



Oracle

PL/SQL

Auteur : Clotilde Attouche

Version 1.1

Du 6 Mars 2010

Sommaire

1	La base de données Oracle 11g.....	5
2	La documentation.....	10
3	Les Outils d'accès à la base	11
1.1	L'outil iSQL*Plus	11
1.2	L'outil SQL*Plus	12
1.2.1	Environnement de travail.....	13
1.2.2	Le prompt SQL*Plus.....	15
1.2.3	Quelques commandes SQL*Plus	16
1.2.4	Générer un fichier résultat appelé « spool »	16
1.2.5	Déclarer un éditeur.....	17
1.3	Utilisation de paramètres	17
4	Le dictionnaire de données	18
5	La base Exemple.....	20
1.3.1	Modèle Conceptuel de Données Tahiti.....	20
1.3.2	Les contraintes d'intégrité.....	21
1.3.3	Règles de passage du MCD au MPD.....	21
1.3.4	Modèle Physique de données Tahiti :	22
6	Le langage SQL.....	23
1.4	Notion de schema	23
1.5	Règles de nommage	23
7	Le langage PL/SQL	24
1.6	Structure d'un programme P/SQL.....	24
1.7	Les différentes types de données.....	25
1.7.1	Conversion implicite	26
1.7.2	Conversion explicite.....	26
1.8	Les variables et les constantes	26
1.9	Les structures.....	27
8	Les instructions de bases.....	28
1.10	Condition	28
1.11	Itération	28
1.12	L'expression CASE.....	29
1.13	Expression GOTO	31
1.14	Expression NULL	32
1.15	Gestion de l'affichage	33
9	Tables et tableaux	34
10	Les curseurs	36



1.16	Opérations sur les curseurs	36
1.17	Attributs sur les curseurs	38
1.18	Exemple de curseur	38
1.19	Exemple de curseur avec BULK COLLECT	39
1.20	Variables curseur	40
11	Les exceptions	42
1.21	Implémenter des exceptions utilisateurs	43
1.22	Implémenter des erreurs Oracle	45
1.23	Fonctions pour la gestion des erreurs	47
1.24	Exemples de programmes PL/SQL	48
1.25	Compilation native du code PL/SQL	49
1.26	Transactions autonomes	50
12	Procédures, Fonctions et Packages	54
13	Procédures	55
1.26.1	Créer une procédure	55
1.27	Modifier une procédure	57
1.28	Correction des erreurs	57
14	Fonctions	60
1.29	Créer une fonction	60
15	Packages	62
16	Triggers	66
1.30	Créer un trigger	67
1.31	Activer un trigger	71
1.32	Supprimer un Trigger	71
1.33	Triggers rattaché aux vues	72
1.34	Triggers sur événements systèmes	72
17	Architecture du moteur PL/SQL	74
1.35	Procédures stockées JAVA	77
1.36	Procédures externes	77
1.36.1	Ordres partagés	77
1.37	Vues du dictionnaire de données	78
18	Les dépendances	79
1.38	Dépendances des procédures et des fonctions	79
1.38.1	Dépendances directes	79
1.38.2	Dépendances indirectes	80
1.38.3	Dépendances locales et distantes	80
1.38.4	Impacte et gestion des dépendances	80
1.39	Packages	81
19	Quelques packages intégrés	82
1.40	Le package DBMS_OUTPUT	82





1.41 Le package UTL_FILE..... 82

1.42 Le package DBMS_SQL 83



1 La base de données Oracle 11g

Oracle Database 11g représente la nouvelle génération de la gestion des informations en entreprise, qui permet de faire face aux exigences qu'imposent la croissance rapide des volumes de données, l'évolution constante de l'environnement et la nécessité de fournir une qualité de service maximale tout en réduisant et en contrôlant les coûts informatiques.

Oracle Database 11g reste centré sur le grid computing : il permet de constituer des matrices de serveurs et de systèmes de stockage économiques capables de traiter les données de façon rapide, fiable et évolutive, en supportant les environnements les plus exigeants, qu'il s'agisse de datawarehouse, de transactionnel ou de gestion de contenus.

Oracle Database 11g intègre de multiples nouveautés et améliorations. Ainsi, Oracle 11g offre une performance améliorée du stockage sur fichiers, des fonctionnalités renforcées pour la sécurité, d'importantes améliorations de performances pour Oracle XML DB, et des fonctions nouvelles pour l'OLAP et le datawarehouse.

Oracle 11g multiplie les outils de gestion et introduits de nouvelles fonctionnalités d'auto gestion et d'automatisation. *Automatic SQL*, *Partitioning Advisor* ou *Support Workbench* accompagnent les administrateurs pour améliorer les performances et les informer le plus rapidement possible des incidents. Ainsi

- *Oracle Flashback Transaction* permet de revenir plus facilement sur une erreur de transaction et des dépendances.

- Parallel Backup and Restore* augmente les performances des sauvegardes sur les grosses bases.

- *Hot Patching* permet d'appliquer les mises à jour sans arrêter les bases.

- *Data Recovery Advisor* accompagne les administrateurs pour déterminer intelligemment les plans de secours.

- *Oracle Fast Files* adopte un comportement proche des systèmes de fichiers (file systems), ce qui est un gage de performances avec les LOBs (Large Objects) ou des fichiers contenant du texte, des images, ou des données XML, objets tridimensionnels, etc.

- *Oracle XML DB* permet de stockées et manipulées nativement les données XML. Le langage XML se révèle lourd, et avec cette approche Oracle 11g limite la dégradation de ses performances. De même la base supporte les interfaces standard *XQuery*, *Java Specification Requests* (JSR)-170 et *SQL/XML*.

- *Oracle Transparent Data Encryption* permet de crypter les données des tables, des index ou encore les données stockées de type LOB.

- *Cubes OLAP*, apporte des fonctionnalités de datawarehouse (fermes de données), Oracle 11g embarque les cubes OLAP pour visualiser les informations stockées, ce qui autorise le développement de requêtes au format SQL.

- *Continuous Query Notification* notifie immédiatement les changements apportés dans la base.

- avec *Query Result Caches*, requêtes et fonctionnalité de la base ou d'applications tierces sont placées en cache afin de les accélérer ou de les réutiliser.

- *Database Resident Connection Pooling* est destiné aux applications qui ne sont pas multithreadées (ou les développeurs qui ne maîtrisent pas cette technologie parallèle), en particulier pour les systèmes web, Oracle 11g permet de créer des 'pool' de connexions.



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 destinée à des serveurs possédant 4 processeurs.
- ◆ **Standard Edition ONE** - la gamme destinée à un bi-processeur.
- ◆ **Personal Edition** - La gamme pour l'utilisateur indépendant (développeur, consultant, ...), elle utilise un noyau Enterprise Edition.

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.

Quatre nouvelles options offrent des possibilités exclusives de gestion des données pour Oracle Database 11g Enterprise Edition :

- ⇒ . Oracle Real Application Testing
- ⇒ . Oracle Advanced Compression
- ⇒ . Oracle Total Recall
- ⇒ . Oracle Active Data Guard

Oracle Real Application Testing aide ces clients à réduire les délais, les risques et les coûts de test de ces modifications de leur environnement informatique, de façon contrôlée et économique. Outil de tests et de gestion des changements, cet outil est bienvenu là où les infrastructures et environnements sont plus que jamais multiples.

Oracle Advanced Compression intègre de nouveaux mécanismes de compression applicables à tous les types de données permettant d'atteindre des taux de compression de 2x ou 3x, et parfois plus. Associé à de nouveaux mécanismes de partitionnement, Oracle Advanced Compression permet de déployer dans la base de données des stratégies de gestion du cycle de vie des informations, sans avoir à modifier les applications, afin de réduire encore plus les besoins de stockage.

Oracle Total Recall permet de conserver et de retrouver les historiques des données modifiées, mais aussi d'en simplifier l'accès. Les administrateurs peuvent intervenir plus tôt dans les processus, ce qui apporte une nouvelle dimension de temps dans la gestion des données, comme le tracking (*suivi, en temps réel des flux d'informations*), les audits ou le respect des règles.

