Université de Nice Sophia-Antipolis

Langage SQL

version 5.7 du polycopié

Richard Grin

4 janvier 2008

Table des matières

Présentation du polycopié **vi**

1 Introduction **1**
1.1 Présentation de SQL 1
1.2 Normes SQL 1
1.3 Utilitaires associés 2
1.4 Connexion et déconnexion 2
1.5 Objets manipulés par SQL 3
1.5.1 Identificateurs 3
1.5.2 Tables 3
1.5.3 Colonnes 4
1.6 Types de données 5
1.6.1 Types numériques 5
1.6.2 Types chaîne de caractères 6
1.6.3 Types temporels 7
1.6.4 Types binaires 7
1.6.5 Valeur NULL 8
1.7 Sélections simples 8
1.8 Expressions 9
1.8.1 Contenu d'une expression, opérateurs et fonctions . . . 9
1.8.2 Expressions NULL 9

2 Création d'une table et contraintes **11**
2.1 Création d'une table 11
2.2 Contrainte d'intégrité 12
2.2.1 Types de contraintes 12
2.2.2 Ajouter, supprimer ou renommer une contrainte 15
2.2.3 Enlever, différer des contraintes 15

3 Langage de manipulation des données 19
3.1 Insertion . . . 19
3.2 Modification . . . 20
3.3 Suppression . . . 21
3.4 Transactions . . . 22
3.4.1 Généralités sur les transactions . . . 22
3.4.2 Les transactions dans SQL . . . 23
3.4.3 Autres modèles de transactions . . . 24

4 Interrogations 26
4.1 Syntaxe générale . . . 26
4.2 Clause SELECT . . . 26
4.2.1 select comme expression . . . 27
4.2.2 Pseudo-colonnes . . . 28
4.3 Clause FROM . . . 29
4.4 Clause WHERE . . . 31
4.4.1 Clause WHERE simple . . . 31
4.4.2 Opérateurs logiques . . . 33
4.5 Jointure . . . 33
4.5.1 Jointure naturelle . . . 35
4.5.2 Jointure d'une table avec elle-même . . . 35
4.5.3 Jointure externe . . . 36
4.5.4 Jointure "non équi" . . . 37
4.6 Sous-interrogation . . . 38
4.6.1 Sous-interrogation à une ligne et une colonne . . . 38
4.6.2 Sous-interrogation ramenant plusieurs lignes . . . 39
4.6.3 Sous-interrogation synchronisée . . . 40
4.6.4 Sous-interrogation ramenant plusieurs colonnes . . . 41
4.6.5 Clause EXISTS . . . 41
4.7 Fonctions de groupes . . . 45
4.8 Clause GROUP BY . . . 45
4.9 Clause HAVING . . . 47
4.10 Fonctions . . . 47
4.10.1 Fonctions arithmétiques . . . 48
4.10.2 Fonctions chaînes de caractères . . . 48
4.10.3 Fonctions de travail avec les dates . . . 51
4.10.4 Fonction de choix (CASE) . . . 51
4.10.5 Nom de l'utilisateur . . . 53
4.11 Clause ORDER BY . . . 53
4.12 Opérateurs ensemblistes . . . 54
4.12.1 Opérateur UNION . . . 55

4.12.2 Opérateur INTERSECT 55
4.12.3 Opérateur EXCEPT 55
4.12.4 Clause ORDER BY 55
4.13 Limiter le nombre de lignes renvoyées 56
4.14 Injection de code SQL 56

5 Langage de définition des données 59
5.1 Schéma 59
5.2 Tables 59
5.2.1 CREATE TABLE AS 59
5.2.2 ALTER TABLE 60
5.2.3 Supprimer une table - DROP TABLE 62
5.2.4 Synonyme public de table ou de vue 62
5.3 Vues 62
5.3.1 CREATE VIEW 63
5.3.2 DROP VIEW 64
5.3.3 Utilisation des vues 64
5.3.4 Utilité des vues 65
5.4 Index 66
5.4.1 CREATE INDEX 66
5.4.2 Utilité des index 67
5.4.3 DROP INDEX 68
5.4.4 Types d'index 68
5.4.5 Dictionnaire des données 69
5.5 Génération de clés primaires 70
5.5.1 Utilisation de tables de la base 70
5.5.2 Autres solutions 71
5.5.3 Séquences 71
5.6 Procédure et fonction stockée 73
5.6.1 Procédure stockée 73
5.6.2 Fonction stockée 74
5.6.3 Tables du dictionnaire 75
5.7 Triggers 75
5.7.1 Types de triggers 77
5.7.2 Exemple 77
5.7.3 Restriction sur le code des triggers 78
5.7.4 Clause WHEN 78
5.7.5 Triggers INSTEAD OF 79
5.7.6 Dictionnaire de données 80
5.8 Dictionnaire de données 81
5.9 Privilèges d'accès à la base 82

