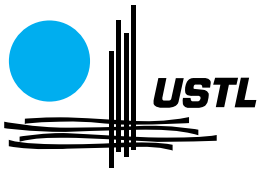


TP de Bases de Données
SQL - PL/SQL

Guide de l'utilisateur

Jean-Claude Marti

1997

Correction 2007

UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

SQL

Le langage SQL est constitué de plusieurs couches indépendantes mais présentant des similarités syntaxiques marquées.

On distingue :

- le Langage de Définition des Données (LDD (fr) ou DDL(us))
- le Langage de Manipulation des Données (LMD ou DML)
- le Langage de Sécurité des Données (LSD ou DSL)
- le Langage de Gestion de Transaction (LGT ou TML)

1 Le Langage de Definition des Données (LDD/DDL)

1.1 Types d'attributs

Les difficultés d'entente au sein de la commission de normalisation du langage ont eu pour résultat que toutes les déclarations des types de données inventées par les principaux éditeurs de logiciel ont été incluses dans la norme. Certains SGBD les acceptent toutes ou presque, d'autres sont plus limitatifs.

Le tableau suivant dresse la liste des plus courants pour Oracle, MySQL et PostgreSQL. Les types spécifiques à ORACLE sont donnés en italique, les types spécifiques à MYSQL sont marqués d'une astérisque.

1.1.1 Type alphanumérique

CHAR	Un caractère
CHAR(n)	Chaîne de `n` caractères. (1 par défaut)
VARCHAR(n), *VARCHAR2(n)*	Chaîne dynamique d'au plus n caractères
Text, *LONG*	Chaîne pouvant atteindre jusqu'à 65000 caractères.

1.1.2 Type numérique

INT^*, INTEGER	Nombre entier signé
SMALLINT, $INT2^*$	entier codé sur 2 octets
MEDIUMINT, $INT4^*$	entier 4 octets
BIGINT, $INT8^*$	entier 8 octets
NUMBER[(n[,d])]	Nombre de `n` chiffres dont éventuellement `d` après la virgule.
REAL, FLOAT, $FLOAT4^*$	Réels 4 octets
DOUBLE PRECISION, $FLOAT8^*$	Réels 8 octets
NUMERIC[(m,e)], DECIMAL[(m [,e])]	nombre réel de mantisse à m chiffres et d'exposant e
RAW (n)	Binaire de longueur `n`.

Le type RAW est utilisé pour stocker des images par exemples. Les valeurs correspondantes sont échangées sous forme hexadécimale.

1.2 Autres types

DATE	structure composée de champs et stockant le siècle, l'année, le mois, le jour, l'heure, les minutes et les secondes
TIME	autre format date
TIMESTAMP	estampille tamporelle
$INTERVAL^*$	intervalle de dates
ROWID	Pointeur de ligne

REM1 : La représentation du type `DATE` ne permet pas directement la saisie et l'affichage. Il faut pour cela passer par des fonctions de conversion telles que `to_date` et `to_char` sous Oracle.

REM2 : Indépendemment des types, la valeur `NULL` est une valeur spéciale qui dénote l'absence de valeurs. Un entier, une chaîne, une date peuvent avoir pour valeur `NULL`.

1.3 Gestion des tables

1.3.1 Création de tables

Syntaxe générale :

```
CREATE TABLE [user.]nomTable (attribut type [contrainte], ...) [AS SELECT ...];
```

L'ordre de création SQL d 'une table peut prendre deux formes.
exemple de la 1ère forme (Oracle) :

```
CREATE TABLE EMP(numemp number, nom varchar2(30), dateNaiss date,
                        salaire number(10,2), emploi varchar2(20));
```

exemple de la 2de forme (Oracle), utilisant l'option `AS SELECT`, qui permet créer une table à partir du résultat de l'interrogation de tables ou de vues :

```
CREATE TABLE VENDEURS(numvd number, nom char(20), dateNaiss date,
    salaire number(10,2)) AS SELECT * FROM EMP where emploi='VENDEUR';
```

1.4 Modification d'une table

1.4.1 Ajout d'une colonne

Syntaxe :

```
ALTER TABLE <nomTable> ADD ( <attribut> <type>, ...);
```

cette commande permet de rajouter des attributs (ou des contraintes) oublié(e)s dans une première approche. Si la table contient déjà des données, les nouveaux attributs auront la valeur NULL pour les tuples présents.

1.4.2 Modification d'un type d'attribut

Syntaxe :

```
ALTER TABLE <nomTabl>e MODIFY (<attribut> <type>, ...);
```

Il est possible de modifier le type d'un ou de plusieurs attributs (ou d'une contrainte si son nom est connu), par exemple d'étendre la dimension d'un type chaîne.

```
ALTER TABLE emp MODIFY nom varchar(50);
```

Aucun problème si la table est vide. S'il y a déjà des données, il faut que le changement de type soit compatible avec celui des données présentes. Les type tels que `integer`,`number`, ... peuvent être automatiquement convertis en `char(n)` ou `varchar(n)`, à condition que n soit assez grand ; l'inverse est également vrai si la chaîne ne contient que des caractères numériques. En cas de problèmes, il faut sauvegarder les données dans une table transitoire et les rapatrier en utilisant une fonction de conversion.

1.4.3 Supression d'une colonne

```
ALTER TABLE <nomTable>  DROP [COLUMN] <nomColonne>;
```

1.4.4 Modification du nom d'une colonne

```
ALTER TABLE <nomTable> RENAME <ancienNom> TO <nouveauNom>;
```

1.4.5 Modification du nom d'une table

```
RENAME <ancienNom> TO <nouveauNom>
```

1.4.6 Suppression d'une table

```
DROP TABLE <nomTable>;
```

1.5 Contraintes d'intégrité

Il existe plusieurs types de contraintes d'intégrité : d'unicité, de référence et de domaine.
La syntaxe générale d'une déclaration de contrainte est de la forme :
`CONSTRAINT <nomContrainte> <typeContrainte>`

1. **Clé primaire**
   Les contraintes d'unicité permettent d'assurer l'unicité de la clé et son existence par la clauses : `PRIMARY KEY` Elle est parfois accompagnée par la clause `NOT NULL` précisant que la valeur de l'attribut correspondant ne peut être absente. La clause `PRIMARY KEY` entraîne la création automatique d'un index sur le ou les attributs correspondants. Lorsqu'elle est invoquée, il est évident que la clause `NOT NULL` devient inutile.
   exemples : `Create table T1(num number PRIMARY KEY, ...);`
   `Create Table T2(nump number, numt number, PRIMARY KEY(nump, numt), ...);`
   Dans les deux exemples précédents, il est évident que l'index gérant la clé primaire sera anonyme, c.à.d nommé par le SGBD, et à sa manière. Il vaut bien mieux nommer systématiquement toutes contraintes en utilisant la clause (`CONSTRAINT nomContrainte`), de façon à pouvoir les gèrer par la suite (clause `Alter table`), ou les retrouver facilement dans le dictionnaire des données.
   exemple :

```
CREATE TABLE DetailVentes
  (venteId NUMBER,
   numPiece NUMBER,
   ...,
    CONSTRAINT PK_dtvnt PRIMARY KEY (venteId, numPiece));
```