Oracle DATA GUARD porte la protection des données jusqu'aux risques de défaillances des systèmes et de désastres. L'application permet simultanément d'écrire et récupérer les données



d'une base de données, ce qui augmente les performances et apporte une solution économique de 'Disaster Recovery'. **Oracle Active Data Guard** peut être employé pour améliorer la performance des bases de données de production en transférant vers une base de données physique secondaire des opérations requérant beaucoup de ressources, telles que certaines requêtes ou les sauvegardes. Cette solution améliore fortement le retour sur investissement pour une base de données physique de secours, car celle-ci peut être utilisée à la fois pour la protection en cas de panne générale et pour l'amélioration de la qualité de service de l'environnement de production.

Notion de Grid Computing

La base de données 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 11g permet à ces machines d'interopérer ; l'ensemble étant considéré comme une seule ressource unifiée.

- ⇒ Chaque ressource est vue comme un service.

Il est possible de mettre en place des réseaux grille nationaux, voire mondiaux.

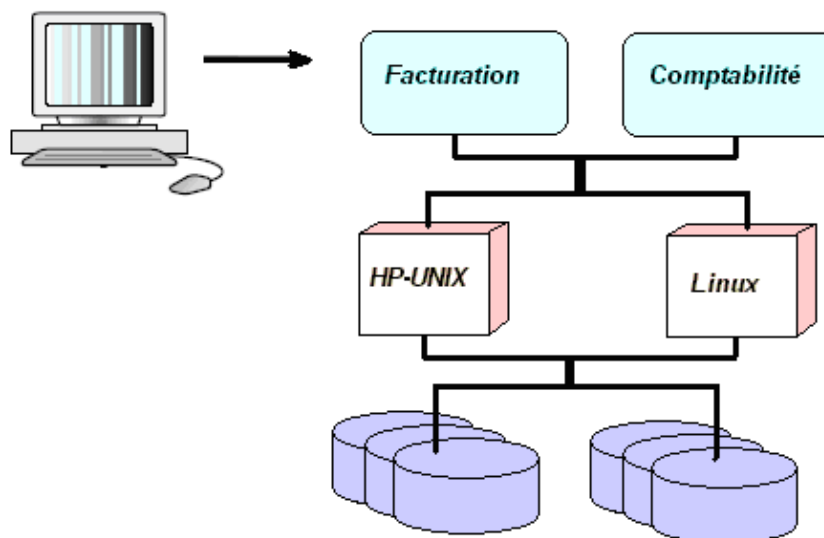
Ainsi chaque nouveau système peut être rapidement mis à disposition à partir du pool de composants

Exemple d'application en Grid Computing

Les deux applications présentées ci-dessous, Facturation et Comptabilité se partagent des ressources de deux serveurs.

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





Informatique en Grille



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.

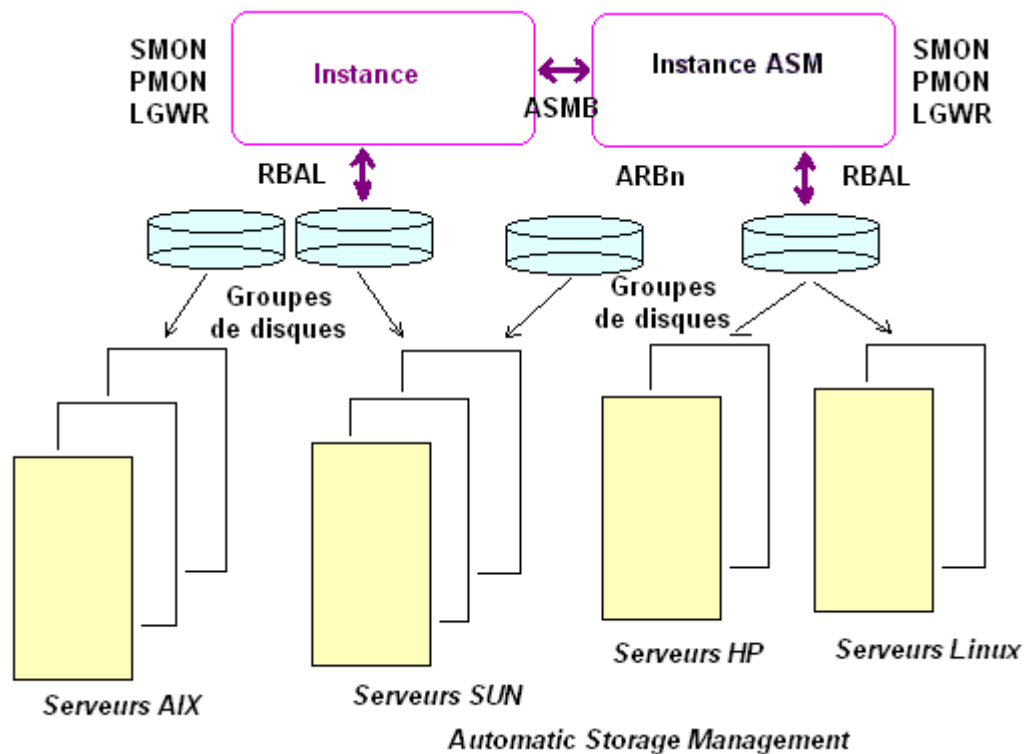
Au fur et à mesure de l'ajout et de l'abandon de disques, l'ASM actualise automatiquement la répartition des données.

Pour utiliser ASM vous devez démarrer une instance 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 base de données (ensemble de fichiers constituant la base) mais gère les *metadata*s 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.





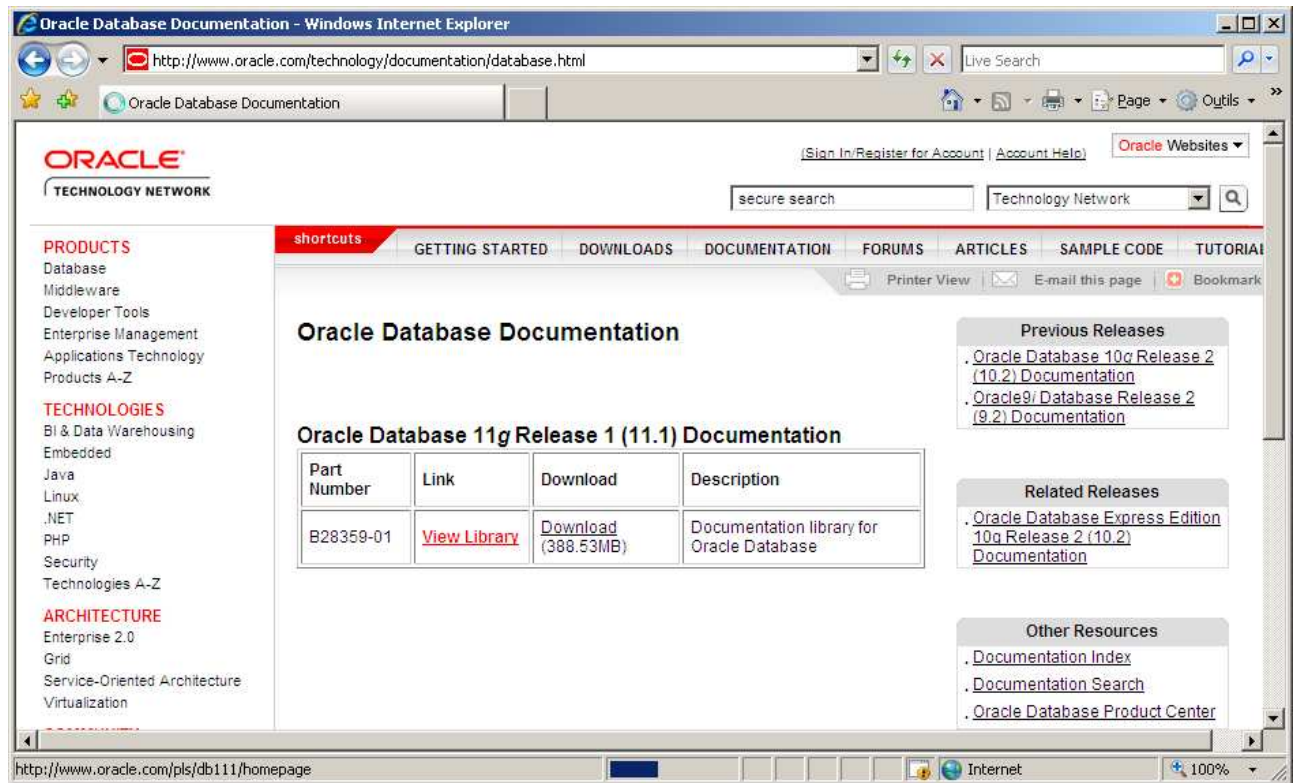
Outils de développement

Oracle offre l'accès à un choix d'outils et processus de développement, avec de nouvelles fonctionnalités comme **Client Side Caching**, **Binary XML**, un nouveau compilateur Java, l'intégration native avec *Microsoft Visual Studio 2005* pour les applications .NET, **Oracle Application Express** pour les outils de migration, ou encore **SQL Developer** pour coder rapidement les routines SQL et PL/SQL.



