVEL pour un JOB CLASS peut avoir une des valeurs constantes suivantes :

- ♦ LOGGING_OFF : aucun log est créé pour tous les jobs de ce CLASS.
- ♦ LOGGING_RUNS : le scheduler écrit des infos détaillées dans le log du job pour chaque exécution de chaque job dans ce CLASS.
- ♦ LOGGING_FULL : en plus de tracer chaque exécution dans le logue du job, le scheduler logue aussi toutes autres opérations exécutées pour tous les JOBS de ce JOB CLASS. Par exemple la création de nouveaux JOBS, l'activation ou la désactivation de nouveaux JOBS etc...



La vue DBA_SCHEDULER_JOB_LOG stocke une ligne pour chaque opération « loguée » du JOB.

Le paramètre LOG_HISTORY spécifie combien de jours une entrée de logue reste dans le fichier de log avant d'être supprimée.

Le job PURGE_LOG est créé automatiquement. Il supprime les anciennes entrées de logue une fois par jour. Ces entrées peuvent aussi être supprimées manuellement en utilisant la procédure : DBMS_SCHEDULER.PURGE_LOG.

```
EXECUTE DBMS_SCHEDULER.CREATE_JOB_CLASS
(
  JOB_CLASS_NAME      => 'class_jobs' ,
  RESOURCE_CONSUMER_GROUP => 'Group_jobs' ,
  LOGGING_LEVEL       => DBMS_SCHEDULER.LOGGING_RUNS ,
  LOG_HISTORY         => 30
);
```




14.6.2. Les logs des JOBS

La vue `DBA_SCHEDULER_JOB_LOG` affiche une ligne pour chaque opération effectuée par le `JOB` pendant son exécution.

```
SELECT job_name, operation, owner
FROM DBA_SCHEDULER_JOB_LOG
/
```

Vous pouvez purger les logs de `JOBS` :

- ⇒ Automatiquement à travers la valeur du paramètre `PURGE_LOG`
Le paramètre `PURGE_LOG` définit les conditions de purge des logs
Identique à l'utilisation de la procédure `DBMS_SCHEDULER.PURGE_LOG`

```
EXEC DBMS_SCHEDULER.PURGE_LOG(
    Log_history => 1,
    Job_name    => 'DEV_TEST_JOB1' ) ;
```

La durée de purge par défaut est 30 jours mais elle peut être modifiée en utilisant le paramètre `LOG_HISTORY`. Les valeurs pouvant être utilisées vont de 1 à 999.

Par exemple, pour spécifier la durée de purge de 60 jours pour le job `APPL_JOB_CLASS` il suffit d'utiliser la procédure :

```
EXEC DBMS_SCHEDULER.SET_ATTRIBUTE(
    'APPL_JOB_CLASS', 'log_history', '60' ) ;
```

La procédure `DBMS_SCHEDULER.PURGE_LOG` permet d'effectuer des purges manuellement. Elle contient les paramètres suivants :

- ♦ `LOG_HISTORY` : spécifie la durée de conservation. Cette valeur va de 0 à 999. Si la valeur « 0 » est positionnée, alors toutes les `LOGs` sont purgées.
- ♦ `WHICH_LOG` : définit le `JOB` ou la fenêtre à supprimer. Il prend les valeurs qui sont dans `JOB_LOG`, `WINDOW_LOG` et `JOB_AND_WINDOWS_LOG`.
- ♦ `JOB_NAME` : précise le nom du `JOB` des `LOGs` à purger. Vous pouvez spécifier une liste de noms de *Jobs* ou de *Job Classes*, séparés par des virgules

14.7. Les Windows

14.7.1. Créer une *WINDOW*

Le but d'une `WINDOW` est de changer le `RESSOURCE PLAN` qui est actif pour une période spécifique. La `WINDOW` est représentée par un intervalle de temps comme par exemple « chaque jour de 8H00 à 18H00 ». Ce type de `WINDOW` est paramétré comme un `schedule` qui spécifie un modèle de date de démarrage et une durée (en minutes).