2. Les contraintes de référence permettent d'assurer la liaison entre des attributs de 2 tables différentes :
   `FOREIGN KEY (attribut, ...) REFERENCES [user.]table(attribut,...)`
   Le processeur d'intégrité du SGBD permet d'assurer la cohérence des données de liaison en cas de mise à jour ou de suppression d'une clé. A cet effet, on utilise, avec les actions référentielles (`ON DELETE` et `ON UPDATE`), les clauses additionnelles :
   - `CASCADE` : L'opération (DELETE ou UPDATE) effectuée sur la table référencée est lancée sur les attributs de même valeur de la table référençante.
   - `SET NULL` : donne à l'attribut référençant la valeur NULL lorsque l'on manipule la valeur correspondante de l'attribut référencé.
   - `SET DEFAULT` : impose une valeur par défaut à l'attribut référençant.
   - `NO ACTION` : inhibe le processeur d'intégrité. L'opération menée sur la table référencée est sans objet sur la table référençante.

```
CREATE TABLE DetailVentes
  (venteId NUMBER,
   numPiece NUMBER,
   ...,
    CONSTRAINT PK_dtvnt PRIMARY KEY (venteId, numPiece),
   CONSTRAINT FK_ref1 FOREIGN KEY (venteId) REFERENCES scott.vente(venteId)
      ON DELETE SET NULL ON UPDATE CASCADE,
   CONSTRAINT FK_ref2 FOREIGN KEY (numPiece) REFERENCES scott.piece(numpiece)
      ON DELETE SET NULL ON UPDATE CASCADE);
```

3. Les contraintes de domaines permettent de vérifier l'appartenance des valeurs d'un attribut à son domaine de définition. Elles s'expriment par la clause : `CHECK <condition>`

```
CONSTRAINT CK_sxsexe CHAR,  check (sexe IN ('M','F'))
CONSTRAINT CK_sal check (salaire NOT NULL)
CONSTRAINT CK_vidck CHECK (venteId > 0)
CONSTRAINT CK_npck CHECK (numPiece > 0)
```

Il arrive qu'il soit impossible de déclarer des contraintes de référence lors de la création des tables (cas des références croisées), ou qu'un oubli ait été fait à ce moment, ou encore que l'on décide, au niveau d'une base existante, de rajouter ou de modifier des contraintes.

L'ajout, comme la suppression peut se faire par altération du profil de la table :

```
alter table emp add constraint fk_empag foreign key(numagence) references agences.numagence;
alter table detailVentes drop constraint fk_ref2;
```

La modification de définition d'une contrainte se fera par suppression de la contrainte, suivi d'un ajout avec la nouvelle définition.

1.6 Gestion des vues

Une vue est une pseudo-table. Du point de vue de l'utilisateur, il n'y a pas de différence par rapport à une table de la base. Du point de vue du système, c'est du code précompilé et stocké, prêt à être exécuté. Lors de l'interrogation d'une vue, le texte de l'interrogation est combiné à celui de la déclaration pour constituer la requête effective. On peut construire une vue à partir d'autres tables et vues.

Syntaxe de création d'une vue :

`CREATE VIEW nomVue [(alias, alias, ...)] AS interrogation [WITH CHECK OPTION] ;`

L'interrogation se fait par un SELECT (voir plus loin le DML de SQL)

La clause WITH CHECK OPTION spécifie que les insertions et mises à jour réalisées à travers la vue ne pourront affecter des tuples que la vue ne peut atteindre. Elle peut être utilisée dans le cas d'une vue construite sur une autre vue.

exemples :

```
CREATE VIEW e AS SELECT nomEmp, 12*salaire as salaireAnnuel
      FROM emp WHERE numDept=20;‘

CREATE VIEW x(ne, nom, ndpt)
   AS SELECT numEmp, nomEmp, numDept FROM emp
    WHERE numDept in (SELECT DISTINCT numDept FROM dept)
     WITH CHECK OPTION CONSTRAINT vx;
```

Cet exemple montre qu'on a le choix entre le fait de définir le nom des colonnes de la vue en les précisant lors de sa création ou en utilisant des alias au sein de la requête d'interrogation associée.

La suppression de la vue est réalisée par :

`drop view <nomVue>`

1.7 Séquences

Une séquence joue le rôle de distributeur de numéro. Elle constitue une série unique de numéros, chaque tirage modifiant la valeur du numéro courant.

Syntaxe :

`CREATE SEQUENCE [user.]nomSequence [INCREMENT BY n] [START WITH n]`

Par défaut, l'incrément est de 1 et la valeur de départ vaut 0.

Une séquence est munie de 2 pseudo-colonnes : NEXTVAL et CURRVAL. Leur syntaxe d'appel est nomSequence.NEXTVAL et nomSequence.CURRVAL.

```
CREATE SEQUENCE seq_emp;
INSERT INTO EMP VALUES(seq_emp.nextval, 'TOTO', 10000.00);
```

L'invocation de NEXTVAL remet automatiquement à jour la valeur de CURRVAL, ce qui permet d'attribuer à chaque nouvel employé un numéro à chaque fois différent.

REM : sous ORACLE, Pour connaître la valeur de CURRVAL, on peut utiliser la pseudo-table `DUAL`. `DUAL` est une pseudo-table à un seul tuple et d'un attribut `DUMMY`, de valeur 'X', dont le seul rôle est de permettre le respect de la syntaxe de l'interrogation qui impose la clause FROM lors de l'utilisation de l'instruction SELECT : `SELECT eseq.currval FROM dual;`

La suppression de la séquence est réalisée par : `drop sequence <nomSequence>`

1.8 Création de synonymes

Syntaxe :
`CREATE SYNONYM [user.]nomSym FOR [proprietaire.]nomTable;`
Un synonyme est un lien vers une autre table et permet d'utiliser cette dernière à partir d'un alias :

```
SQL > CREATE SYNONYM piece FOR TP.PIECE;

      synonym created

SQL > select * from piece;
```

Dans cet exemple, la table dont le véritable nom est TP.PIECE sera désormais désignée par PIECE.
La destruction de ce synonyme se ferait par la commande :
`drop synonym piece`

1.9 Création d'un index

Le rôle d'un index est de permettre un accès soit direct, soit direct-séquentiel à des valeurs d'une table. C'est une structure liée à la table, qui doit être gérée (avec un coût non nul) lors de toute opération d'écriture. Elle peut correspondre à une table de hachage ou un B-arbre. Actuellement, ce dernier type de structure est le plus couramment adopté. Dans le cas ou les valeurs sont uniques (clés primaires par exemple), il faut utiliser le mot-clé UNIQUE qui garantit le respect de cette unicité.

`CREATE [UNIQUE] INDEX nomIndex ON table(attribut [ASC | DESC], ...);`
exemple : `CREATE INDEX indemp ON emp(numDept);`

2 Le Langage de Manipulation des Données (LMD)

Ce langage est constitué de quatres commandes : l'interrogation (SELECT), l'insertion (INSERT), la mise à jour (UPDATE) et la suppression de lignes (DELETE).
La commande d'interrogation fait apparaître à divers niveaux des expressions et des conditions.
Une *expression* peut avoir plusieurs formes :

- une colonne, une constante, ou une valeur spéciale (ROWID sous Oracle), du texte, un nombre, sequence.CURRVAL, sequence.NEXTVAL, NULL, ROWNUM (Oracle), LEVEL, SYSDATE (Oracle), UID, USER.
- une fonction
- une combinaison d'expressions liées par des opérateurs arithmétiques, la clause PRIOR, l'opérateur de concaténation de chaîne ||
- une liste d'expressions entre parenthèses.

On peut trouver une expression derrière une clause SELECT, dans la condition des clauses WHERE et HAVING, dans les clauses CONNECT BY, START WITH, ORDER BY, dans la clause VALUES d'un INSERT, dans la clause SET d'un UPDATE.

Une *condition* est constituée d'expressions reliées par des opérateurs logiques et de comparaison, les opérateurs [NOT] IN, [NOT] BETWEEN, IS [NOT] NULL, [NOT] EXISTS, ANY, ALL.
On rencontre des conditions après les clauses WHERE, START WITH, CONNECT BY et HAVING.
Expressions et conditions utilisent tous deux des opérateurs et des fonctions SQL.

2.1 Opérateurs

2.1.1 Opérateurs communs

On dispose des opérateurs arithmétiques usuels (+, -, *, /) ainsi que de l'opérateur de concaténation de chaînes ||.

```
SELECT (1.186*prix)-ristourne FROM produits ...
SELECT nom || prenom FROM emp ...
```

2.1.2 opérateurs de comparaison

Ce sont les opérateurs usuels du langage C (la différence est représentée par `!=`), plus quelques autres.