5.9.1 GRANT 82
5.9.2 REVOKE 83
5.9.3 Changement de mot de passe 83
5.9.4 Synonyme 84
5.9.5 Création d'un utilisateur, rôle 84

6 Gestion des accès concurrents **86**
6.1 Problèmes liés aux accès concurrents 86
6.1.1 Mises à jour perdues 86
6.1.2 Lecture inconsistante ou lecture impropre 87
6.1.3 Lecture non répétitive, ou non reproductible, ou incohérente 87
6.1.4 Lignes fantômes 88
6.2 Isolation des transactions 88
6.2.1 Sérialisation des transactions 88
6.2.2 Niveaux d'isolation 89
6.3 Traitement pour les accès concurrents 90
6.3.1 Traitement pessimiste 90
6.3.2 Traitement optimiste 91
6.4 Traitement par défaut des accès concurrents 92
6.5 Blocages explicites 94
6.5.1 Blocages explicites d'Oracle 95

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 ont été testés avec le SGBD Oracle, version 10g mais n'utilisent qu'exceptionnellement les particularités de ce SGBD par rapport aux normes SQL.

Ce cours concerne la norme SQL2. SQL3, le relationnel-objet et l'interface avec le langage Java sont étudiés dans un autre cours.

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" et la version du document.

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

La norme SQL2 (appelée aussi SQL92) date de 1992. Le niveau "Entry Level" est à peu près respecté par tous les SGBD relationnels qui dominent actuellement le marché.

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)

SQL3 (appelée aussi SQL99) est la nouvelle norme SQL.

Malgré ces normes, il existe des différences non négligeables entre les syntaxes et fonctionnalités des différents SGBD. En conséquence, écrire du code portable n'est pas toujours simple dans le domaine des bases de données.

1.3 Utilitaires associés

Comme tous les autres SGBD, Oracle fournit plusieurs utilitaires qui facilitent l'emploi du langage SQL et le développement d'applications de gestion s'appuyant sur une base de données relationnelle. En particulier SQLFORMS facilite grandement la réalisation des 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. Pour taper et faire exécuter des commandes SQL, les séances de travaux pratiques utiliseront soit un petit utilitaire, SQL*PLUS, fourni par Oracle, soit un utilitaire gratuit que l'on peut récupérer gratuitement sur le Web et qui permet d'envoyer des commandes SQL à un SGBD distant.

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

1.4 Connexion et déconnexion

On entre dans SQL*PLUS 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, SQL*PLUS 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 SQL à taper est **EXIT** (ou `exit` car on peut taper les commandes SQL en majuscules ou en minuscules).

1.5 Objets manipulés 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, REF, REFERENCES, SELECT, SEQUENCE, SESSION, SET, TABLE, TYPE.

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 5.1).

(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

Table DUAL

C'est une particularité d'Oracle. Cette pseudo-table ne contient qu'une seule ligne et une seule colonne et ne peut être utilisée qu'avec une requête "select".

Elle permet de faire afficher une expression dont la valeur ne dépend d'aucune table en particulier.

Exemple 1.2

Afficher la date d'aujourd'hui, le nom de l'utilisateur et le résultat d'un calcul :

```
select sysdate, user, round(3676 / 7) from dual
```

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 2.1).

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

1.6.1 Types numériques

Types numériques SQL-2

Les types numériques de la norme SQL-2 sont :

- Nombres entiers : `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 DECIMAL 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.

Exemple 1.3

SALAIRE DECIMAL(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}).

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 chiffres 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*.

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 constante.
  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 ; par défaut, *longueur* est égale à 1. 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). Tous les SGBD imposent une valeur maximum pour *longueur* (255 pour Oracle).
- le type `VARCHAR` pour les colonnes qui contiennent des chaînes de longueurs variables.
  On déclare ces colonnes par :
  `VARCHAR(`*longueur*`)`
  longueur indique la longueur maximale des chaînes contenues dans la colonne. Tous les SGBD imposent une valeur maximum pour *longueur* (plusieurs milliers de caractères).

