Langage SQL

Richard Grin

Université de Nice Sophia-Antipolis

Langage SQL

(version 2.0.2 du polycopié)

Richard Grin

8 avril 1998

Table des matières

Présentation du polycopié

Ce polycopié présente le langage SQL. Il ne suppose que quelques connaissances de bas
exposées dans le polycopié d'introduction aux bases de données.

Les exemples s'appuient sur la version du langage SQL implantée dans le SGBD Oracl
version 7.

Les remarques et les corrections d'erreurs peuvent être envoyées par courrier électr
nique à l'adresse "`grin@unice.fr`", en précisant le sujet "Poly SQL".

..

Chapitre 1

Introduction

1.1 Présentation de SQL

SQL signifie "*Structured Query Language*" c'est-à-dire "Langage d'interrogation struc
turé".

En fait SQL est un langage complet de gestion de bases de données relationnelles. Il
été conçu par IBM dans les années 70. Il est devenu le langage standard des systèmes d
gestion de bases de données (SGBD) relationnelles (SGBDR).

C'est à la fois :

- un langage de manipulation des données (LMD)
  - un langage d'interrogation (ordre SELECT)
  - un langage de manipulation des données (ordres UPDATE, INSERT, DELETE
- un langage de définition des données (LDD ; ordres CREATE, ALTER, DROP),
- un langage de contrôle de l'accès aux données (LCD ; ordres GRANT, REVOKE).

Le langage SQL est utilisé par les principaux SGBDR : DB2, Oracle, Ingres, RDB,
Chacun de ces SGBDR a cependant sa propre variante du langage. Ce support de cou
présente un noyau de commandes disponibles sur l'ensemble de ces SGBDR, et leur im
plantation dans Oracle Version 7.

1.2 Normes SQL

SQL a éte normalisé dès 1986 mais les premières normes, trop incomplètes, ont ét
ignorées par les éditeurs de SGBD.

La norme actuelle SQL-2 (appelée aussi SQL-92) date de 1992. Elle est acceptée par tou
mais les SGBD relationnels qui dominent actuellement le marché (en particulier Oracl
ne sont toujours pas totalement adaptés à cette norme.

SQL-2 définit trois niveaux :

- Full SQL (ensemble de la norme)
- Intermediate SQL
- Entry Level (ensemble minimum à respecter pour se dire à la norme SQL-2)

1.3 Utilitaires associés

Comme tous les autres SGBD, Oracle comprend plusieurs utilitaires qui facilitent l'emploi du langage SQL et le développement d'applications de gestion s'appuyant sur une base de données relationnelle. En particulier SQLFORMS facilite grandement la réalisation des procédures transactionnelles (pour les traitements effectués pendant la saisie ou la modification des données en interactif par l'utilisateur). Il permet de dessiner les écrans de saisie et d'indiquer les traitements associés à cette saisie. D'autres utilitaires permettent de décrire les états de sorties imprimés, de sauvegarder les données, d'échanger des données avec d'autres logiciels, de travailler en réseau ou de constituer des bases de données réparties entre plusieurs sites.

Ce cours se limitant strictement à l'étude du langage SQL, nous n'étudierons pas tous ces utilitaires. Nous verrons les commandes essentielles d'un seul utilitaire, SQLPLUS, qui facilite l'utilisation interactive du langage SQL par un utilisateur. Ces commandes permettent de modifier les ordres SQL et de constituer des fichiers de commandes SQL que l'on peut réutiliser ensuite.

Les commandes du langage SQL peuvent être tapées directement au clavier par l'utilisateur ou elles peuvent être incluses dans un programme écrit dans un langage de troisième génération (Cobol, Langage C, Fortran, Ada,...) grâce à un précompilateur fourni par Oracle.

1.4 Connexion et déconnexion

On entre dans SQLPLUS par la commande :

`SQLPLUS` *`nom/mot-de-passe`*

Les deux paramètres, *nom* et *mot-de-passe*, sont facultatifs. Si on les omet sur la ligne de commande, SQLPLUS les demandera, ce qui est préférable pour *mot-de-passe* car une commande "*ps*" sous Unix permet de visualiser les paramètres d'une ligne de commande et donc de lire le mot de passe.

Pour se déconnecter, l'ordre à taper est **EXIT** (ou `exit` car on peut taper les commandes SQL en majuscules ou en minuscules).

1.5 Objets manipulées par SQL

1.5.1 Identificateurs

SQL utilise des identificateurs pour désigner les objets qu'il manipule : utilisateur tables, colonnes, index, fonctions, etc.

Un identificateur est un mot formé d'au plus 30 caractères, commençant obligatoireme par une lettre de l'alphabet. Les caractères suivants peuvent être une lettre, un chiffre, c l'un des symboles # $et _. SQL ne fait pas la différence entre les lettres minuscules majuscules. Les voyelles accentuées ne sont pas acceptées.

Un identificateur ne doit pas figurer dans la liste des mot clés réservés (voir manuel référence Oracle). Voici quelques mots clés que l'on risque d'utiliser comme identificateurs ASSERT, ASSIGN, AUDIT, COMMENT, DATE, DECIMAL, DEFINITION, FILE, FO MAT, INDEX, LIST, MODE, OPTION, PARTITION, PRIVILEGES, PUBLIC, SELEC SESSION, SET, TABLE.

1.5.2 Tables

Les relations (d'un schéma relationnel ; voir polycopié du cours sur le modèle relationne sont stockées sous forme de tables composées de lignes et de colonnes.

Si on veut utiliser la table créée par un autre utilisateur, il faut spécifier le nom de c utilisateur devant le nom de la table :

BERNARD.DEPT

Remarques 1.1

(a) Selon la norme SQL-2, le nom d'une table devrait être précédé d'un nom de schém (voir 4.1).

(b) Il est d'usage (mais non obligatoire évidemment) de mettre les noms de table a singulier : plutôt EMPLOYE que EMPLOYES pour une table d'employés.

Exemple 1.1

Table DEPT des départements :

DEPT	NOMD	LIEU
10	FINANCES	PARIS
20	RECHERCHES	GRENOBLE
30	VENTES	LYON
40	FABRICATION	ROUEN

1.5.3 Colonnes

Les données contenues dans une colonne doivent être toutes d'un même type de donnée Ce type est indiqué au moment de la création de la table qui contient la colonne (voir 1.7

Chaque colonne est repérée par un identificateur unique à l'intérieur de chaque table. Deux colonnes de deux tables différentes peuvent porter le même nom. Il est ainsi fréquent de donner le même nom à deux colonnes de deux tables différentes lorsqu'elles correspondent à une clé étrangère à la clé primaire référencée. Par exemple, la colonne "Dept" des tables DEPT et EMP.

Une colonne peut porter le même nom que sa table.

Le nom complet d'une colonne est en fait celui de sa table, suivi d'un point et du nom de la colonne. Par exemple, la colonne DEPT.LIEU

Le nom de la table peut être omis quand il n'y a pas d'ambiguïté sur la table à laquelle elle appartient, ce qui est généralement le cas.

1.6 Types de données

Les types de données d'Oracle ne suivent pas la norme SQL-2.

1.6.1 Type numérique

Types numériques SQL-2

Les types numériques de la norme SQL-2 sont :

- Types exacts : INTEGER, SMALLINT, NUMERIC, DECIMAL
- Types non exacts : REAL, DOUBLE PRECISION, FLOAT

La définition de ces types dépend du SGBD. Reportez-vous au manuel du SGBD que vous utilisez pour plus de précisions.

Type numérique d'Oracle

Oracle n'a qu'un seul type numérique NUMBER. Par soucis de compatibilité, Oracle permet d'utiliser les types SQL-2 mais ils sont ramenés au type NUMBER.

Lors de la définition d'une colonne de type numérique, on peut préciser le nombre maximum de chiffres et de décimales qu'une valeur de cette colonne pourra contenir :

```
NUMBER
NUMBER(taille_maxi)
NUMBER(taille_maxi, décimales)
```

Si le paramètre *décimales* n'est pas spécifié, 0 est pris par défaut. Si le paramètre *taille_maxi* n'est pas spécifié il n'y aura aucun contrôle effectué par ORACLE sur le nombre de chiffres des valeurs contenues dans la colonne (maximum 38 chiffres). La valeur absolue du nombre doit être inférieure à 10^{128}.

L'insertion d'un nombre avec plus de chiffres que *taille_maxi* sera refusée (*taille_ma* ne prend en compte ni le signe ni le point décimal). Les décimales seront éventuellemer arrondies en fonction des valeurs données pour *taille_maxi* et *décimales*.

Exemple 1.2

SALAIRE NUMBER(8,2)
définit une colonne numérique SALAIRE. Les valeurs auront au maximum 2 déc males et 8 chiffres au plus au total (donc 6 chiffres avant le point décimal).

Les constantes numériques ont le format habituel : -10, 2.5, 1.2E-8 (ce dernier représer tant 1.2 x 10^{-8}).

1.6.2 Type chaîne de caractères

Les constantes chaînes de caractères sont entourées par des apostrophes ('). Si la chaîr contient une apostrophe, celle-ci doit être doublée. Exemple : `'aujourd''hui'`.

Il existe deux types pour les colonnes qui contiennent des chaînes de caractères :

- le type `CHAR` pour les colonnes qui contiennent des chaînes de longueur constantes r contenant pas plus de 255 caractères.

  La déclaration de type chaîne de caractères de longueur constante a le format suivant

  `CHAR (longueur)`

  où *longueur* est la longueur maximale (en nombre de caractères) qu'il sera possib de stocker dans le champ. *longueur* doit être obligatoirement spécifiée. L'insertic d'une chaîne dont la longueur est supérieure à *longueur* sera refusée. Une chaîne plu courte que *longueur* sera complétée par des espaces (important pour les comparaisor de chaînes).

- le type `VARCHAR2` pour les colonnes qui contiennent des chaînes de longueurs variable ne contenant pas plus de 2000 caractères. On déclare ces colonnes par :

  `VARCHAR2 (longueur)`

  longueur indique la longueur maximale des chaînes contenues dans la colonne (VAF CHAR dans la norme SQL-2).

1.6.3 Type date

La déclaration de type date a le format suivant :

`DATE`

Une constante de type "date" est une chaîne de caractères entre apostrophes. Le forma dépend des options que l'administrateur a choisies au moment de la création de la bas S'il a choisi de "franciser" la base, le format d'une date est "jour/mois/année", par exempl '25/11/92' (le format "américain" par défaut donnerait '25-NOV-92'). L'utilisateur peu saisir des dates telles que '3/8/93' mais les dates enregistrées dans la base ont toujou deux chiffres pour chacun des nombres, par exemple, '03/08/93'.

Dans Oracle une donnée de type DATE inclut un temps en heures, minutes et secondes. En SQL-2 ce n'est pas le cas et il existe les types TIME et TIMESTAMP pour indiquer un temps en heures minutes, secondes et fractions de secondes.

1.6.4 Valeur NULL

Une colonne qui n'est pas renseignée, et donc vide, est dite contenir la valeur "**NULL**". Cette valeur n'est pas zéro, c'est une absence de valeur.

1.7 Création d'une table

L'ordre CREATE TABLE permet de créer une table en définissant le nom et le type (voir 1.6) de chacune des colonnes de la table. Nous ne verrons ici que trois des types de données utilisés dans SQL : numérique, chaîne de caractères et date. Nous nous limiterons dans cette section à une syntaxe simplifiée de cet ordre (voir 4.2.1 et 1.8 pour des compléments) :

```
CREATE TABLE table
(colonne1 type1,
colonne2 type2,
.........
.........)
```

table est le nom que l'on donne à la table ; *colonne1, colonne2,..* sont les noms des colonnes ; *type1, type2,..* sont les types des données qui seront contenues dans les colonnes.

On peut ajouter après la description d'une colonne l'option NOT NULL qui interdira que cette colonne contienne la valeur NULL. On peut aussi ajouter des contraintes d'intégrité portant sur une ou plusieurs colonnes de la table (voir 1.8).

Exemple 1.3

```
CREATE TABLE article
( ref     CHAR (10) NOT NULL,
  prix    NUMBER (9,2),
  datemaj DATE)
```

1.8 Contrainte d'intégrité

Dans la définition d'une table, on peut indiquer des contraintes d'intégrité portant sur une ou plusieurs colonnes. Les contraintes possibles sont :

UNIQUE, PRIMARY KEY, FOREIGN KEY...REFERENCES, CHECK

Chaque contrainte doit être nommée (ce qui permettra de la désigner par un ordre ALTER, et ce qui est requis par les nouvelles normes SQL) en la faisant précéder de :

```
CONSTRAINT nom-contrainte contrainte
```


Il existe des contraintes :

sur une colonne : la contrainte porte sur une seule colonne. Elle suit la définition d la colonne dans un ordre CREATE TABLE (pas possible dans un ordre ALTE TABLE).

sur une table : la contrainte porte sur une ou plusieurs colonne. Elles se place au mêm niveau que les définition des colonnes dans un ordre CREATE TABLE ou ALTE TABLE.

1.8.1 Types de contraintes

```
(pour une contrainte sur une table :)
PRIMARY KEY (colonne1, colonne2,...)
(pour une contrainte sur une colonne :)
PRIMARY KEY
```

indique la clé primaire de la table (contrainte d'intégrité d'entité). Aucune des colonn qui composent cette clé ne doit avoir une valeur NULL.

```
(pour une contrainte sur une table :)
UNIQUE (colonne1, colonne2,...)
(pour une contrainte sur une colonne :)
UNIQUE
```

interdit qu'une colonne (ou la concaténation de plusieurs colonnes) contienne deux valeu identiques.

```
(pour une contrainte sur une table :)
FOREIGN KEY (colonne1, colonne2,...)
REFERENCES tableref[(col1, col2,...)]
[ON DELETE CASCADE]
(pour une contrainte sur une colonne :)
REFERENCES tableref[(col1)]
[ON DELETE CASCADE]
```

indique que la concaténation de *colonne1*, *colonne2*,... (ou la colonne que l'on définit pou une contrainte sur une colonne) est une clé étrangère qui fait référence à la concaténatio des colonnes *col1*, *col2*,... de la table *tableref* (contrainte d'intégrité référentielle). Si aucun colonne de *tableref* n'est indiquée, c'est la clé primaire de *tableref* qui est prise par défau

Cette contrainte ne permettra pas d'insérer une ligne de la table si la table *tabler* ne contient aucune ligne dont la concaténation des valeurs de *col1*, *col2*,... est égale à concaténation des valeurs de *colonne1*, *colonne2*,...

col1, *col2*,... doivent avoir la contrainte PRIMARY KEY ou UNIQUE. Ceci impliqu qu'une valeur de *colonne1*, *colonne2*,... va référencer une et une seule ligne de *tableref*.

L'option "**ON DELETE CASCADE**" indique que la suppression d'une ligne de *t bleref* va entrainer automatiquement la suppression des lignes qui la référencent dans

table. Si cette option n'est pas indiquée, il est impossible de supprimer des lignes de *tableref* qui seraient référencées par des lignes de la table.

`CHECK(`*condition*`)`

donne une condition que la ou les colonnes devront vérifier (exemples dans la section suivante). On peut ainsi indiquer des contraintes d'intégrité de domaines dont on a des exemples dans la section 1.8.2.

Des contraintes d'intégrité peuvent être ajoutées (uniquement des contraintes liées à la table et pas à une colonne) ou supprimées par la commande ALTER (exemple à la fin de la section 1.8.2). Mais pour modifier une contrainte, il faut la supprimer et ajouter ensuite la contrainte modifiée.

Il est parfois intéressant d'enlever temporairement des contraintes. Oracle le permet par la commande ALTER TABLE ... DISABLE/ENABLE mais n'utilise pas la commande de la norme SQL-2 (SET CONSTRAINTS ... DEFERRED/IMMEDIATE).

1.8.2 Exemples de contraintes

Quelques contraintes sur des colonnes :

```
CREATE TABLE EMP (
  MATR NUMBER(5) CONSTRAINT KEMP PRIMARY KEY,
  NOME CHAR(9) CONSTRAINT NOM_UNIQUE UNIQUE
    CONSTRAINT MAJ CHECK (NOME = UPPER(NOME)),
  .........
  DEPT NUMBER(2) CONSTRAINT R_DEPT REFERENCES DEPT(DEPT)
    CONSTRAINT NDEPT CHECK (DEPT IN (10, 20, 30, 35, 40)))
```

Des contraintes de colonne peuvent être mises au niveau de la table :

```
CREATE TABLE EMP (
  MATR NUMBER(5),
  NOME CHAR(9) CONSTRAINT NOM_UNIQUE UNIQUE
     CONSTRAINT MAJ CHECK (NOME = UPPER(NOME)),
  .........
  CONSTRAINT NDEPT DEPT NUMBER(2) CHECK (DEPT IN (10, 20, 30, 35, 40)),
  CONSTRAINT KEMP PRIMARY KEY (MATR),
  CONSTRAINT R_DEPT FOREIGN KEY (DEPT) REFERENCES DEPT(DEPT))
```

Certaines contraintes portent sur plusieurs colonnes et ne peuvent être indiquées que comme contraintes de table :

```
CREATE TABLE PARTICIPATION (
MATR NUMBER(5) CONSTRAINT R_EMP REFERENCES EMP,
CODEP VARCHAR2(10) CONSTRAINT R_PROJET REFERENCES PROJET,
....,
CONSTRAINT PKPART PRIMARY KEY(MATR, CODEP))
```

Exemple avec ALTER TABLE :

```
ALTER TABLE EMP
  DROP CONSTRAINT NOM_UNIQUE
```



```
  ADD (CONSTRAINT SAL_MIN CHECK(SAL + NVL(COMM,0) > 5000))
```

1.9 Sélection simple

L'ordre pour retrouver des informations stockées dans la base est "SELECT".