OPERATEURS	EXEMPLES
Comparaison	WHERE salaire = 10000
différence	WHERE salaire != 10000
<, >, <=, >=	WHERE salaire >= 10000
IN	WHERE job IN ('PROGRAMMEUR','ANALYSTE')
NOT IN	WHERE salaire NOT IN (SELECT salaire FROM emp WHERE ...
ANY	WHERE salaire=ANY (SELECT salaire ...
ALL	WHERE (salaire,commission) >= ALL ((14000,3000),(12000,6000))
[NOT] BETWEEN	WHERE A BETWEEN 1 AND 9
[NOT] EXISTS	WHERE EXISTS (SELECT salaire FROM emp WHERE numdept=300)
[NOT] LIKE	WHERE nom LIKE 'HITCH%
IS [NOT] NULL	WHERE job IS NOT NULL

IN est un opérateur d'appartenance à un sous-ensemble. Ce dernier peut être défini par une liste de constantes entre parenthèses ou être constitué par le résultat d'une interrogation (Select).

L'opérateur LIKE utilise le caractère % pour désigner un nombre indéterminé de caractères dans la valeur examinée, et le caractère _ pour repérer exactement 1 caractère inconnu :

`LIKE '_ITCH%'` ou encore `LIKE '%TCH%'`, `LIKE '__TCH%'`.

2.1.3 Opérateurs ensemblistes

UNION	tuples de 2 ensembles de même schéma SELECT ... UNION SELECT ...
INTERSECT	tuples communs à 2 ensembles de même schéma SELECT ... INTERSECT SELECT ...
MINUS	tuples de A qui ne sont pas présents aussi dans B SELECT ... MINUS SELECT ...

2.1.4 Autres opérateurs

DISTINCT élimine les tuples en double dans le résultat d'une interrogation.

PRIOR détermine la relation père-fils dans une association réflexive.

2.2 Fonctions

2.2.1 Fonctions arithmétiques

ABS	retourne la valeur absolue de n	ABS(n)
CEIL (CEILING en MySQL)	fonction plafond	CEIL(n)
		SELECT CEIL(15.7) FROM DUAL renvoit 16
FLOOR	fonction plancher	FLOOR(n) SELECT FLOOR(15.7) FROM DUAL renvoit 15

MOD	fonction modulo	MOD(m,n) renvoit m modulo n
POWER (POW MySQL et Postgres)	Puissance	POWER(m,n) renvoit m puissance n
ROUND	n arrondit à m position à droite si m est omis, à 0 positions si m est négatif, l'arrondi porte sur les chiffres à gauche du point décimal	ROUND(n [,m]) SELECT ROUND(15.193,1) FROM DUAL renvoit 15.2 SELECT ROUND(15.193,-1) FROM DUAL renvoit 20
SIGN (pas Postgres)	renvoit +/- 1 selon le signe de n	SIGN(n)
SQRT	racine carrée	SQRT(n)
TRUNC (TRUNCATE en MySQL)	troncature en position m	TRUNC(n [,m]) SELECT TRUNC(15.79,1) FROM DUAL renvoit 15.7 SELECT TRUNC(15.79,-1) FROM DUAL renvoit 10

2.2.2 Fonctions de traitement de chaîne

CHR (CHAR en MySQL)	renvoit un caractère	SELECT CHR(65) FROM DUAL renvoit 'A'
ASCII	renvoit le code ASCII du caractère	SELECT ASCII('A') FROM DUAL renvoit 65
LENGTH	renvoit la longueur de la chaîne	LENGTH(ch)
INITCAP (pas MySQL)	renvoit la chaîne avec le premier caractère en majuscule	SELECT INITCAP('TOTO') FROM DUAL renvoit Toto
LOWER (pas MySQL)	retourne la chaîne en minuscules	
UPPER (pas MySQL)	renvoit la chaîne en majuscules	
LTRIM	retire tous les caractères de l'ensemble dans la chaîne jusqu'au premier n'y appartenant pas à partir de la gauche	LTRIM(chaîne [,ensemble]) SELECT LTRIM('xxxXxxFIN','x') FROM DUAL renvoit XxxFIN
RTRIM	même fonction agissant à droite	
LPAD	retourne la chaîne1 aughmentée jusqu'à la longueur n de la chaîne2	LPAD(chaîne1, n [,chaîne2]) SELECT LPAD('page 1',14,'*.') FROM DUAL renvoit `*.*.*.*.page 1`
RPAD	même fonctionnement à droite	
REPLACE	remplace dans une chaîne les occurences de ch1 par ch2	REPLACE (chaîne,ch1,ch2) REPLACE('JACK and JUE','J','BL') donne `BLACK and BLUE`
SUBSTR	retourne la sous-chaîne de n caractères en position m	SUBSTR(ch,m [,n]) SELECT SUBSTR('ABCDEFG',2,3) FROM DUAL renvoit `BCD`
TRANSLATE (Oracle)	remplace dans ch toutes les occurences de *from* par les caractères correspondants de *to*	TRANSLATE(ch, from, to) SELECT TRANSLATE('A ! B !',' !','-') FROM DUAL donne `A- B-`
SOUNDEX (Oracle)	approximation phonétique	WHERE soundex(nom)=soundex(dupon)

2.2.3 Fonctions de groupe

AVG	moyenne	SELECT AVG(salaire) FROM emp ;
COUNT	comptage	SELECT COUNT(*) FROM emp ;
MAX		SELECT MAX(salaire) FROM emp ;
MIN		SELECT MIN(salaire) FROM emp ;
SUM		SELECT SUM(salaire) FROM emp ;
VARIANCE		SELECT VARIANCE(salaire) FROM emp ;

2.2.4 Fonctions de conversion (Oracle)

TO_CHAR	TO_CHAR(n [,fmt]) où n est un nombre SELECT TO_CHAR(12345,'0999,99') renvoit 0123,45 Le second paramètre est un *format* constituant un modèle de transfert. TO_CHAR(d [,fmt]) où d est une date SELECT TO_CHAR(datenaiss,'DD Month YYYY') renvoit 19 MAY 1947
TO_DATE	TO_DATE(ch [,fmt]) SELECT TO_DATE('01 06 1995','DD MM YYYY') FROM DUAL
TO_NUMBER	TO_NUMBER(ch) SELECT TO_NUMBER('123') FROM DUAL renvoit 123

Les formats numériques utilisent les caractères 0 (pour marquer les zéros explicites), 9 (pour indiquer les positions d'affichage des chiffres, "$","." et ","

Le format de date par défaut est 'DD-MON-YY' qui utilise les 3 premières lettres du mois en anglais. La ponctuation est reproduite dans le résultat. Les symboles possibles sont : `-`, `/`, `.`, `:` En ce qui concerne les éléments de format de date, on a :

Format Oracle/Postgres	signification	Format MySQL
YYYY	année sur 4 chiffres	%Y
YY	année sur 2 chiffres	%y
MM	mois de l'année (1-12)	%m
MONTH	mois en toutes lettres (en anglais)	%M
MON	abbréviation (anglaise) du mois sur 3 lettres	%b
WW	numéro de la semaine dans l'année (1-52)	
W	numéro de la semaine dans le mois (1-5)	
DDD	jour de l'année (1-366)	%d
DD	jour du mois (1-31)	
D	jour de la semaine (1-7)	
DAY	nom du jour (anglais)	%a
J	date julienne (nombre de jours comptés à partir du 1er janvier 4712 avant J-C)	
HH	heure anglaise (0-12)	%h
HH24	heur européenne (0-24)	%H
MI	minutes (0-59)	%i
SS	secondes (0-59)	%S
SSSSS (oracle) ou SSSS(postgres)	nombre de secondes écoulées depuis minuit (0-86400)	

2.2.5 Fonctions de traitement de date (Oracle)

Les opérations arithmétiques (+/-) sont autorisées avec le type date. Les opérations correspondent à un ajout ou un retrait de jours. Si d1 correspond au 12 décembre 1991, d1 + 1 a pour résultat : `13-DEC-91`.

