Université de Nice Sophia-Antipolis

Langage SQL

version 2.3

Richard Grin

6 novembre 2000

Table des matières

Présentation du polycopié

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

Les exemples s'appuient sur la version du langage SQL implantée dans le SGBD Oracle, version 7. Les exemples et remarques sur la version 8i n'ont pas été testées (cette version n'est toujours pas installée...).

Les remarques et les corrections d'erreurs peuvent être envoyées par courrier électronique 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 structuré".

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

C'est à la fois:

- un langage d'interrogation de la base (ordre SELECT)
- un langage de manipulation des données (LMD; 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, Informix, Ingres, RDB,... Chacun de ces SGBDR a cependant sa propre variante du langage. Ce support de cours présente un noyau de commandes disponibles sur l'ensemble de ces SGBDR, et leur implantation dans Oracle Version 7.

1.2 Normes SQL

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

La norme actuelle SQL-2 (appelée aussi SQL-92) date de 1992. Elle est acceptée par tous mais les SGBD relationnels qui dominent actuellement le marché (en particulier Oracle) 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)

SQL-3 est en cours d'homologation.

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 SQL-FORMS facilite grandement la réalisation des 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.

Sous Unix, la variable d'environnement `ORACLE_SID` donne le nom de la base sur laquelle on se connecte.

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: utilisateurs, tables, colonnes, index, fonctions, etc.

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

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

1.5.2 Tables

Les relations (d'un schéma relationnel; voir polycopié du cours sur le modèle relationnel) 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 cet 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 schéma (voir 4.1).

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

Exemple 1.1

Table DEPT des départements:

DEPT	NOMD	LIEU
10	FINANCES	PARIS
20	RECHERCHE	GRENOBLE
30	VENTE	LYON
40	FABRICATION	ROUEN

1.5.3 Colonnes

Les données contenues dans une colonne doivent être toutes d'un même type de données. 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 Types numériques

Types numériques SQL-2

Les types numériques de la norme SQL-2 sont:

- Nombres entiers: TINYINT (sur 1 octet, de 0 à 255), SMALLINT (sur 2 octets, de -32.768 à 32.767), INTEGER (sur 4 octets, de -2.147.483.648 à 2.147.483.647).
- Nombres décimaux avec un nombre fixe de décimales: NUMERIC, DECIMAL (la norme impose à NUMERIC d'être implanté avec exactement le nombre de décimales indiqué alors que l'implantation de DE-

CIMAL peut avoir plus de décimales) : DECIMAL(p, d) correspond à des nombres décimaux qui ont p chiffres significatifs et d chiffres après la virgule ; NUMERIC a la même syntaxe.

- Numériques non exacts à virgule flottante : REAL (simple précision, avec au moins 7 chiffres significatifs), DOUBLE PRECISION ou FLOAT (double précision, avec au moins 15 chiffres significatifs).

La définition des types non entiers dépend du SGBD (le nombre de chiffres significatifs varie). Reportez-vous au manuel du SGBD que vous utilisez pour plus de précisions.

Le type BIT permet de ranger une valeur booléenne (un bit) en SQL-2.

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. La valeur absolue du nombre doit être inférieure à 10^{128}. NUMBER est un nombre à virgule flottante (on ne précise pas le nombre de chifres après la virgule) qui peut avoir jusqu'à 38 chiffres significatifs.

L'insertion d'un nombre avec plus de chiffres que *taille_maxi* sera refusée (*taille_maxi* ne prend en compte ni le signe ni le point décimal). Les décimales seront éventuellement 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 décimales 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 représentant 1.2 x 10^{-8}).

1.6.2 Types chaîne de caractères

Types chaîne de caractères de SQL-2

Les constantes chaînes de caractères sont entourées par des apostrophes ('). Si la chaîne 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 ne contenant pas plus de 255 caractères (255 pour Oracle, cette longueur maximum varie suivant les SGBD).
  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 possible de stocker dans le champ. *longueur* doit être obligatoirement spécifiée. L'insertion d'une chaîne dont la longueur est supérieure à *longueur* sera refusée. Une chaîne plus courte que *longueur* sera complétée par des espaces (important pour les comparaisons de chaînes).
- le type `VARCHAR` pour les colonnes qui contiennent des chaînes de longueurs variables. Tous les SGBD ont une longueur maximum pour ces chaînes (plusieurs milliers de caractères).
  On déclare ces colonnes par :
  `VARCHAR(longueur)`
  longueur indique la longueur maximale des chaînes contenues dans la colonne

Types chaîne de caractères d'Oracle

Les types Oracle sont les types SQL-2 mais le type `VARCHAR` s'appelle `VARCHAR2` dans Oracle (la taille maximum est de 2000 caractères).

1.6.3 Types temporels

Types temporels SQL-2

Les types temporels de SQL-2 sont :

DATE réserve 2 chiffres pour le mois et le jour et 4 pour l'année ;

TIME pour les heures, minutes et secondes (les secondes peuvent comporter un certain nombre de décimales) ;

TIMESTAMP permet d'indiquer un moment précis par une date avec heures, minutes et secondes (6 chiffres après la virgule ; c'est-à-dire en microsecondes).

Types temporels d'Oracle

La déclaration de type date a le format suivant :

```
DATE
```

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

1.6.4 Types binaires

Ce type permet d'enregistrer des données telles que les images et les sons, de très grande taille et avec divers formats.

SQL-2 n'a pas normalisé ce type de données. Les différents SGBD fournissent un type pour ces données mais les noms varient : `LONG RAW` pour Oracle, mais `IMAGE` pour Sybase, `BYTE` pour Informix, etc.

Nous n'utiliserons pas ce type de données dans ce cours.

1.6.5 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 1.8 et 4.2.1pour 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    DECIMAL(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 TABLE, 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 de la colonne dans un ordre CREATE TABLE (pas possible dans un ordre ALTER TABLE).

sur une table : la contrainte porte sur une ou plusieurs colonnes. Elles se place au même niveau que les définition des colonnes dans un ordre CREATE TABLE ou ALTER 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 colonnes 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 valeurs 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 pour une contrainte sur une colonne) est une clé étrangère qui fait référence à la concaténation des colonnes *col1*, *col2*,... de la table *tableref* (contrainte d'intégrité référentielle). Si aucune colonne de *tableref* n'est indiquée, c'est la clé primaire de *tableref* qui est prise par défaut.

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

col1, *col2*,... doivent avoir la contrainte PRIMARY KEY ou UNIQUE. Ceci implique 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 *tableref* va entrainer automatiquement la suppression des lignes qui la référencent dans la table. Si cette option n'est pas indiquée, il est impossible de supprimer des lignes de *tableref* qui sont 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. Cette contrainte peut être une contrainte de colonne ou de table.

Des contraintes d'intégrité peuvent être ajoutées ou supprimées par la commande ALTER TABLE (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" qui permet de choisir si le moment où sont vérifiées les contraintes (immédiatement après chaque instruction ou seulement à la fin de la transaction).