Nous étudions dans ce chapitre une partie simplifiée de la syntaxe, suffisant néanmoi à la plupart des interrogations courantes. Une étude plus complète sera vue au chapitre

```
SELECT exp1, exp2,...
FROM table
WHERE prédicat
```

table est le nom de la table sur laquelle porte la sélection. *exp1, exp2,...* est la liste d expressions (colonnes, constantes,... ; voir 1.10) que l'on veut obtenir. Cette liste peut êt "*", auquel cas toutes les colonnes de la table sont sélectionnées.

Exemples 1.4

(a)
```
SELECT * FROM DEPT
```

(b)
```
SELECT NOME, POSTE FROM EMP
```

(c)
```
SELECT NOME , SAL + NVL(COMM,0) FROM EMP
```

La clause WHERE permet de spécifier quelles sont les lignes à sélectionner.

Le prédicat peut prendre des formes assez complexes. La forme la plus simple est "exp *op* exp2", où *exp1* et *exp2* sont des expressions (voir 1.10) et *op* est un des opérateurs != (différent), >, >=, <, <=.

Exemple 1.5

```
SELECT MATR, NOME, SAL * 1.15
FROM EMP
WHERE SAL + NVL(COMM,0) >= 12500
```

1.10 Expressions

Les expressions acceptées par SQL portent sur des colonnes, des constantes, des fon tions.

Ces trois types d'éléments peuvent être reliés par des opérateurs arithmétiques (+ - * maniant des chaînes de caractères (|| pour concaténer des chaînes), des dates (- donne nombre de jours entre deux dates).

Les priorités des opérateurs arithmétiques sont celles de l'arithmétique classique (* / , puis + et -). Il est possible d'ajouter des parenthèses dans les expressions pour obten l'ordre de calcul que l'on désire.

Les expressions peuvent figurer :

– en tant que colonne résultat d'un SELECT

- dans une clause WHERE
- dans une clause ORDER BY (étudiée en 3.11)
- dans les ordres de manipulations de données (INSERT, UPDATE, DELETE étudiés au chapitre 2)

Le tableau qui suit donne le nom des principales fonctions (voir 3.10 pour plus de détails).

DE GROUPE	ARITHMETIQUES	DE CHAINES	DE DATES
SUM	NVL	NVL	NVL
COUNT	TO_CHAR	SUBSTR	TO_CHAR
VARIANCE	SQRT	LENGTH	ADD_MONTHS
MAX	ABS	INSTR	MONTHS_BETWEEN
MIN	POWER	TO_NUMBER	NEXT_DAY

Remarque 1.2
Toute expression dont au moins un des termes a la valeur NULL donne comme résultat la valeur NULL.

La fonction NVL (*Null VaLue*) permet de remplacer une valeur NULL par une valeur par défaut :

```
NVL (expr1, expr2)
```

prend la valeur *expr1*, sauf si *expr1* a la valeur NULL, auquel cas NVL prend la valeur *expr2*.

Exemple 1.6 `NVL(COMM, 0)`

Chapitre 2

Langage de manipulation des données

Le langage de manipulation de données est le langage permettant de modifier les info mations contenues dans la base.

Il existe trois commandes SQL permettant d'effectuer les trois types de modificatio des données :

INSERT	ajout de lignes
UPDATE	mise à jour de lignes
DELETE	suppression de lignes

Ces trois commandes travaillent sur la base telle qu'elle était au début de l'exécutio de la commande. Les modifications effectuées par les autres utilisateurs entre le début la fin de l'exécution ne sont pas prises en compte (même pour les transactions validées).

2.1 Insertion

```
INSERT INTO table (col1,..., coln)
VALUES (val1,...,valn)
```

ou

```
INSERT INTO table (col1,..., coln)
SELECT ...
```

table est le nom de la table sur laquelle porte l'insertion. *col1,..., coln* est la liste des nom des colonnes pour lesquelles on donne une valeur. Cette liste est optionnelle. Si elle e omise, ORACLE prendra par défaut l'ensemble des colonnes de la table dans l'ordre elles ont été données lors de la création de la table. Si une liste de colonnes est spécifié les colonnes ne figurant pas dans la liste auront la valeur NULL.

Exemples 2.1

(a)
```
INSERT INTO dept VALUES (10, 'FINANCES', 'PARIS')
```

(b)
```
INSERT INTO dept (lieu, nomd, dept)
   VALUES ('GRENOBLE', 'RECHERCHE', 20)
```

La deuxième forme avec la clause SELECT permet d'insérer dans une table des lignes provenant d'une table de la base. Le SELECT a la même syntaxe qu'un SELECT normal.

Exemple 2.2

Enregistrer la participation de MARTIN au groupe de projet numéro 10 :

```
INSERT INTO PARTICIPATION (MATR, CODEP)
SELECT MATR, 10
FROM EMP
WHERE NOME = 'MARTIN';
```

2.2 Modification

La commande UPDATE permet de modifier les valeurs d'un ou plusieurs champs, dans une ou plusieurs lignes existantes d'une table.

```
UPDATE table
SET col1 = exp1, col2 = exp2, ...
WHERE prédicat
```

ou

```
UPDATE table
SET (col1, col2,...) = (SELECT ...)
WHERE prédicat
```

table est le nom de la table mise à jour ; *col1*, *col2*, ... sont les noms des colonnes qui seront modifiées ; *exp1*, *exp2*,... sont des expressions. Elles peuvent aussi être un ordre SELECT renvoyant les valeurs attribuées aux colonnes (deuxième variante de la syntaxe).

Les valeurs de *col1*, *col2*... sont mises à jour dans toutes les lignes satisfaisant le prédicat. La clause WHERE est facultative. Si elle est absente, *toutes* les lignes sont mises à jour.

Exemples 2.3

(a) Faire passer MARTIN dans le département 10 :

```
UPDATE EMP SET DEPT = 10
WHERE NOME = 'MARTIN';
```

(b) Augmenter de 10 % les commerciaux :

```
UPDATE EMP
SET SAL = SAL * 1.1
WHERE POSTE = 'COMMERCIAL';
```

(c) Donner à CLEMENT un salaire 10 % au dessus de la moyenne des salaires des secrétaires :

```
UPDATE EMP
```



```
SET SAL = (SELECT AVG(SAL) * 1.10
           FROM EMP
           WHERE POSTE = 'SECRETAIRE')
WHERE NOME = 'CLEMENT'
```

On remarquera que la moyenne des salaires sera calculée pour les valeurs qu'avaie les salaires au début de l'exécution de la commande UPDATE et que les modificatio effectuées sur la base pendant l'exécution de cette commande ne seront pas prises e compte.

(d) Enlever (plus exactement, mettre à la valeur "NULL") la commission de MARTIN

```
UPDATE EMP
SET COMM = NULL
WHERE NOME = 'MARTIN'
```

2.3 Suppression

L'ordre DELETE permet de supprimer des lignes d'une table.

```
DELETE FROM table
WHERE prédicat
```

La clause WHERE indique quelles lignes doivent être supprimées. ATTENTION : cet clause est facultative ; si elle n'est pas précisée, TOUTES LES LIGNES DE LA TABL SONT SUPPRIMEES (heureusement qu'il existe ROLLBACK !).

Exemple 2.4 `DELETE FROM dept WHERE dept = 10`

2.4 Transactions : Commit/Rollback

Une transaction est un ensemble de modifications de la base qui forment un tout ind visible : il faut effectuer ces modifications entièrement ou pas du tout, sous peine de laiss la base dans un état incohérent.

Une transaction commence au début d'une session de travail ou juste après la fin de transaction précédente. L'utilisateur peut à tout moment valider (et terminer) la transa tion en cours par la commande **COMMIT**. Les modifications deviennent alors définitiv et visibles à toutes les autres transactions.

IMPORTANT : tant que la transaction n'est pas validée, les insertions, modificatio et suppressions qu'elle a exécutées n'apparaissent pas aux autres transactions.

L'utilisateur peut annuler (et terminer) la transaction en cours par la comman **ROLLBACK**. Toutes les modifications depuis le début de la transaction sont annulées

Ce mécanisme est utilisé par Oracle pour assurer l'intégrité de la base en cas de f anormale d'une tâche utilisateur : Oracle lance automatiquement un ROLLBACK des tra sactions non terminées.

Si une erreur survient lors d'une transaction, un ROLLBACK est automatiquement effectué. Cela signifie par exemple que, si une erreur survient au cours d'une instruction UPDATE qui modifie plusieurs lignes, aucune ligne ne sera modifiée.

Remarque 2.1

Certains ordres SQL, notamment ceux de définitions de données, provoquent une validation automatique de la transaction en cours. Le fait de quitter SQLPLUS par EXIT provoque également un COMMIT.

On peut indiquer si la transaction écrira ou non dans la base :

```
SET TRANSACTION {READ ONLY | READ WRITE}
```

On peut aussi indiquer le degré de concurrence que l'on accepte pour cette transaction par rapport aux autres transactions (voir chapitre 5).

Chapitre 3

Interrogations

3.1 Syntaxe générale

L'ordre SELECT possède six clauses différentes, dont seules les deux premières sor obligatoires. Elles sont données ci-dessous, dans l'ordre dans lequel elles doivent apparaîtr quand elles sont utilisées :

```
SELECT ...
FROM ...
WHERE ...
GROUP BY ...
HAVING ...
ORDER BY ...
```

3.2 Clause SELECT

Cette clause permet d'indiquer quelles colonnes, ou quelles expressions doivent êt retournées par l'interrogation.

```
SELECT [DISTINCT] *
```

ou

```
SELECT [DISTINCT] exp1 [[AS] nom1], exp2 [[AS] nom2], .....
```

exp1, *exp2*, ... sont des expressions, *nom1*, *nom2*, ... sont des noms facultatifs de 30 cara tères maximum donnés aux expressions. Chacun de ces noms est inséré derrière l'expressio séparé de cette dernière par un blanc ; il constituera le titre de la colonne dans l'affichag du résultat de la sélection. Le mot clé AS est optionnel.

"*" signifie que toutes les colonnes de la table sont sélectionnées.

Le mot clé facultatif DISTINCT ajouté derrière l'ordre SELECT permet d'éliminer l duplications : si, dans le résultat, plusieurs lignes sont identiques, une seule sera conservé

Exemples 3.1

(a) `SELECT * FROM DEPT`

(b)
```
SELECT DISTINCT POSTE FROM EMP
```

(c)
```
SELECT NOME, SAL + NVL(COMM,0) AS Salaire
FROM EMP
```

Si le nom contient des séparateurs (espace, caractère spécial), ou s'il est identique à un mot réservé SQL (exemple : DATE), il doit être mis entre guillemets.

Exemple 3.2

```
SELECT NOME, SAL + NVL(COMM,0) "Salaire Total"
FROM EMP
```

Le nom complet d'une colonne d'une table est le nom de la table suivi d'un point et du nom de la colonne. Par exemple : EMP.MATR, EMP.DEPT, DEPT.DEPT

Le nom de la table peut être omis quand il n'y a pas d'ambiguïté. Il doit être précisé s'il y a une ambiguïté, ce qui peut arriver quand on fait une sélection sur plusieurs tables à la fois et que celles-ci contiennent des colonnes qui ont le même nom (voir en particulier 3.5).

Remarque 3.1

La norme SQL-2, mais pas Oracle, permet d'avoir des SELECT emboîtés parmi les expressions.

3.3 Clause FROM

La clause FROM donne la liste des tables participant à l'interrogation. Il est possible de lancer des interrogations utilisant plusieurs tables à la fois.

`FROM` *table1* `[`*synonyme1*`] , `*table2* `[`*synonyme2*`] , ...`

synonyme1, *synonyme2*,... sont des synonymes attribués facultativement aux tables pour le temps de la sélection. On utilise cette possibilité pour lever certaines ambiguïtés, quand la même table est utilisée de plusieurs façons différentes dans une même interrogation (voir 3.5.1 ou 3.6.3 pour des exemples caractéristiques). Quand on a donné un synonyme à une table dans une requête, elle n'est plus reconnue sous son nom d'origine dans cette requête.

Le nom complet d'une table est celui de son créateur (celui du nom du shéma selon la norme SQL-2 : voir 4.1), suivi d'un point et du nom de la table. Par défaut, le nom du créateur est celui de l'utilisateur en cours. Ainsi, on peut se dispenser de préciser ce nom quand on travaille sur ses propres tables. *Mais il faut le préciser dès que l'on se sert de la table d'un autre utilisateur.*

Quand on précise plusieurs tables dans la clause FROM, on obtient le produit cartésien des tables. On verra en étudiant les jointures (voir 3.5) que l'on peut se restreindre à un sous-ensemble de ce produit cartésien grâce à la clause WHERE.

Exemple 3.3

Produit cartésien des noms des départements par les numéros des départements :

```
SQL> SELECT B.dept, A.nomd
```



```
  2  FROM dept A,dept B;

DEPT NOMD
---- ------------------
  10 FINANCES
  20 FINANCES
  30 FINANCES
  10 RECHERCHE
  20 RECHERCHE
  30 RECHERCHE
  10 VENTES
  20 VENTES
  30 VENTES
```

La norme SQL-2 (et les dernières versions d'Oracle) permet d'avoir un SELECT emboî à la place d'un nom de table.

Exemple 3.4

Pour obtenir la liste des employés avec le pourcentage de leur salaire par rapport a total des salaires, il fallait auparavant utiliser une vue (voir 4.3). Il est maintenan possible d'avoir cette liste avec une seule instruction SELECT :

```
select nome, sal, sal/total*100
from emp,
     (select sum(sal) as total
      from emp)
```

3.4 Clause WHERE

La clause WHERE permet de spécifier quelles sont les lignes à sélectionner dans u table ou dans le produit cartésien de plusieurs tables. Elle est suivie d'un prédicat (expre sion logique ayant la valeur vrai ou faux) qui sera évalué pour chaque ligne. Les lignes pou lesquelles le prédicat est vrai seront sélectionnées.

La clause where est étudiée ici pour la commande SELECT. Elle peut se rencontr aussi dans les commandes UPDATE et DELETE avec la même syntaxe.

3.4.1 Clause WHERE simple

`WHERE` *prédicat*

Un prédicat simple est la comparaison de deux expressions ou plus au moyen d'u opérateur logique :

```
WHERE exp1 = exp2
WHERE exp1 != exp2
WHERE exp1 < exp2
WHERE exp1 > exp2
WHERE exp1 <= exp2
WHERE exp1 >= exp2
WHERE exp1 BETWEEN exp2 AND exp3
WHERE exp1 LIKE exp2
WHERE exp1 NOT LIKE exp2
WHERE exp1 IN (exp2, exp3,...)
WHERE exp1 NOT IN (exp2, exp3,...)
WHERE exp IS NULL
WHERE exp IS NOT NULL
```

Les trois types d'expressions (arithmétiques, caractères, ou dates) peuvent être comparées au moyen des opérateurs d'égalité ou d'ordre (=, !=, <, >, <=, >=) : pour les types date, la relation d'ordre est l'ordre chronologique ; pour les types caractères, la relation d'ordre est l'ordre lexicographique.

Il faut ajouter à ces opérateurs classiques les opérateurs suivants BETWEEN, IN, LIKE, IS NULL :

exp1 BETWEEN *exp2* AND *exp3*
est vrai si *exp1* est compris entre *exp2* et *exp3*, bornes incluses.

exp1 IN (*exp2*, *exp3*...)
est vrai si *exp1* est égale à l'une des expressions de la liste entre parenthèses.

exp1 LIKE *exp2*
teste l'égalité de deux chaînes en tenant compte des caractères jokers dans la 2ème chaîne :

- "_" remplace 1 caractère exactement
- "%" remplace une chaîne de caractères de longueur quelconque, y compris de longueur nulle

Le fonctionnement est le même que celui des caractères joker ? et * pour le shell sous Unix. Ainsi l'expression 'MARTIN' LIKE '_AR%' sera vraie.

L'opérateur IS NULL permet de tester la valeur NULL :

exp IS [NOT] NULL
est vrai si l'expression a la valeur NULL (ou l'inverse avec NOT).

Remarque 3.2
Le prédicat "expr = NULL" est toujours faux, et ne permet donc pas de tester si l'expression est NULL.