ADD_MONTH(d,n) ajoute n mois à la date d.

LAST_DAY(d) renvoit le dernier jour du mois de d.

MONTH_BETWEEN(d,e) le nombre de mois compris entre les dates d et e. Ce nombre est un réel dont la partie décimale correspond à la portion d'un mois de 31 jours restant.

NEXT_DAY(d,ch)renvoit la date du jour suivant celui qui est indiqué par la chaîne :
`NEXT_DAY('17-MAR-89','TUESDAY')` va renvoyer le : 21 mars 89.

SYSDATE renvoit la date courante. On peut l'afficher en tapant : `Select SYSDATE drom DUAL;`

2.2.6 Autres SGBD

MySQL : CURDATE(), CURTIME(), DATE_ADD(date, INTERVAL durée period), (DATE_SUB(date, INTERVAL durée period), DAYOFMONTH(date), DAYOFWEEK(date), DAYOFYEAR(date), MINUTE(datechaine), MONTH(date), MONTHNAME(date), PERIOD_ADD(date, nbmois), PERIOD_SUB, SECOND(datechaine), TIME_TO_SECOND(datechaine), WEEK(date), YEAR(date)
FORMAT (date_chaîne, format) transforme une chaîne en date

POSTGRESQL : CURRENT_DATE(), CURRENT_TIME(), DATE_PART(chaine,intervalle)

2.2.7 Autres fonctions (oracle)

DECODE(exp,s1,r1[,s2,r2], ...[,defaut]) L'expression exp est analysée. si on y trouve s1, on remplace sa valeur par r1, sinon, si on trouve s2, on remplace par r2, ... Si on ne trouve rien, la valeur par défaut est attribuée.
Exemple : SELECT nom, DECODE(numdir, 100,'SMITH',200,'JONES',300,'KING','AUCUN')

GREATEST(exp1 [,exp2] ...) retourne la plus grande d'une liste de valeurs.

LEAST(exp1 [,exp2] ...) effectue l'opération inverse.

NVL(exp1,exp2) renvoit exp2 si exp1 vaut NULL.
```
SELECT nom,NVL(commission,'SANS') FROM emp;
```

2.3 Interrogations

L'ordre `SELECT` permet de réaliser une combinaison très large d'interrogations utilisant les opérateurs relationnels de projection, restriction, jointure, union, tris, ...

```
SELECT (ALL|DISTINCT) {*| table.*| expr [alias] [,  ...]}
FROM [user.]table [alias] [, ...]
[WHERE condition ]
[CONNECT BY condition [PRIOR attribut][START WITH condition] ]
[GROUP BY expr,expr,... [HAVING condition] ]
[ {UNION|INTERSECT|MINUS} SELECT ... ]
[ORDER BY {expr|position} [ASC | DESC], {expr|position],...];
```

`SQL` évalue les clauses dans l'ordre suivant FROM, WHERE, GROUP BY, HAVING, ORDER BY ... A chaque évaluation d'une clause, `SQL` produit une relation intermédiaire qui est utilisée en flux d'entrée pour l'évaluation de la clause suivante.
Voici quelques exemples d'utilisation du SELECT, utilisant les tables suivantes :

```
emp(numemp, nom, dir, datemb, salaire, comm, numdept)
dept(numdept, nomdept, localisation)
```

Opération de restriction :

```
SELECT * FROM emp WHERE numDept=20;
```

NUMEMP	NOM	DIR	DATEMB	SALAIRE	COMM	NUMDEPT
7499	ALLEN	7698	20-FEB-81	10260.87	543	20

```
     7521  WARD         7698   22-FEB-81  12956.72  1320       20
     7654  MARTIN       7698   28-SEP-81  14713.00   863       20
     7844  TURNER       7698   01-MAY-82   9983.58   820       20
```

Opération de projection :

```
SELECT nomDept, numDept  FROM dept;

  NOMDEPT      NUMDEPT
  -----------  -------
  VENTES            10
  RECHERCHES        20
  MARKETING         30
  FABRICATION       40
```

Combinaison d'une restriction et d'une projection :

```
SELECT nom,salaire FROM emp WHERE salaire BETWEEN 12000 AND 15000 AND numdept=20;

    NOM          SALAIRE
    ------------ --------
    WARD         12956.72
    MARTIN       14713.00
    MILLER       13825.74
```

Utilisation d'opérateurs dans la clause Where :

```
 SELECT * FROM dept WHERE numDept IN (10,30);

     NUMDEPT   NOMDEPT   LOCALISATION
     -------   --------- -----------------
          10   VENTES    LILLE
          30   MARKETING VILLENEUVE D'ASCQ
```

Equi-jointure :

```
SELECT nom FROM emp, dept where emp.numDept=dept.numDept
            and  localisation  like 'Lil%';

      NOM
      ---------------
      AUSTIN
      BRADLEY
      HOPMAN
```

Une telle requête peut aussi être écrite en utilisant une sous requête (non corrélative) :

```
SELECT nom FROM emp where numDept in (SELECT numdept FROM dept
           WHERE localisation like 'Lil%');
```

Sous-requête corrélative :

```
SELECT nom FROM emp e1 WHERE salaire >
      (SELECT avg(salaire) FROM emp e2
              WHERE e1.numdept=e2.numdept);
```

Cette requête recherche les noms des employés dont le salaire est supérieur à la moyenne des salaires de *leur* département. Le traitement de la requête proncipale et celui de la sous-requête est mené "en parallèle" sur la base de l'auto-jointure qu'on a exprimé ici à l'aide de deux alias.

Utilisation d'expressions :

```
SELECT  nom,salaire+comm  "TOTAL" FROM emp WHERE  numDept=20;
```

```
NOM            TOTAL
-------------- -----------
ALLEN          10803.87
WARD           14276.72
MARTIN         15576.00
TURNER         10803.58
```

Opérateur de tri :

```
SELECT nom, salaire from EMP
WHERE numDept=20
ORDER BY nom;
```

```
NOM          SALAIRE
------------ --------
ALLEN        10260.87
MARTIN       14713.00
MILLER       13825.74
WARD         12956.72
```

Utilisation de fonctions d'agrégat :

```
SELECT nomDept,job,sum(salaire) "MASSE SAL", count(*), AVG(salaire) "MOY"
FROM emp e,dept d
WHERE e.numDept=d.numDept
GROUP BY nomDept,job;
```

```
NOMDEPT      JOB       MASSE SAL   COUNT(*)  MOY
---------- ---------- ----------- --------  --------
MARKETING SECRETAIRE  123468.98         10   9357.34
MARKETING DIRECTEUR    27875.32          1  27875.32
MARKETING PRESIDENT    76268.97          1  76268.97
RECHERCHE CHERCHEUR    47914.09          4  11978.52
RECHERCHE SECRETAIRE   16923.45          2   8461.72
RECHERCHE DIRECTEUR    22965.36          1  22965.36
VENTES    SECRETAIRE   45988.12          5   8121.32
VENTES    VENDEUR     105438.35          7  15062.62
VENTES    DIRECTEUR    53288.00          1  53288.00
```

```
SELECT nomDept,job,sum(salaire) "MASSE SAL", count(*), AVG(salaire) "MOY"
FROM emp e,dept d
WHERE e.numDept=d.numDept
GROUP BY nomDept,job
HAVING count(*)>= 2;
```

```
 NOMDEPT      JOB       MASSE SAL   COUNT(*)  MOY
---------- ---------- ----------- --------  --------
MARKETING SECRETAIRE  123468.98         10   9357.34
```

```
RECHERCHE CHERCHEUR    47914.09         4  11978.52
RECHERCHE SECRETAIRE   16923.45         2   8461.72
VENTES    SECRETAIRE   45988.12         5   8121.32
VENTES    VENDEUR     105438.35         7  15062.62
```