1.8.2 Exemples de contraintes

Quelques contraintes sur des colonnes:

```
CREATE TABLE EMP (
  MATR NUMBER(5) CONSTRAINT KEMP PRIMARY KEY,
  NOME VARCHAR(10) 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 VARCHAR(10) 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(5) CONSTRAINT R_PROJET REFERENCES PROJET,
....,
CONSTRAINT PKPART PRIMARY KEY(MATR, CODEP))
```

Avec ALTER TABLE on peut ajouter, enlever ou modifier des contraintes de colonnes ou de tables :

```
ALTER TABLE EMP
  DROP CONSTRAINT NOM_UNIQUE
  ADD (CONSTRAINT SAL_MIN CHECK(SAL + NVL(COMM,0) > 5000))

ALTER TABLE EMP
  MODIFY NOME CONSTRAINT NOM_UNIQUE UNIQUE
```

On peut aussi provisoirement supprimer et ensuite rétablir une contrainte par les clauses DISABLE et ENABLE de ALTER TABLE:

```
ALTER TABLE EMP
  DISABLE CONSTRAINT NOM_UNIQUE UNIQUE
```

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 néanmoins à la plupart des interrogations courantes. Une étude plus complète sera vue au chapitre 3.

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

table est le nom de la table sur laquelle porte la sélection. *exp1, exp2,...* est la liste des expressions (colonnes, constantes,... ; voir 1.10) que l'on veut obtenir. Cette liste peut être "*", 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 "exp1 *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 fonctions.

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

Les priorités des opérateurs arithmétiques sont celles de l'arithmétique classique (`*` et `/`, puis `+` et `-`). Il est possible d'ajouter des parenthèses dans les expressions pour obtenir 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 informations contenues dans la base.

Il existe trois commandes SQL permettant d'effectuer les trois types de modification 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'exécution de la commande. Les modifications effectuées par les autres utilisateurs entre le début et 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 noms des colonnes pour lesquelles on donne une valeur. Cette liste est optionnelle. Si elle est omise, ORACLE prendra par défaut l'ensemble des colonnes de la table dans l'ordre où elles ont été données lors de la création de la table. Si une liste de colonnes est spécifiée, 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.

Le prédicat peut contenir des sous-interrogations (voir 3.6).

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'avaient les salaires au début de l'exécution de la commande UPDATE et que les modifications effectuées sur la base pendant l'exécution de cette commande ne seront pas prises en 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: cette clause est facultative; si elle n'est pas précisée, TOUTES LES LIGNES DE LA TABLE SONT SUPPRIMEES (heureusement qu'il existe ROLLBACK!).

Le prédicat peut contenir des sous-interrogations (voir 3.6).

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 indivisible: il faut effectuer ces modifications entièrement ou pas du

tout, sous peine de laisser la base dans un état incohérent.

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

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

L'utilisateur peut annuler (et terminer) la transaction en cours par la commande **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 fin anormale d'une tâche utilisateur: Oracle lance automatiquement un ROLLBACK des transactions 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).

Les instructions SQL sont atomiques: quand une instruction provoque une erreur (par exemple si une contrainte d'intégrité n'est pas vérifiée), toutes les modifications déjà effectuées par cette instruction sont annulées. Mais cette erreur ne provoque pas de rollback automatique de la transaction.

Chapitre 3

Interrogations

3.1 Syntaxe générale

L'ordre SELECT possède six clauses différentes, dont seules les deux premières sont obligatoires. Elles sont données ci-dessous, dans l'ordre dans lequel elles doivent apparaître, 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 être 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 caractères maximum donnés aux expressions. Chacun de ces noms est inséré derrière l'expression, séparé de cette dernière par un blanc ; il constituera le titre de la colonne dans l'affichage 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 les duplications : si, dans le résultat, plusieurs lignes sont identiques, une seule sera conservée.

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 à la place d'un nom de table.

Exemples 3.4

(a) Pour obtenir la liste des employés avec le pourcentage de leur salaire par rapport au total des salaires, il fallait auparavant utiliser une vue (voir 4.3). Il est maintenant 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)
```

(b) On peut même donner un synonyme comme nom de table au select :

```
select nome, sal, sal/total*100
```



```
from emp, (select dept, sum(sal) as total from emp
group by dept)  TOTALDEPT
where emp.dept = TOTALDEPT.dept
```

Remarque 3.2

Ce select n'est pas une sous-interrogation ; on ne peut synchroniser le select avec les autres tables du from (voir 3.6.3).

3.4 Clause WHERE

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

La clause where est étudiée ici pour la commande SELECT. Elle peut se rencontrer 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'un 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 3.3

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 un 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 3.4

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 employés 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 table à laquelle la colonne appartient, d'une ligne fictive qui réalise la correspondance avec les 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'une ligne nulle virtuelle à la table `EMP`, pour laquelle le prédicat

`DEPT.DEPT = EMP.DEPT` sera vrai pour tout département sans employé. Les départements sans employé feront maintenant partie de cette sélection.

Remarque 3.5

Cette syntaxe est particulière à Oracle. La norme SQL2 spécifie une syntaxe différente (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 dans la table de gauche (avant le "`LEFT OUTER JOIN`") et `FULL OUTER JOIN` pour l'ajout de lignes fictives dans les deux tables.

3.5.3 Jointure non "équi"

Les jointures autres que les équi-jointures peuvent être représentées en SQL-2 en utilisant les opérateurs de comparaison $=$, $>$, $<$, $>=$, $<=$, et aussi between et in.

Exemples 3.8

(a) Si la table tranche contient les informations sur les tranches d'impôts, on peut obtenir le taux de la tranche maximum liée à un salaire par la requête suivante :

```
select nomE, sal, pourcentage
from emp, join tranche on sal between min and max
```

(b) Sous Oracle, on écrit :

```
select nomE, sal, pourcentage
from emp, tranche
where sal between min and max
```

3.6 Sous-interrogation

Une caractéristique puissante de SQL est la possibilité qu'un prédicat employé dans une clause WHERE (expression à droite d'un opérateur de comparaison) comporte un SELECT emboîté.

Par exemple, la sélection des employés ayant même poste que MARTIN peut s'écrire 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 3.9

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 3.10

(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 3.11

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 ; dans le cas où plusieurs lignes, ou pas de ligne du tout seraient ramenées, un message d'erreur 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 de 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 comparaison classique `=, !=, <, >, <=, >=`.

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

  ALL : la comparaison sera vraie si elle est vraie pour tous les éléments de l'ensemble (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 3.12

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 3.6

L'opérateur IN est équivalent à "= ANY", et l'opérateur NOT IN est équivalent à "!= 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 3.13

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

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

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 3.14

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 3.15

```
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 3.7

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 J{Matr} EMP) [Dept, CodeP] ("J{Matr}" 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) ; ce 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 3.16

(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 la somme des valeurs qui ne sont pas égales à NULL de la colonne 'col'. De même, AVG est 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 niveau de regroupement. Par exemple, si on veut le nom et le salaire des employés qui gagnent le 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 regroupement 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 des lignes ayant une valeur commune.

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

groupe en une seule ligne toutes les lignes pour lesquelles *exp1*, *exp2*,... ont la même valeur. 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 3.17

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

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

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

(d)
```
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 peut porter que sur des caractéristiques de groupe: fonction de groupe ou expression figurant dans la clause GROUP BY.

Exemple 3.18

```
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-interrogations. La requête suivante donne le nom du département (et son nombre de secrétaires) qui a 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 SELECT.