3.4.2 Opérateurs logiques

Les opérateurs logiques AND et OR peuvent être utilisés pour combiner plusieurs prédicats (l'opérateur AND est prioritaire par rapport à l'opérateur OR). Des parenthèses

peuvent être utilisées pour imposer une priorité dans l'évaluation du prédicat, ou simplement pour rendre plus claire l'expression logique.

L'opérateur NOT placé devant un prédicat en inverse le sens.

Exemples 3.5

(a) Sélectionner les employés du département 30 ayant un salaire supérieur à 1500 frs.

```
SELECT NOME FROM EMP
WHERE DEPT = 30 AND SAL > 1500
```

(b) Afficher une liste comprenant les employés du département 30 dont le salaire est supérieur à 11000 Frs *et* (attention, à la traduction par OR) les employés qui ne touchent pas de commission.

```
SELECT nome
FROM emp
WHERE dept = 30 AND sal > 11000 OR comm IS NULL
```

(c)
```
SELECT * FROM EMP
WHERE (POSTE = 'DIRECTEUR' OR POSTE = 'SECRETAIRE')
    AND DEPT = 10
```

La clause WHERE peut aussi être utilisée pour faire des jointures (vues dans le cours sur le modèle relationnel) et des sous-interrogations (une des valeurs utilisées dans une WHERE provient d'une requête SELECT emboîtée) comme nous allons le voir dans les sections suivantes.

3.5 Jointure

Quand on précise plusieurs tables dans la clause FROM, on obtient le produit cartésien des tables. Le produit cartésien de deux tables offre en général peu d'intérêt. Ce qui est normalement souhaité, c'est de joindre les informations de diverses tables, en précisant quelles relations les relient entre elles. C'est la clause WHERE qui permet d'obtenir ce résultat. Elle vient limiter cette sélection en ne conservant que le sous-ensemble du produit cartésien qui satisfait le prédicat.

On retrouve l'opération de jointure du modèle relationnel.

Exemple 3.6
Liste des employés avec le nom du département où ils travaillent :

```
SELECT NOME, NOMD
FROM EMP, DEPT
WHERE EMP.DEPT = DEPT.DEPT
```

La clause WHERE indique de ne conserver dans le produit cartésien des tables EMP et DEPT que les éléments pour lesquels le numéro de département provenant de la table DEPT est le même que le numéro de département provenant de la table EMP. Ainsi, on obtiendra bien une jointure entre les tables EMP et DEPT d'après le numéro de département.

L'exemple 3.6 est un exemple d'équi-jointure. On peut aussi faire des jointures "non-équi" en remplaçant le signe "=" par un des opérateurs de comparaison (< <= > >=).

Remarque 3.3

La norme SQL-2 a introduit une nouvelle syntaxe pour la jointure qui permet de séparer les conditions de jointure des conditions de sélection des lignes. L'exemple 3.6 devient :

```
select nome, nomd
from emp join dept on dept.dept = emp.dept
```

Oracle n'admet pas cette syntaxe.

3.5.1 Jointure d'une table à elle-même

Il peut être utile de rassembler des informations venant d'une ligne d'une table avec des informations venant d'une autre ligne de la même table.

Dans ce cas il faut renommer au moins l'une des deux tables en lui donnant un synonyme (voir 3.3), afin de pouvoir préfixer sans ambiguïté chaque nom de colonne.

Exemple 3.7

Lister les employés qui ont un supéreur, en indiquant pour chacun le nom de son supérieur :

```
SELECT EMP.NOME EMPLOYE, SUPE.NOME SUPERIEUR
FROM EMP, EMP SUPE
WHERE EMP.SUP = SUPE.MATR
```

3.5.2 Jointure externe

Le SELECT suivant donnera la liste des employés et de leur département :

```
SELECT DEPT.DEPT, NOMD, NOME
FROM DEPT, EMP
WHERE DEPT.DEPT = EMP.DEPT
```

Dans cette sélection, un département qui n'a pas d'employé n'apparaîtra jamais dans la liste, puisqu'il n'y aura dans le produit cartésien des deux tables aucun élément où l'on trouve une égalité des colonnes DEPT.

On pourrait pourtant désirer une liste des divers départements, avec leurs employé s'ils en ont, sans omettre les départements vides. On écrira alors :

```
SELECT DEPT.DEPT, NOMD, NOME
FROM DEPT, EMP
WHERE DEPT.DEPT = EMP.DEPT (+)
```

Le (+) ajouté après un nom de colonne peut s'interpréter comme l'ajout, dans la tab à laquelle la colonne appartient, d'une ligne fictive qui réalise la correspondance avec l lignes de l'autre table, qui n'ont pas de correspondant réel.

Ainsi, dans l'ordre ci-dessus, le (+) après la colonne `EMP.DEPT` provoque l'ajout d'un ligne nulle virtuelle à la table `EMP`, pour laquelle le prédicat `DEPT.DEPT = EMP.DEPT` se vrai pour tout département sans employé. Les départements sans employé feront maint nant partie de cette sélection.

Remarque 3.4

Cette syntaxe est particulière à Oracle. La norme SQL2 spécifie une syntaxe différen (non admise par Oracle). L'exemple précédent s'écrirait :

```
SELECT DEPT.DEPT, NOMD, NOME
FROM emp RIGHT OUTER JOIN dept ON emp.dept = dept.dept
```

De même, il existe `LEFT OUTER JOIN` qui correspond à l'ajout d'une ligne fictive dan la table de gauche (avant le "`LEFT OUTER JOIN`") et `FULL OUTER JOIN` pour l'ajou de lignes fictives dans les deux tables.

3.6 Sous-interrogation

Une caractéristique puissante de SQL est la possibilité qu'un critère de recherche em ployé dans une clause WHERE (expression à droite d'un opérateur de comparaison) so lui-même le résultat d'un SELECT.

Par exemple, la sélection des employés ayant même poste que MARTIN peut s'écri en joignant la table EMP avec elle-même :

```
SELECT NOME
FROM EMP, EMP MARTIN
WHERE POSTE = MARTIN.POSTE
  AND MARTIN.NOME = 'MARTIN'
```

mais on peut aussi la formuler au moyen d'une sous-interrogation :

```
SELECT NOME
FROM EMP
WHERE POSTE = (SELECT POSTE
               FROM EMP
               WHERE NOME = 'MARTIN')
```

Les sections suivantes exposent les divers aspects de ces sous-interrogations.

3.6.1 Sous-interrogation à une ligne et une colonne

Dans ce cas, le SELECT imbriqué équivaut à une valeur.

```
WHERE exp op (SELECT ...)
```

où *op* est un des opérateurs = != < > <= >= *exp* est toute expression légale.

Exemple 3.8

Liste des employés travaillant dans le même département que MERCIER :

```
SELECT NOME FROM EMP
WHERE DEPT = (SELECT DEPT FROM EMP
              WHERE NOME = 'MERCIER')
```

Un SELECT peut comporter plusieurs sous-interrogations, soit imbriquées, soit au même niveau dans différents prédicats combinés par des AND ou des OR.

Exemples 3.9

(a) Liste des employés du département 10 ayant même poste que quelqu'un du département VENTES :

```
SELECT NOME, POSTE
FROM EMP
WHERE DEPT = 10
  AND POSTE IN
    (SELECT POSTE
     FROM EMP
     WHERE DEPT = (SELECT DEPT
                   FROM DEPT
                   WHERE NOMD = 'VENTES'))
```

(b) Liste des employés ayant même poste que MERCIER ou un salaire supérieur à CHATEL :

```
SELECT NOME, POSTE, SAL
FROM EMP
WHERE POSTE = (SELECT POSTE FROM EMP
               WHERE NOME = 'MERCIER')
   OR SAL > (SELECT SAL FROM EMP WHERE NOME = 'CHATEL')
```

Jointures et sous-interrogations peuvent se combiner.

Exemple 3.10

Liste des employés travaillant à LYON et ayant même poste que FREMONT.

```
SELECT NOME, POSTE
FROM EMP, DEPT
WHERE LIEU = 'LYON'
  AND EMP.DEPT = DEPT.DEPT
  AND POSTE = (SELECT POSTE FROM EMP
               WHERE NOME = 'FREMONT')
```


Attention : une sous-interrogation à une seule ligne doit ramener une seule ligne ; dan le cas où plusieurs lignes, ou pas de ligne du tout seraient ramenées, un message d'erreu sera affiché et l'interrogation sera abandonnée.

3.6.2 Sous-interrogation ramenant plusieurs lignes

Une sous-interrogation peut ramener plusieurs lignes à condition que l'opérateur c comparaison admette à sa droite un ensemble de valeurs.

Les opérateurs permettant de comparer une valeur à un ensemble de valeurs sont :

- l'opérateur IN
- les opérateurs obtenus en ajoutant ANY ou ALL à la suite des opérateurs de com paraison classique =, !=, <, >, <=, >=.

  ANY : la comparaison sera vraie si elle est vraie pour au moins un élément de l'en semble (elle est donc fausse si l'ensemble est vide).

  ALL : la comparaison sera vraie si elle est vraie pour tous les éléments de l'ensemb (elle est vraie si l'ensemble est vide).

```
WHERE exp op ANY (SELECT ...)
WHERE exp op ALL (SELECT ...)
WHERE exp IN (SELECT ...)
WHERE exp NOT IN (SELECT ...)
```

où *op* est un des opérateurs =, !=, <, >, <=, >=.

Exemple 3.11

Liste des employés gagnant plus que tous les employés du département 30 :

```
SELECT NOME, SAL FROM EMP
WHERE SAL > ALL (SELECT SAL FROM EMP
                 WHERE DEPT=30)
```

Remarque 3.5

L'opérateur IN est équivalent à "= ANY", et l'opérateur NOT IN est équivalent "!= ALL".

3.6.3 Sous-interrogation synchronisée

Il est possible de synchroniser une sous-interrogation avec l'interrogation principale.

Dans les exemples précédents, la sous-interrogation pouvait être évaluée d'abord, puis le résultat utilisé pour exécuter l'interrogation principale. SQL sait également traiter une sous-interrogation faisant référence à une colonne de la table de l'interrogation principale.

Le traitement dans ce cas est plus complexe car il faut évaluer la sous-interrogation pour chaque ligne de l'interrogation principale.

Exemple 3.12

Liste des employés ne travaillant pas dans le même département que leur supérieur.

```
SELECT NOME FROM EMP X
WHERE DEPT != (SELECT DEPT FROM EMP
               WHERE X.SUP = MATR)
```

Il a fallu renommer la table EMP de l'interrogation principale pour pouvoir la référencer dans la sous-interrogation.

3.6.4 Sous-interrogation ramenant plusieurs colonnes

Il est possible de comparer le résultat d'un SELECT ramenant plusieurs colonnes à une liste de colonnes. La liste de colonnes figurera entre parenthèses à gauche de l'opérateur de comparaison.

Avec une seule ligne sélectionnée :

```
WHERE (exp, exp,...) op (SELECT ...)
```

Avec plusieurs lignes sélectionnées :

```
WHERE (exp, exp,...) op ANY (SELECT ...)
WHERE (exp, exp,...) op ALL (SELECT ...)
WHERE (exp, exp,...) IN (SELECT ...)
WHERE (exp, exp,...) NOT IN (SELECT ...)
WHERE (exp, exp,...)
```

où *op* est un des opérateurs "=" ou "!="

Les expressions figurant dans la liste entre parenthèses seront comparées à celles qui sont ramenées par le SELECT.

Exemple 3.13

Employés ayant même poste et même salaire que MERCIER :

```
SELECT NOME, POSTE, SAL FROM EMP
WHERE (POSTE, SAL) =
  (SELECT POSTE, SAL FROM EMP
   WHERE NOME = 'MERCIER')
```


On peut utiliser ce type de sous-interrogation pour retrouver les lignes qui correspondent à des optima sur certains critères pour des regroupements de lignes (voir un exemple dans 3.8).

3.6.5 Clause EXISTS

La clause **EXISTS** est suivie d'une sous-interrogation entre parenthèses, et prend la valeur vrai s'il existe au moins une ligne satisfaisant les conditions de la sous-interrogation.

Exemple 3.14

```
SELECT NOMD FROM DEPT
WHERE EXISTS (SELECT NULL FROM EMP
              WHERE DEPT = DEPT.DEPT AND SAL > 10000);
```

Cette interrogation liste le nom des départements qui ont au moins un employé ayant plus de 10.000 comme salaire ; pour chaque ligne de DEPT la sous-interrogation synchronisée est exécutée et si au moins une ligne est trouvée dans la table EMP, EXISTS prend la valeur vrai et la ligne de DEPT satisfait les critères de l'interrogation.

Souvent on peut utiliser IN à la place de la clause EXISTS. Essayez sur l'exemple précédent.

Remarque 3.6

Il faut se méfier lorsque l'on utilise EXISTS en présence de valeurs NULL. Si on veut par exemple les employés qui ont la plus grande commission par la requête suivante :

```
select nome from emp e1
where not exists
  (select matr from emp
   where comm > e1.comm)
```

on aura en plus dans la liste tous les employés qui ont une commission NULL.

Division avec la clause EXISTS

NOT EXISTS permet de spécifier des prédicats où le mot "tous" intervient dans un sens comparable à celui de l'exemple suivant. Elle permet d'obtenir la division de deux relations.

On rappelle que la division de R par S sur l'attribut B (notée $R \div_B S$, ou $R \div S$ s'il n'y a pas d'ambiguïté sur l'attribut B) est la relation D définie par :

$D = \{a \in R[A] \ / \ \forall\, b \in S, (a,b) \in R\} = \{a \in R[A] \ / \ \nexists\, b \in S, (a,b) \notin R\}$

Faisons une traduction "mot à mot" de cette dernière définition en langage SQL :

```
select A from R  R1
where not exists
  (select C from S
```

```
   where not exists
     (select A, B from R
      where A = R1.A and B = S.C))
```

En fait, on peut remplacer les colonnes des select placés derrière des "not exists" par ce que l'on veut puisque seule l'existence ou non d'une ligne compte. On peut écrire par exemple :

```
select A from R  R1
where not exists
  (select null from S
   where not exists
     (select null from R
      where A = R1.A and B = S.C))
```

On arrive souvent à optimiser ce type de select en utilisant les spécificités du cas. Voyons un exemple concret : la réponse à la question "Quels sont les départements qui participent à tous les projets ?" est fourni par R $\div_{Dept}$ S où R = (PARTICIPATION JMatr EMP) [Dept, CodeP] ("JMatr" indique une jointure sur l'attribut Matr) et S = PROJET [CodeP]

La traduction donne (en remplaçant presque mot à mot R par PARTICIPATION, EMP et par la condition de jointure "PARTICIPATION.Matr = EMP.Matr", S par PROJET, A par EMP.Dept, B par PARTICIPATION.CodeP et C par PROJET.CodeP) :

```
SELECT DEPT FROM PARTICIPATION, EMP E1
WHERE PARTICIPATION.MATR = EMP.MATR
  AND NOT EXISTS
    (SELECT CODEP FROM PROJET
     WHERE NOT EXISTS
       (SELECT DEPT, CODEP FROM PARTICIPATION, EMP
        WHERE PARTICIPATION.MATR = EMP.MATR
          AND DEPT = E1.DEPT AND CODEP = PROJET.CODEP))
```

Sur ce cas particulier on voit qu'il est inutile de travailler sur la jointure de PARTICIPATION et de EMP pour le SELECT externe. On peut travailler sur la table DEPT. Il en est de même sur tous les cas où la table "R" est une jointure. D'après cette remarque, le SELECT précédent devient :

```
SELECT DEPT FROM DEPT
WHERE NOT EXISTS
  (SELECT CODEP FROM PROJET
   WHERE NOT EXISTS
     (SELECT DEPT, CODEP FROM PARTICIPATION, EMP
      WHERE PARTICIPATION.MATR = EMP.MATR
        AND DEPT = DEPT.DEPT AND CODEP = PROJET.CODEP))
```


3.7 Fonctions de groupes

Les fonctions de groupes peuvent apparaître dans le Select ou le Having (voir 3.9) ; c sont les fonctions suivantes :

AVG	moyenne
SUM	somme
MIN	plus petite des valeurs
MAX	plus grande des valeurs
VARIANCE	variance
STDDEV	écart type (déviation standard)
COUNT(*)	nombre de lignes
COUNT(*col*)	nombre de valeurs non nulles de la colonne
COUNT(**DISTINCT** *col*)	nombre de valeurs non nulles différentes

Exemples 3.15

(a)
```
SELECT COUNT(*)
FROM EMP
```

(b)
```
SELECT SUM(COMM)
FROM EMP
WHERE DEPT = 10
```

Les valeurs NULL sont ignorées par les fonctions de groupe. Ainsi, SUM(col) est somme des valeurs qui ne sont pas égales à NULL de la colonne 'col'. De même, AVG e la somme des valeurs non "NULL" divisée par le nombre de valeurs non "NULL".