Le parcours des arborescences, qui correspondent à des associations réflexives sur lien fonctionnel dans le M.C.D. est assuré par la clause `CONNECT`. Considérons l'exemple suivant, construit sur une table d'employés appelée EMPLOYE :

NOM	NUM	JOB	CHEF
DUBOIS	0309	EMPLOYE	0902
ALLARD	0409	VENDEUR	0298
WHITE	0501	VENDEUR	0298
JOSEPH	0506	CHEFDEPT	0809
MARTIN	0604	VENDEUR	0298
BARON	0298	CHEFDEPT	0809
CLAIR	0702	CHEFDEPT	0809
SCOTTO	0708	ANALYSTE	0506
KLEIN	0809	PRESIDENT	
TURNO	0804	VENDEUR	0298
ADAM	0806	EMPLOYE	0708
JEAN	0900	EMPLOYE	0298
FABRE	0902	ANALYSTE	0506
MILLET	0904	EMPLOYE	0702

Il existe une variable `LEVEL` qui permet de connaître le niveau d'une donnée dans un tel arbre :

```
  SELECT level, num, nom, job, chef FROM employe
  CONNECT BY  PRIOR num= chef
  START WITH nom='JOSEPH';
}
     LEVEL  NUM   NOM    JOB       CHEF
     -----  ----  ------ --------  ----
         1  0506  JOSEPH CHEFDEPT  0809
         2  0708  SCOTTO ANALYSTE  0506
         3  0806  ADAM   EMPLOYE   0708
         2  0902  FABRE  ANALYSTE  0506
         3  0309  DUBOIS EMPLOYE   0902
```

PRIOR donne le sens du lien fonctionnel en s'accolant à l'attribut faisant référence à la ligne du père. `CONNECT BY  PRIOR num= chef` signifie que chaque CONNECT BY retournera un chef avant chaque employé. `CONNECT BY num=PRIOR chef` signifie que chaque employé sera affiché avant son supérieur.

L'utilisation de la fonction `lpad` permet d'indenter la présentation des niveaux :

```
SELECT LPAD(' ', 2*level)||nom, num, job, chef FROM employe
 CONNECT BY PRIOR num='chef'
 START WITH nom='JOSEPH';
     LPAD(' ',2*LEVEL)||NOM    NUM     JOB    CHEF
     ------------------------  ----  -------  ----
     JOSEPH                    0506  CHEFDEPT 0809
       SCOTTO                  0708  ANALYSTE 0506
         ADAM                  0806  EMPLOYE  0708
       FABRE                   0902  ANALYSTE 0506
         DUBOIS                0309  EMPLOYE  0902
```

2.4 Insertion

La requête d'insertion existe sous 2 formes. Syntaxe :

```
INSERT INTO [user.]table [(att1 [,att2 ...])] VALUES(v1, v2, ...) | interrogation;

INSERT INTO EMP VALUES (7890,'TOTO,'EMPLOYE',0902);

INSERT INTO bonus SELECT nom, job,salaire,commission
  FROM emp WHERE commission < 25*salaire
    OR job IN ('PRESIDENT','CHEFDEPT');


CREATE TABLE anniversaire(nom char(15), datenaiss date);
INSERT INTO anniversaire VALUES
   ('ANNIE', to_date('13-NOV-85 10:56 ', 'DD-MON-YY HH:MI')  );

CREATE TABLE image(img raw( 4));
INSERT INTO image VALUES ('1B5AC2F0');
```

2.5 Mise à jour

```
UPDATE [user.]table [alias] SET attr=expr [,attr=expr]... [WHERE condition];

UPDATE emp SET job='PRESIDENT ' WHERE num=0506;
```

2.6 Destruction de tuples

```
DELETE [FROM] [user.]table [alias] [WHERE condition];

DELETE FROM ventes;

DELETE FROM emp WHERE job='VENDEUR' AND commission < 100;
```

3 Le langage de Sécurité des Données

3.1 Contrôle des accès

La création d'un utilisateur par l'administrateur est réalisée par la commande CREATE USER :
Sous Oracle, par exemple :
`create user toto identified by toto default tablespace LICENCE`
Il existe 2 ordres d'autorisation :

- **Droits généraux**
  `GRANT <Droits> TO utilisateurId IDENTIFIED BY  motDePasse;`
  Cet ordre donne des droits généraux à l'utilisateur et lui permet de modifier son mot de passe. On distingue 3 types de droits généraux : le droit de se connecter, le droit de créer des objets et de les administrer (utilisateur courant) et les droits du DBA.
  Sous Oracle, ils sont définis par les mots-clé CONNECT pour pouvoir se connecter à la base ou changer son mot de passe, RESOURCE pour pouvoir créer, modifier ou détruire des objets, DBA pour un administrateur (tous les droits).
  `GRANT CONNECT TO toto IDENTIFIED BY nouveauMotDePasse;`
- **Droits sur les objets**
  `GRANT droits ON table | vue TO utilisateur [WITH GRANT OPTION];`
  Les droits sur les objets sont : SELECT (lecture), INSERT, UPDATE, DELETE, ALTER (droit de modifier une table), INDEX (droit de créer des index), CLUSTER (droit de créer des clusters), ALL (tous les droits précédants).
  La clause optionnelle WITH GRANT OPTION permet au receveur de transmettre ces droits à son tour.

exemple : `GRANT CONNECT,INSERT,UPDATE,DELETE ON emp TO toto WITH GRANT OPTION;`

La commande REVOKE permet de retirer ses droits à un utilisateur :

`REVOKE droits ON table FROM nom;`

REMARQUE (Oracle) : Outre les droits sur les objets présentés ci-dessus, on trouve une liste de privilèges système, trop large pour être donnée ici, qui indique *l'action* que l'utilisateur peut effectuer sur chaque type d'objet, par exemple :

```
grant create table to toto;
grant alter any table to toto;
```

On trouve aussi sous Oracle la notion de *rôle*, qui permet de simplifier notablement l'administration des utilisateurs en définissant un comportement commun à des groupes d'usagers. Un rôle, qui se manipule comme un pseudo-utilisateur, représente quelque chose d'équivalent à un groupe sous UNIX.

```
create role lambda;
grant select, insert on matable to lambda;
grant create any view to lambda;
grant lambda to toto;
grant lambda to machin;
```

4 Le langage de gestion des transactions (contrôle d'exécution)

4.1 Commit et Rollback

Le poly de cours donne des informations sur la sécurité des données en Client/serveur. Vous pouvez vous y reporter pour comprendre comment sont gérées les transactions dans ce contexte. Deux commandes sont alors indispensables :

`COMMIT` permet l'enregistrement des opérations dans la base et rend donc permanent ce qui n'était modifié qu'au sein du buffer de la transaction.

`ROLLBACK` effectue l'opération inverse, c.à.d défait de la base les résultats de la transaction. On peut ainsi tester une application, insérer des données, en détruire et en modifier, puis revenir à l'état préalable de la base.

4.2 Connexions multiples (Oracle)

Une fois connecté sous un nom d'utilisateur, on peut, sans quitter l'éditeur, passer sous un autre compte, en utilisant la commande `CONNECT uid/pswd@chaineDeConnexion` ou `CONNECT uid` qui réclame alors le mot de passe correspondant en mode invisible.