3.10 Fonctions

Nous allons décrire ci-dessous les principales fonctions disponibles dans Oracle. Il faut remarquer que ces fonctions ne sont pas standardisées et ne sont pas toutes disponibles 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*
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_CHAR(*n*, *format*)	convertit *n* en chaîne de caractères (voir 3.10.2)
TO_NUMBER(*chaîne*)	convertit la chaîne de caractères en numérique

Exemple 3.19

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 3.20

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 numérotées 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 vaut 1 par défaut). Une valeur négative de *pos* signifie une position par rapport à la fin de la chaîne.

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

Exemple 3.21

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 le 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 appartiennent 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 ensembles de caractères. La fonction TRANSLATE remplace chaque caractère de la chaîne *chaîne* présent dans l'ensemble de caractères *car_source* par le caractère correspondant (de même 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 de 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 pouvant 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 3.22

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 3.23

```
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 masques 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 indéterminé. La clause ORDER BY précise l'ordre dans lequel la liste

des lignes sélectionnées sera donnée.

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

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

Le tri se fait d'abord selon la première expression, puis les lignes ayant la même valeur pour la première expression sont triées selon la deuxième, etc. Les valeurs nulles sont 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 de donner le rang relatif de la colonne dans la liste des colonnes, plutôt que son nom. Il est aussi possible de donner un nom d'en-tête de colonne du SELECT (voir 3.2).

Exemple 3.24

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'un 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 3.25

(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 ensemblistes

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 3.26

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 deux interrogations.

Exemple 3.27

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 les lignes obtenues dans une deuxième sélection.

Exemple 3.28

Liste des départements qui ont des employés dans la première filiale mais pas dans 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, autorisations, etc...) gérées ensemble. On pourra ainsi avoir un schéma lié à la gestion du personnel et un autre lié à la gestion des clients. Un schéma est créé par la commande `CREATE SCHEMA AUTHORIZATION`.

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.

Autre notion de SQL-2, le catalogue, est un ensemble de schémas. Un catalogue doit nécessairement comprendre un schéma particulier qui correspond au dictionnaire des données (voir 4.8).

4.2 Tables

4.2.1 CREATE TABLE AS

La commande CREATE a déjà été vue au premier chapitre. Une variante (d'Oracle mais pas dans la norme SQL-2) permet d'insérer pendant la création de la table des lignes venant d'autres tables:

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


On peut aussi spécifier des contraintes d'intégrité de colonne ou de table.

Exemple 4.1

```
CREATE TABLE MINIDEPT(CLE INTEGER, NOM VARCHAR(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 nom de la table. Dans ce cas les colonnes de la table créée auront les mêmes noms, types et 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 modifications sont possibles: ajout d'une colonne (après toutes les autres colonnes), et modification d'une colonne existante.

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

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 sont les mêmes que ceux décrits avec la commande CREATE TABLE. Les parenthèses ne sont 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 table contient déjà des lignes, la nouvelle colonne sera nulle dans ces lignes existantes et donc la 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 déjà exister dans la table. *type1*, *type2*,... sont les nouveaux types que l'on désire attribuer aux colonnes.

Il est possible de modifier la définition d'une colonne, à condition que la colonne ne contiennent que des valeurs NULL ou que la nouvelle définition soit compatible avec le 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'on ne dépasse pas les limites propres à SQL, et on peut dans tous les cas spécifier 'NULL' 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, il est possible de consulter la vue EMP10. Tout se passe comme s'il existait une table EMP10 des employés du département 10 :

```
SELECT * FROM EMP10
```

Mise à jour avec une vue

Sous certaines conditions, il est possible d'effectuer des DELETE, INSERT et des UPDATE à travers des vues.

Les conditions suivantes doivent être remplies :

- pour effectuer un DELETE, le select qui définit la vue ne doit pas comporter de jointure, de group by, de distinct, de fonction de groupe ;
- pour un UPDATE, en plus des conditions précédentes, les colonnes modifiées doivent être des colonnes réelles de la table sous-jacente ;
- pour un INSERT, en plus des conditions précédentes, toute colonne "not null" de la table sous-jacente doit être présente dans la vue.

Ainsi, il est possible de modifier les salaires du département 10 à travers la vue EMP10. 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 ajouter un employé du département 20 avec la vue EMP10.

Si l'on veut éviter cela il faut ajouter "**WITH CHECK OPTION**" dans l'ordre de création de la vue après l'interrogation définissant la vue. Il est alors interdit de créer au moyen de la vue des lignes qu'elle ne pourrait relire. Ce dispositif fonctionne également 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 voient les données, du découpage en tables. On sépare l'aspect externe (ce que voit un utilisateur particulier de la base) de l'aspect conceptuel (comment a été conçu l'ensemble de la base). Ceci favorise l'indépendance entre les programmes et les données. Si la structure des données est modifiée, les programmes ne seront pas à modifier si l'on a pris la précaution d'utiliser des vues (ou si on peut se ramener à travailler sur des vues). Par exemple, si une table est découpée en plusieurs tables après l'introduction de nouvelles données, on peut introduire une vue, jointure des nouvelles tables, et la nommer du nom de l'ancienne table 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 dans la table.

Remarque 4.2

Deux index construits sur des tables d'un même utilisateur ne peuvent avoir le même 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 contenant déjà des lignes. Il sera ensuite tenu à jour automatiquement lors des modifications de la table.

Un index peut porter sur plusieurs colonnes: la clé d'accès sera la concaténation des 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'optimiseur du SGBD qui, au moment de l'exécution de chaque requête, recherche s'il peut s'aider d'un index.

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

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

Une jointure s'effectuera souvent plus rapidement si une des colonnes de jointure est 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 index.

Une autre utilité des index est d'assurer l'unicité d'une clé en utilisant l'option "UNIQUE". Ainsi la création de l'index suivant empêchera l'insertion dans la table EMP d'un 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 plusieurs index doivent être mis à jour. De plus, la recherche d'information n'est accélérée par un index que si l'index ne contient pas trop de données égales. Il n'est pas bon, par 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 et ses paramètres 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.

En Oracle, on peut créer une nouvelle procédure stockée par la commande CREATE PROCEDURE (voir exemple 6.4 de la page 71). Les erreurs éventuelles de la compilation peuvent être vues par la commande SHOW ERRORS. On obtient les paramètres et les types d'une procédure stockée par la commande DESCRIBE.

Les procédures stockées ne sont pas normalisées par SQL-2.

4.7 Trigger

Les *triggers* (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* dans la norme SQL-3.