Il faut remarquer qu'à un niveau de profondeur (relativement aux sous-interrogations d'un SELECT, les fonctions de groupe et les colonnes doivent être toutes du même nivea de regroupement. Par exemple, si on veut le nom et le salaire des employés qui gagnent plus dans l'entreprise, la requête suivante provoquera une erreur :

```
SELECT NOME, SAL
FROM EMP
WHERE SAL = MAX(SAL)
```

Il faut une sous-interrogation car MAX(SAL) n'est pas au même niveau de regroupeme que le simple SAL :

```
SELECT NOME, SAL
FROM EMP
WHERE SAL = (SELECT MAX(SAL) FROM EMP)
```

3.8 Clause GROUP BY

Il est possible de subdiviser la table en groupes, chaque groupe étant l'ensemble d lignes ayant une valeur commune.

`GROUP BY` *exp1, exp2,...*

groupe en une seule ligne toutes les lignes pour lesquelles *exp1*, *exp2*,... ont la même valeu

Cette clause se place juste après la clause WHERE, ou après la clause FROM si la clause WHERE n'existe pas.

Des lignes peuvent être éliminées avant que le groupe ne soit formé grâce à la clause WHERE.

Exemples 3.16

(a)
```
SELECT DEPT, COUNT(*)
FROM EMP
GROUP BY DEPT
```

(b)
```
SELECT DEPT, COUNT(*)
FROM EMP
WHERE POSTE = 'SECRETAIRE'
GROUP BY DEPT
```

(c)
```
SELECT NOME, DEPT
FROM EMP
WHERE (DEPT, SAL) =
  (SELECT DEPT, MAX(SAL)
   FROM EMP
   GROUP BY DEPT)
```

RESTRICTION :

Dans la liste des colonnes résultat d'un SELECT de groupe ne peuvent figurer que des caractéristiques de groupe, c'est-à-dire :

- soit des fonctions de groupe,
- soit des expressions figurant dans le GROUP BY.

Par exemple l'ordre suivant est invalide :
```
SELECT NOMD, SUM(SAL)
FROM EMP, DEPT
WHERE EMP.DEPT = DEPT.DEPT
GROUP BY DEPT.DEPT
```
Il aurait fallu écrire :
```
SELECT NOMD, SUM(SAL)
FROM EMP, DEPT
WHERE EMP.DEPT = DEPT.DEPT
GROUP BY NOMD
```

3.9 Clause HAVING

HAVING *prédicat*

sert à préciser quels groupes doivent être sélectionnés.

Elle se place après la clause GROUP BY.

Le *prédicat* suit la même syntaxe que celui de la clause WHERE. Cependant, il ne peu porter que sur des caractéristiques de groupe : fonction de groupe ou expression figura dans la clause GROUP BY.

Exemple 3.17
```
SELECT DEPT, COUNT(*)
FROM EMP
WHERE POSTE = 'SECRETAIRE'
GROUP BY DEPT
HAVING COUNT(*) > 1
```

On peut évidemment combiner toutes les clauses, des jointures et des sous-interrogatio La requête suivante donne les noms des départements (et leur nombre de secrétaires) q ont le plus de secrétaires :

```
SELECT NOMD Departement, COUNT(*) "Nombre de secretaires"
FROM EMP, DEPT
WHERE EMP.DEPT = DEPT.DEPT AND POSTE = 'SECRETAIRE'
GROUP BY NOMD
HAVING COUNT(*) =
         (SELECT MAX(COUNT(*))
          FROM EMP
          WHERE POSTE = 'SECRETAIRE'
          GROUP BY DEPT)
```

On remarquera que la dernière sous-interrogation est indispensable car MAX(COUNT n'est pas au même niveau de regroupement que les autres expressions du premier SELEC

3.10 Fonctions

Nous allons décrire ci-dessous les principales fonctions disponibles dans Oracle. Il fa remarquer que ces fonctions ne sont pas standardisées et ne sont pas toutes disponibl dans les autres SGBD.

3.10.1 Fonctions arithmétiques

ABS(*n*)	valeur absolue de *n*
MOD(*n1*, *n2*)	*n1* modulo *n2*
POWER(*n*, *e*)	*n* à la puissance *e*
ROUND(*n*[, *p*])	arrondit *n* à la précision *p* (0 par défaut)
SIGN(*n*)	-1 si *n*<0, 0 si *n*=0, 1 si *n*>0
SQRT(*n*)	racine carrée de *n*
TO_CHAR(*n*, *format*)	convertit *n* en chaîne de caractères (voir 3.10.2)
TRUNC(*n*[, *p*])	tronque *n* à la précision *p* (0 par défaut)
GREATEST(*n1*, *n2*,...)	maximum de *n1*, *n2*,...
LEAST(*n1*, *n2*,...)	minimum de *n1*, *n2*,...
TO_NUMBER(*chaîne*)	convertit la chaîne de caractères en numérique

Exemple 3.18

Calcul du salaire journalier :

```
SELECT NOME, ROUND(SAL/22, 2) FROM EMP
```

3.10.2 Fonctions chaînes de caractères

DECODE(*crit*, *val1*, *res1* [,*val2*, *res2*,...], *defaut*)
permet de choisir une valeur parmi une liste d'expressions, en fonction de la valeur prise par une expression servant de critère de sélection : elle prend la valeur *res1* si l'expression *crit* a la valeur *val1*, prend la valeur *res2* si *crit* a la valeur *val2*,... ; si l'expression *crit* n'est égale à aucune des expressions *val1*, *val2*... , DECODE prend la valeur *defaut* par défaut.

Les expressions résultat (*res1*, *res2*, *defaut*) peuvent être de types différents : caractère et numérique, ou caractère et date (le résultat est du type de la première expression rencontrée dans le DECODE).

Les expression "*val*" et "*res*" peuvent être soit des constantes, soit des colonnes ou même des expressions résultats de fonctions.

Exemple 3.19

Liste des employés avec leur catégorie (président = 1, directeur = 2, autre = 3) :

```
SELECT NOME, DECODE(POSTE, 'PRESIDENT', 1, 'DIRECTEUR', 2, 3)
```

LENGTH(*chaîne*)
prend comme valeur la longueur de la *chaîne*.

SUBSTR(*chaîne*, *position* [,*longueur*])
extrait de la chaîne *chaîne* une sous-chaîne de longueur *longueur* commençant en position *position* de la chaîne.

Le paramètre *longueur* est facultatif : par défaut, la sous-chaîne va jusqu'à l'extrémité de la chaîne.

INSTR(*chaîne*, *sous-chaîne* [,*pos* [,*n*]])
prend comme valeur la position de la sous-chaîne dans la chaîne (les positions sont num rotées à partir de 1). 0 signifie que la sous-chaîne n'a pas été trouvée dans la chaîne.

La recherche commence à la position *pos* de la chaîne (paramètre facultatif qui vau 1 par défaut). Une valeur négative de *pos* signifie une position par rapport à la fin de chaîne.

Le dernier paramètre *n* permet de rechercher la *n*ième occurrence de la sous-chaî dans la chaîne. Ce paramètre facultatif vaut 1 par défaut.

Exemple 3.20

Position du deuxième 'A' dans les postes :

```
SELECT INSTR (POSTE, 'A', 1, 2) FROM EMP
```

UPPER(*chaîne*) convertit les minuscules en majuscules

LOWER(*chaîne*) convertit les majuscules en minuscules

LPAD(*chaîne*, *long* [,*car*])
complète (ou tronque) *chaîne* à la longueur *long*. La chaîne est complétée à gauche par caractère (ou la chaîne de caractères) *car*.

Le paramètre *car* est optionnel. Par défaut, *chaîne* est complétée par des espaces.

RPAD(*chaîne*, *long* [,*car*]) a une fonction analogue, *chaîne* étant complétée à droite

Exemple : `SELECT LPAD (NOME, 10, '.') FROM EMP`

LTRIM(*chaîne*, *car*)
supprime les caractères à l'extrémité gauche de la chaîne "chaîne" tant qu'ils appartienne à l'ensemble de caractères "car".

RTRIM(*chaîne*, *car*)
a une fonction analogue, les caractères étant supprimés à l'extrémité droite de la chaîne

Exemple : supprimer les A et les M en tête des noms :

```
SELECT LTRIM(NOME,'AM') FROM EMP
```

TRANSLATE(*chaîne*, *car_source*, *car_cible*)
car_source et *car_cible* sont des chaînes de caractères considérées comme des ensembl de caractères. La fonction TRANSLATE remplace chaque caractère de la chaîne *chaî* présent dans l'ensemble de caractères *car_source* par le caractère correspondant (de mêm position) de l'ensemble *car_cible*.

Exemple : remplacer les A et les M par des * dans les noms des employés :

```
SELECT TRANSLATE (NOME, 'AM', '**') FROM EMP
```

REPLACE(*chaîne*, *ch1*, *ch2*) remplace *ch1* par *ch2* dans *chaîne*.

La fonction **TO_CHAR** permet de convertir un nombre ou une date en chaîne caractère en fonction d'un format :

Pour les nombres :

TO_CHAR (*nombre*, *format*)
nombre est une expression de type numérique, *format* est une chaîne de caractère pouva

contenir les caractères suivants :

9	représente un chiffre (non représente si non significatif)
0	représente un chiffre (présent même si non significatif)
.	point décimal apparent
,	une virgule apparaîtra à cet endroit
$	un $ précédera le premier chiffre significatif
B	le nombre sera représente par des blancs s'il vaut zéro
MI	le signe négatif sera à droite
PR	un nombre négatif sera entre < >

Exemple 3.21

Affichage des salaires avec au moins trois chiffres (dont deux décimales) :

```
SELECT TO_CHAR(SAL, '9990.00') FROM EMP
```

Pour les dates :

TO_CHAR (*date*, *format*)

format indique le format sous lequel sera affichée *date*. C'est une combinaison de codes ; en voici quelques uns :

YYYY	année
YY	deux derniers chiffres de l'année
WW	numéro de la semaine dans l'année
MM	numéro du mois
DDD	numéro du jour dans l'année
DD	numéro du jour dans le mois
D	numéro du jour dans la semaine
HH ou HH12	heure (sur 12 heures)
HH24	heure (sur 24 heures)
MI	minutes

Tout caractère spécial inséré dans le format sera reproduit dans la chaîne de caractère résultat. On peut également insérer dans le format une chaîne de caractères quelconque, à condition de la placer entre guillemets.

Exemple 3.22

```
SELECT TO_CHAR(DATEMB, 'DD/MM/YY HH24')
WHERE TO_CHAR(DATEMB) LIKE '%/05/91'
```

TO_NUMBER (*chaîne*)

convertit une chaîne de caractères en nombre (quand la chaîne de caractères est composée de caractères "numériques".

ASCII(*chaîne*)

donne le code ASCII du premier caractère de *chaîne*.

CHR(n) donne le caractère de code ASCII n.

TO_DATE(*chaîne*, *format*)

permet de convertir une chaîne de caractères en donnée de type date. Le *format* est identique à celui de la fonction TO_CHAR.

3.10.3 Fonctions de travail avec les dates

ROUND(*date*, *précision*)

arrondit la date à la précision spécifiée. La précision est indiquée en utilisant un des masqu de mise en forme de la date.

Exemple : premier jour de l'année où les employés ont été embauchés :

```
SELECT ROUND (DATEMB, 'YY') FROM EMP
```

TRUNC(*date*, *précision*)

tronque la date à la précision spécifiée (similaire à ROUND).

SYSDATE

a pour valeur la date et l'heure courante du système d'exploitation hôte.

Exemple : nombre de jour depuis l'embauche :

```
SELECT ROUND(SYSDATE - DATEMB) FROM EMP
```

3.10.4 Nom de l'utilisateur

USER

a pour valeur le nom sous lequel l'utilisateur est entré dans Oracle.

```
SELECT SAL FROM EMP
WHERE NOME = USER
```

3.11 Clause ORDER BY

Les lignes constituant le résultat d'un SELECT sont obtenues dans un ordre indéte miné. La clause ORDER BY précise l'ordre dans lequel la liste des lignes sélectionnées se donnée.

```
ORDER BY exp1 [DESC], exp2 [DESC], ...
```

L'option facultative **DESC** donne un tri par ordre décroissant. Par défaut, l'ordre e croissant.

Le tri se fait d'abord selon la première expression, puis les lignes ayant la même valeu pour la première expression sont triées selon la deuxième, etc. Les valeurs nulles so toujours en tête quel que soit l'ordre du tri (ascendant ou descendant).

Pour préciser lors d'un tri sur quelle expression va porter le tri, il est possible donner le rang relatif de la colonne dans la liste des colonnes, plutôt que son nom. Il e aussi possible de donner un nom d'en-tête de colonne du SELECT (voir 3.2).

Exemple 3.23

À la place de : `SELECT DEPT, NOMD FROM DEPT ORDER BY NOMD`
on peut taper : `SELECT DEPT, NOMD FROM DEPT ORDER BY 2`

Cette nouvelle syntaxe doit être utilisée pour les interrogations exprimées à l'aide d'u opérateur booléen UNION, INTERSECT ou MINUS.

Elle permet aussi de simplifier l'écriture d'un tri sur une colonne qui contient une expression complexe.

Exemples 3.24

(a) Liste des employés et de leur poste, triée par département et dans chaque département par ordre de salaire décroissant :

```
SELECT NOME, POSTE
FROM EMP
ORDER BY DEPT, SAL DESC
```

(b)
```
SELECT DEPT, SUM(SAL) "Total salaires"
FROM EMP
GROUP BY DEPT
ORDER BY 2
```

(c)
```
SELECT DEPT, SUM(SAL) "Total salaires"
FROM EMP
GROUP BY DEPT
ORDER BY SUM(SAL)
```

(d)
```
SELECT DEPT, SUM(SAL) "Total salaires"
FROM EMP
GROUP BY DEPT
ORDER BY "Total salaires"
```

3.12 Opérateurs booléens

Pour cette section on suposera que deux tables EMP1 et EMP2 contiennent les informations sur deux filiales de l'entreprise.

3.12.1 Opérateur UNION

L'opérateur UNION permet de fusionner deux sélections de tables pour obtenir un ensemble de lignes égal à la réunion des lignes des deux sélections. Les lignes communes n'apparaîtront qu'une fois.

Exemple 3.25

Liste des ingénieurs des deux filiales :

```
SELECT * FROM EMP1 WHERE POSTE='INGENIEUR'
UNION
SELECT * FROM EMP WHERE POSTE='INGENIEUR'
```


3.12.2 Opérateur INTERSECT

L'opérateur INTERSECT permet d'obtenir l'ensemble des lignes communes à deu interrogations.

Exemple 3.26

Liste des départements qui ont des employés dans les deux filiales :

```
SELECT DEPT FROM EMP1
INTERSECT
SELECT DEPT FROM EMP2
```

3.12.3 Opérateur MINUS

L'opérateur MINUS (EXCEPT pour la norme SQL-2) permet d'ôter d'une sélection l lignes obtenues dans une deuxième sélection.

Exemple 3.27

Liste des départements qui ont des employés dans la première filiale mais pas dan la deuxième.

```
SELECT DEPT FROM EMP1
MINUS
SELECT DEPT FROM EMP2
```

Chapitre 4

Langage de définition des données

Le "Langage de Définition des Données" est la partie de SQL qui permet de décrire les tables et autres objets manipulés par ORACLE.

4.1 Schéma

Un schéma est un ensemble d'objets (tables, vues, index, etc...) gérées ensemble. Cette notion correspond au concept habituel de base de données. On pourra ainsi avoir un schéma lié à la gestion du personnel et un autre lié à la gestion des clients.

Cette notion introduite par la norme SQL-2 n'est pas vraiment prise en compte par Oracle qui identifie pour le moment un nom de schéma avec un nom d'utilisateur.

4.2 Tables

4.2.1 CREATE TABLE AS

La commande CREATE a déjà été vue au premier chapitre. Une variante permet d'insérer pendant la création de la table des lignes venant d'autres tables :

```
CREATE TABLE table (col type.....)
AS SELECT .....
```

Exemple 4.1

```
CREATE TABLE MINIDEPT(CLE NUMBER, DATA CHAR(20))
AS SELECT DEPT, NOMD FROM DEPT
```

Cet ordre créera une table MINIDEPT et la remplira avec deux colonnes des lignes de la table DEPT.

Il faut évidemment que les définitions des colonnes de la table créée et du résultat de la sélection soient compatibles en type et en taille.

On peut également spécifier le mot-clé AS et l'interrogation directement derrière le no de la table. Dans ce cas les colonnes de la table créée auront les mêmes noms, types tailles que celles de l'interrogation :

```
CREATE TABLE DEPT10 AS SELECT * FROM DEPT WHERE DEPT = 10
```

4.2.2 ALTER TABLE

Oracle offre la possibilité de modifier la définition d'une table. Deux types de modifica tions sont possibles : ajout d'une colonne (après toutes les autres colonnes), et modificatio d'une colonne existante.