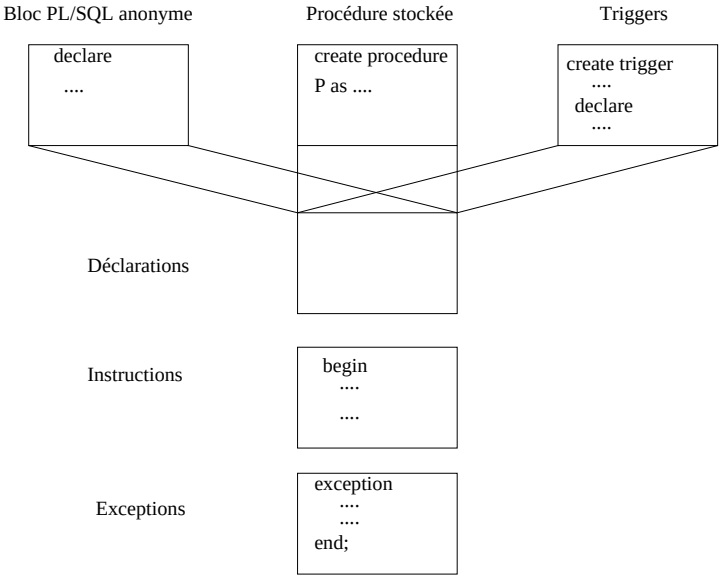
Exemple : `connect toto/1234@filora10`

PL/SQL

Ce langage, créé par `Oracle`, est une tentative de création d'un langage spécialisé pour manipuler une base de données relationnelle tout en conservant l'approche procédurale, les notions de boucle et de récursivité. PL/SQL possède une syntaxe très proche de celle d'ADA. Seule une petite partie du noyau ADA est utilisée, et celle-ci est de plus très proche du langage PASCAL qui a servit de substrat syntaxique pour la définition d'ADA.

ADA, comme PASCAL, est un langage impératif (mais il permet aussi la programmation fonctionnelle ou orientée-objet). C'est de plus un langage dit *de bloc*, car sa structure de base consiste en blocs emboîtés.

PL/SQL permet donc d'écrire des procédures , fonctions, paquetages ou triggers compilés et stockés dans la base, mais aussi d'écrire des séquences d'instructions anonymes à interprèter. Cette dernière possibilité s'avère très utile lors de la mise au point d'un programme avant de le stocker effectivement dans la base.



1 Structure d'un bloc

la structure générale d'un bloc est la suivante :

```
<En-tête de bloc>
    -- zone de déclarations de variables
BEGIN
    -- zone d'instructions
[EXCEPTION
    -- traitements d'exception (optionnel)]
END;
```

L'en-tête d'un bloc anonyme est définie par le mot-clé : `DECLARE`

Toutefois, lorsqu'aucune déclaration n'est nécessaire, un bloc anonyme peut commencer par le BEGIN délimitant la zone d'instructions.

Les en-tête de fonctions et de procédure commencent respectivement par les mots-clé : `create function` ou `create procedure`.

Les en-têtes de paquetage, qui comportent une variante syntaxique par raport aux précédents (mot-clé AS à la place de IS) commencent par `create package`, et enfin les en-tête de trigger par `create trigger`.

Voici un exemple de bloc anonyme, réalisant un traitement sans intérêt particulier, et comportant un traite-exception. Il intègre une instruction `commit` de façon à assurer la prise en compte effective des écritures sur le disque du serveur. Il diffère cependant de l'écriture d'une transaction car il ne met en place aucun des protocoles et actions de verouillages qui seraient indispensables dans un contexte d'accès multi-utilisateurs :

```
Declare
  numdeptv Dept.numdept%type = 10;
  nbenr    NUMBER(5);
  minimum  NUMBER;
  valeur   NUMBER;
begin
  SELECT count(*) INTO nbenr FROM emp
    WHERE dept=numdeptv;

  if nbenr > 0 then
    UPDATE emp SET dept = null WHERE dept=numdeptv;
  end if;

  DELETE FROM dept WHERE numdept=numdeptv;

  SELECT min(salaire) INTO minimum
        FROM emp WHERE dept=5;
  if valeur < minimum
    then augmentationSalaire;
    else  message(' valeur superieure a:  '|| to_char(minimum)||'.');
  end if;

  COMMIT;

exception
  when no_data_found then INSERT INTO erreur_log VALUES(sysdate, numdeptv);
  when others then
    err_code:=sqlcod;
    err_txt := sqlerrm;
    INSERT INTO errlog2 VALUES(sysdate, err_code,err_txt);
end;
```

2 Partie déclarative

Elle permet de déclarer des types, des variables, des constantes, des curseurs, des exceptions. Les types autorisés des variables et constantes sont tous les types SQL reconnus par le SGBD (par ex. `number` et `varchar2` sous ORACLE).

La syntaxe de déclaration d'une variable est :

```
<nomVariable> <typeSQL>;
```

La syntaxe d'une déclaration de constante est :

```
<nomConstante> constant <typeSQL> <initialisation>
```

où `<initialisation> ::= [not null] default <valeur> | := <valeur>`

Voici un ensemble de définitions valides de variables et de constantes :

```
x varchar2(50);
OK boolean;
date_naissance date;
v constant number(8) default 500;
t constant number(5) not null := 0;
```

Les déclarations de type permettent de définir des types nouveaux, comme en ADA, mais d'une façon très limitée. On ne peut ainsi construire que des types *tableaux* ou *enregistrements*. Un tableau est une structure indexée d'objets de même type (même occupation mémoire pour chacune des éléments du tableau). Un enregistrement est une structure comportant des champs nommés contenant chacun des

valeurs correspondant à des types simples (SQL) différents. Dans l'exemple qui suit, on construit le type tableau `txt_tab_type`, afin de définir 2 variables : `texte` et `err_msg`.

```
type txt_tab_type is table of varchar2(80) index by binary_integer;
texte txt_tab_type;
err_msg txt_tab_type;
```

Après ces déclarations, on pourra écrire :

```
texte(1) :='PL/SQL est le langage procedural d''Oracle';
```

Dans l'exemple qui suit, on construit maintenant un type d'enregistrement nommé `emp_rec_type` permettant de définir la variable : `emp_rec` .

Cette possibilité est particulièrement utile pour construire des variables de réception pour les rangées d'une table ou d'une vue (c.à.d des *gabarits d'enregistrement*) :

```
type emp_rec_type is record (nom varchar2(30), datenaiss date, num number(8));
emp_rec emp_rec_type;
```

Lors de la déclaration d'une variable ou constante, il est également possible de se référer au type de l'attribut d'une table par une expression telle que `nomTable.attribut%type`. On peut se référer à ligne complète de la table T, correspondant à un type d'enregistrement, par l'expression : `T%rowtype`.

On peut de même faire référence à celui d'une variable déjà déclarée par : `variable%type`. Cette notation `%type` correspond à une notation d'attribut. Chaque variable possède au moins un attribut (complément d'information) précisant à quel type elle se rattache. L'utilisation du symbole `%` à la place de l'apostrophe constitue une première petite différence syntaxique avec le langage ADA.

Voici des exemples de déclarations de variables exploitant cette possibilité :

```
declare
  nom employes.nom%type;
  libelle toto.produits.libelle%type;
  messages err_msg%type;
  -- fait référence à la déclaration de err_msg dans l'exemple précédent
```

Il sera fait allusion plus loin aux déclarations d'exception et de curseur.

3 Instructions

Elles suivent à 99% près la syntaxe ADA. Le langage ne possède pas d'instruction d'aiguillage (CASE). On remarquera qu'une simple instruction SQL est en fait une instruction PL/SQL, puisque SQL est complètement intégré à PL/SQL.