2 La documentation

La documentation Oracle est également consultable à partir du serveur : <http://www.oracle.com>



Oracle Database Documentation - Windows Internet Explorer

http://www.oracle.com/technology/documentation/database.html

Oracle Database Documentation

ORACLE TECHNOLOGY NETWORK

(Sign In/Register for Account | Account Help) Oracle Websites

secure search Technology Network

PRODUCTS

- Database
- Middleware
- Developer Tools
- Enterprise Management
- Applications Technology
- Products A-Z

TECHNOLOGIES

- BI & Data Warehousing
- Embedded
- Java
- Linux
- .NET
- PHP
- Security
- Technologies A-Z

ARCHITECTURE

- Enterprise 2.0
- Grid
- Service-Oriented Architecture
- Virtualization

shortcuts GETTING STARTED DOWNLOADS DOCUMENTATION FORUMS ARTICLES SAMPLE CODE TUTORIAL

Printer View E-mail this page Bookmark

Oracle Database Documentation

Oracle Database 11g Release 1 (11.1) Documentation

Part Number	Link	Download	Description
B28359-01	View Library	Download (388.53MB)	Documentation library for Oracle Database

Previous Releases

- [Oracle Database 10g Release 2 \(10.2\) Documentation](#)
- [Oracle9i Database Release 2 \(9.2\) Documentation](#)

Related Releases

- [Oracle Database Express Edition 10g Release 2 \(10.2\) Documentation](#)

Other Resources

- [Documentation Index](#)
- [Documentation Search](#)
- [Oracle Database Product Center](#)

http://www.oracle.com/pls/db111/homepage

La documentation Oracle est également consultable à partir du serveur : <http://tahiti.oracle.com>



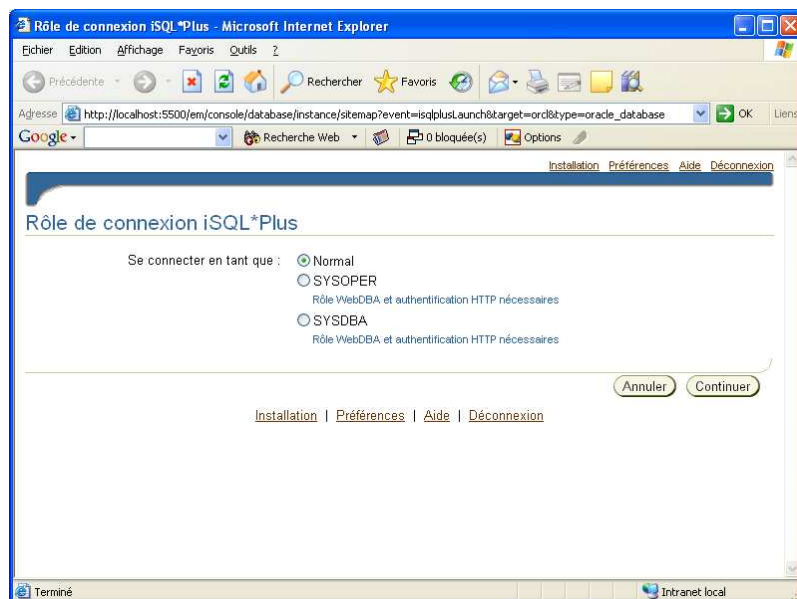
3 Les Outils d'accès à la base

Trois outils sont présents pour accéder à une base de données Oracle

- ⇒ iSQL*Plus, peut être utilisé en application indépendante ou connecté à un référentiel Oracle *Management Server* (OMS)
- ⇒ SQL*Plus (sqlplus), interface d'accès à la base de données en mode commande
- ⇒ *Oracle Enterprise Manager* (OEM), appelé Grid Control ou Database Control.
 - *Database control* est créé à la création d'une base oracle et ne permet d'administrer graphiquement que cette base de données
 - *Grid control* est un outil qui permet d'administrer une ferme de bases de données (oracle ou non oracle).

1.1 L'outil iSQL*Plus

Outil Internet d'accès à une base de données Oracle, permettant d'écrire des requêtes SQL (d'une façon plus ou moins graphique).



Par défaut, seule la connexion en tant qu'utilisateur « normal » (non SYSDBA ou SYSOPER) est autorisée.

Par contre, la connexion en tant qu'utilisateur SYSDBA ou SYSOPER est protégée par une authentification au niveau du serveur HTTP



Pour l'autoriser, il faut au choix :

- ◆ Ajouter des entrées (utilisateur / mot de passe) à l'aide de l'utilitaire *htpasswd* dans un fichier d'authentification du serveur HTTP (défini par défaut dans le fichier de configuration *isqlplus.conf* à :
ORACLE_HOME\sqlplus\admin\iplusdba.pw
- ◆ Désactiver l'authentification du serveur HTTP pour ce type de connexion
(directive <Location /isqlplusdba> dans le fichier de configuration *isqlplus.conf*)

Lors d'une connexion SYSDBA ou SYSOPER, l'URL est modifiée en :

⇒ [http://serveur\[:port\]/isqlplusdba](http://serveur[:port]/isqlplusdba)

1.2 L'outil SQL*Plus

Outil ligne de commande nommé SQLPLUS.

```
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

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

-- sous dos -----
set ORACLE_SID=TAHITI

-- connection sans fichier de mot de passe
SQL> connect /as sysdba
Connecté.

SQL> show user
USER est "SYS"

-- sous unix -----
Export ORACLE_SID=TAHITI

-- Connexion avec un fichier de mots de passe
SQL> connect sys/secret as sysdba
Connecté.

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



1.2.1 Environnement de travail

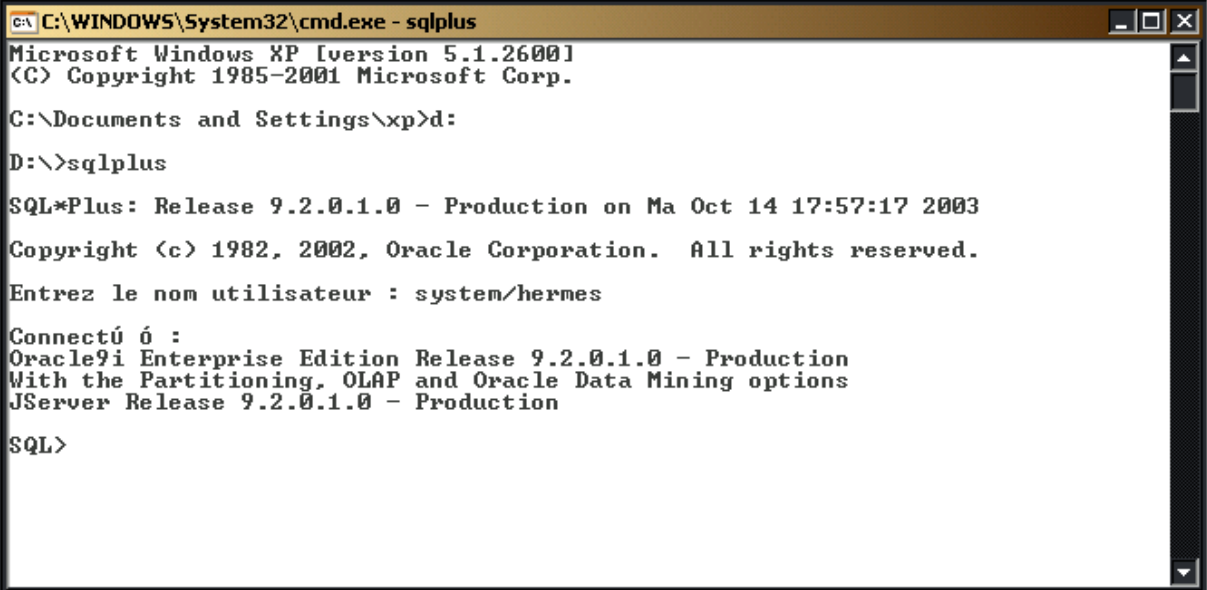
SQL*PLUS est avant tout un interpréteur de commandes SQL. Il est également fortement interfacé avec le système d'exploitation. Par exemple, sous UNIX, on pourra lancer des commandes UNIX sans quitter sa session SQL*PLUS.

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.

Pour lancer SQL Plus sans se connecter à une base de données utilisez la commande :

```
C:\> sqlplus /nolog
```

Pour démarrer une session SQL Plus sous dos il suffit de se mettre en commande DOS puis d'exécuter la commande SQL PLUS .