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, sortie 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_NAME
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}
```

donne le niveau d'isolation de la transaction qui vient de commencer (doit être la première instruction de la transaction).

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

SERIALIZABLE: un ordre de modification échoue 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. 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.

Sauf raison impérieuse, il n'est pas recommandé de modifier le fonctionnement par défaut d'Oracle.

SQL-2 définit d'autres niveaux d'isolation:

READ UNCOMMITED: les autres transactions voient les modifications d'une transaction avant même le COMMIT ; c'est le niveau d'isolation

le plus bas.

REPEATABLE READ : niveau placé entre READ COMMITTED et SERIALIZABLE ; il empêche les lectures non répétitives (voir polycopié "Introduction aux bases de données"). Sous Oracle on peut le simuler en bloquant en mode ROW SHARE (voir refrowshare) les tables des données lues (avec le plus souvent une importante perte de performances).

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 une même ligne. Si une transaction est en train de modifier une ligne d'une table, les autres transactions peuvent lire les données telles qu'elles étaient avant ces dernières modifications (jamais de temps d'attente pour la lecture), mais les autres transactions sont bloquées automatiquement par Oracle si elles veulent modifier cette même ligne.

Plus précisément, certaines commandes provoquent un blocage implicite sur la table et les lignes impliquées : DELETE, UPDATE, INSERT, SELECT FOR UPDATE et les ordres de définitions et de contrôle des données : LDD (Langage de Définition des Données) 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'un ordre SQL ; par exemple, un ordre SELECT ou UPDATE va travailler sur les lignes telles qu'elles étaient au moment du début de l'exécution de la commande, même si entre-temps une autre transaction validée par un COMMIT a modifié certaines de ces lignes. De même, si une commande UPDATE comporte un SELECT emboîté, les modifications de la commande UPDATE ne sont pas prises en compte par le SELECT emboîté.

On peut aussi obtenir une lecture consistante pour toute une transaction et pas seulement pour un ordre SQL en l'indiquant explicitement à Oracle.

Cependant, dans ce cas, la 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, on peut effectuer des blocages au niveau des lignes ou des tables. Par exemple, si l'utilisateur veut que les valeurs des lignes sur lesquelles il va travailler ne soient pas modifiées par une autre transaction, il ne peut se fier aux blocages implicites de Oracle. En effet ceux-ci ne peuvent lui assurer ceci que durant l'exécution d'un commande et pas durant toute une transaction.

Il existe cinq sortes de blocage au niveau des tables. Ces blocages sont étudiés en détails 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 la table.

On peut l'utiliser lorsque l'on veut, par exemple, établir un bilan des données contenues dans une table. L'avantage par rapport au mode exclusif est que l'on n'est pas mis en attente 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 ordre INSERT, DELETE ou UPDATE.

Il permet de modifier certaines lignes pendant que d'autres utilisateurs modifient d'autres 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 UPDATE dont la syntaxe est :

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

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

On remarquera que ce sont les lignes entières qui sont réservées et pas seulement les 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 la commande SELECT avec la clause FOR UPDATE OF.
2. on peut effectuer une préparation avant de modifier les lignes réservées en tenant 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 langage :

- 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 dans sa version 1, distribuée avec le JDK 1.1. Les nouvelles possibilités offertes par la version 2 sont abordées rapidement à la fin de ce chapitre (en attendant l'installation d'une version 8 d'Oracle pour les tester avec des nouveaux drivers). Pour plus de précisions, vous pouvez consulter, le guide JDBC en ligne à l'adresse Web
`http://www.javasoft.com/products/jdk/1.2/docs/guide/jdbc/`

JDBC ("*Java Data Base Connectivity*") offre des classes et des interfaces pour permettre l'utilisation d'un ou plusieurs SGBD à partir d'un programme en Java. Il est fourni par le paquetage Java "`java.sql`".

Les premières versions de JDBC supportent le standard "SQL-2 Entry Level" (le niveau le moins élevé, mais suffisant pour les traitements habituels). La version 2 de JDBC supporte de nombreuses fonctionnalités de la future norme SQL-3.

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

6.1 Drivers et gestionnaire de drivers

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

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

Le gestionnaire de drivers gère la liste des drivers disponibles. En chargeant plusieurs 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 auprès du gestionnaire de drivers, instance de la classe `DriverManager`, fournie par JDBC. Lorsque l'on veut ouvrir une connexion avec une base de données dans un programme, le gestionnaire de drivers 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 "propriété" 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 qui permettent de travailler avec le SGBD que l'on veut utiliser (le plus souvent fournies avec le SGBD).
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.

6.1.2 Types de drivers et applet *untrusted*

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

Les navigateurs Web se méfie (à juste titre) des applets qui proviennent d'un site distant. Ils leur interdisent par exemple

- d'écrire dans le système de fichier local,
- de communiquer par "*sockets*" avec d'autres machine que la machine d'où elles proviennent,
- de charger du code exécutable non Java.

Java 2 a introduit une nouvelle architecture de sécurité qui permet d'étendre les actions autorisées aux applets. On appelle applet "*untrusted*" les applets qui n'ont aucune autorisation spéciale. Malheureusement cette nouvelle architecture de sécurité est arrivée un peu tard et les navigateurs ont ajouté à Java leur propre façon d'étendre les actions autorisées aux applets. On peut espérer que les prochaines versions des navigateurs adopteront tous la nouvelle architecture de sécurité de Java 2.

Nous nous limitons ici aux applets *untrusted*.

Ces 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.

Elles ne peuvent échanger des données par "*sockets*" qu'avec les machines d'où elles proviennent. Donc, 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'applet car les échanges entre l'applet et le serveur se font par *sockets*. Le SGBD peut être n'importe où.

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

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

6.1.3 Servlets et pages JSP

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

Les JSP (*Java Server Page*) sont des pages Web qui peuvent contenir du code HTML (ou XML) et du code Java. Elles offrent les mêmes fonctionna-

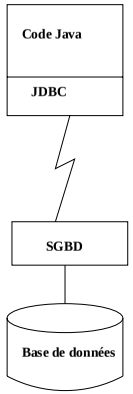

FIG. 6.1 – *Modèle 2-tiers*

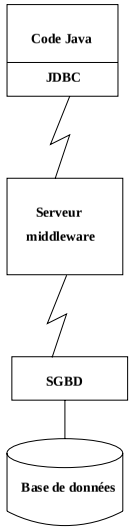

FIG. 6.2 – *Modèle 3-tiers*

lités que les servlets (elles provoquent d'ailleurs le plus souvent la création de servlets lors de leur première utilisation), en apportant souvent plus de souplesse.

6.2 Modèles de connexion à une base de données

Modèle 2-tiers Le terme 2-tiers provient de la traduction de l'anglais "*2 tiers*" qui pourrait être traduit en français par 2 niveaux.

Dans le modèle 2-tiers, 2 entités interviennent (application Java ou applet, et SGBD) : une application Java ou une applet utilise JDBC pour parler directement avec le SGBD qui gère la base de données.

On parle alors de mode client-serveur avec client "lourd" car le client effectue tous les traitements à part la gestion des données.

On parle de client "léger" quand celui-ci ne s'occupe que des traitements liés à l'interface "Homme-Machine" entre l'application et l'utilisateur de l'application.

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

Nous avons alors 3 entités qui interviennent : un client pour l'interface homme-machine (IHM), un serveur "*middleware*" pour les traitements techniques et métier, et le SGBD pour l'enregistrement et l'interrogation des données.

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.

Si l'IHM est simple, on peut se passer du langage Java pour écrire un client léger avec le langage HTML.

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 clientes 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 applet "*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 à ma connaissance que des solutions "propriétaires" qui obligent les applications à utiliser les protocoles du serveur, au détriment de la portabilité. On attend des solutions plus portables avec l'arrivée de produits *middleware* adaptés aux EJB (*Entreprise Java Beans*). Les EJB sont le pendant des *Java beans* ; ils sont déployés sur les serveurs et pas du côté client comme les *Java beans*.

Modèle logique n-tiers Dans un modèle n-tiers typique, les clients sont des clients "légers" : ils ne s'occupent essentiellement que de l'interface entre l'utilisateur et l'application. À l'autre bout se trouvent les serveurs de données.

Le *middleware* est une couche de logiciels qui se trouvent sur le réseau entre les clients et les serveurs de données. Il va assurer des services communs à toutes les applications, comme

- le nommage, pour désigner une ressource sur le réseau par un nom logique et pas par un emplacement physique,

- la sécurité pour l'accès aux données (filtrer par exemple les appels de méthodes distants),
- la gestion des transactions, comme dans les SGBD, par exemple pour annuler éventuellement les effets d'actions effectuées sur plusieurs types de données enregistrées dans des SGBD différents,
- la répartition de la charge des traitements sur plusieurs machines.

On peut parler d'architecture n-tiers car ces tâches sont effectuées par des entités du *middleware* bien distinctes, et souvent réparties sur différentes machines du réseau.

L'avantage d'un modèle logique n-tiers (n >2) est qu'il facilite l'utilisation de clients légers et que les gestions complexes citées ci-dessus sont encapsulées dans la couche "*middleware*". Ainsi, par exemple, un changement dans les règles de sécurité n'entraine pas une modification dans les autres couches. Les inconvénients sont ceux évoqués pour le modèle 3-tiers ; on peut y ajouter une mise en place complexe et une mise au point difficile des programmes car les outils de débogage ne sont souvent pas adaptés à la programmation distribuée ; il peut être alors difficile de trouver la source des erreurs parmi les divers programmes qui interviennent dans chaque traitement.

Le concept de serveur d'application est très en vogue en ce moment. Un tel serveur fournit les traitements de base de toute application (traitement des transactions, de la distribution, des messages, de la sécurité, etc...) et peut accueillir des "objets métier" (ces serveurs d'application sont le plus souvent liés au monde de l'objet) qui font les traitements spécifiques à chaque application.

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 (qui s'enregistre automatiquement auprès du gestionnaire de drivers). Pour celui que nous allons utiliser :

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

   Pour améliorer la portabilité, il est fortement conseillé de lire le nom du driver dans un fichier de propriétés Java ou d'utiliser la propriété système Java `jdbc.drivers`.

6.4 Étapes du travail avec une base de données avec JDBC

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

1. Ouvrir une connexion (`Connection`)
2. Créer les objets Java qui vont contenir les commandes SQL : `Statement`, `PreparedStatement` ou `CallableStatement`
3. Lancer l'exécution de ces commandes : interroger avec un ordre select (`executeQuery()`), modifier la base avec un ordre insert, update ou delete, ou un ordre DDL (`executeUpdate()`), 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, "de la classe `Connection`" pour désigner une classe du driver qui implémente l'interface `Connection` de JDBC, 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 drivers essaie 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");
```