3.1 Instruction conditionnelle

```
IF <condition> THEN
  --bloc ALORS
[ELSIF <condition2> THEN
  -- bloc DANS CE CAS ALORS 2
[ELSIF <condition3> THEN
  -- bloc DANS CE CAS ALORS 3
... ]]
[ELSE
 -- bloc SINON]
END IF;
```

Exemples :

```
if nom is null or numemp=1000
  then
    salaire := salaire_de_base;
  elseif nom like 'BigBoss%' then
    salaire := salaire_max*4;
  else
    salaire := salaire_moyen;
end if;
```

3.2 boucles

1. On distingue la boucle "brute" :

   ```
   LOOP
    -- bloc répétitif
   END LOOP;
   ```

   Pour ne pas constituer une boucle infinie, un tel bloc doit obligatoirement comporter une instruction :

   `exit when <condition>`

2. Une seconde instruction de boucle est constituée par la boucle TANT QUE :

   ```
   WHILE <condition> LOOP
    -- bloc répétitif
   END LOOP;
   ```

   Un TANT QUE peut lui aussi comporter une instruction `exit when`.

3. La boucle POUR, quoique très simple à écrire, comporte deux pièges :
   La variable de contrôle de boucle est déclarée implicitement au sein de la boucle. ELLE NE NECESSITE DONC AUCUNE AUTRE DECLARATION PREALABLE (dans la zone de déclaration). Le fait de faire une telle déclaration revient à déclarer deux variables, différentes bien que comportant le même nom (ce qui est une caractéristique des langages de bloc : l'existence et la portée d'une variable n'a de sens qu'au sein du bloc où elle a été déclarée).
   L'intervalle de comptage (une boucle POUR revient à compter le nombre de cycles répétitifs) est toujours exprimé dans le sens ascendant, de la valeur minimale à la valeur maximale. Pour compter dans le sens descendant, il faut utiliser le mot-clé `REVERSE` placé juste avant la définition de l'intervalle.

   ```
   FOR <variableDeControle> IN [REVERSE] <valeurMini> .. <valeurMaxi> LOOP
    -- bloc répétitif
   END LOOP;
   ```

   On veillera à ne jamais placer d'instruction de rupture de boucle (par exemple `exit when`) dans une boucle POUR.

Exemples de boucles :

```
loop
  i := i + 1;
  exit when i=1000;
end loop;

i := 0;
while i < 1000 loop
  i := i + 1;
end loop;

for i in reverse 1 .. 1000 loop
  j := j + 2*i;
end loop;
```

3.3 Affectation

Elle utilise l'opérateur d'affectation `:=`, le `=` étant réservé à la comparaison. Attention donc à ne pas utiliser le double égale `==` du JAVA pour faire une comparaison ! Toutes les fonctions et opérateurs SQL peuvent être utilisés. Voici quelques exemples d'instructions :

```
entier := 10;
sousChaine:= substr(v5,1,10);
condition := salaire > 10000;
datenaiss := to_date('19-05-47', 'DD-MM-YYYY');
```

3.4 Select

En PL/SQL, une instruction SQL `select` ne peut retourner plusieurs lignes, mais une seule à la fois. Il est en effet hors de question, en mode programme, de rediriger les valeurs lues dans une table vers l'écran, du moins sans l'invocation d'une instruction d'affichage. Les valeurs lues doivent être récupérées dans des *variables de réception* appropriées, c.à.d de même type que les valeurs qu'elles doivent recevoir. Ces variables sont des scalaires. Même losqu'on transfère des attributs d'une table relationelle dans un tableau, on ne peut le faire qu'une seule ligne à la fois.

De ce fait, la syntaxe d'une instruction SQL est modifiée et possède, dans ce contexte, une clause supplémentaire : INTO

```
Select employes.nom INTO un_nom from employes where numero_employe=1000;
```

4 Exceptions

Un grand nombre d'exceptions sont pré-définies, de même que les codes d'erreur qu'il est possible de manipuler. On peut également en créer selon ses besoins. Pour *lever* une exception, on utilise la clause *raise*. Cette exception devra alors être récupérée dans la zone de *traite-exception*, associée à un traitement approprié, sous peine d'interrompre le programme avec pour seule indication l'affichage d'un code d'erreur associé lorsque l'exception est prédéfinie.

```
declare
  trop        exception;
  pas_assez   exception;
  en_panne    exception;
begin
  ...
  if mauvais_contact then raise en_panne;
  ...
exception
  when en_panne then declencher_sonnerie;
end;
```

La clause `when others` permet la récupération de toutes les exception non invoquées dans le traite-exception. Voici un extrait de la liste des exceptions pré-définies :

```
cursor_already_open   program_error
dup_val_on_index      storage_error
invalid_cursor        time_out_on_resource
invalid_number        too_many_rows
logon_denied          value_error
no_data_found         divide_error
not_logged_on
```

Il peut-être judicieux, dans le traite-exception, de récupérer les messages et codes d'erreur afin de faciliter le deboggage :

```
exception
  when others then
    err_code := sqlcode; -- fonction SQL retournant le code d'erreur
    err_text := sqlerrm; -- fonction SQL retournant le message d'erreur
    insert into erreurs values(err_code, err_text, sysdate);
```

5 Curseurs

La recherche d'une information par SELECT commence par une initialisation du point d'accès à la table. Ceci signifie que si on place l'ordre SELECT dans une boucle, les mêmes informations seront recherchées autant de fois, peut-être même indéfiniment. Ainsi la boucle :

```
for i in 1 .. 10 loop
  select nom into nom_emp from employes;
  DBMS_OUTPUT.put(nom_emp);
end loop;
```

afficherait 10 fois de suite le nom du même premier employé (à condition de pouvoir s'exécuter. Il est bien évident que la requête SQL retournant un ensemble que l'on cherche à stocker dans un scalaire, c'est l'exception `too_many_rows` qui sera déclenchée).

Le seul moyen de poursuivre l'exploration d'une table par déplacements successifs pilotés par une boucle est de manipuler un `curseur`.

Un curseur est une structure fondamentale de la programmation dès lors que l'on manipule les données d'une base relationnelle. Il permet, comme son nom le suggère, d'explorer une table itérativement, ligne par ligne, en conservant la mémoire de la position, dans les fichiers de la base, de la ligne précédemment explorée. Son exploitation impose l'utilisation de variables de récupération des données retournées. Celles-ci doivent correspondre à un type *enregistrement (record)* lorsque le *select* sur lequel le curseur est défini retourne une ligne entière.

Un curseur doit etre déclaré, ouvert, utilisé, puis refermé, exactement comme on opère avec un fichier exploré séquentiellement.

La syntaxe de la déclaraction est la suivante :
`Cursor nom[(parametre)] is <requete d'interrogation>`
Par exemple :

```
cursor c1 is select numEmp from emp where nom like '%x%
                                              order by nom;
cursor c2(nomDept varchar2) is select nom,departement,salaire
                              from emp where nom = nomDept order by nom;
```

Un curseur est munis d'attributs repérés par le caractère % :
`%found`, `%notfound`, `%rowcount` et `%isopen`.

Le type du curseur peut être invoqué pour la définition de la variable de récupération du résultat. Exemple :
`c2_rec c2%rowtype`.

L'ouverture du curseur se fait par une instruction OPEN . Exemple : `open c1`

Son utilisation est commandée par l'instruction :`FETCH curseur INTO variable`

Enfin, on le ferme avec CLOSE : `close c1`

Voici une séquence complète de manipulation d'un curseur :

```
declare
  nb number;
  cursor c1 is
    select * from emp order by nom;
  c1_rec c1%rowtype
begin
  open c1;
  loop
```

```
    fetch c1 into c1_rec;
    exit when c1%notfound
  end loop;
  nb := c1%rowcount;
  close c1;
  DBMS_OUTPUT.put(nb);
end;
```

REM : Un curseur avec paramètre doit être ouvert en lui transmettant la valeur de ce dernier : `open c2('Comptabilite');`

Il existe une forme spéciale de la boucle POUR, la *boucle PourCurseur* dont la syntaxe est :
`For <Enregistrement> in <NomCurseur> loop ... end loop;`
Dans ce cas, comme pour une boucle POUR ordinaire, le gabarit d'enregistrement est implicitement déclaré dans la boucle elle-même et ne doit pas faire l'objet d'une déclaration séparée.
En voici un exemple de boucle POUR CURSEUR utilisant la variable locale `c1_rec` :

```
declare
  Cursor cur_for is select * from emp order by nom;
begin
  for c1_rec in cur_for loop
   -- PAS DE FETCH - il est automatique
    DBMS_OUTPUT.put(c1_rec.nom);
  end loop;
  total := cur_for%rowcount;
end;
```

Le curseur va, par l'intermédiaire de la boucle POUR, explorer tout son intervalle de définition, qui correspond au nombre de lignes de la table EMP. Il est ouvert implicitement, manipulé et fermé implicitement après le `end loop`.