Il n'est pas possible de supprimer une colonne mais les valeurs d'une colonne qui n'es plus utilisée peuvent être mises à la valeur NULL.

4.2.3 Ajout d'une colonne - ADD

```
ALTER TABLE table
ADD (col1 type1, col2 type2, ...)
```

permet d'ajouter une ou plusieurs colonnes à une table existante. Les types possibles so les mêmes que ceux décrits avec la commande CREATE TABLE. Les parenthèses ne so pas nécessaires si on n'ajoute qu'une seule colonne.

L'attribut 'NOT NULL' peut être spécifié seulement si la table est vide (si la tab contient déjà des lignes, la nouvelle colonne sera nulle dans ces lignes existantes et donc condition 'NOT NULL' ne pourra être satisfaite).

4.2.4 Modification d'une colonne - MODIFY

```
ALTER TABLE table
MODIFY (col1 type1, col2 type2, ...)
```

col1, *col2*... sont les noms des colonnes que l'on veut modifier. Elles doivent bien sûr dé exister dans la table. *type1*, *type2*,... sont les nouveaux types que l'on désire attribuer au colonnes.

Il est possible de modifier la définition d'une colonne, à condition que la colonne contiennent que des valeurs NULL ou que la nouvelle définition soit compatible avec contenu de la colonne :

- on ne peut pas diminuer la taille maximale d'une colonne.
- on ne peut spécifier 'NOT NULL' que si la colonne ne contient pas de valeur nulle

Mais il est toujours possible d'augmenter la taille maximale d'une colonne, tant qu'o ne dépasse pas les limites propres à SQL, et on peut dans tous les cas spécifier 'NUL pour autoriser les valeurs nulles.

4.2.5 DROP TABLE

`DROP TABLE table`
permet de supprimer une table : les lignes de la table et la définition elle-même de la table sont détruites. L'espace occupé par la table est libéré.

4.3 Vues

On peut enregistrer un ordre SELECT en tant que vue. Les utilisateurs pourront consulter la base, ou modifier la base (avec certaines restrictions) à travers la vue, c'est-à-dire manipuler la table résultat du SELECT comme si c'était une table réelle.

Seule la définition de la vue est enregistrée dans la base, et pas les données de la vue. On peut parler de table virtuelle.

4.3.1 CREATE VIEW

La commande CREATE VIEW permet de créer une vue en spécifiant le SELECT constituant la définition de la vue :
`CREATE VIEW vue (col1, col2...) AS SELECT ...`

La spécification des noms des colonnes de la vue est facultative : par défaut, les colonnes de la vue ont pour nom les noms des colonnes résultat du SELECT. Si certaines colonnes résultat du SELECT sont des expressions sans nom, il faut alors obligatoirement spécifier les noms de colonnes de la vue.

Le SELECT peut contenir toutes les clauses d'un SELECT, sauf la clause ORDER BY.

Exemple 4.2

Vue constituant une restriction de la table EMP aux employés du département 10 :

```
CREATE VIEW EMP10 AS
SELECT * FROM EMP
WHERE DEPT = 10
```

Remarque 4.1

Dans l'exemple ci-dessus il aurait été plus prudent et plus souple d'éviter d'utiliser "*" et de le remplacer par les noms des colonnes de la table EMP. En effet, si la définition de la table EMP est modifiée, il y aura une erreur à l'exécution si on ne reconstruit pas la vue EMP10.

4.3.2 DROP VIEW

`DROP VIEW vue`
supprime la vue "*vue*".

4.3.3 Utilisation des vues

Une vue peut être référencée dans un SELECT de la même façon qu'une table. Ainsi, est possible de consulter la vue EMP10. Tout se passe comme s'il existait une table EMP1 des employés du département 10 :

```
SELECT * FROM EMP10
```

Mise à jour avec une vue

Il est possible d'effectuer des INSERT et des UPDATE à travers des vues, sous deu conditions :

- le SELECT définissant la vue ne doit pas comporter de jointure,
- les colonnes résultats du SELECT doivent être des colonnes réelles et non des e pressions composées.

Ainsi, il est possible de modifier les salaires du département 10 à travers la vue EMP1 Toutes les lignes de la table EMP avec DEPT = 10 seront modifiées :

```
UPDATE EMP10
SET SAL = SAL * 1.1
```

Une vue peut créer des données qu'elle ne pourra pas visualiser. On peut ainsi ajout un employé du département 20 avec la vue EMP10.

Si l'on veut éviter cela il faut ajouter "**WITH CHECK OPTION**" dans l'ordre c création de la vue après l'interrogation définissant la vue. Il est alors interdit de créer a moyen de la vue des lignes qu'elle ne pourrait relire. Ce dispositif fonctionne égalemen pour les mises à jour.

Exemple 4.3

```
CREATE VIEW EMP10 AS
SELECT * FROM EMP
WHERE DEPT = 10
WITH CHECK OPTION
```

4.3.4 Utilité des vues

De façon générale, les vues permettent de dissocier la façon dont les utilisateurs voie les données, du découpage en tables. On sépare l'aspect externe (ce que voit un utilisa teur particulier de la base) de l'aspect conceptuel (comment a été conçu l'ensemble de base). Ceci favorise l'indépendance entre les programmes et les données. Si la structure d données est modifiée, les programmes ne seront pas à modifier si l'on a pris la précautio d'utiliser des vues (ou si on peut se ramener à travailler sur des vues). Par exemple, si u table est découpée en plusieurs tables après l'introduction de nouvelles données, on pe introduire une vue, jointure des nouvelles tables, et la nommer du nom de l'ancienne tab pour éviter de réécrire les programmes qui utilisaient l'ancienne table.

Une vue peut aussi être utilisée pour restreindre les droits d'accès à certaines colonnes et à certaines lignes d'une table : un utilisateur peut ne pas avoir accès à une table mais avoir les autorisations pour utiliser une vue qui ne contient que certaines colonnes de la table ; on peut de plus ajouter des restrictions d'utilisation sur cette vue comme on le verra en 4.5.1.

Dans le même ordre d'idées, une vue peut être utilisée pour implanter une contrainte d'intégrité grâce à l'option "WITH CHECK OPTION".

Une vue peut également simplifier la consultation de la base en enregistrant des SELECT complexes.

Exemple 4.4

En créant la vue :

```
CREATE VIEW EMPSAL AS
  SELECT NOME, SAL + NVL(COMM, 0) GAINS, NOMD
  FROM EMP, DEPT
  WHERE EMP.DEPT = DEPT.DEPT
```

La liste des employés avec leur rémunération totale et le nom de leur département sera obtenue simplement par :

```
SELECT * FROM EMPSAL
```

4.4 Index

Considérons le SELECT suivant :

```
SELECT * FROM EMP
WHERE NOME  = 'MARTIN'
```

Un moyen de retrouver la ou les lignes pour lesquelles NOME est égal à 'MARTIN' est de balayer toute la table.

Un tel moyen d'accès conduit à des temps de réponse prohibitifs pour des tables dépassant quelques milliers de lignes.

Une solution est la création d'index, qui permettra de satisfaire aux requêtes les plus fréquentes avec des temps de réponses acceptables.

Un index est formé de clés auxquelles SQL peut accéder très rapidement. Comme pour l'index d'un livre, ces clés permettent de lire ensuite directement les données repérées par les clés.

4.4.1 CREATE INDEX

Un index se crée par la commande CREATE INDEX :

```
CREATE [UNIQUE] INDEX nom-index ON table (col1, col2,...)
```


On peut spécifier par l'option "UNIQUE" que chaque valeur d'index doit être unique dan la table.

Remarque 4.2

Deux index construits sur des tables d'un même utilisateur ne peuvent avoir le mêm nom (même s'ils sont liés à deux tables différentes).

4.4.2 Utilisation des index

Un index peut être créé juste après la création d'une table ou sur une table contenan déjà des lignes. Il sera ensuite tenu à jour automatiquement lors des modifications de table.

Un index peut porter sur plusieurs colonnes : la clé d'accès sera la concaténation d différentes colonnes.

On peut créer plusieurs index indépendants sur une même table.

Les requêtes SQL sont transparentes au fait qu'il existe un index ou non. C'est l'op timiseur du SGBD qui, au moment de l'exécution de chaque requête, recherche s'il peu s'aider d'un index.

La principale utilité des index est d'accélérer les recherches d'informations dans la bas Une ligne est retrouvée instantanément si la recherche peut utiliser un index. Sinon, un recherche séquentielle sur toutes les lignes de la table doit être effectuée. Il faut cependan noter que les données à retrouver doivent correspondre à environ moins de 20 % de toute les lignes sinon une recherche séquentielle est préférable.

Les index concaténés permettent même dans certains cas de récupérer toutes les donnée cherchées sans accéder à la table.

Une jointure s'effectuera souvent plus rapidement si une des colonnes de jointure es indexée (s'il n'y a pas trop de valeurs égales dans les colonnes indexées).

Les index accélèrent le tri des données si le début de la clé de tri correspond à un inde

Une autre utilité des index est d'assurer l'unicité d'une clé en utilisant l'option "UN QUE". Ainsi la création de l'index suivant empêchera l'insertion dans la table EMP d'u nom d'employé existant :

```
CREATE UNIQUE INDEX NOME ON EMP (NOME)
```

Il faut cependant savoir que les modifications des données sont ralenties si un ou plu sieurs index doivent être mis à jour. De plus, la recherche d'information n'est accélér par un index que si l'index ne contient pas trop de données égales. Il n'est pas bon, pa exemple, d'indexer une colonne SEXE qui ne pourrait contenir que des valeurs "M" ou "F

4.4.3 DROP INDEX

Un index peut être supprimé par la commande DROP INDEX :

```
DROP INDEX nom_index [ON table]
```

Le nom de la table est obligatoire si vous voulez supprimer un index d'une table d'un autre utilisateur alors que vous possédez un index du même nom.

Remarque 4.3

Un index est automatiquement supprimé dès qu'on supprime la table à laquelle il appartient.

4.5 Privilèges d'accès à la base

Oracle permet à plusieurs utilisateurs de travailler en toute sécurité sur la même base.

Chaque donnée peut être confidentielle et accessible à un seul utilisateur, ou partageable entre plusieurs utilisateurs.

Les ordres GRANT et REVOKE permettent de définir les droits de chaque utilisateur sur les objets de la base.

Tout utilisateur doit communiquer son nom d'utilisateur et son mot de passe pour pouvoir accéder à la base. C'est ce nom d'utilisateur qui déterminera les droits d'accès aux objets de la base.

L'utilisateur qui crée une table est considéré comme le propriétaire de cette table. Il a tous les droits sur cette table et son contenu. En revanche, les autres utilisateurs n'ont aucun droit sur cette table (ni lecture ni modification), à moins que le propriétaire ne leur donne explicitement ces droits avec un ordre GRANT.

4.5.1 GRANT

L'ordre GRANT du langage SQL permet au propriétaire d'une table ou d'une vue de donner à d'autres utilisateurs des droits d'accès à celles-ci :

```
GRANT privilège ON table/vue TO utilisateur [WITH GRANT OPTION]
```

Les privilèges qui peuvent être donnés sont les suivants :

SELECT	droit de lecture
INSERT	droit d'insertion de lignes
UPDATE	droit de modification de lignes
UPDATE (*col1*, *col2*, ...)	droit de modification de lignes limité à certaines colonnes
DELETE	droit de suppression de lignes
ALTER	droit de modification de la définition de la table
INDEX	droit de création d'index
ALL	tous les droit ci-dessus

Les privilèges SELECT, INSERT et UPDATE s'appliquent aux tables et aux vues. Les autres s'appliquent uniquement aux tables.

Un utilisateur ayant reçu un privilège avec l'option facultative "**WITH GRANT OPTION**" peut le transmettre à son tour.

Exemple 4.5

L'utilisateur DUBOIS peut autoriser l'utilisateur CLEMENT à lire sa table EMP :


```
GRANT SELECT ON EMP TO CLEMENT
```

Dans un même ordre GRANT, on peut accorder plusieurs privilèges à plusieurs utilisateurs :

```
GRANT SELECT, UPDATE ON EMP TO CLEMENT, CHATEL
```

Les droits peuvent être accordés à tous les utilisateurs en utilisant le mot réservé **PUBLIC** à la place d'un nom d'utilisateur :

```
GRANT SELECT ON EMP TO PUBLIC
```

4.5.2 REVOKE

Un utilisateur ayant accordé un privilège peut le reprendre à l'aide de l'ordre REVOKE

```
REVOKE privilège ON table/vue FROM utilisateur
```

Remarque 4.4

Si on enlève un privilège à un utilisateur, ce privilège est automatiquement retiré à tout autre utilisateur à qui il aurait accordé ce privilège.

4.5.3 Changement de mot de passe

Tout utilisateur peut modifier son mot de passe par l'ordre GRANT CONNECT :

```
GRANT CONNECT TO utilisateur IDENTIFIED BY mot-de-passe
```

4.6 Procédure stockée

Une procédure stockée est un programme qui comprend des instructions SQL précompilées et qui est enregistré dans la base de données (plus exactement dans le dictionnaire des données, notion étudiée en 4.8).

Le plus souvent le programme est écrit dans un langage spécial qui contient à la fois des instructions procédurales et des ordres SQL. Ces instructions ajoutent les possibilités habituelles des langages dits de troisième génération comme le langage C ou le Pascal (boucles, tests, fonctions et procédures,...). Oracle offre ainsi le langage PL/SQL qui se rapproche de la syntaxe du langage Ada et qui inclut des ordres SQL. Malheureusement aucun de ces langages (ni la notion de procédure stockée) n'est normalisé et ils sont donc liés à chacun des SGBD.

Les procédures stockées offrent des gros avantages pour les applications client/serveur surtout au niveau des performances :

- le trafic sur le réseau est réduit car les clients SQL ont seulement à envoyer l'identification de la procédure au serveur sur lequel elle est stockée.

- les procédures sont précompilées une seule fois quand elles sont enregistrées. L'optimisation a lieu à ce moment et leurs exécutions ultérieures n'ont plus à passer par cette étape et sont donc plus rapides.

– la gestion et la maintenance des procédures sont facilitées car elles sont enregistrées sur le serveur et ne sont pas dispersées sur les postes clients.

4.7 Trigger

Les *triggers* (on dirait "déclencheurs" en français) ressemblent aux procédures stockées car ils sont eux aussi enregistrés dans le dictionnaire des données de la base et ils sont le plus souvent écrits dans le même langage. La différence est qu'ils sont déclenchés automatiquement par des événements liés à des actions sur la base, par exemple, une insertion dans une table si certaines conditions sont remplies.

Les *triggers* complètent les contraintes d'intégrité en permettant des contrôles et des traitements plus complexes.

Il est prévu une future normalisation des *triggers*.

L'outil SQL*FORMS d'Oracle (construction et utilisation d'écrans de saisie) utilise aussi le terme de *trigger* mais pour des programmes dont l'exécution est déclenchée par des actions de l'utilisateur pas toujours liées aux données enregistrées dans la base (par exemple, sorie d'une zone de saisie ou entrée dans un bloc logique de l'écran de saisie).

4.8 Dictionnaire de données

Le dictionnaire de données est un ensemble de tables dans lesquelles sont stockées les descriptions des objets de la base. Il est tenu à jour automatiquement par Oracle.

Les tables de ce dictionnaire peuvent être consultées au moyen du langage SQL.

Des vues de ces tables permettent à chaque utilisateur de ne voir que les objets qui lui appartiennent ou sur lesquels il a des droits. D'autres vues sont réservées aux administrateurs de la base.

Voici les principales vues et tables du dictionnaire de données qui sont liées à un utilisateur (certaines ont des synonymes qui sont donnés entre parenthèses) :

DICTIONARY (DICT)	vues permettant de consulter le dictionnaire de données
USER_TABLES	tables et vues créées par l'utilisateur
USER_CATALOG (CAT)	tables et vues sur lesquelles l'utilisateur courant a des droits, à l'exclusion des tables et vues du dictionnaire de données
USER_TAB_COLUMNS (COLS)	colonnes de chaque table ou vue créée par l'utilisateur courant
USER_INDEXES (IND)	index créés par l'utilisateur courant ou indexant des tables créées par l'utilisateur
USER_VIEWS	vues créées par l'utilisateur
USER_TAB_GRANTS	objets sur lesquels l'utilisateur est propriétaire donneur ou receveur d'autorisation
USER_CONSTRAINTS	définition des contraintes pour les tables de l'utilisateur
USER_CONS_COLUMNS	colonnes qui interviennent dans les définitions des contraintes

Exemples 4.6

(a) Colonnes de la table EMP :

```
SELECT * FROM COLS WHERE TNANE = 'EMP'
```

(b) Informations sur les contraintes de type UNIQUE, PRIMARY ou REFERENCES

```
SELECT T.TABLE_NAME, T.CONSTRAINT_NAME, T.CONSTRAINT_TYPE, COLUMN_NAI
FROM USER_CONSTRAINTS T, USER_CONS_COLUMNS C
WHERE T.CONSTRAINT_NAME = C.CONSTRAINT_NAME
  AND CONSTRAINT_TYPE  IN ('U', 'P', 'R')
```

(c) Informations sur les contraintes de type CHECK ou NOT NULL :

```
SELECT TABLE_NAME, CONSTRAINT_NAME, SEARCH_CONDITION
FROM USER_CONSTRAINTS
WHERE CONSTRAINT_TYPE = 'C'
```

Chapitre 5

Gestion des accès concurrents

5.1 Niveaux d'"isolation" des transactions

`SET TRANSACTION ISOLATION LEVEL {SERIALIZABLE | READ COMMITTED}`
indique le niveau d'isolation de la transaction qui vient de commencer.