```
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>
```

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

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

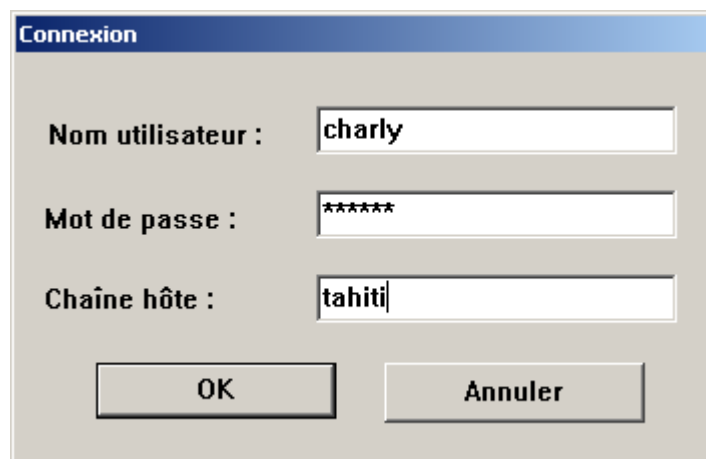


Nous avons installé au préalable les couches clients ORACLE sur le Poste Client. Cette installation est simple, il suffit de renseigner les modules à installer et le protocole TCP/IP à utiliser (il peut y en avoir plusieurs sur une même machine).

Depuis le groupe ORACLE, double cliquer sur l'icône SQL*Plus ...



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



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.



1.2.2 Le prompt 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.



SQL*Plus est un outil composé de commandes de mise en forme et d'affichage

⇒ A ne pas confondre avec des commandes SQL.



1.2.3 Quelques commandes SQL*Plus

- ◆ `COL ADRESSE FORMAT A20`, formater l'affichage d'une colonne ADRESSE sur 20 caractères
- ◆ `COL PRIXUNIT FORMAT 99.99`, formater l'affichage d'une colonne PRIXUNIT
- ◆ `CLEAR COL`, ré-initialiser la taille des colonnes par défaut
- ◆ `SET LINESIZE 100`, reformater la taille de la ligne à 100 caractères
- ◆ `SET PAUSE ON`, afficher un résultat page par page
- ◆ `SHOW USER`, visualiser le user sous lequel on est connecté
- ◆ `CONNECT` , se connecter à l'instance
- ◆ `User/MotPass@adresseServeur` , permet de changer de session utilisateur
- ◆ `CLEAR SCREEN`, ré-initialiser l'écran
- ◆ `SET SQLPROMPT TEST>` , afficher le prompt SQL en : TEST>
- ◆ `DESC Nom_Table`, afficher la structure d'une table ou d'une vue
- ◆ `@ nom_fichier`, permet d'exécuter le contenu d'un fichier sql
- ◆ `/` , ré-active la dernière commande
- ◆ `SET ECHO ON/OFF`, affiche ou non le texte de la requête ou de la commande à exécuter
- ◆ `SAVE nom_fichier [append|create|replace]` , permet de sauvegarder le contenu du buffer courant dans un fichier « .sql ».
- ◆ `TIMING ON|OFF`, provoque l'affichage d'informations sur le temps écoulé, le nombre d'E/S après chaque requête
- ◆ `TI ON|OFF`, provoque l'affichage de l'heure avec l'invite
- ◆ `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.
- ◆ `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

1.2.4 Générer un fichier résultat appelé « spool »

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
```

1.2.5 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.

1.3 Utilisation de paramètres

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

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

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



4 Le dictionnaire de données

C'est un ensemble de tables et de vues qui donne 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
- ◆ Sont en fait uniquement 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 de performances sont stockées dans le fichier de contrôle, et disponibles à l'ouverture de celui-ci (voir démarrage et arrêt d'une base Oracle).



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

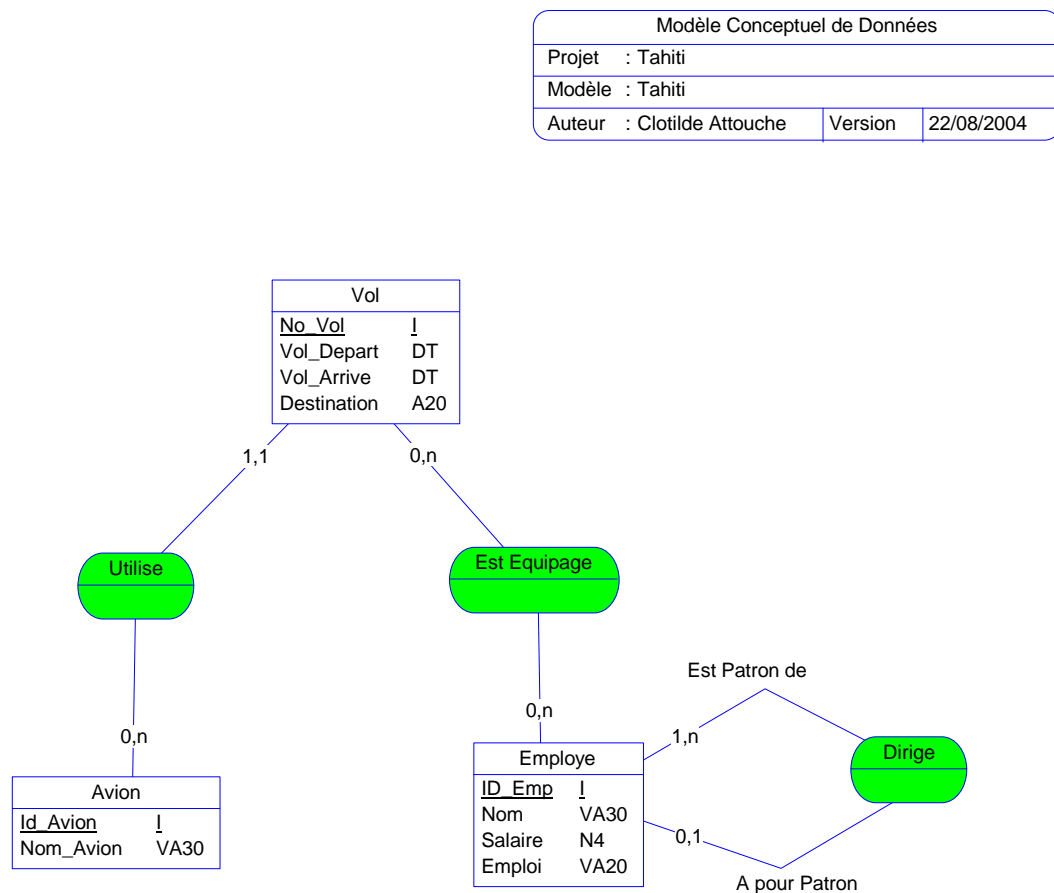
```
V$INSTANCE  
V$DATABASE  
V$SGA  
V$DATABASE  
V$PARAMETER
```



5 La base Exemple

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

1.3.1 Modèle Conceptuel de Données Tahiti



1.3.2 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

1.3.3 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 :