6 Procedures PL/SQL

6.1 Procédures et fonctions

Un bloc PL/SQL peut-être muni d'un en-tête de procédure ou de fonction :

```
create procedure p(...) is
create or replace procedure p(...) is
create or replace function f(...) return <type> is
```

Exemple :

```
create or replace function f(num number, txt varchar2) return number is
  total number;
begin
  select count(*) into total from emp where emp.num=num;
  if total>20 then return total else return 0; end if;
end;
```

Les types des paramètres des fonctions et procédures ne doivent pas être dimensionnés. Dans l'exemple, txt est un varchar2 et non un varchar2(30) par exemple. D'autre part, les déclarations éventuelles suivent l'en-tête sans être précédées d'une clause syntaxique particulière comme c'est le cas en PASCAL.

Le source est stocké dans la table `sys.Source$` du dictionnaire des données. Les messages d'erreurs éventuellement produits par la compilation sont également stockés (`sys.error$`), et un compte-rendu de compilation est fourni. En cas d'erreur(s) de compilation, l'unité a le statut 'invalid'.
Le compte-rendu, disponible dans les tables du dictionnaire, peut être plus facilement examiné par l'intermédiaire des instructions :

```
show errors function <f>;
show errors procedure <p>;
select * from user_errors where name = '<p>';
select * from all_errors where name='<p>';
```

En ce qui concerne la mise au point , il existe un paquetage `DBMS_OUTPUT` qui contient les procédures d'entrées/sorties suivantes :

```
ENABLE
PUT(<variable>)
PUT_LINE(<variable>)
NEW_LINE
GET_LINE(<variable>,status)
GET_LINES(<variable>,nombreDeLignes,status)
DISABLE
```

`Put` affiche son unique paramètre à l'écran et laisse le curseur en place. `Put_line` affiche et passe à la ligne. `New_line` se contente de sauter de ligne. Les données à affichées sont stockées dans une structure représentant un pipe-line en mémoire. Elles en sont extraites par les procédures de type `get`. `get_line` lit une ligne dans le pipe alors qeu `get_lines` en lit le nombre précisé en paramètre. La procédure `enable` ouvre l'accès au pipe d'E/S.

REM : ce paquetage ne devient disponible qu'en activant le serveur gérant la sortie standard par la commande : `set serveroutput on`.

6.2 paquetages

La déclaration est conforme à la syntaxe utilisée pour les procédures, au mot clé AS (à la place de IS) près :

`create [or replace] package [body] <nom_du_paquetage> as ... end <nom_du_paquetage>;`

Un paquetage comprend deux parties : la partie spécification et la partie implémentation ou corps (body). On peut recompiler (option `or replace`) séparément soit la spécification, soit le corps si on le souhaite.

La spécification du paquetage contient les *profils* des fonctions et procédures ainsi que les déclarations de variables et de constantes. Le corps reprend, comme c'est le cas en ADA, les détails de l'implémentation. La surcharge des noms de procédures et fonctions est autorisée si les profils sont différents.

```
create or replace package test as
-- partie spécification
  function fgetval(num number) return number;
  function fgetval(text varchar2) return number;
  function fgetval(jour date) return number;
end test;

create or replace package body test as
-- Corps du paquetage
  function fgetval(num number) return number is
    -- code pl/sql de fgetval
  end;

  function fgetval(text varchar2) return number is
    -- code pl/sql
  end;

  function fgetval(jour date) return number is
    -- code pl/sql
  end;
end test;
```

Lors de l'utilisation, il faut préciser le nom complet des objets :
`[proprietaire].[paquetage].procedure[(parametres)]`
par exemple :

```
create synonym ouv for tp.pack.lister;
ouv('Moliere','seuil');
```

dans cet exemple, lister est une procédure à deux paramètres du paquetage pack dont tp est propriétaire.

7 Triggers

Voici la syntaxe d'un trigger :

```
Create or replace trigger <trigger_name>
before/after
  insert or update of <colonne1>, <colonne2>, ...
   or delete
on <tab_name>
  [for each row [when (<trigger_condition>)]]
[declare
    types, constantes ou variables éventuelles]
begin

   -- texte du trigger
end;
```

Les conditions d'exploitation d'un trigger peuvent se faire à l'aide d'une forme spéciale du IF, qui pourra avoir la forme suivante :

```
if inserting then ...
if updating then ...
if deleting then ...
```

Voici un exemple de texte de trigger :

```
Create or replace trigger commande
  after insert or update on commande_usine for each row
begin
  if inserting  then
    insert into commande@magasin values(
      :new.numcom, :new.numclient, :new.numarticle, :new.montant);
  elsif updating then
    update commande@magasin set
      numcom = :new numcom, numcli= :new.numcli,
      numarticle = :new.numarticle, montant = :new.montant
      where numcom = :old.numcom
  end if;
end;
```

On observera que lors d'une opération d'écriture déclenchant le trigger , la valeur à écrire est désignée par la syntaxe : `:next.attribut`, alors que la valeur supprimée (par un update ou un delete) est désignée par la syntaxe : `:old.attribut`. Un trigger peut éventuellement être muni d'une clause de restriction (`when ...`). Lorsque celle-ci doit exploiter les anciennes ou nouvelles valeurs, leur désignation se fait avec les préfixes OLD et NEW exempt du caractère ' :'.
Exemple :

```
Create or replace trigger correction
  after insert on employes
   for each row when lower(new.nom)=new.nom
    -- le nom a été rentré en minuscules
begin
  update employes set nom=initcap(nom) where numemp=:new.numemp;
end;
```

Le Dictionnaire des Données

Le **Dictionnaire des données** d'Oracle est composé de tables de base, propriétés du super-utilisateur SYS.

Ce dernier fournit aux autres utilisateurs les vues ALL, USER et DBA (principalement). Leur liste exhausitive est donnée par la vue : `DICTIONARY`

- Les vues ALL renseignent sur les objets propriétés de l'utilisateur, ainsi que ceux pour lesquels des droits d'accès lui ont été fournis.
- Les vues USER permettent d'accèder exclusivement aux objets créés par l'utilisateur.
- Les vues DBA fournissent des informations sur les objets auxquels seul un administrateur a accès.

Ces vues sont respectivement préfixées par : `ALL_`, `USER_` et `DBA_`.

Voci une liste de vues USER (à utiliser donc avec le prefixe correct, par exemple `USER_ALL_TABLES`) qui peuvent s'avérer utiles :

ALL_TABLES	description des tables
CATALOG	tables, vues, synonymes et séquences
COL_PRIVS	provilèges sur les colonnes des tables
CONS_COLUMNS	informations sur les colonnes associées à des contraintes
CONSTRAINTS	définition des contraintes des tables
ERRORS	dexcriptif des erreurs de compilation ou d'exécution d'une procédure PL/SQL
INDEXES	descritiondes index sur les tables
OBJECTS	descriptioj de la totalité des objets
PROCEDURES	définition des procedures
SEQUENECS	description des séquences
SYBONYMS	description des synonymes
TABLES	description des tables
TABLESPACES	descripton des tablespaces accessibles
TAB_COLUMNS	colonnes des tables et vues accessibles
TAB_PRIVS	droits sur les objets
USERS	infomations sur les utilisateurs
VIEWS	Description et définition des vues