Un *ressource plan* valable pour tout le system peut être associé avec une *Window* pour gérer l'utilisation globale des ressources pour les jobs et les sessions qui tournent dans cette *Window*.

Par exemple la priorité des JOBS change après une certaine période de temps. Ainsi il peut être important dans certaines situations de charger des jobs qui tournent la nuit et de leur allouer un pourcentage important des ressources de la base de données.

Or pendant la journée, les jobs applicatifs sont plus importants et doivent se voir allouer un pourcentage plus important de ressource.

Pour réaliser ceci, vous devez changer le *ressource plan* en utilisant un *schedule*.

⇒ le concept de *WINDOW* vous permet de le faire.



Pour créer une *Window*, il faut avoir le privilège `system MANAGE SCHEDULER`.
Les *Window* sont créées dans le schéma `SYS`.

Une *Window* est « ouverte » pendant une période de temps définie.

Le paramètre `DURATION` spécifie combien de temps cette *Window* restera ouverte.

La durée est spécifiée comme une donnée de type `INTERVAL DAY TO SECOND`.

Pour savoir quand ouvrir à nouveau la *Window*, Oracle utilise la valeur du paramètre `REPEAT_INTERVAL` qui sera évalué par rapport au paramètre `START_DATE`.

La priorité est spécifiée en utilisant le paramètre `WINDOW_PRIORITY`, la valeur par défaut étant `LOW_PRIORITY`.

```
BEGIN
DBMS_SCHEDULER.CREATE_WINDOW
(
  WINDOW_NAME => 'dec_nuit' ,
  RESOURCE_PLAN => 'fin_annee' ,
  START_DATE => '01-DEC-05 06.00.00 PM EST' ,
  REPEAT_INTERVAL => 'FREQ=DAILY; BYHOUR=18' ,
  DURATION => '0 12/000/00' ,
  END_DATE => '31-DEC-05 06.00.00 AM EST' ,
  COMMENTS => 'Every day at 6:00 PM'
) ;
END;
/
```

La *Window* devient active à 6H00 le 1er décembre 2005.

Le paramètre `DURATION` spécifie combien de temps cette *Window* restera ouverte.

La durée est spécifiée comme une donnée de type `INTERVAL DAY TO SECOND`.

La valeur « 0 12 :00 :00 » représente 0 jours 12 heures 0 minutes et 0 secondes. Ceci signifie que la *WINDOW* ferme à 6 heure le 2 decembre 2003.

La prochaine fois que la *WINDOW* sera ouverte est calculée en utilisant la valeur du `REPEAT_INTERVAL` qui sera évalué par rapport à 6 heures le 2 décembre 2005.

A 6h00 le 31 décembre 2005 la *WINDOW* sera fermée et désactivée.

Elle ne sera plus ouverte par la suite.

Pendant que la *Window* est appelée, `DEC_NUIT` sera ouverte, les ressources allouées pour les jobs seront déterminées par les éléments spécifiés dans le `RESSOURCE PLAN` appelé `FIN_ANNEE`.

Il n'y a aucune priorité spécifiée pour cette *Window*.



la priorité des *Windows* est significative quand plusieurs *Windows* définissent la même période.

14.7.2. Attribuer des priorités aux JOBS dans les WINDOWS

Les `JOB CLASS` sont utilisés pour catégoriser les jobs.

Un `JOB CLASS` est associé à un *Resource Consumer group*.

Les *resources plan* actifs déterminent les ressources allouées à chaque *Resource Consumer Group* et par là même à chaque *job class*.

Lorsque vous créez plusieurs *jobs* dans la base de données, vous devez spécifier quels *jobs* ont la priorité la plus haute.

Pour une *Window* en particulier, vous pouvez faire tourner plusieurs *jobs*, chacun ayant sa propre priorité.

Ainsi, il est possible d'avoir différents niveaux de classe ou de job.