Remarque 6.1

Pour améliorer la portabilité, il est fortement conseillé de lire l'URL

dans un fichier de propriétés Java.

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 :

- `executeQuery()` pour une consultation (select) (étudiée dans la section suivante).
- `executeUpdate()` pour une modification des données (insert, delete, update) ou un autre ordre SQL, de type DLL par exemple (create table). Cette méthode renvoie le nombre de lignes qui ont été affectées par la commande.
- `execute()` est utilisée dans certains cas particuliers. On peut ne connaître qu'à l'exécution la nature de l'ordre SQL à exécuter, par exemple, si l'ordre SQL à exécuter a été entré par l'utilisateur.
  Cette dernière méthode doit aussi être employée dans certains cas particuliers pour lesquels l'ordre SQL renvoie plusieurs résultats (voir la section 6.10 sur les procédures stockées).

6.8.1 Consultation des données (SELECT)

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

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

Types de données JDBC/SQL

Comme les SGBD n'ont pas tous les mêmes types SQL, JDBC a des types génériques dont les noms sont donnés dans la classe `java.sql.Types`. Ces types sont utilisés, par exemple, pour indiquer les types des paramètres des procédures stockées (voir 6.10). Ils donnent une certaine indépendance envers les types des SGBD. Ce sont les drivers particuliers des SGBD qui feront la correspondance entre les types JDBC et les types particuliers des SGBD.

Le tableau 6.1 donne les types principaux JDBC/SQL et les types Java qu'il est conseillé d'utiliser pour convertir des types JDBC/SQL en types Java.

Récupération des données

La classe `ResultSet` fournit des méthodes "getXXX(int numéroColonne)" (colonne repérée par son numéro ; la première colonne a le numéro 1 et pas 0) ou "getXXX(String nomColonne)" (colonne repérée par son nom) pour récupérer dans le code Java les valeurs des colonnes des lignes renvoyées par le SELECT. Le "XXX" de getXXX() est le nom du type Java correspondant au type JDBC attendu (voir tableau 6.1). Par exemple, la première ligne du tableau indique que, pour lire une valeur SQL du type CHAR, il est conseillé d'utiliser la méthode `getString()` qui renvoie une instance de la classe `String`.

Remarque 6.2

Les tableaux d'octets (`byte[]`) sont récupérés par la méthode `getBytes()`.

C'est la responsabilité du programmeur de choisir les bonnes méthodes getXXX() pour récupérer les données de la base, compatible avec le type

Type JDBC/SQL	**Type Java**
CHAR	String
VARCHAR	String
LONGVARCHAR	récupération sous forme de "*streams*"
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	récupération sous forme de "*streams*"
BIT	boolean
TINYINT	byte si <127, short sinon
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
DOUBLE ou FLOAT	double
DECIMAL	java.math.BigDecimal
NUMERIC	java.math.BigDecimal
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

TAB. 6.1 – *Conversion JDBC vers Java*

passé par le driver. Une certaine tolérance est prévue mais il est préférable de choisir la méthode getXXX() correspondant au tableau 6.1. Si le programmeur choisit une mauvaise méthode getXXX() et que la conversion est impossible, la méthode lance une `SQLException`.

Les données de très grande taille peuvent être récupérées sous forme de flot de données (*stream*) avec les méthodes `getAsciiStream()`, `getUnicodeStream()` ou `getBinaryStream()`.

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

Les méthodes `getObject()` retournent une valeur de la classe Java correspondant au type SQL de la colonne spécifiée (suivant le tableau 6.1). La classe Java sera une classe "enveloppante" si le type de la colonne correspond à un des types primitifs Java ; par exemple, `Integer` à la place de `int`. Cette méthode peut être utile quand on ne connait pas le type d'une donnée de la base au moment de l'écriture du programme.

`getObject()` peut aussi être utilisée pour récupérer une donnée d'un type spécifique au SGBD utilisé, non cité par le tableau 6.1. Depuis la version 2 de JDBC, elle est aussi utilisée pour récupérer les données d'un type SQL 3 défini par l'utilisateur.

Remarques 6.3

(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.

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, ou ordre DDL)

Pour lancer un ordre INSERT, UPDATE ou DELETE (ou pour un ordre DDL comme CREATE TABLE), il faut utiliser la méthode `executeUpdate(String)`. Cette méthode renvoie le nombre de lignes modifiées par la requête SQL.