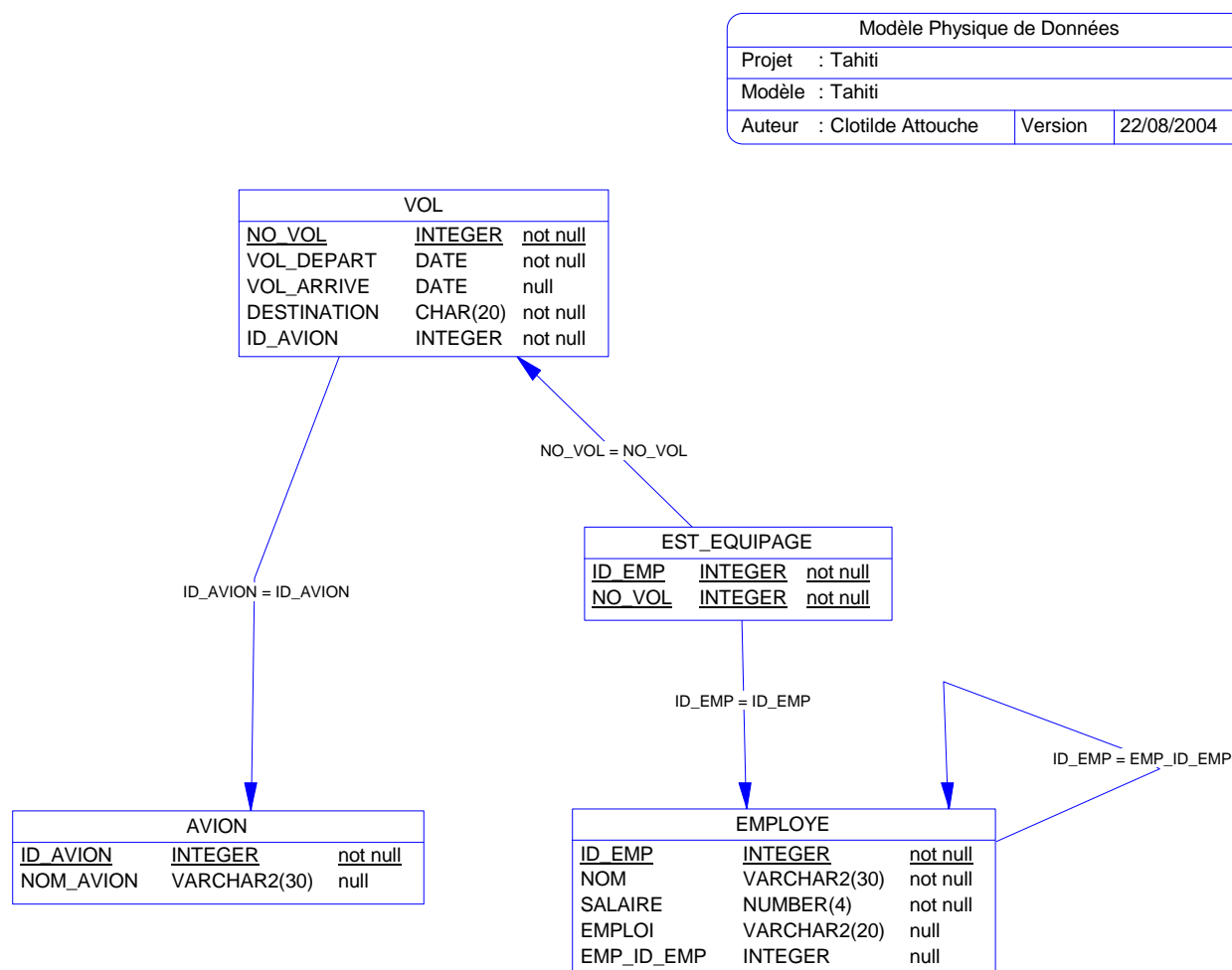
- ⇒ Les entités deviennent des tables
- ⇒ Les identifiants des entités deviennent les clés primaires de ces tables
- ⇒ Les relations ternaires, dont toutes les cardinalités sont 0,N ou 1,N de chaque côté de la relation deviennent des tables
- ⇒ La concaténation des identifiants des entités qui concourent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



1.3.4 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.



6 Le 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`

1.4 Notion de schema

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, séquences et synonymes
- ◆ Programmes PL/SQL (*procédures, fonctions, packages, triggers*)

1.5 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



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

1.6 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.



1.7 Les différents types de données

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

TYPE	VALEURS
BINARY-INTEGER	entiers allant de -2^{31} à 2^{31})
POSITIVE / NATURAL	entiers positifs allant jusqu'à $2^{31} - 1$
NUMBER	Numérique (entre -2^{418} à 2^{418})
INTEGER	Entier stocké en binaire (entre -2^{126} à 2^{126})
CHAR (n)	Chaîne fixe de 1 à 32767 caractères (différent pour une colonne de table)
VARCHAR2 (n)	Chaîne variable (1 à 32767 caractères)
LONG	idem VARCHAR2 (maximum 2 gigaoctets)
DATE	Date (ex. 01/01/1996 ou 01-01-1996 ou 01-JAN-96 ...)
RAW	Permet de stocker des types de données binaire relativement faibles(≤ 32767 octets) idem VARCHAR2. Les données RAW ne subissent jamais de conversion de caractères lors de leur transfert entre le programme et la base de données.
LONG RAW	Idem LONG mais avec du binaire
CLOB	Grand objet caractère. Objets de type long stockés en binaire (maximum 4 giga octets) Déclare une variable gérant un pointeur sur un grand bloc de caractères, mono-octet et de longueur fixe, stocké en base de données.
BLOB	Grand objet binaire. Objets de type long (maximum 4 giga octets) Déclare une variable gérant un pointeur sur un grand objet binaire stocké dans la base de données (Son ou image).



NCLOB	Support en langage nationale (NLS) des grands objets caractères. Déclare une variable gérant un pointeur sur un grand bloc de caractères utilisant un jeu de caractères mono-octets, multi-octets de longueur fixe ou encore multi-octets de longueur variable et stocké en base de données.
ROWID	Composé de 6 octets binaires permettre d'identifier une ligne par son adresse physique dans la base de données.
UROWID	Le U de UROWID signifie Universel, une variable de ce type peut contenir n'importe quel type de ROWID de n'importe quel type de table.
BOOLEAN	Bouléen, prend les valeurs TRUE, FALSE, NULL

1.7.1 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.

1.7.2 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.

1.8 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.

1.9 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;
/
```



8 Les instructions de bases

1.10 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;
```

1.11 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;
```



1.12 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 mariée
                        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 mariée = 'O' THEN 'Madame'
        WHEN sexe = 'F' AND mariée = 'N' THEN 'Mademoiselle'
        ELSE 'Inconnu'
    END CASE
FROM
    employe
/
```

Il est possible d'avoir des instructions CASE imbriquées.

1.13 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

1.14 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_detaille;
Else
  NULL ; -- ne rien faire
END IF ;
```



1.15 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 re-direction (attention aux fichiers de spool qui grossissent inconsidérément).



9 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;  
/
```



10 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
```

1.16 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;
```