- ⇒ La première priorité est le niveau de la classe dans le *Ressource Plan*.
- ⇒ La seconde priorité est la priorité du job dans la `JOB CLASS`.



Le niveau de priorité est approprié quand deux jobs de la même classe sont supposés démarrer au même moment.
La priorité n'est pas gérée entre des jobs de différentes `JOB CLASS`.

Pour définir la priorité il faut utiliser la procédure `SET_ATTRIBUTE` du package `DBMS_SCHEDULER`

Les priorités sont définies en utilisant des rangs allant de 1 à 5, le rang 1 définissant la priorité la plus haute.

```
BEGIN
DBMS_SCHEDULER.SET_ATTRIBUTE
(
  NAME    => 'job3',
  ATTRIBUTE => 'job_priority' ,
  VALUE   => 2
);
END;
/
```



Si aucune priorité n'est affectée au job, le rang 3 est affecté par défaut.
Pour afficher la priorité en cours des différents jobs, utiliser la requête :

```
SELECT JOB_NAME, JOB_PRIORITY
FROM    DBA_SCHEDULER_JOBS ;
```



14.8. Activer un composant du scheduler

Pour activer un composant du *Scheduler*, utiliser la procédure `ENABLE` du package `DBMS_SCHEDULER`, que ce soit pour un *Job*, un *Program* ou une *Window*.

```
Exemple
--activer le programme stats3
EXEC DBMS_SCHEDULER.ENABLE( 'clo.stats3' ) ;

-- desactiver le job job_stat du schema clo
EXEC DBMS_SCHEDULER.DISABLE( 'clo.job_stat' ) ;
```

De la même façon pour désactiver un composant du scheduler, utiliser la procédure `DISABLE` du package `DBMS_SCHEDULER`.

14.9. Gérer des composants du scheduler

14.9.1. Gérer un JOB

Pour exécuter un Job, utiliser la procédure `RUN_JOB` du package `DBMS_SCHEDULER`. Le Job s'exécute immédiatement dans votre session.

Le *Job* s'exécute dans votre session au lieu d'être exécuté par le Job Coordinateur et est exécuté comme un *Job* esclave.

De la même façon, pour arrêter l'exécution d'un Job, utiliser la procédure `STOP_JOB`. L'arrêt du *job* détruit sa définition.

Cette procédure possède deux arguments :

- ⇒ `JOB_NAME` : le nom du job
- ⇒ `FORCE` : définit la méthode avec laquelle le Job est arrêté.
- ⇒ Si elle est à `FALSE` (par défaut), le scheduler essaie de terminer le Job proprement. S'il échoue, une erreur est retournée.
- ⇒ Si elle est à `TRUE`, le Job s'arrête brutalement.



Pour utiliser le paramètre `FORCE`, il faut avoir le privilège système `MANAGE SCHEDULER`.

Pour supprimer un Job, utiliser la procédure `DROP_JOB` du package `DBMS_SCHEDULER`. Cette procédure possède deux arguments :

- ⇒ `JOB_NAME` : le nom du job
- ⇒ `FORCE` : supprime le Job même s'il est en train de s'exécuter. Positionné à `FALSE` par défaut.



Si l'argument `FORCE` est positionné à `TRUE`, tous les *Jobs* en cour d'exécution sont supprimés après avoir été arrêtés.

Si l'on spécifie le nom d'un `JOB CLASS` à la place du nom d'un Job, alors la totalité des Jobs contenus dans le `JOB CLASS` sont supprimé, mais pas le `JOB CLASS` lui-même.

```
-- executer un job
DBMS_SCHEDULER.RUN_JOB('clo.job3') ;

-- arrêter un job
DBMS_SCHEDULER.STOP_JOB('clo.job3') ;

-- supprimer des jobs en cours d'exécution
DBMS_SCHEDULER.DROP_JOB('clo.job3, clo.job5, sys.jobclass2') ;
```

14.9.2. Gérer un PROGRAM

Pour pouvez utiliser les procédures `ENABLE` et `DISABLE` du package `DBMS_SCHEDULER` pour activer ou désactiver un composant du scheduler.