Exemple 6.2

```
Statement stmt2 = conn.createStatement();
```



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

6.8.3 Ordre SQL quelconque

Les méthodes `execute(`*ordreSQL*`)` de la classe `Statement` et `execute()` (sans paramètre) de la classe `PreparedStatement` doivent être utilisées lorsque l'on ne connaît pas à l'avance le type de l'ordre SQL (select, DML, DDL ou autre) dont on doit lancer l'exécution ou lorsque l'ordre SQL n'est ni un SELECT ni un ordre de manipulation de données, ni un ordre DDL.

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

1. Les méthodes `execute()` retournent un booléen. Si le résultat est `true`, c'est que l'ordreSQL était une requête SELECT. On utilise alors un `Resultset` pour récupérer les lignes.
2. Sinon, on lance la méthode `getUpdateCount()` qui renvoie un entier. Si cet entier est strictement positif, l'ordre était un ordre DML (modification de lignes) ; l'entier est le nombre de lignes modifiées par l'ordre SQL.
3. Sinon, si cet entier est égal à 0, il s'agissait soit d'un ordre DML qui n'a modifié aucune ligne, soit d'un ordre DDL (définition des données tel qu'un CREATE TABLE) ou autre (grant, création de procédure stockée,...).

En fait, la méthode `getUpdateCount()` suffit pour déterminer le type car elle renvoie 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 la meilleure façon de l'exécuter. Avec la plupart des SGBD (dont Oracle) on peut s'arranger 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)" où XXX désigne le type Java de la 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 JDBC donnés par le tableau 6.2 (presque le tableau 6.1 inversé) avant d'envoyer les données au SGBD. Par exemple, `setShort(3, 28)` passera un SMALLINT à la base de données.

Type Java	**Type JDBC/SQL**
String	VARCHAR (ou LONGVARCHAR si trop long)
byte[]	VARBINARY (LONGVARBINARY)
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
java.math.BigDecimal	NUMERIC
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

TAB. 6.2 – *Conversion Java vers JDBC*

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 JDBC particulier en utilisant la méthode `setObject(int n, Object x, int typeSQL)`. Si on utilise `setObject()` sans le troisième paramètre, la conversion se fait dans le type JDBC donné par le tableau 6.2, selon le type Java de l'objet `x`. Cette méthode peut être utile quand on ne connait pas le type d'une donnée de la base au moment de l'écriture du programme. Le type SQL est indiqué par une constante entière définie dans la classe `java.sql.Types`. Par exemple, `Types.VARCHAR`.

De même que pour les méthodes "getXXX()" on peut utiliser des "*streams*" Java pour passer des valeurs de très grande dimension.

6.10 Procédure stockée

Les procédures stockées sont représentées par des classes qui implémentent l'interface `CallableStatement` qui hérite de l'interface `PreparedStatement`.

La première étape pour utiliser une procédure stockée est de créer une instance de la classe qui implémente l'interface `CallableStatement`. Cette instance est créée par la méthode `prepareCall()` de la classe Connection. On passe à cette méthode une chaîne de caractères qui décrit comment sera appelée la procédure stockée.

L'appel des procédures stockées n'est pas standardisée dans les différents SGBD. JDBC appelle les procédures stockées avec une syntaxe spéciale uniforme pour tous les SGBD (voir 6.11). Ce sera la tâche du driver particulier à chaque SGBD de la traduire avec la syntaxe appropriée.

Voici la syntaxe:

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

Remarque 6.4

Certains paramètres peuvent être remplacés par des valeurs littérales si on est certain que la procédure sera toujours appelée avec ces valeurs:
`{call augmenter[(10, ?,...)}`

Avant l'appel de la procédure, il faut donner les valeurs des paramètres "IN" et IN/OUT" et indiquer le type des paramètres "OUT" et "IN/OUT".

Au contraire des paramètres des requêtes paramétrées qui n'ont que des paramètres "IN", les procédures stockées peuvent aussi avoir des paramètres "IN/OUT" ou "OUT". C'est au programmeur d'utiliser les méthodes (de la classe `CallableStatement`) "setXXX()" ou "getXXX()" qui conviennent à la définition de la procédure stockée. le numéro correspond à l'ordre d'apparition des "?" dans

Le programme Java doit indiquer le type JDBC de tous les paramètres "IN/OUT" et "OUT" par la méthode `registerOutParameter()`. On utilise pour cela les constantes (entières) de la classe `java.sql.Types` dont les noms sont donnés par la tableau 6.2 ; par exemple, `Types.CHAR` pour le type JDBC `CHAR` et `Types.VARCHAR` pour le type JDBC `VARCHAR` (`VARCHAR2` pour Oracle).

Le lancement de la procédure se fait par l'appel des méthodes `execute()` (le plus souvent), `executeQuery()` ou `executeUpdate()`, suivant le type des commandes SQL que la procédure contient.

Remarques 6.5

(a) Il est recommandé de récupérer les valeurs des "Resultset" avant de récupérer les paramètres "OUT" et "INOUT".

(b) Avec Oracle une procédure stockée est écrite dans le langage PL/SQL qui ne peut comporter de SELECT sans ranger les valeurs trouvées dans des variables PL/SQL (clause INTO) et on ne peut pas récupérer de "Resultset". Le type non standard d'Oracle `REFCURSOR` permet cependant d'obtenir des "ResultSet" avec une procédure stockée. Attention, pour travailler avec ce type il faut importer un paquetage Java fourni par Oracle avec le driver, et la version de ce driver de Oracle 7 est incompatible avec le JDK 1.2.

(c) Il ne faut pas se faire d'illusions, si on utilise des procédures stockées on perd beaucoup de portabilité malgré les efforts de JDBC pour réduire les différences de syntaxe. En effet, les différences se trouve aussi bien dans le comportement que dans la syntaxe (voir la remarque précédente). Malgré tout, le gain de performance apporté par certaines procédures stockées peut les rendre indispensables dans des cas précis.

(d) Pour le moment il est à peu près impossible d'écrire du code portable si on utilise des procédures stockées. Tous les SGBD n'en ont pas et ceux qui en ont les utilisent d'une façon trop différente.

Exemple 6.4

Voici une procédure stockée Oracle qui prend en paramètre un numéro de département et un pourcentage, augmente tous les salaires des employés de ce département 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;
```

Remarques 6.6

(a) Voici un ordre SQL pour avoir, sous Oracle, les noms des procédures stockées et le nom de leur propriétaire :

```
select owner, object_name
from all_objects
where object_type = 'PROCEDURE'
order by owner, object_name
```

(b) Sous Oracle, on peut avoir une description des paramètres d'une procédure stockée par la commande "DESC *nom-procédure*".

(c) ... et, si on a les autorisations "DBA" (administrateur de la base), le code de la procédure est donné par :

```
select text
from dba_source
where name = 'nom-procedure'
order by line
```

(d) Les erreurs de compilation des procédures stockées peuvent être vues sous Oracle par la commande SQL*PLUS "`show errors`".

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.execute();
int x = csmt.getFloat(3);
System.out.println("Cout total de l'augmentation : " + x);
```

Remarque 6.7

Si la procédure stockée ne comporte qu'un ordre SELECT, on peut utiliser executeQuery() pour récupérer un ResultSet.