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) ;

INTERVAL permet d'indiquer un intervalle de temps.

Types temporels d'Oracle

Oracle offre le type DATE comme en SQL-2 mais pour Oracle une donnée de type DATE inclut un temps en heures, minutes et secondes.

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, '25/11/1992' (le format "américain" par défaut donnerait '25-NOV-1992'). L'utilisateur peut saisir des dates telles que '3/8/1993' mais les dates sont toujours affichées avec deux chiffres pour le jour et le mois, par exemple, '03/08/1993'.

Remarque 1.2

Ne pas oublier de donner 4 chiffres pour l'année, sinon la date risque d'être mal interprétée par Oracle (voir remarque 4.12 page 51).

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 fournit les types BIT et BIT VARYING (longueur constante ou non).

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. Voir aussi 1.8.2.

1.7 Sélections simples

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 pour les interrogations courantes. Une étude plus complète sera vue au chapitre 4.

```
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 1.8) 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 1.8) et *op* est un des opérateurs `=`, `!=` (différent), `>`, `>=`, `<`, `<=`.

Exemple 1.5

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

1.8 Expressions

1.8.1 Contenu d'une expression, opérateurs et fonctions

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 4.11)
- dans les ordres de manipulations de données (INSERT, UPDATE, DELETE étudiés au chapitre 3)

Le tableau qui suit donne le nom des principales fonctions (voir 4.10 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

1.8.2 Expressions NULL

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

Une exception à cette règle est la fonction COALESCE.

`COALESCE (`*`expr1, expr2,...`*`)`

renvoie la première valeur qui n'est pas `null` parmi les valeurs des expressions *expr1*, *expr2*,....

Elle permet de remplacer la valeur `null` par une autre valeur.

Exemple 1.6

```
select coalesce(salaire, commission, 0) from emp
```

Cette requête renvoie le salaire s'il n'est pas `null`. S'il est `null`, renvoie la commission si elle n'est pas `null`. Sinon elle renvoie 0.

Il faut préférer la fonction SQL2 `COALESCE` à la fonction NVL (*Null Value*) fournie par Oracle (`COALESCE` est disponible avec Oracle 10g mais pas avec Oracle 9i) qui permet de remplacer une valeur NULL par une valeur par défaut ; en fait, c'est `COALESCE` avec seulement 2 expressions :

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

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

Exemple 1.7 `NVL(COMM, 0)`

Au contraire, la fonction SQL2 NULLIF renvoie la valeur `null` si une expression est égale à une certaine valeur. Elle peut être utile lorsque des données sont reprises d'un ancien SGBD non relationnel qui ne supportait pas la valeur null.

Exemple 1.8

```
select NULLIF(salaire, -1) from emp
```

Chapitre 2

Création d'une table et contraintes d'intégrité

2.1 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 2.2 et 5.2.1 pour des compléments) :

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

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

On peut ajouter après la description d'une colonne l'option `NOT NULL` qui interdira que cette colonne contienne la valeur `NULL`. On peut aussi ajouter des contraintes d'intégrité portant sur une ou plusieurs colonnes de la table (voir 2.2).

Exemple 2.1

```
CREATE TABLE article (
  ref     VARCHAR(10) constraint pk_article primary key,
  nom     VARCHAR(30) NOT NULL,
  prix    DECIMAL(9,2),
  datemaj DATE)
```

On peut donner une valeur par défaut pour une colonne si la colonne n'est pas renseignée.

Exemple 2.2

```
CREATE TABLE article (
  ref     VARCHAR(10) constraint pk_article primary key,
  nom     VARCHAR(30) NOT NULL,
  prix    DECIMAL(9,2),
  datemaj DATE DEFAULT CURRENT_DATE)
```

2.2 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 :

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

Toute définition de table doit comporter au moins une contrainte de type PRIMARY KEY.

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) :

`CONSTRAINT` *nom-contrainte définition-contrainte*

Le nom d'une contrainte doit être unique parmi toutes les contraintes de toutes les tables de la base de données.

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.

2.2.1 Types de contraintes

Contrainte de clé primaire

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

définit la clé primaire de la table. Aucune des colonnes qui forment cette clé ne doit avoir une valeur NULL.

Contrainte d'unicité

```
(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.