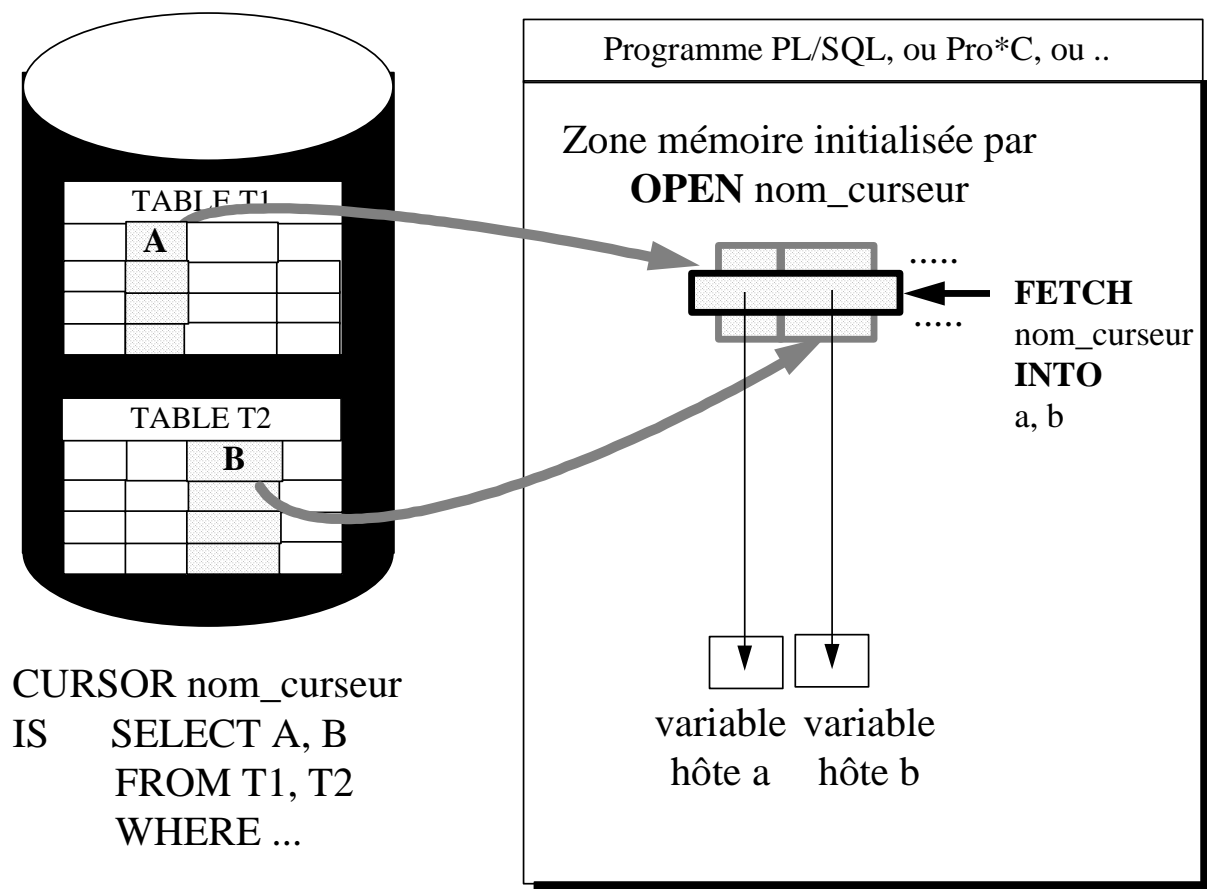


⇒ **CLOSE**

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 :



1.17 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`

1.18 Exemple de curseur

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

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

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

Voyage_marquise.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 voyage 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.

1.19 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;
/

```

1.20 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;
/
```



11 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
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
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 initialisé d'un tableau PL/SQL.
- ◆ `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.
- ◆ `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.
- ◆ `ZERO_DIVIDE` : tentative de division par zéro.

1.21 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
```

1.22 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;
```



1.23 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 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.

```



1.24 Exemples de programmes PL/SQL

Script insérant des lignes dans la table `EMPLOYEE` 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 ;
/
```

1.25 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 librairies 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.`

1.26 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.

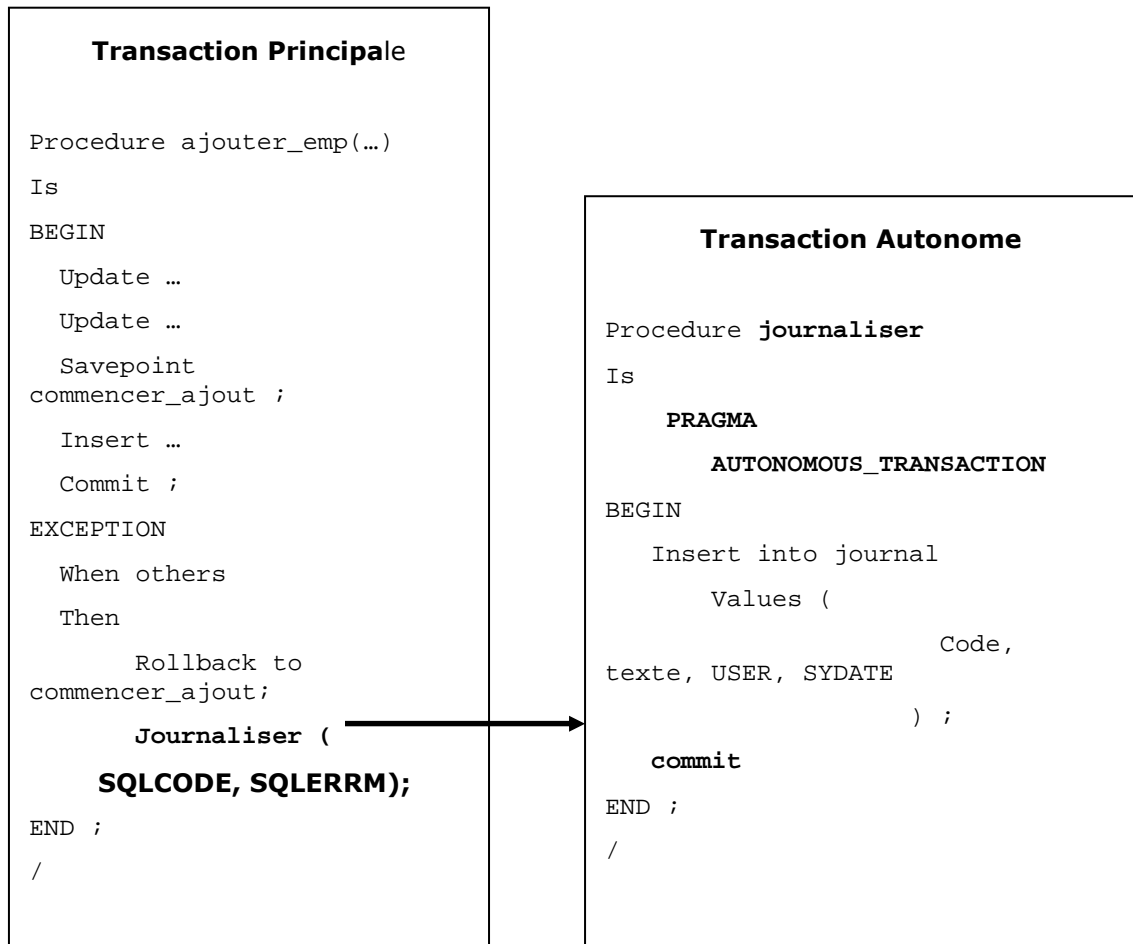
A partir de la version 8i d'Oracle, il est possible d'exécuter et de sauvegarder ou d'annuler certains ordres DML (`INSERT`, `UPDATE`, `DELETE`) sans affecter la totalité de la transaction.

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.

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.*



12 Procédures, Fonctions et Packages

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

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

Les procédures (fonctions) permettent de :