Oracle permet deux niveaux d'isolation des transactions les unes par rapport aux autres :

SERIALIZABLE : si une transaction SERIALIZABLE essaie de modifier une donnée qui pourrait avoir été modifiée par une autre transaction non validée au début de la transaction SERIALIZABLE alors l'ordre de modification échoue. On évite ainsi le problème dit de lecture non répétitive : la transaction lit une valeur et quand elle veut l'utiliser pour la modifier cette valeur a changé car l'autre transaction a été validée entre temps. Ce mode de fonctionnement permet d'isoler complètement les transactions les unes des autres : elles s'exécutent concurremment comme si elles s'exécutaient les unes après les autres. Ce mode est cependant coûteux puisqu'il limite le fonctionnement en parallèle des transactions.

READ COMMITTED : c'est le mode de fonctionnement par défaut des transactions d'Oracle. Il empêche les principaux problèmes de concurrence mais pas les lectures non répétitives. Il est décrit dans la section suivante.

5.2 Traitement par défaut des accès concurrents par Oracle

Il faut tout d'abord savoir que les modifications effectuées par une transaction ne sont connues des autres transactions que lorsque la transaction a été confirmée (ordre COMMIT).

Oracle gère automatiquement les accès concurrents de plusieurs transactions sur les mêmes lignes des tables. Si une transaction est en train de modifier des lignes d'une table,

les autres transactions peuvent lire les données telles qu'elles étaient avant ces dernière modifications (jamais de temps d'attente pour la lecture), mais les autres transactions son bloquées automatiquement par Oracle si elles veulent modifier ces mêmes lignes.

Plus précisément, certaines commandes provoquent un blocage implicite sur la tab et les lignes impliquées : DELETE, UPDATE, INSERT, SELECT FOR UPDATE et l ordres de définitions et de contrôle des données : LDD (Langage de Définition des Donnée et LCD (Langage de Contrôle des Données) : ALTER TABLE, GRANT, etc.

Ordre SQL	Blocage niveau ligne	Blocage niveau table
DELETE, UPDATE	Exclusif	Row Exclusive
INSERT		Row Exclusive
SELECT FOR UPDATE	Exclusif	Row Share
LDD/LCD		Exclusif

De plus, Oracle assure une "lecture consistante" des données pendant l'exécution d'u ordre SQL ; par exemple, un ordre SELECT ou UPDATE va travailler sur les lignes tell qu'elles étaient au moment du début de l'exécution de la commande, même si entre-temp une autre transaction validée par un COMMIT a modifié certaines de ces lignes. De mêm si une commande UPDATE comporte un SELECT emboîté, les modifications de la com mande UPDATE ne sont pas prises en compte par le SELECT emboîté.

On peut aussi obtenir une lecture consistante pour toute une transaction et pas seul ment pour un ordre SQL en l'indiquant explicitement à Oracle. Cependant, dans ce cas, transaction ne pourra pas modifier des données (voir 5.9).

5.3 Autres possibilités de blocages

Si le comportement par défaut d'Oracle ne convient pas pour un certain traitement, o peut effectuer des blocages au niveau des lignes ou des tables. Par exemple, si l'utilisateu veut que les valeurs des lignes sur lesquelles il va travailler ne soient pas modifiées par un autre transaction, il ne peut se fier aux blocages implicites de Oracle. En effet ceux-ci n peuvent lui assurer ceci que durant l'exécution d'un commande et pas durant toute un transaction.

Il existe cinq sortes de blocage au niveau des tables. Ces blocages sont étudiés en détai dans les sections suivantes. Voici les cinq variantes de la commande **LOCK TABLE** :

- IN EXCLUSIVE MODE
- IN SHARE ROW EXCLUSIVE MODE
- IN SHARE MODE
- IN ROW EXCLUSIVE
- IN ROW SHARE MODE

Au niveau des lignes il n'existe que le blocage exclusif.

Les blocages explicites peuvent conduire à des interblocages. Oracle détecte et résoud les interblocages (en annulant certaines transactions bloquées/bloquantes par un ROLLBACK).

Un verrouillage dure le temps d'une transaction. Il prend fin au premier COMMIT ou ROLLBACK (explicite ou implicite).

5.4 Verrouillage d'une table en mode exclusif (Exclusive)

`LOCK TABLE table IN EXCLUSIVE MODE [NOWAIT]`

Ce mode est utilisé lorsque l'on veut modifier des données de la table en s'assurant qu'aucune autre modification ne sera apportée sur la table par les autres utilisateurs ; en effet, dans ce mode

- celui qui a bloqué la table peut insérer, supprimer ou consulter,
- les autres ne peuvent que consulter la table. Ils sont aussi bloqués s'ils veulent bloquer la même table.

Ce mode étant très contraignant pour les autres utilisateurs (ils sont mis en attente et ne reprennent la main qu'au déblocage de la table), le blocage doit être le plus court possible. Un blocage en mode exclusif doit être suivi rapidement d'un COMMIT ou d'un ROLLBACK.

Remarque 5.1

Pour les 5 modes de blocage, l'option NOWAIT (ajoutée à la fin de la commande LOCK) permet de reprendre immédiatement la main au cas où la table que l'on veut bloquer serait déjà bloquée. Par exemple :

```
LOCK TABLE EMP IN EXCLUSIVE MODE NOWAIT
```

C'est au processus de relancer plus tard la commande LOCK.

5.5 Verrouillage d'une table en mode Share Row Exclusive

`LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE [NOWAIT]`

Ce blocage empêche tous les autres blocages sur les tables sauf le mode Row Share. Il empêche donc toute modification sur les tables. Il n'est pas très utilisé.

5.6 Verrouillage d'une table en mode partagé (Share)

`LOCK TABLE table IN SHARE MODE [NOWAIT]`

Ce mode interdit toute modification de la table, y compris par celui qui a bloqué table.

On peut l'utiliser lorsque l'on veut, par exemple, établir un bilan des données contenu dans une table. L'avantage par rapport au mode exclusif est que l'on n'est pas mis en attent si la table est déjà bloquée en mode partagé.

5.7 Verrouillage de lignes pour les modifier (mode Row Exclusive)

`LOCK TABLE table IN ROW EXCLUSIVE MODE [NOWAIT]`

C'est le blocage qui est effectué automatiquement par Oracle avant d'exécuter un ordr INSERT, DELETE ou UPDATE.

Il permet de modifier certaines lignes pendant que d'autres utilisateurs modifient d'autr lignes.

Il empêche les autres de bloquer en mode Share, Share Row Exclusive et Exclusive.

5.8 Verrouillage de lignes en mode Row Share

Ce mode empêche le blocage de la table en mode Exclusif.

`LOCK TABLE table IN ROW SHARE MODE [NOWAIT]`

C'est le mode implicite de blocage de la table de la commande SELECT FOR UPDAT dont la syntaxe est :

```
SELECT colonnes FROM table
WHERE condition
FOR UPDATE OF colonnes
```

Cette commande réserve les lignes sélectionnée pour une modification ultérieure (l lignes sont bloquées en mode exclusif) et les affiche. On peut ainsi préparer tranquilleme les modifications à apporter à ces lignes en étant certain qu'aucune modification ne leu sera apportée pendant cette préparation. On ne gêne pas trop les autres transactions q peuvent elles aussi se réserver d'autres lignes.

On remarquera que ce sont les lignes entières qui sont réservées et pas seulement l colonnes spécifiées à la suite de FOR UPDATE OF.

La réservation des lignes et leur modification s'effectuent en plusieurs étapes :

1. on indique les lignes que l'on veut se réserver. Pour cela on utilise la variante de commande SELECT avec la clause FOR UPDATE OF.
2. on peut effectuer une préparation avant de modifier les lignes réservées en tenan compte des valeurs que l'on vient de lire par le SELECT précédent.

3. on modifie les lignes réservées

   ```
   UPDATE table
   SET ...........
   WHERE condition
   ```

   ou

   ```
   DELETE FROM table
   WHERE condition
   ```

4. on lance un COMMIT ou un ROLLBACK pour libérer les lignes et ne pas trop gêner les autres transactions.

5.9 Lecture consistante pendant une transaction : set transaction read only

L'instruction "**SET TRANSACTION READ ONLY**" permet une lecture consistante durant toute une transaction et pas seulement pendant une seule instruction. Les données lues sont telles qu'elles étaient au début de la transaction.

La transaction ne pourra effectuer que des lectures. Les modifications des données de la base sont interdites. Les autres transactions peuvent faire ce qu'elles veulent.

On ne peut revenir en arrière. La transaction sera "read only" jusqu'à ce qu'elle se termine.

5.10 Tableau récapitulatif

Ce tableau indique les commandes qui sont autorisées dans chacun des modes de blocage.

Modes / **Commandes**	X	RSX	S	RX	RS
LOCK EXCLUSIVE (X)	non	non	non	non	non
LOCK ROW SHARE EXCLUSIVE (RSX)	non	non	non	non	OUI
LOCK SHARE (S)	non	non	OUI	non	OUI
LOCK ROW EXCLUSIVE (RX)	non	non	non	OUI	OUI
LOCK ROW SHARE (RS)	non	OUI	OUI	OUI	OUI
INSERT DELETE UPDATE	non	non	non	OUI *	OUI *
SELECT FOR UPDATE	non	OUI *	OUI *	OUI *	OUI *

* à condition que l'on ne travaille pas sur des lignes déjà bloquées par une autre transaction.

Chapitre 6

Java et JDBC

Nous allons prendre deux exemples d'insertion de commandes SQL dans un autre lan gage :

- insertion de commandes SQL en Java avec JDBC (étudiée dans ce chapitre),
- insertion de commandes SQL dans le langage C (décrite au chapitre suivant).

Ce chapitre décrit les fonctionnalités les plus importantes de JDBC. Pour plus de pr cisions, vous pouvez consulter, par exemple, le guide JDBC en ligne à l'adresse Web `http://www.javasoft.com/products/jdk/1.1/docs/guide/jdbc/`

JDBC ("*Java Data Base Connection*") offre des classes et des interfaces pour perme trel'utilisation d'un ou plusieurs SGBD à partir d'un programme en Java. Il est fourni pa le package Java "`java.sql`".

Les premières versions de JDBC supportent le standard "SQL-2 Entry Level" (le nivea le moins élevé, mais suffisant pour les traitements habituels). Il est prévu un support d niveaux supérieurs de SQL-2.

JDBC est indépendant du SGBD. Pour des raisons d'efficacité JDBC permet d'utilis les possibilités particulières d'un SGBD si l'implémentation de JDBC l'a prévu, mais a détriment de la portabilité.

6.1 Drivers et gestionnaire de drivers

Pour travailler avec un SGBD particulier il faut disposer de classes qui implémente les interfaces de JDBC (voir 6.5). Ces ensembles de classes sont désignés sous le nom d *drivers*.

Une de ces interfaces les plus importantes s'appelle `Driver`. les classes qui l'impl mentent doivent fournir une méthode `connect()` qui renvoie une instance d'une classe q implémente l'interface `Connection`. Cette instance permettra ensuite de lancer des requêt vers le SGBD en créant des instances de (classes qui implémentent) l'interface `Statemen`

Le gestionnaire de drivers gère la liste des drivers disponibles. En chargeant plusieu drivers on peut travailler en même temps avec plusieurs SGBD de types différents.

Quand une classe "Driver" est chargée en mémoire, un bloc *static* de la classe crée une instance de la classe et enregistre le driver pour le DriverManager. Lorsque l'on veut ouvrir une connexion avec une base de données dans un programme, le DriverManager essaie d'effectuer la connexion avec chacun des drivers qui se sont enregistrés et utilise le premier qui fonctionne pour la base de données. Les drivers sont testés dans l'ordre de leur enregistrement par le gestionnaire de drivers. On peut changer cet ordre en donnant les noms des drivers dans une "propriéte" Java `jdbc.drivers` (noms de drivers séparés par des ":").

6.1.1 Types de drivers

Il existe plusieurs types de drivers habituellement numérotés de 1 à 4 :

1. Le pont JDBC-ODBC permet l'accès aux bases de données en passant par les drivers ODBC (standard Microsoft). Les appels JDBC sont traduits en appels ODBC. Presque tous les SGBD peuvent être accédés par un driver ODBC.

2. Driver qui fait appel à des fonctions natives non Java de l'API (les "*Application Program Interface* sont un ensemble de fonctions ou procédures qui permettent d'utiliser les services d'un système d'exploitation ou d'un système ; ici il s'agit du SGBD) du SGBD que l'on veut utiliser.

3. Driver qui permet l'utilisation d'un protocole pour travailler avec un service dit "*middleware*" d'accès à plusieurs SGBD (voir 6.2) ou autres sources de données. Ce protocole est indépendant des SGBD. Le serveur "*middleware*" accèdera par un moyen quelconque aux différents SGBD (par exemple, JDBC s'il est écrit en Java).

4. Driver qui utilise le protocole réseau du SGBD. Il est écrit entièrement en Java.

   Une applette "*untrusted*" peut l'utiliser si le SGBD est installé sur la même machine que le serveur Web (sécurité pour les utilisations de "*sockets*").

6.1.2 Types de drivers et applet *untrusted*

Une application Java peut travailler avec tous les types de drivers.

Les applets ne peuvent pas charger à distance du code natif (non Java). Elle ne peuvent donc pas utiliser les drivers de type 1 et 2.

Pour des raisons de sécurité, les applets "*untrusted*" [1] ne peuvent échanger des données par "*sockets*" qu'avec les machines d'où elles proviennent. Pour être utilisé par ce type d'applet, le serveur "*middleware*" (type de driver 3) doit être sur la même machine que le serveur Web d'où vient l'applette car les échanges entre l'applet et le serveur se font par *sockets*. Le SGBD peut être n'importe où.

1. La plupart des navigateurs ne peuvent fonctionner qu'avec ce type d'applette. Seuls les dernières générations acceptent d'autres types d'applettes.

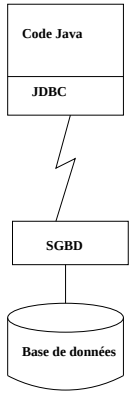

FIG. 6.1 – *Modèle 2-tiers*

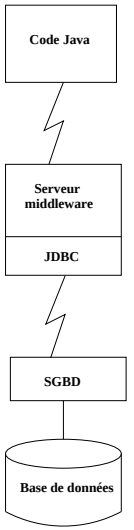

FIG. 6.2 – *Modèle 3-tiers*

De même, avec un driver de type 4, le serveur Web doit être sur la même machine qu le SGBD.

En fait certain SGBD (Oracle version 8 mais pas la version 7, par exemple) offrent possibilité de placer sur la machine du serveur Web un programme qui relaie les ordr SQL vers des serveurs du SGBD, qui peuvent donc se trouver sur une autre machine.

Une autre solution souple pour accèder à une base de données à travers le Web e d'utiliser des *servlets* (programmes Java qui travaillent directement avec le serveur Web qui ne sont pas des applets mais des applications Java et qui n'ont pas les contraintes d sécurité des applets. Ces servlets peuvent générer des pages Web (comme les programm CGI) qui contiennent des données de la base récupérées grâce à JDBC. Les servlets so utilisables avec la plupart des serveurs Web.

6.2 Modèles de connexion à une base de données

Modèle 2-tiers Dans ce modèle, une application Java ou une applette utilise JDB pour parler directement avec le SGBD qui gère la base de données.

Modèle 3-tiers Dans ce modèle, un serveur "middleware" est l'interlocuteur direct d code Java client. C'est ce serveur qui échange des données avec le SGBD.

Ce serveur n'est pas nécessairement écrit en Java. Si c'est le cas, il utilisera le plus souvent JDBC pour l'interface avec le SGBD.

Cette solution a de nombreux avantages :

- Le serveur peut ajouter un niveau de sécurité ;
- Il permet de centraliser des demandes venant de différents types de programmes (navigateurs HTML, programmes écrits en différents langages,...) vers différents types de stockages de données ;
- Les échanges entre les applications et le serveur peuvent avoir plusieurs supports (sockets, RMI Java, CORBA, HTML,...) ;
- Il permet plus de souplesse pour l'emplacement du serveur Web et du SGBD pour des accès par une applette "*unsecure*" : ils peuvent se trouver sur des machines différentes. Il suffit que le serveur Web et le serveur "*middleware*" se trouvent sur la même machine.