Remarque 2.1

Cette contrainte UNIQUE convient à des clés candidates. Cependant une colonne UNIQUE peut avoir des valeurs NULL et une contrainte UNIQUE ne correspond donc pas toujours à un identificateur.

Contrainte de clé étrangère

```
(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 entraîner 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.

A la place de "ON DELETE CASCADE" on peut donner l'option "**ON DELETE SET NULL**". Dans ce cas, la clé étrangère sera mise à NULL si la ligne qu'elle référence dans *tableref* est supprimée.

La norme SQL2 offre 4 autres options qui ne sont pas implémentée dans Oracle :

ON DELETE SET DEFAULT met une valeur par défaut dans la clé étrangère quand la clé primaire référencée est supprimée.

ON UPDATE CASCADE modifie la clé étrangère si on modifie la clé primaire (ce qui est à éviter).

ON UPDATE SET NULL met NULL dans la clé étrangère quand la clé primaire référencée est modifiée.

ON UPDATE SET DEFAULT met une valeur par défaut dans la clé étrangère quand la clé primaire référencée est modifiée.

Contrainte "CHECK"

`CHECK(`*condition*`)`

donne une condition que les colonnes de chaque ligne devront vérifier (exemples dans la section suivante). On peut ainsi indiquer des contraintes d'intégrité de domaines. Cette contrainte peut être une contrainte de colonne ou de table. Si c'est une contrainte de colonne, elle ne doit porter que sur la colonne en question.

Exemples de contraintes

Exemples 2.3

(a) Quelques contraintes sur des colonnes :

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

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

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

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

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

2.2.2 Ajouter, supprimer ou renommer une contrainte

Des contraintes d'intégrité peuvent être ajoutées ou supprimées par la commande ALTER TABLE. On peut aussi modifier l'état de contraintes (voir la section suivante) par MODIFY CONSTRAINT. On ne peut ajouter que des contraintes de table. Si on veut ajouter (ou modifier) une contrainte de colonne, il faut modifier la colonne (voir exemple 5.4 page 61).

Exemple 2.4

```
ALTER TABLE EMP
  DROP CONSTRAINT NOM_UNIQUE
  ADD CONSTRAINT SAL_MIN CHECK(SAL + NVL(COMM,0) > 1000)
  RENAME CONSTRAINT NOM1 TO NOM2
  MODIFY CONSTRAINT SAL_MIN DISABLE
```

2.2.3 Enlever, différer des contraintes

Enlever une contrainte

Les contraintes d'intégrité sont parfois gênantes. On peut vouloir les enlever pour améliorer les performances durant le chargement d'une grande quantité de données dans la base. Pour cela, Oracle fournit la commande ALTER TABLE ... DISABLE/ENABLE. Il faut l'utiliser avec précaution lorsque l'on est certain qu'aucune donnée ajoutée pendant l'invalidation de la contrainte ne violera la contrainte.

Exemple 2.5

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

La clause ENABLE a une option qui permet de conserver dans une table les lignes qui ne respectent pas la contrainte réactivée.

Différer des contraintes

Normalement, les contraintes d'intégrité sont vérifiées à chaque requête SQL.

Dans certaines circonstances particulières on peut préférer différer la vérification d'une ou plusieurs contraintes à la fin d'un ensemble de requêtes regroupées en une transaction (les transactions sont étudiées en 3.4).

Supposons par exemple, que l'on veuille ajouter un grand nombre d'employés. La table des employés contient une contrainte sur la colonne sup : le matricule du supérieur d'un employé doit correspondre à un matricule d'un des employés de table. La contrainte sera violée si un employé est ajouté avant son supérieur. Tout se passera bien si tous les employés sont ajoutés en une seule transaction et si cette contrainte n'est vérifiée qu'à la fin de la transaction.

Si une contrainte est différée et si elle n'est pas vérifiée au moment du commit de la transaction, toute la transaction est invalidée (rollback) automatiquement par le SGBD.

Pour différer la vérification d'une contrainte à la fin de la transaction, on procède en 2 étapes :

1. Au moment de la déclaration de la contrainte, on indique qu'on pourra différer sa vérification :
   ```
   CONSTRAINT truc def-contrainte [NOT] DEFERRABLE
   [INITIALLY DEFERRED | IMMEDIATE]
   ```
   La valeur par défaut est `NOT DEFERRABLE`.

   Si une contrainte peut être différée, elle ne le sera qu'après la deuxième étape. On peut changer ça en donnant la clause `INITIALLY DEFERRED` qui diffère tout de suite la contrainte pour toutes les transactions.

2. On diffère la vérification de contraintes juste pour la transaction en cours, avec la commande :
   ```
   set CONSTRAINTS {liste de contraintes | ALL} DEFERRED
   ```
   Si elle est `INITIALLY DEFERRED`, elle sera différée par défaut au début de chaque transaction. On peut ne plus la différer pour le temps d'une transaction en lançant la commande :
   ```
   set CONSTRAINTS {liste de contraintes | ALL} IMMEDIATE
   ```

On peut modifier l'état d'une contrainte différable par la commande ALTER TABLE ; par exemple :