Pour supprimer un programme, utiliser la procédure `DROP_PROGRAM` du package `DBMS_SCHEDULER`. Cette procédure contient deux arguments :

- ⇒ `PROGRAM_NAME` : nom du programme
- ⇒ `FORCE` : argument permettant de forcer la suppression des programmes
- ⇒ positionné à `TRUE`, tous les jobs qui référencent ce programme sont désactivés avant la suppression du programme
- ⇒ positionné à `FALSE` (par défaut), job référençant les programmes à supprimer qui doivent être désactivés un par un avant de pouvoir les supprimer.

```
-- activer un programme
EXECUTE DBMS_SCHEDULER.ENABLE('clo.prog1', 'clo.prog2');

-- désactiver un programme
EXECUTE DBMS_SCHEDULER.DISABLE('clo.prog1', 'clo.prog2');

-- supprimer un programme
EXECUTE DBMS_SCHEDULER.DROP_PROGRAM(-
    PROGRAM_NAME => 'clo.prog1', -
    FORCE         => TRUE);
```

14.9.3. Gérer un schedule

Un schedule peut être créé, modifié ou supprimé en utilisant les procédures adéquates :

- ♦ Pour Créer ⇒ `CREATE_SCHEDULE`
- ♦ Pour Modifier ⇒ `SET_ATTRIBUTE`
- ♦ Pour Supprimer ⇒ `DROP_SCHEDULE`



La procédure `SET_ATTRIBUTE` permet de modifier les attributs du *schedule*.

La suppression de plusieurs *schedules* est possible en listant leurs noms dans les arguments de la procédure `DROP_SCHEDULE`.

Si des jobs ou des Windows utilisent le *schedule* qui doit être supprimé, il faut alors forcer la suppression en utilisant le paramètre `FORCE`. Dans ce cas, les jobs ou les *windows* sont désactivées (`DISABLE`) avant la suppression du *schedule*.



Si vous n'êtes pas propriétaire du *schedule* vous devez posséder le privilège `ALTER` ou `CREATE ANY JOB`.

```
-- modifier l'heure de démarrage du schedule

EXECUTE DBMS_SCHEDULER.SET_ATTRIBUTE(-
    NAME    => 'clo.schedul3', -
    ATTRIBUTE => 'start_date',
    VALUE    => '01-JAN-2005 9:00:00
```

14.9.4. Gérer une fenêtre de temps : *Window*

Seule une Window peut être active à un moment donné. Lorsque le temps de démarrage d'une fenêtre est atteint la fenêtre de temps Window s'ouvre automatiquement.

La procédure `OPEN_WINDOW` du package `DBMS_SCHEDULER` permet d'ouvrir une fenêtre prématurément. L'ouverture d'une *Window* ferme n'importe quelle autre Window ouverte, même si la *window* a une priorité plus importante car l'attribut `FORCE` est positionné à `TRUE`.

L'intervalle de temps spécifié pour la *window* n'est pas altéré. Elle s'ouvrira donc normalement la fois suivante.

On peut modifier les paramètres de durée d'ouverture d'une fenêtre déjà ouverte à une durée plus importante. Les nouvelles valeurs sont prises en compte immédiatement.

```
-- ouverture de la window Win_nuit

DBMS_SCHEDULER.OPEN_WINDOW
(
    WINDOW_NAME => 'Win_nuit' ,
    DURATION    => '1 0:00:00'
) ;

-- fermeture de la window win_nuit

DBMS_SCHEDULER.CLOSE_WINDOW ( 'Win_nuit' ) ;
```

Pour fermer une fenêtre prématurément, il faut utiliser la procédure `CLOSE_WINDOW` du package `DBMS_SCHEDULER`. Les jobs sont arrêtés lorsque la Window se ferme s'ils contiennent le paramètre `stop_on_window_close` positionné à `TRUE`. Autrement, ils continuent de s'exécuter. Par contre, comme le plan de ressource change à cause de la fermeture de la Window, les ressources affectées aux jobs changent également.