Cependant ce type de serveur est bien souvent assez coûteux et il n'existe que des solutions "propriétaires" qui obligent les applications à utiliser les protocoles du serveur, au détriment de la portabilité.

6.3 Utilisation pratique de JDBC

Les exemples pratiques donnés dans ce cours utilisent une connexion avec une base de données Oracle. Ils utilisent un driver de type 4 fourni (gratuitement) par Oracle.

Pour travailler avec JDBC et un certain driver, on doit :

1. Ajouter le chemin des classes qui implémentent JDBC dans la variable `CLASSPATH`. Pour notre cas :

   ```
   ORACLE_HOME=/net4/oracle
   CLASSPATH=$CLASSPATH:$ORACLE_HOME/jdbc/lib/classes111.zip
   ```

2. Le plus souvent on importera le package `java.sql` pour éviter de préfixer ses éléments dans le programme :

   ```
   import java.sql.*;
   ```

3. Charger en mémoire la classe du driver. Pour celui que nous allons utiliser :

   ```
   Class.forName("oracle.jdbc.driver.OracleDriver");
   ```


6.4 Étapes du travail avec une base de données ave JDBC

Un des drivers précédemment enregistrés doit permettre d'accéder à la base. Les étap du travail avec la base sont :

1. Ouvrir une connexion (`Connection`)
2. Créer des instructions SQL (`Statement`, `PreparedStatement` ou `CallableStatemer`
3. Lancer l'exécution de ces instructions : interroger ou modifier la base (`executeQuery` `executeUpdate()` ou `execute()`), ou lancer tout autre ordre SQL (`execute()`)
4. Fermer la connexion (`close()`).

Ces étapes sont détaillées dans les sections suivantes.

6.5 Classes et interfaces de JDBC

Interfaces	Description
Driver	renvoie une "Connection" utilisée par le "DriverManager"
Connection	connexion à une base
Statement	lié à un ordre SQL
PreparedStatement	lié à un ordre SQL paramétré
CallableStatement	lié à une procédure stockée sur le serveur
ResultSet	lignes récupérées par un ordre SELECT
ResultSetMetaData	description des lignes récupérées par un SELECT
DatabaseMetaData	informations sur la base de données

Classes	Description
DriverManager	gère les drivers, lance les connexions aux bases
Date	date SQL
Time	heures, minutes, secondes SQL
TimeStamp	comme Time avec une grande précision
Types	constantes pour les types SQL
DriverPropertyInfo	informations utilisées pour la connexion (pas utilisé pour la programmation ordinaire)

Exceptions	Description
SQLException	erreurs SQL
SQLWarning	avertissements SQL
DataTruncation	avertit quand une valeur est tronquée

6.6 Connexion à une base de données

Une connexion à un SGBD est représentée par une instance de la classe implémentant l'interface `Connection`. Dans la suite de ce cours on dira, en raccourci, "par une instance de la classe `Connection`" et on utilisera le même type de raccourci pour les autres classes qui implémentent les autres interfaces de JDBC.

On obtient une telle instance de `Connection` en retour de la méthode *static* `getConnection()` de la classe `DriverManager`. On doit passer à cette méthode l'URL de la base de données. En fait, le gestionnaire de driver essaye tous les drivers qui se sont enregistrés (au moment de leur chargement en mémoire par "Class.forName()") jusqu'à ce qu'il trouve un driver qui peut se connecter à la base.

Une URL pour une base de données est de la forme :

jdbc:*sous-protocole*:*base de donnée*

Par exemple, si on utilise le pont JDBC-ODBC, on pourrait avoir l'URL "`jdbc:odbc:base1`".

En fait la syntaxe de l'URL dépend du driver utilisé. Il faut se reporter à la documentation du driver.

Pour notre exemple avec Oracle, on aura, si `erato` est la machine qui héberge la base, 1521 est le port sur lequel le SGBD écoute les requêtes et MINFO est le nom de la base Oracle :

```
static final String url = "jdbc:oracle:thin:@erato:1521:MINFO";
conn = DriverManager.getConnection(url, "toto", "mdptoto");
```

6.7 Transactions

Quand on a ouvert une connexion, On peut positionner certaines options si elles sont disponibles avec le SGBD utilisé. La plus fréquente est l'"auto commit". Par exemple, on peut indiquer que l'on ne veut pas un *commit* automatique après chaque modification de la base par :

```
conn.setAutoCommit(false);
```

Par défaut la connexion est en "*auto commit*".

Si on n'a pas choisi l'*auto commit*, on valide ou annule la transaction en cours avec les méthodes `commit()` et `rollback` de la classe `Connection`.

6.8 Instruction SQL simple

Les ordres SQL simples sont représentés par des instances des classes qui implémentent l'interface `Statement`. Pour lancer un ordre SQL il faut commencer par créer une instance d'une telle classe (dont l'implémentation est fournie avec le driver utilisé pour accéder à la base particulière avec laquelle on travaille).

La méthode à appeler est différente suivant la nature de l'ordre SQL : consultation ("`executeQuery()`") ou modification des données ou autre ordre SQL ("`executeUpdate()`"). Dans certains cas on ne peut connaître qu'à l'exécution la nature de l'ordre SQL à exécuter

("`execute()`") ; cette dernière méthode doit aussi être employée dans certains cas particu liers pour lesquels l'ordre SQL renvoie plusieurs résultats (voir ci-après la section 6.10 su les procédures stockées).

6.8.1 Consultation des données (SELECT)

Une requête SELECT est associée à une instance d'une classe qui implémente l'interfac `Statement` en la passant en paramètre de la méthode `executeQuery(String)` (de la class `Statement`).

La méthode `executeQuery()` renvoie une instance de la classe `ResultSet`. Cette in tance est une représentation des lignes renvoyées par le SELECT, que l'on parcourt par méthode `next()`.

Exemple 6.1

```
Statement stmt1 = conn.createStatement();
// rset contient les lignes renvoyees par le select
ResultSet rset = stmt1.executeQuery("select NOME from EMP");
// On récupère chaque ligne une à une.
// getString(1) récupère la 1ère colonne (voir sect. suivante)
while (rset.next())
  System.out.println (rset.getString(1));
stmt1.close();
```

Récupération des données

La classe `ResultSet` fournit des méthodes "getXXX(int numéroColonne)" ou "getXXX nomColonne)" pour récupérer dans le code Java les valeurs des colonnes des lignes renvoyé par le SELECT. Le "XXX" de getXXX() est le nom du type Java de la valeur renvoyée pa la méthode. Par exemple, `getString()` renvoie une instance de la classe `String`.

C'est la responsabilité du programmeur de choisir les bonnes méthode getXXX() pou récupérer les données de la base. Une certaine tolérance est prévue mais il est préférab de choisir la méthode getXXX() donnée dans le tableau ci-dessous. Si la conversion e impossible, la méthode getXXX() lance une `SQLException`.

Oracle n'ayant qu'un seul type numérique, on peut utiliser les méthodes `getShort(` `getInt()`, `getLong()`, `getFloat()`, `getDouble()` pour récupérer des valeurs numériques l'on est sûr que les données s'adaptent au format. Si le nombre a une très grande précisio on peut utiliser la classe `java.math.BigDecimal` qui permet une précision aussi grand que l'on veut.

Méthodes Java pour récupérer les différents types SQL Oracle :

Type SQL	Méthode Java conseillée
CHAR(n)	getString()
VARCHAR2	getString()
NUMBER	getXXX() où XXX est un type numérique Java adapté
DATE	getDate()

Les valeurs des types de données de très grande dimension (par exemple associées à des images ; elles ne sont pas étudiées dans ce cours) peuvent être recupérées par des "Streams" Java. Des types spécialement adaptés aux fichiers binaires de très grande dimension sont prévus dans la future (au mois de mars 1998) norme JDBC 2.0.

Les méthodes `getObject()` retournent une valeur du type Java correspondant au type SQL de la colonne spécifiée.

Remarques 6.1

(a) Quand on récupère un "CHAR(n)" dans un objet de la classe `String`, la valeur est complétée par des espaces.

(b) La méthode `getDate()` renvoie un objet de la classe `java.sql.Date`. Cette classe hérite de la classe `java.util.Date`. Attention, de nombreuses méthodes de cette dernière classe sont "*deprecated*" dans JDK 1.1. Un objet de la classe `java.sql.Date` correspond à un objet de la classe `java.util.Date` dont les heures, minutes et secondes sont mises à 0.

(c) Oracle n'utilise que le type numérique NUMBER, mais dans la norme SQL-2 on trouve aussi d'autres types numériques tels que TINYINT, SMALLINT, INTEGER, REAL, FLOAT, DOUBLE qui correspondent respectivement aux types Java byte, short, int, float, double. Si vous travaillez avec un autre SGBD qu'Oracle et si vous voulez une liste complète des correspondances, consultez le guide JDBC dont l'adresse est donnée au début de ce chapitre.

Valeur NULL

Pour repérer les valeurs NULL de la base de données, on doit utiliser la méthode `wasNull()` (qui renvoie un booléen). Il faut d'abord appeler la méthode getXXX(). Voici un exemple :

```
Statement stmt1 = conn.createStatement();
ResultSet rset = stmt1.executeQuery("select NOME, COMM from EMP");
Float commission
while (rset.next()) {
  nom = rset.getString(1);
  commission = rset.getFloat(2);
  if (rset.wasNull())
    System.out.println(nom + " n'a pas de commission");
  else
```



```
    System.out.println(nom + " a " + commission + "F de commission");
}
```

6.8.2 Modification des données (INSERT, UPDATE, DELETE)

Pour lancer un ordre INSERT, UPDATE ou DELETE, il faut utiliser la méthod `executeUpdate(String)` à la place de la méthode `executeQuery(String)`. Cette méthod renvoie le nombre de lignes modifiées par la requête SQL.

Exemple 6.2

```
Statement stmt2 = conn.createStatement();
String ville = new String("NICE");
int nbLignesModifiees = stmt.executeUpdate(
  "INSERT INTO dept(DEPT, NOMD, LIEU)
   VALUES(70, 'DIRECTION', '"
   + ville + "')");
stmt2.close();
```

De plus, `executeUpdate()` permet de lancer un ordre SQL qui ne renvoie rien comm les ordres DDL tels que `CREATE TABLE`.

6.8.3 Ordre SQL quelconque

Dans certains cas on ne sait pas à l'avance si l'ordre SQL modifiera les données o effectuera une simple consultation.

Voici un algorithme pour retrouver le type d'un ordre SQL non connu à l'avance :

1. La méthode `execute(`*ordreSQL*`)` retourne un booléen. Si le résultat est `true`, c'e que *ordreSQL* était une requête SELECT. On utilise alors un `Resultset` pour récu pérer les lignes.

2. Sinon, on lance la méthode `getUpdateCount()` qui renvoie un entier. Si cet enti est strictement positif, l'ordre était un ordre DML (modification de lignes).

3. Sinon, si cet entier est égal à 0, il s'agissait soit d'un ordre DML qui n'a modifié aucu ligne, soit d'un ordre DDL (définition des données tel qu'un CREATE TABLE).

En fait, la méthode `getUpdateCount()` suffit pour déterminer le type car elle renvo l'entier -1 si l'ordre est un ordre SELECT.

6.9 Instruction SQL paramétrée

Chaque instruction SQL envoyée au SGBD est analysée par le SGBD pour trouver meilleure façon de l'exécuter. Avec la plupart des SGBD (dont Oracle) on peut s'arrang

pour que le SGBD n'analyse qu'une seule fois une requête si elle est exécutée un grand nombre de fois avec des valeurs différentes pour certaines valeurs considérées comme des paramètres.

JDBC a prévu de profiter de ce type de fonctionnalité par l'utilisation de requêtes paramétrées ou de procédures stockées (voir section suivante).

Les instructions paramétrées peuvent avoir des paramètres qui peuvent recevoir des valeurs données par le programme Java. Elles n'ont d'intérêt que si elles sont exécutées plusieurs fois dans le programme avec des paramètres différents. La stratégie d'exécution d'une requête ne dépendant pas de ces paramètres, la plupart des SGBD ne choisiront cette stratégie qu'une fois et exécuteront donc plus vite les multiples exécutions de la requête.

Les procédures paramétrées sont associées aux instances des classes qui implémentent l'interface `PreparedStatement` qui hérite de l'interface `Statement`.

Les valeurs des paramètres sont données par les méthodes "setXXX(n, valeur)". On n'a pas la même flexibilité que pour les méthodes "getXXX()" pour les correspondances entre types Java et SQL. Le driver JDBC traduit les types Java dans les types SQL donnés par le tableau suivant :

Type Java	Type SQL	Méthode
String	VARCHAR	setString()
Numérique adapté	NUMBER	correspondant au type numérique
java.math.BigDecimal	NUMBER	setBigDecimal()
java.sql.Date	DATE	setDate()

Exemple 6.3

```
PreparedStatement pstmt = conn.prepareStatement(
  "UPDATE emp SET sal = ?
   WHERE name = ?");
int nbLignesModifiees;
for (int i=0; i<10; i++) {
  psmt.setFloat(1, salaire[i]);
  psmt.setString(2, nom[i]);
  nbLignesModifiees = psmt.executeUpdate();
}
```

Pour passer la valeur NULL on peut utiliser la méthode `setNull()` ou passer la valeur Java "`null`" si la méthode "setXXX()" attend un objet en paramètre comme, par exemple, la méthode `setString()`, mais pas comme la méthode `setFloat()`.

On peut imposer la conversion d'une valeur Java en un type SQL particulier en utilisant la méthode `setObject(int n, Object x, int typeSQL)`.

Le types SQL est indiquer par une constante entière définie dans la classe `java.sql.Types`. Par exemple, `Types.VARCHAR`. Si on omet le type SQL, le driver traduit le type Java dans le type SQL senblable au tableau ci-dessus associé à "setXXX()" mais les types numériques Java sont les classes correspondant aux types primitifs. Par exemple, `Integer` au lieu de `int`.

De même que pour les méthodes "getXXX()" on peut utiliser des "streams" Java pou passer des valeurs de très grandes dimensions.

6.10 Procédure stockée

L'appel de procédures stockées n'est pas standardisée dans les différents SGBD. JDB appelle les procédures stockées avec une syntaxe spéciale uniforme pour tous les SGBI Ce sera la tâche de l'implémentation de JDBC particulière à un SGBD de la traduire ave la syntaxe appropriée.

`{ [? = ]call` *nom-procédure* `[(?,?,...)] }`

Les procédures stockées sont associées aux instances des classes qui implémentent l'i terface `CallableStatement` qui hérite de l'interface `PreparedStatement`.

Au contraire des paramètres des requêtes paramétrées qui n'ont que des paramètr "IN", les procédures stockées peuvent aussi avoir des paramètres "IN/OUT" ou "OUT". C'e au programmeur d'utiliser les méthodes (de la classe `CallableStatement`) "setXXX()" c "getXXX()" qui conviennent à la définition de la procédure stockée.

Le programme Java doit indiquer le type SQL de tous les paramètres "IN/OUT" "OUT" par la méthode `registerOutParameter()`. On utilise pour cela les constantes en tières de la classe `java.sql.Types`.

Java	Types JDBC	SQL Oracle
String	Types.CHAR	CHAR
String	Types.VARCHAR	VARCHAR2
type numérique Java adapté	Types.DECIMAL	NUMBER
java.sql.date	Types.DATE	DATE

Remarque 6.2

Pour la portabilité, il est recommandé de récupérer les valeurs des "Resultset" avan de récupérer les paramètres "OUT" et "INOUT".

Avec Oracle le problème ne se pose pas car une procédure stockée est écrite dan le langage PL/SQL qui ne peut comporter de consultation sans ranger les valeu trouvées dans des variables PL/SQL et on ne peut donc avoir à traiter de "Resultset

Exemple 6.4

Voici une procédure stockée Oracle qui prend en paramètre un numéro de départ ment et un pourcentage, augmente tous les salaires des employés de ce départemen de ce pourcentage et renvoie dans un paramètre le coût total pour l'entreprise.

```
create or replace procedure augmentation
  (unDept in integer, pourcentage in number,
   cout out number) is
begin
  update emp
```

```
  set sal = sal * (1 + pourcentage / 100)
  where dept = unDept;

  select sum(sal) * pourcentage / 100
  into cout
  from emp
  where dept = unDept;
end;
```

Exemple 6.5

Voici un exemple d'utilisation de la procédure stockée de l'exemple 6.4 :

```
CallableStatement csmt =
  conn.prepareCall("{ call augmentation(?, ?, ?) }");
// 2 chiffres après la virgule
csmt.registerOutParameter(3, Types.DECIMAL, 2);
// Augmentation de 2,5 % des salaires du dept 10
csmt.setInt(1, 10);
csmt.setFloat(2, 2.5);
csmt.executeQuery();
int x = csmt.getFloat(3);
System.out.println("Cout total de l'augmentation : " + x);
```

Procédure stockée comprenant plusieurs ordres SQL

Quand une procédure stockée comporte plus d'un ordre SQL, il faut utiliser la méthode `execute()` pour exécuter le premier ordre SQL et des méthodes `getMoreResults()` pour récupérer le résultat suivant d'une instruction SQL.