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

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

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



13 Procédures

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

1.26.1 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.
```

1.27 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.

1.28 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.



14 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)`.

1.29 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;

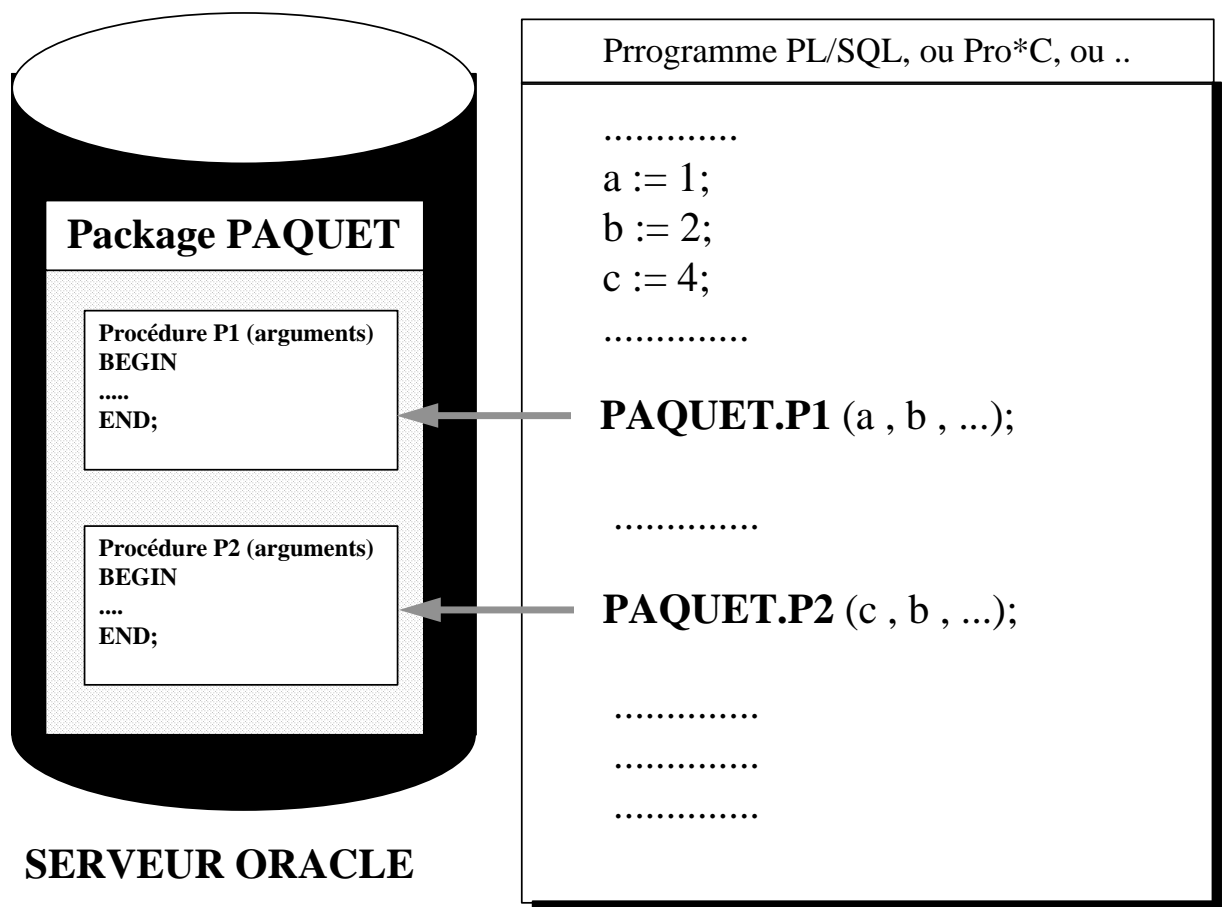
begin

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



15 Packages

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



Principe général d'utilisation d'un package



La structure générale d'un package est la suivante :

```
package_general.sql
CREATE OR REPLACE PACKAGE nom_package IS

définitions des types utilisés dans le package;
prototypes de toutes les procédures et fonctions du package;

END nom_package;
/

CREATE OR REPLACE PACKAGE BODY nom_package IS

déclaration de variables globales;

définition de la première procédure;

définition de la deuxième procédure;

etc. ...

END nom_package;
/
```

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

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

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

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

Le package se termine par / sur la dernière ligne.



Exemple

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

```
paquet1.sql
create or replace package ges_emp is

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

function moyenne_salaire (v_Id_employe IN NUMBER)
return NUMBER;

end ges_emp;
/

create or replace package body ges_emp is

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

end augmente_salaire;

function moyenne_salaire (v_Id_employe IN NUMBER)
return NUMBER
is
valeur NUMBER;
begin
select avg(salaire)
into valeur
from employe
groupe by emp_id_emp;
return (valeur);
end moyenne_salaire;

end ges_emp;
/
```



Compilation

```
SQL> @paquet1

Package created.

Package body created.
```

Exécution

Exécution de la procédure augmente_salaire du package ges_emp

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

PL/SQL procedure successfully completed.
```

Tests

```
SQL> select * from 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.



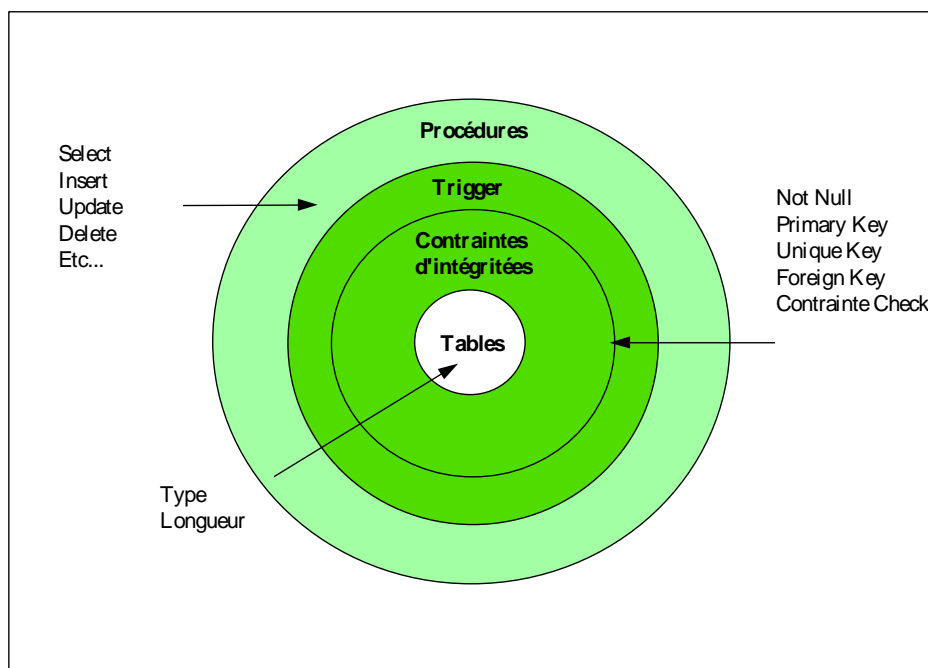
16 Triggers

Un trigger permet de spécifier les réactions du système d'information lorsque l'on « touche » à ses données. Concrètement il s'agit de définir un traitement (un bloc PL/SQL) à réaliser lorsqu'un événement survient. Les événements sont de six types (dont trois de base) et ils peuvent porter sur des tables ou des colonnes :

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

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

Serveur :



Vue générale des contraintes



Les TRIGGERS permettent de :

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

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

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

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

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

1.30 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



1.31 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.
```

1.32 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.



1.33 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, son 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.

1.34 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
- ⇒ Before logoff
- ⇒ {before|after} alter
- ⇒ {before|after} analyse
- ⇒ {before|after} {audit|noaudit}
- ⇒ {before|after} {comment|DDL|drop|rename|truncate}
- ⇒ {before|after} create
- ⇒ {before|after} grant
- ⇒ {before|after} revoke




```
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
before 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;
/
```



17 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.

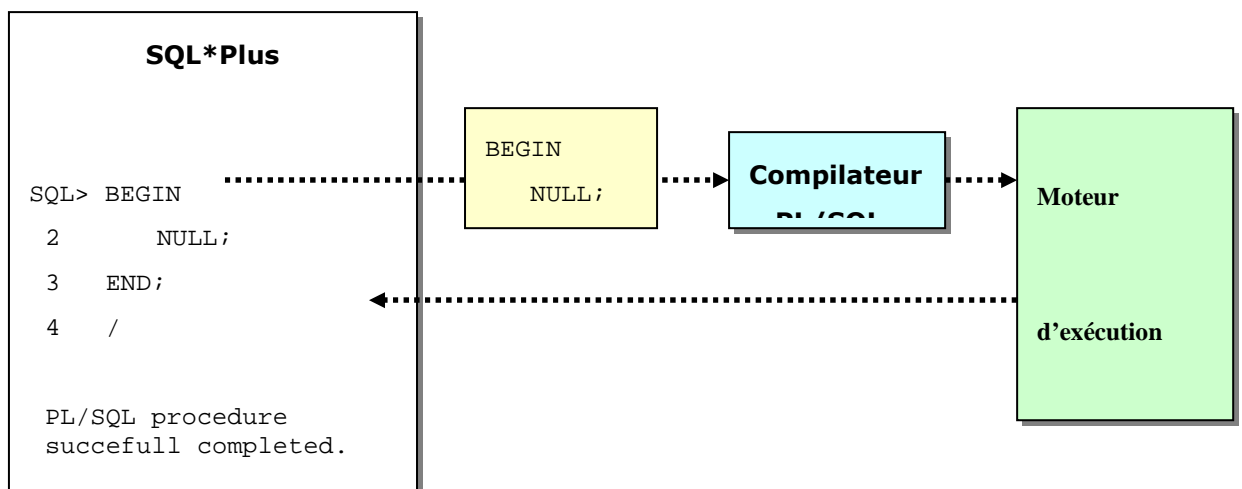
⇒ **Moteur d'exécution PL/SQL**

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.