Une Window peut être positionnée à `DISABLE` si elle n'est pas référencée par un job ou si elle est fermée. Pour rendre inactive une Window, il faut positionner le paramètre `FORCE` à `TRUE` en utilisant la procédure `DISABLE` du package `DBMS_SCHEDULER`.

La procédure `DROP_WINDOW` du package `DBMS_SCHEDULER` permet de supprimer une Window. La suppression d'une Window entraîne la désactivation de tous les jobs référencés par cette Window. Pour supprimer une Window active il faut positionner le paramètre `FORCE` à `TRUE`. La Window est alors fermée puis supprimée. Cette Window est également supprimée de tous les Windows Groups.

```
DBMS_SCHEDULER.DISABLE
(
  WINDOW_NAME => 'sys.Win_nuit' ,
  FORCE       => TRUE
) ;

DBMS_SCHEDULER.DROP_WINDOW
(
  WINDOW_NAME => 'Win_nuit, Win_mois',
  FORCE       => TRUE
) ;
```



Pour ouvrir, fermer, désactiver ou supprimer une *Window*, il faut avoir le privilège système `MANAGE_SCHEDULER`. Comme la *Window* est dans le schéma `SYS`, il faut la préfixer par `[SYS.]`.

14.9.5. *Priorité des Windows*

Le Scheduler ne valide pas les dates de démarrage et de fin gérées dans les Windows. Il est donc possible de créer des Windows avec des programmes de recouvrement.

Les schedulers ne sont pas arrêtés si la Window se ferme, sauf s'ils le paramètre `STOP_ON_CLOSE` est positionné à `TRUE`.

Les règles concernant ces recouvrements sont définies comme suit :

- ⇒ Si des fenêtres de même priorité se chevauchent, la fenêtre qui est active reste ouverte. Cependant, si le chevauchement se fait avec une fenêtre d'une priorité plus élevée, la fenêtre de plus faible priorité se ferme et la fenêtre de priorité plus élevée s'ouvre.
- ⇒ Si à la fin d'une fenêtre il y a plusieurs fenêtres définies, la fenêtre qui a le pourcentage de temps le plus élevé s'ouvre.
- ⇒ Une fenêtre ouverte qui est supprimée est automatiquement fermée.



14.9.6. Gérer les attributs des composants du scheduler

La procédure `SET_ATTRIBUTE` modifie les attributs d'un composant du *Scheduler*. Elle peut accepter des valeurs de plusieurs `DATA TYPE`.

```
BEGIN
DBMS_SCHEDULER.SET_ATTRIBUTE(
  Name      => 'MAX_FAILURES',
  Attribute => 'MAX_FAILURES',
  Value     => 3 );
END;
```

Chaque composant de Scheduler possède différents attributs qui peuvent être modifiés. Par exemple il est possible de modifier le `PROGRAM_TYPE` d'un programme. Par contre cet attribut n'est pas adapté à un `JOB` ou à une *Window*. De la même façon, il est possible de modifier le `JOB_CLASS` d'un `JOB`.

Pour positionner un attribut à la valeur `NULL`, utilisez la procédure `SET_ATTRIBUTE_NULL`. Cette procédure peut être utilisée pour initialiser tous les attributs.

```
EXEC DBMS_SCHEDULER.SET_ATTRIBUTE_NULL(
  'CHARLY.CALC_STATS', 'COMMENTS' );
```

Il existe 3 attributs qui peuvent être gérés au niveau global :

`DEFAULT_TIMEZONE` : utilisé par les `JOBs` et les *Windows* qui emploient une syntaxe calendaire pour leurs intervalles de répétition. Elle est utilisée pour `START_DATE` mais si la date n'est pas fournie, le fuseau horaire est recherché à partir de cet attribut.