Exemple 6.6

Voici un exemple schématique de traitement de "`Statement`" qui renvoie plusieurs résultats :

```
stmt.execute(ordreSQL);
while (true) {
  int nbLignes = stmt.getUpdateCount();
  if (nbLignes > 0) {      // ordre DML
    ....
  }
  else if (nbLignes = 0) { // ordre DML sans ligne traitée
                           // ou autre ordre SQL non SELECT
    ....
  }
  else {                   // ordre SELECT
```



```
                           // ou plus de résultat
    ResultSet rs = stmt.getResultSet();
    if (rs != null) { // ordre SQL
      ....
    }
    else              // plus de résultat à traiter
      break;
  }
  stmt.getMoreResults(); // récupère le résultat suivant
}
```

6.11 Syntaxe spéciale SQL ("*SQL Escape Syntax*")

Ce type de syntaxe "{*mot-clé paramètre...*}" est utilisée par JDBC pour définir ur syntaxe commune à tous les SGBD quand ils ont des syntaxes différentes. Une date e ainsi représentée par `{d 'yyyy-mm-dd'}`, par exemple, `{d '1998-02-18'}`.

De même de nombreuses fonctions peuvent être supportées par les différents driver Reportez-vous à leur documentation pour savoir lesquelles (ou utilisez les méthodes d `DataBaseMetaData`).

Exemples 6.7

(a) `{fn concat("debut", "fin")}`

(b) `{fn user()}`

Remarque 6.3

Ce type de syntaxe est aussi associé à la jointure externe qui a une syntaxe différen dans les différents SGBD. Oracle n'est d'ailleurs pas à la norme SQL-2 pour la jointu externe. De plus, le driver fourni par Oracle ne supporte pas ce type de syntaxe. O doit donc utiliser la syntaxe particulière à Oracle "(+)" et on perd donc en portabilit

6.12 Les "Meta données"

JDBC permet de récupérer des informations sur le type de données que l'on vie de récupérer par un SELECT (interface `ResultSetMetaData`), mais aussi sur la base d données elle-même (interface `DataBaseMetaData`).

Les données que l'on peut récupérer dépendent du SGBD avec lequel on travaill surtout pour `DataBaseMetaData`. Dans la version actuelle d'Access, on ne peut ainsi pa obtenir la liste de toutes les tables d'une base de données.

Dans ce cours d'introduction à JDBC, nous ne détaillerons pas les nombreuses possib lités offertes par ces interfaces ; nous donnerons seulement des exemples d'utilisation.

6.12.1 ResultSetMetaData

L'exemple suivant montre comment obtenir quelques informations sur les colonnes renvoyées par un SELECT :

```
ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
ResultSetMetaData rsmd = rs.getMetaData();
int nbColonnes = rsmd.getColumnCount();
for (int i = 1; i <= nbColonnes; i++) {
  // Les colonnes sont numérotées à partir de 1 (et pas de 0)
  String typeColonne = rsmd.getColumnTypeName(i);
  String nomColonne = rsmd.getColumnName(i);
  System.out.println("La colonne " + i + " de nom "
                     + nomColonne + " est de type " + typeColonne);
}
```

6.12.2 DatabaseMetaData

Voici un extrait de programme, qui ajoute dans une liste (classe List de l'AWT) les noms des tables et vues disponibles dans une base de données :

```
private DatabaseMetaData metaData;
private List listTables = new java.awt.List(10);
.... {
  metaData = conn.getMetaData();
  String[] types = { "TABLE", "VIEW" };
  // % : joker SQL pour désigner tous les noms
  ResultSet rs = metaData.getTables(null, null, "%", types);
  String nomTables;
  while (rs.next()) {
    // Le nom des tables est la 3ème colonne du ResultSet
    nomTable = rs.getString(3);
    listTables.add(nomTable);
  }
...
}
```


Chapitre 7

Langage C et Pro*C d'Oracle

Pour obtenir un programme exécutable à partir des programmes sources qui contienner des instructions en Langage C et des requêtes SQL, il faut utiliser des outils annexes fourn le plus souvent par la société qui commercialise le SGBD.

Avec Oracle, le programmeur a deux solutions pour insérer des ordres SQL dans u programme écrit en Langage C :

- l'utilisation dans ses programmes en langage C de fonctions C contenues par ur bibliothèque écrite par Oracle. Le programmeur construit les ordres SQL dans programme et les passe en paramètres de ces fonctions. Le programme doit être lié la bibliothèque pour obtenir un programme exécutable.

- à un plus haut niveau qui simplifie la tâche du programmeur, celui-ci insert dans s programmes des pseudo-instructions qui ne sont pas du langage C. Un programm fourni par Oracle, appelé précompilateur, remplace alors ces pseudo-instructions pa des instructions en langage C ou par des appels aux fonctions C citées dans la premiè solution. Le programme peut alors être compilé et lié aux bibliothèques fournies pa Oracle.

  Le module "Pro*C" d'Oracle fournit les outils et les bibliothèques nécessaires.

Des exemples de programmes sont donnés à la section 7.5. Les sections 7.1 et 7 donnent quelques précisons pour expliquer ces programmes. La section 7.4 montre comme construire les programmes exécutables et indique l'environnement nécessaire.

7.1 Variables hôtes

Les échanges de données entre les instructions en langage C et les ordres SQL se fo grâce à des variables "hôtes" (du langage C) qui sont déclarées dans une zone spécia du programme encadrée par les deux instructions EXEC SQL BEGIN DECLARE SECTION; EXEC SQL END DECLARE SECTION;. Ces variables sont préfixees par ":" dans les ordr SQL (mais pas dans les instructions en langage C).

VARCHAR est un pseudo-type C qui sert à conserver des valeurs SQL de type VARCHAR2. Par exemple, VARCHAR uid[20] sera remplacé par le précompilateur par la structure C

```
struct {
  unsigned short len;
  unsigned char arr[20];
} uid
```

Il ne faut pas oublier de donner la bonne valeur à "len" lorsque l'on récupère une valeur à partir du langage C et qu'on l'utilise ensuite dans Oracle. En sens inverse, il faut ajouter le caractère "\0" en fin de valeur dans le tableau "arr" lorsque l'on veut utiliser en langage C une valeur obtenue par Oracle.

À chaque variable hôte associée à une colonne d'une table de la base on peut lier une *variable indicatrice* de type "short". Cette variable permettra de repérer et de donner une valeur 'NULL' ou tronquée (voir figure 7.3).

7.2 Zone SQLCA

Pour récupérer les codes et messages d'erreur et d'avertissement de SQL, on doit insérer dans le programme l'instruction EXEC SQL INCLUDE SQLCA; qui inclut la zone de communication "SQLCA". Cette zone contient (entre autres)

- le code retour de la derniere commande SQL exécutée (contenu dans la variable SQLCODE). Il est nul si la commande s'est bien passée, négatif si une erreur s'est produite et positive si la commande a provoqué un message d'avertissement.
- le nombre de lignes traitées par la dernière commande SQL (contenu dans SQLERRD[2]).

L'instruction EXEC SQL WHENEVER utilise cette zone pour traiter les erreurs.

7.3 Curseur

Lorsqu'un ordre SELECT ramène plus d'une seule ligne, il est nécessaire d'utiliser un curseur pour récupérer ces lignes dans le programme en langage C.

Un curseur est une zone temporaire dans laquelle sont stockées les lignes ramenées par le SELECT. On ne peut lire cette zone que séquentiellement. La ligne courante est désignée par le mot clé "CURRENT", ce qui permet eventuellement de la modifier par un ordre UPDATE ou de la supprimer par un ordre DELETE.

** Donner un exemple en utilisant un curseur sans tableau : ajouter 10 **

Oracle a ajouté la possibilité de récupérer directement dans un tableau du langage C les lignes renvoyées par un SELECT (un exemple est donné dans la figure 7.6).

Les étapes de l'utilisation d'un curseur sont les suivantes :

1. déclaration et définition du curseur par l'ordre DECLARE. Par exemple,

   ```
   EXEC SQL DECLARE C1 CURSOR FOR
     SELECT MATR, NOME, SAL FROM EMP;
   ```


2. ouverture par l'ordre OPEN, ce qui exécute l'ordre SELECT associé au curseur . Pa exemple,

   ```
   EXEC SQL OPEN C1;
   ```

3. récupèration des lignes dans les variables C par l'ordre FETCH. Par exemple,

   ```
   EXEC SQL FETCH C1 INTO :matricule, :nom, :salaire;
   ```

4. fermeture du curseur quand on n'en a plus besoin par l'ordre CLOSE (EXEC SQ COMMIT WORK RELEASE ferme automatiquement tous les curseurs). Par exemp

   ```
   EXEC SQL CLOSE C1;
   ```

7.4 Pré-compilation, compilation et exécution de pro grammes écrits avec PRO*C

La principale difficulté est de trouver les nombreuses bibliothèques qui sont nécessair à l'éditeur de lien pour créer un exécutable. Il faut pour cela se créer un fichier "make".

L'environnement de travail doit comporter la variable ORACLE_HOME car elle est utilis dans le fichier "make" (et par les outils Oracle).

Avant de lancer l'exécution il faut donner une valeur correcte à la variable d'enviro nement LD_LIBRARY_PATH pour que les liens dynamiques puissent être établis.

Voici un exemple de petit shellscript "deboracle" à lancer sous Unix (par ". deboracl pour avoir un environnement correct quand on travaille avec Oracle et une base de donné nommée "MINFO" :

```
ORACLE_SID=MINFO
TWO_TASK=MINFO  # si la base est sur une autre machine
ORACLE_HOME=/net4/oracle
PATH=$PATH:$ORACLE_HOME/bin:$ORACLE_HOME/orainst
export ORACLE_SID TWO_TASK ORACLE_HOME PATH
LD_LIBRARY_PATH=/usr/X11R6.1/lib/:/usr/dt/lib:/net4/oracle/lib
export LD_LIBRARY_PATH
```

7.5 Exemples

Des exemples commentés sont donnés dans les figures 7.1 à 7.6.

La figure 7.7 donne les étapes pour obtenir un programme exécutable.

```
# include <stdio.h>

/* Declaration des variables hotes */
EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR uid[20];
  VARCHAR pwd[20];
  int matricule;
  VARCHAR nom[15];
  VARCHAR poste[10];
  float salaire;
  int nodept;
EXEC SQL END DECLARE SECTION;

/* Inclusion de la zone de communication SQLCA */
EXEC SQL INCLUDE SQLCA;

main() {
  int sret;
  /* Entree dans Oracle */
  printf("Nom d'utilisateur : ");
  sret = scanf("%s", uid.arr);
  uid.len = strlen(uid.arr);
  printf("Mot de passe : ");
  sret = scanf("%s", pwd.arr);
  pwd.len = strlen(pwd.arr);
  EXEC SQL WHENEVER SQLERROR GOTO mauvaisnom;
  EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
  printf("\nConnexion a Oracle sous le nom %s\n", uid.arr);
  EXEC SQL WHENEVER SQLERROR GOTO erreur;
```

FIG. 7.1 – *Insertion de nouvelles lignes dans la table EMP (début)*


```
  while (1) {
    printf("Matricule de l'employe (0 pour sortir) : ");
    sret = scanf("%d", &matricule);
    if (sret == EOF || sret == 0 || matricule == 0)
      break;
    printf("Nom de l'employe : ");
    scanf("%s", &nom.arr);
    nom.len = strlen(nom.arr);
    printf("Poste de l'employe : ");
    scanf("%s", poste.arr);
    poste.len = strlen(poste.arr);
    printf("Salaire de l'employe : ");
    scanf("%f", &salaire);
    printf("Numero du departement de l'employe : ");
    scanf("%d", &nodept);
    EXEC SQL INSERT INTO EMP
      (matr, nome, poste, sal, dept)
      VALUES (:matricule, :nom, :poste, :salaire, :nodept);
    EXEC SQL COMMIT WORK;
    printf("Employe %s ajoute\n", nom.arr);
  }

  /* Quitter Oracle */
  EXEC SQL COMMIT WORK RELEASE;
  exit(0);

mauvaisnom:
  printf("Mot de passe incorrect");
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);

erreur:
  printf("\n%.70s\n", sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);}
```

FIG. 7.2 – *Insertion de nouvelles lignes dans la table EMP (suite)*

```
# include <stdio.h>
EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR uid[20], pwd[20], nom[15];
  float salaire, commission;
  /* Variables indicatrices */
  short salairei, commissioni;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;

main() {
  int sret;
  /* Entree dans Oracle */
  .........
  /* On recupere la ligne */
  printf("Nom de l'employe : ");
  scanf("%s", &nom.arr);
  nom.len = strlen(nom.arr);
  EXEC SQL WHENEVER NOT FOUND pastrouve;
  EXEC SQL SELECT SAL, COMM INTO :salaire, :commission:commissioni
    FROM EMP
    WHERE NOME = :nom;

  /* Affichage de la ligne */
  if (commissioni == -1)
    /* Valeur NULL */
    printf("Employe %s, salaire %6.2f\n", nom.arr, salaire);
  else
    printf("Employe %s, salaire %6.2f\n commission %6.2f",
          nom.arr, salaire, commission);
```

FIG. 7.3 – *Mise à jour d'une ligne dans la table EMP (début)*


```
  /* Entree des nouvelles valeurs */
  printf("Nouveau salaire de l'employe : ");
  sret = scanf("%f", &salaire);
  if (sret == EOF || sret == 0)
    salairei = -1; /* valeur NULL */
  printf("Nouvelle commission de l'employe : ");
  sret = scanf("%f", &commission);
  if (sret == EOF || sret == 0)
    commissioni = -1; /* valeur NULL */
  else
    commissioni = 1;
  EXEC SQL UPDATE EMP
    SET SAL = :salaire:salairei,
    COMM = :commission:commissioni
    WHERE NOME = :nom;
  printf("Employe %s mis a jour", nom.arr);

  /* Quitter Oracle */
  EXEC SQL COMMIT WORK RELEASE;
  exit(0);

pastrouve:
  printf("Pas d'employe %s dans la base", nom.arr);
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}
```

FIG. 7.4 – *Mise à jour d'une ligne dans la table EMP (suite)*

```
# include <stdio.h>
void afficher_lignes(int);
EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR uid[20], pwd[20], nom[10][15];
  int matricule[10];
  float salaire[10];
  short salairei[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;

main() {
  int sret;
  long nb_lignes;
  /* Entree dans Oracle */
  ............
  EXEC SQL DECLARE C1 CURSOR FOR
    SELECT MATR, NOME, SAL FROM EMP;
  EXEC SQL OPEN C1;
  EXEC SQL WHENEVER NOT FOUND GOTO finboucle;
  nb_lignes = 0;
  while (1) {
    EXEC SQL FETCH C1 INTO :matricule, :nom, :salaire;
    /* sqlca.sqlerrd[2] est le nombre de lignes ramenees
       depuis l'ouverture du curseur */
    afficher_lignes(sqlca.sqlerrd[2] - nb_lignes);
    nb_lignes = sqlca.sqlerrd[2];
  }
finboucle:
  if (sqlca.sqlerrd[2] > nb_lignes)
    /* Dernieres lignes a afficher */
    afficher_lignes(sqlca.sqlerrd[2] - nb_lignes);

  /* Quitter Oracle */
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL COMMIT WORK RELEASE;

  exit(0);
}
```

FIG. 7.5 – *Utilisation d'un "curseur" pour ramener plusieurs lignes de la table EMP (par paquets de 10) (début)*


```
void afficher_lignes(n)
int n;
{
  int i;
  printf("\nMatricule Nom        Salaire\n");
  printf(  "--------- ---------- --------\n");
  for (i=0; i<n; i++)
    printf("%-10d%-11s%9.2f\n",
   matricule[i], nom[i].arr, salaire[i]);
}
```

FIG. 7.6 – *Utilisation d'un "curseur" pour ramener plusieurs lignes de la table EMP (p paquets de 10) (suite)*

```
~/oracle> make -f proc.mk ex3
proc ex3.pc

Pro*C/C++: Release 2.2.2.0.0 - Production on Thu Feb 20 11:43:07 1997

Copyright (c) Oracle Corporation 1979, 1994.  All rights reserved.

System default option values taken from: /net4/oracle/precomp/admin/pcscf

gcc -DSLXMX_ENABLE -DSLTS_ENABLE -D_REENTRANT -I. -I/precomp/public -c ex
gcc -o ex3 ex3.o -LRACLE_HOME/lib -lclntsh -lsql -lncr -lsqlnet -lclient\
 -lcommon -lgeneric  -lepc -lc3v6 -lcore3 -lnlsrtl3 -lnlsrtl3\
`cat /net4/oracle/rdbms/lib/sysliblist` -lm -lthread
```

FIG. 7.7 – *Exemple de traitement pour obtenir un programme exécutable*

Index