Procédure stockée comprenant plusieurs ordres SQL

Quand une procédure stockée comporte plus d'un ordre SQL, il faut utiliser la méthode `execute()` pour exécuter la procédure et des méthodes `getMoreResults()` pour récupérer le résultat suivant d'une instruction SQL. Un résultat est soit un ResultSet, soit un "updateCount" (nombre de lignes modifiées par un ordre DML).

Exemple 6.6

Voici un exemple schématique de traitement d'une procédure stockée qui renvoie plusieurs résultats:

```
csmt.execute(ordreSQL);
while (true) {
  int nbLignes = csmt.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 = csmt.getResultSet();
    if (rs != null) { // ordre SQL
      ....
    }
    else              // plus de résultat à traiter
      break;
  }
  csmt.getMoreResults(); // récupère le résultat suivant
}
```


Cet algorithme ne fonctionne pas sous Oracle. On ne peut récupérer les résultats intermédiaires qu'avec des variables (clause INTO de SELECT ; voir la remarque 6.5 page 71).

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

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

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

Exemples 6.7

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

(b) `{fn user()}`

Remarque 6.8

Ce type de syntaxe est aussi associé à la jointure externe qui a une syntaxe différente dans les différents SGBD.

Oracle n'est pas à la norme SQL-2 pour la jointure externe. De plus, le driver fourni par Oracle ne supporte pas ce type de syntaxe. On doit donc utiliser la syntaxe particulière à Oracle "(+)" et on perd donc en portabilité.

6.12 Les exceptions liées à SQL

Les exceptions liées à SQL sont représentées par la classe `SQLException` qui hérite de la classe `Exception`. Cette classes a 3 méthodes utiles pour avoir des informations sur les exceptions:

- `getMessage()` renvoie un message d'erreur (String),
- `getSQLState()` renvoie un code d'erreur (qui suit la spécification XOpen SQL ; c'est une String),
- `getErrorCode()` renvoie un code d'erreur spécifique au SGBD (int),
- `getNextException()` renvoie une éventuelle SQLException suivante, ou `null` (pour avoir plus d'informations)

Exemple 6.8

```
try {
  // Code SQL qui peut générer une exception
  . . .
}
catch(SQLException e) {
  System.out.println(" Les SQLException :");
  while (e != null) {
    System.out.println("\nMessage : " + e.getMessage ());
    System.out.println("Code d'erreur : " + e.getSQLState ());
    System.out.println("Numéro d'erreur : " + e.getErrorCode ());
    e = e.getNextException();
  }
}
```

6.13 Les avertissements liés à SQL

Le programme Java peut recevoir un avertissement du SGBD indiquant une anomalie de fonctionnement. Une telle anomalie n'est pas aussi grave qu'une exception et elle n'interrompt pas l'exécution normale du programme. Ces avertissements sont chaînés à l'instruction SQL qui a provoqué le problème (objet de type `Connection`, `Statement`, y compris `PreparedStatement` et `CallableStatement`, ou `ResultSet`) .

Le type d'avertissement le plus fréquent indique qu'une donnée a été tronquée durant une lecture ou une écriture dans la base (en général à cause d'un mauvais choix du type de la destination).

Les classes Java `SQLWarning` (classe fille de `SQLException`) et `DataTruncation` (classe fille de `SQLWarning`) représentent les avertissements. Bien qu'héritant de `Exception`, on ne les attrapent pas avec un bloc try - catch mais en envoyant le message `getWarnings()` à l'instruction SQL fautive.

Exemple 6.9

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("select ...");
while (rs.next()) {
  String valeur = rs.getString(1);
  . . .
  SQLWarning warning = stmt.getWarnings();
  if (warning != null) {
    System.out.println("Avertissements sur l'instruction :");
    while (warning != null) {
```



```
      System.out.println("\nMessage: "
                         + warning.getMessage());
      System.out.println("Code d'erreur : "
                         + warning.getSQLState());
      System.out.println("Numéro d'erreur : "
                         + warning.getErrorCode());
      warning = warning.getNextWarning();
    }
  }
  SQLWarning warn = rs.getWarnings();
  if (warn != null) {
    System.out.println("Avertissements sur le ResultSet");
    while (warn != null) {
      System.out.println("Message: " + warn.getMessage());
      System.out.println("Code d'erreur : " + warn.getSQLState());
      System.out.println("Numéro d'erreur : " + warn.getErrorCode());
      warn = warn.getNextWarning();
    }
  }
}
```

6.14 Les "Meta données"

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

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

6.14.1 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);
}
```

Remarque 6.9

Les drivers Oracle actuels renvoient une chaîne vide en retour de la méthode `getTableName(int)`.

6.14.2 DatabaseMetaData

Cette interface comporte des dizaines de méthodes.

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 java.awt.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);
  }
...
}
```

6.15 Les ajouts de JDBC 2

Voici un catalogue des principales nouvelles fonctionnalités, avec des exemples :

- ResultSet amélioré: il peut être parcouru dans les 2 sens et on peut modifier des données d'un ResultSet directement par des méthodes Java sans utiliser SQL,
- amélioration des performances par regroupement de plusieurs ordres SQL,

- manipulation de types de données de SQL 3 (données de très grande taille et types créés par le développeur)

Avertissement : les exemples donnés dans les sections suivantes n'ont pu être testés faute de version 8 d'Oracle installée.

6.15.1 ResultSet

Dans la première version de JDBC, les ResultSet ne peuvent être parcourus que dans un sens. JDBC 2 a ajouté la possibilité de se positionner n'importe où dans un ResultSet ou de le parcourir dans les 2 sens.

On peut aussi maintenant modifier les données de la base par le ResultSet, sans écrire d'ordre SQL.

Il y a 3 types de ResultSet :

- TYPE_FORWARD_ONLY: ne peut être parcouru que dans un sens,
- TYPE_SCROLL_INSENSITIVE: peut être parcouru dans les 2 sens, mais ne reflète pas les modifications faites après la récupération du ResultSet,
- TYPE_SCROLL_SENSITIVE: peut être parcouru dans les 2 sens, et reflète les modifications faites par ailleurs après la récupération du ResultSet.

Parallèllement à ces types, un ResulSet peut être

- CONCUR_READ_ONLY: on ne peut pas modifier les données en passant par le ResultSet,
- CONCUR_UPDATABLE: on peut modifier les données en passant par le ResultSet.

On peut donc avoir 6 types (3 x 2) de ResultSet en combinant toutes ces possibilités. En fait, tous les drivers ne permettent pas toutes ces possibilités, ou peuvent les permettre mais avec de très mauvaises performances (c'est le cas des drivers fournis par Oracle).

Voici un exemple de manipulation d'un ResulSet :

```
Statement stmt = con.createStatement(
      ResultSet.TYPE_SCROLL_SENSITIVE,
      ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery(
      "SELECT nomE, salaire FROM emp");
while (srs.next()) {
  String nomE = srs.getString("nomE");
    float price = srs.getFloat("salaire");
    System.out.println(nomE + " ; " + salaire);
```