`MAX_JOB_SLAVE_PROCESSES` : permet de placer un nombre maximum des processus définis pour une configuration et un chargement particulier. Même si le scheduler détermine automatiquement le nombre de processus optimal. Vous pouvez l'utiliser pour fixer une limite pour le Scheduler.

`LOG_HISTORY` : détermine le nombre de jours maintenus pour un `JOB` ou une *Window* avant une purge automatique. La valeur par défaut est 30.

```
EXEC DBMS_SCHEDULER.SET_SCHEDULER_ATTRIBUTE(
  Attribute => 'log_history',
  Value     => '14' );
```



La modification de la valeur d'un attribut prend effet immédiatement, même si le résultat ne se voit pas tout de suite

La procédure `GET_SCHEDULER_ATTRIBUTE` du package `DBMS_SCHEDULER` permet de retrouver les valeurs des différents attributs.



14.10. Vues du dictionnaire de données

Pour retrouver les valeurs des attributs d'un composant de Scheduler, interroger les vues

- *_SCHEDULER_JOBS
- *_SCHEDULER_PROGRAMS
- DBA_SCHEDULER_SCHEDULE
- DBA_SCHEDULER_WINDOWS

```
SELECT max_failures, job_priority, schedule_limit, logging_level
FROM DBA_SCHEDULER_JOBS
WHERE job_name = 'GET_STATS'
      AND job_creator = 'CHARLY'
/
```

Les vue du dictionnaire de données permettant de visualiser mes informations concernant le Scheduler ou ses composants sont :

- DBA_SCHEDULER_JOBS : informations sur les jobs
- DBA_SCHEDULER_JOBS_ARGS : liste des arguments d'un job
- DBA_SCHEDULER_RUNNING_JOBS : jobs en train de s'exécuter
- DBA_SCHEDULER_JOBS_LOG : LOGs des JOBS.
 - DBA_SCHEDULER_JOB_RUN_DETAILS : informations détaillées sur les jobs
 - DBA_SCHEDULER_PROGRAMS : informations sur les programmes
- DBA_SCHEDULER_PROGRAMS_ARGS : informations sur les arguments des programmes
- DBA_SCHEDULER_JOBS_CLASSES : informations sur les Job Classes.
- DBA_SCHEDULER_WINDOWS : informations sur les Windows
- DBA_SCHEDULER_WINDOW_DETAILS : informations détaillées sur les Windows
- DBA_SCHEDULER_WINDOW_LOG : informations sur les Logs des Windows

La vue DBA_SCHEDULER_JOB_LOG stocke une ligne pour chaque opération « *loguée* » du JOB.

La vue DBA_SCHEDULER_JOB_RUN_DETAILS affiche une ligne par JOB en cours d'exécution. Chacune des lignes contient des informations concernant l'exécution du JOB dans l'instance.



```
-- Visualiser le statut d'un job
--
SELECT job_name, program_name, job_type, state
FROM DBA_SCHEDULER_JOBS
WHERE owner = 'CHARLY'
/

-- Visualiser l'instance dans laquelle s'exécute le job
--
SELECT owner, job_name, running_instance, resource_consumer_group
FROM DBA_SCHEDULER_RUNNING_JOBS
WHERE owner = 'CHARLY'
/

-- Visualiser les arguments d'un job
--
SELECT value
FROM DBA_SCHEDULER_JOB_ARGS
WHERE owner = 'CHARLY'
      AND job_name = 'MON_JOB'
/

-- Auditer l'activité des jobs
--
SELECT owner, job_name, job_class, operation, status
FROM DBA_SCHEDULER_JOB_LOG
/

-- Visualiser les différents statuts d'un job pendant son exécution
--
SELECT job_name, status, error#, run_duration, cpu_used
FROM DBA_SCHEDULER_JOB_RUN_DETAILS
/

-- Visualiser les informations des programmes qui s'exécutent
--
SELECT program_name, program_type, program_action
FROM DBA_SCHEDULER_PROGRAMS
/
```