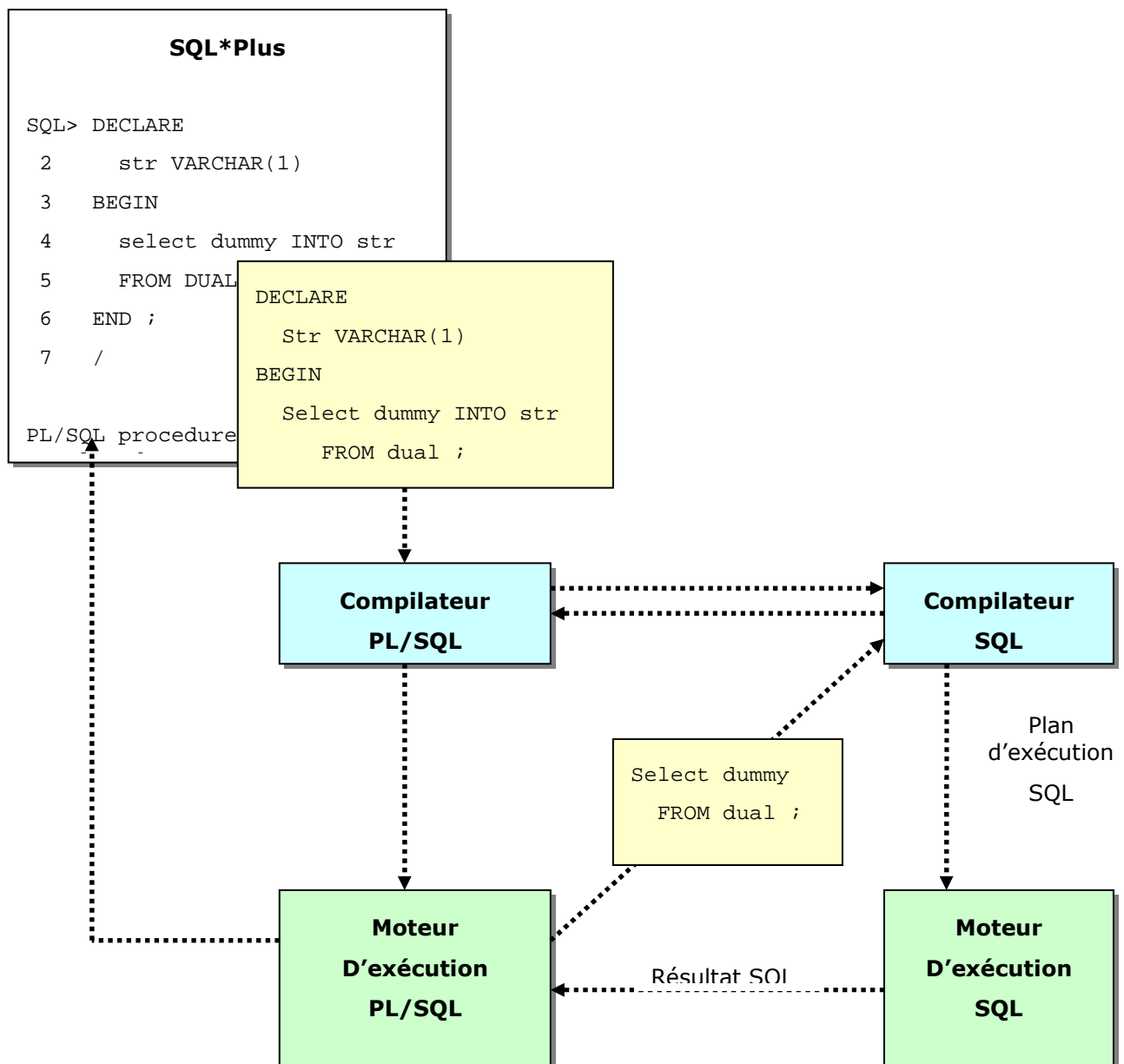
Regardons l'exécution d'un bloc anonyme qui ne fait rien présentée ci-dessous :



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.

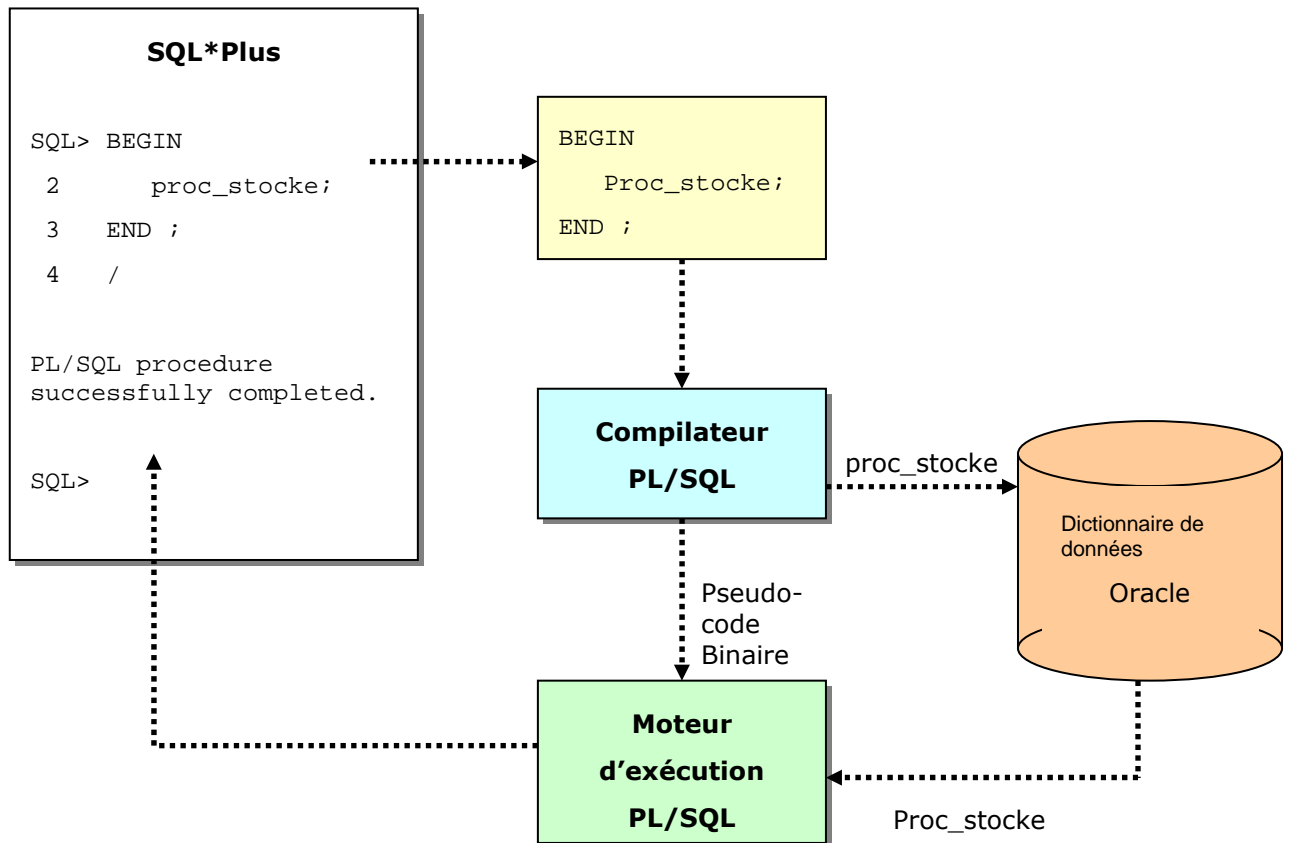
Ajoutons une requête imbriquée au bloc PL/SQL et regardons les modifications qu'elle entraîne.



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 générait 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.



1.35 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.

1.36 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.

1.36.1 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
```



1.37 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.



18 Les dépendances

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

1.38 Dépendances des procédures et des fonctions

Il existe deux type de dépendance :

- ◆ Dépendance directe
- ◆ Dépendance indirecte

1.38.1 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`



1.38.2 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.

1.38.3 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.

1.38.4 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 :




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

1.39 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  
;
```



19 Quelques packages intégrés

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

1.40 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.

1.41 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.

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.

1.42 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.