```
}
// Parcours de la fin vers le début
srs.afterLast();
while (srs.previous()) {
  String nomE = srs.getString("nomE");
    float price = srs.getFloat("salaire");
    System.out.println(nomE + " ; " + salaire);
}
// Divers positionnement dans le ResultSet
srs.absolute(-2); // avant-dernière ligne
srs.absolute(4);
int numLigne = srs.getRow(); // rowNum = 4
srs.relative(-3);
int numLigne = srs.getRow(); // rowNum = 1
srs.relative(2);
int numLigne = srs.getRow();  // rowNum = 3
// ResultSet "modifiable"
Statement stmt = con.createStatement(
      ResultSet.TYPE_SCROLL_SENSITIVE,
      ResultSet.CONCUR_UPDATABLE);
ResultSet srs = stmt.executeQuery(
      "SELECT nomE, salaire FROM emp");
// Modifier des lignes
uprs.last();
uprs.updateFloat("SALAIRE", 10000);
uprs.cancelRowUpdates();
uprs.updateFloat("SALAIRE", 12000);
uprs.updateRow();
// Insérer des lignes
uprs.moveToInsertRow();
uprs.updateInt("MATR", 150);
uprs.updateString("NOME", "Kleber");
uprs.updateFloat("SALAIRE", 10000);
. . .
uprs.insertRow();
// Supprimer des lignes
uprs.absolute(4);
uprs.deleteRow();
```


6.15.2 Regrouper des modifications pour les envoyer au SGBD

On peut améliorer les performances en regroupant des ordres SQL (le vérifier par des tests!).

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch("INSERT INTO DEPT" +
              "VALUES(70, 'Finances', 'Nancy')");
stmt.addBatch("INSERT INTO DEPT" +
              "VALUES(80, 'Comptabilité', 'Nice')");
int[] updateCounts = stmt.executeBatch();
```

6.15.3 Nouveaux types supportés par JDBC

JDBC supporte les types suivants de SQL3:

- BLOB (Binary Large Object)
- CLOB (Character Large Object)
- ARRAY
- types structurés définis par l'utilisateur
- références vers des instances de types structurés

BLOB et CLOB

2 nouveaux types SQL ont été introduits: BLOB et CLOB. Les interfaces Java `Blob` et `Clob` permettent de travailler en Java avec ces types.

Par exemple:

```
Clob clob = rset.getClob("commentaires");
long n = clob.length();
for (long i = 0; i < n; i += 100) {
  Sring ch = clob.getSubString(i, 100);
}
```

Array

Ce type permet d'avoir des colonnes multi-valuées composées d'un certain nombre d'informations de même type.

On a accès aux éléments du tableau par leur indice ou par un ResultSet.

```
Array tab = rset.getArray("tableau");
```

```
// ne récupère que les éléments 5 et 6 du tableau
data = (String[])tab.getArray(4, 2);
for (int i = 0; i < data.length; i++) {
  System.out.println(data[i]);
}
```

```
Array tab = rset.getArray("tableau");
// ne récupère que les éléments 5 et 6 du tableau
ResultSet data = tab.getResultSet(4, 2);
while (data.next()) {
  System.out.println(data.getString(2));
}
```

Types structurés définis par l'utilisateur

SQL 3 a introduit les types "distinct" pour distinguer des types de données qui s'appuient sur le même type de base. Par exemple, pour distinguer un âge et un matricule qui sont tous les deux des entiers mais qui ne correspondent pas à un même domaine :

```
create type matricule as integer;
```

Pour faciliter l'enregistrement d'objets dans un SGBDR, le programmeur peut créer ses propres types structurés SQL, à partir des types de base fournis par SQL :

```
create type departement (
  numDept integer,
  nomDept varchar(15),
  lieu varchar(20))
```

On peut ensuite utiliser ces types distincts et structurés comme les types de base :

```
create table emp (
  matr matricule constraint pkEmp primary key,
  nomE varchar(15),
  . . .
  dept departement)
```

Si on veut pouvoir facilement enregistrer et relire dans la base les instances d'une classe Java

1. On crée un type structuré SQL de même structure que la classe.
2. La classe Java doit implémenter l'interface `java.sql.SQLData`.
3. On indique au driver JDBC la correspondance entre la classe Java et le type structuré.

Quand on implémente l'interface `SQLData`, l'ordre de lecture (et d'écriture) doit être celui donné dans la définition du type structuré.

```
public class Dept implements SQLData {
  private int numDept;
  private String nomDept, lieu;
  private String nomTypeSQL;
  . . . // (constructeurs et autres méthodes de Dept)

  // Les 3 méthodes de l'interface SQLData
  public void readSQL(SQLInput flot, String type)
      throws SQLException {
    nomTypeSQL = type;
    numDept = flot.readInt();
    nomDept = flot.readString();
    lieu = flot.readString();
  }
  public void writeSQL(SQLOutput flot)
      throws SQLException {
    flot.writeInt(numDept);
    flot.writeString(nomDept);
    flot.writeString(lieu);
  }

  public String getSQLTypeName() {
    return sqlTypeName;
  }
}
```

Pour faciliter l'enregistrement dans la base des données d'un type structuré, il faut indiquer au driver JDBC le nom de la classe Java qui correspond à ce type. On utilise pour cela une Map utilisée en interne par la connexion :

```
Map map = conn.getTypeMap();
map.put("departement", dept.class);
```

Il est alors simple de lire et d'écrire des valeurs du type structuré :

```
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery("SELECT * FROM EMP");
while (rset.next()) {
  Dept numDept = (Dept)rset.getObject("DEPT");
```

```
  System.out.println("Lieu du dept : " + dept.getLieu());
}

Dept dept = new Dept(...);
. . .
PreparedStatement ps = conn.createStatement(
  "INSERT INTO EMP" +
  "(matricule, ... , dept)" +
  " VALUES (?, ... , ?)");
ps.setInt(1, matricule);
. . .
ps.setObject(8, dept);
```

Références

Pour améliorer les performances en réduisant les jointures, on peut conserver dans les tables une référence à une valeur plutôt qu'une clé étrangère qui permet de retrouver cette valeur.

Si on veut pouvoir référencer les lignes d'une table dans une autre table, il faut créer cette table à partir d'un type structuré.

Par exemple, à partir du type structuré `departement`, on peut créer la table dept :

```
create table dept of departement (
  refDept ref(departement)
  values are system generated
)

create table emp (
  matr matricule
    constraint pkEmp primary key,
  nomE varchar(15),
  . . .
  refDept ref(departement) )
```

On ne peut récupérer directement un objet référencé. On doit d'abord récupérer une référence et ensuite, on interroge la table de définition de la référence pour trouver l'objet :

```
java.sql.Ref refDept;
Statement stmt = conn.createStatement();
ResultSet rset1 = stmt.executeQuery(
  "SELECT refDept from emp" +
  "where matricule = 105");
```



```
if (rset1.next()) {
  refDept = rset1.getRef("refDept");
  PreparedStatement ps =
    conn.prepareStatement("SELECT lieu FROM emp" +
                          " WHERE deptRef = ?");
  ps.setRef(1, refDept);
  ResultSet rset2 = ps.executeQuery();
  if (rset2.next()) {
    Struct struct = (Struct)rset2.getObject("LIEU");
    Object[] attributs = struct.getAttributes();
    System.out.println(rset1.getString("nomE") +
                       "travaille à " +
                       attributs[2]);
  } // if (rset2
} // if (rset1
```

Index