```
ALTER TABLE emp
MODIFY CONSTRAINT ref_dept_emp INITIALLY DEFERRED
```

Si on veut rendre une contrainte différable (ou l'inverse), on doit supprimer la contrainte et la recréer ensuite.

Exemple 2.6

```
create table emp (
  matr integer constraint emp_pk primary key,
  . . .,
  sup integer CONSTRAINT EMP_REF_SUP REFERENCES EMP DEFERRABLE,
  . . .);

SET CONSTRAINTS EMP_REF_SUP DEFERRED;

insert into emp values
  (7499, 'Biraud', 'commercial', 7698, '20/2/1981',
   12800.00, 2400.00, 30);

. . .

insert into emp values
  (7698, 'Leroy', 'directeur', null, '19/3/1988',
   15000.00, null, 30);

COMMIT SET CONSTRAINTS EMP_REF_SUP IMMEDIATE;
```

Les contraintes dans le dictionnaire des données

Voici une procédure SQL*PLUS Oracle pour faire afficher les contraintes en utilisant des tables du dictionnaire des données. Si vous voulez faire afficher en plus si la contrainte est différable, il suffit d'ajouter la colonne `DEFERRABLE` de la vue `USER_CONSTRAINTS`.

```
COLUMN TABLE FORMAT A9
COLUMN COLONNE FORMAT A9
COLUMN CONTRAINTE FORMAT A12
COLUMN "TABLE REF" FORMAT A9
COLUMN "COL REF" FORMAT A9
COLUMN "TEXTE CONTRAINTE" FORMAT A20

PROMPT 'Contraintes de type CLE PRIMAIRE ou UNIQUE :'
SELECT T.TABLE_NAME "TABLE",COLUMN_NAME "COLONNE",
       T.CONSTRAINT_NAME CONTRAINTE, T.CONSTRAINT_TYPE "TYPE"
FROM USER_CONSTRAINTS T, USER_CONS_COLUMNS C
WHERE T.CONSTRAINT_NAME = C.CONSTRAINT_NAME
  AND CONSTRAINT_TYPE  IN ('U', 'P')
/
```

```
PROMPT 'Contraintes de type REFERENCES :'
SELECT T.TABLE_NAME "TABLE", C.COLUMN_NAME "COLONNE",
       T.CONSTRAINT_NAME CONTRAINTE,
       DELETE_RULE,
       REF.TABLE_NAME "TABLE REF", REF.COLUMN_NAME "COL REF",
        'REFERENCE' "TYPE"
FROM USER_CONSTRAINTS T, USER_CONS_COLUMNS C, USER_CONS_COLUMNS REF
WHERE T.CONSTRAINT_NAME = C.CONSTRAINT_NAME
  AND REF.CONSTRAINT_NAME = T.R_CONSTRAINT_NAME
  AND CONSTRAINT_TYPE  = 'R'
/
PROMPT 'Contraintes de type CHECK ou NOT NULL :'
SELECT TABLE_NAME "TABLE", CONSTRAINT_NAME CONTRAINTE,
       SEARCH_CONDITION "TEXTE CONTRAINTE", CONSTRAINT_TYPE "TYPE"
FROM USER_CONSTRAINTS
WHERE CONSTRAINT_TYPE = 'C'
/
```

Chapitre 3

Langage de manipulation des données

Le langage de manipulation de données (LMD) 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).

3.1 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 3.1

(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 3.2

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

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

Remarque 3.1

Les dernières versions d'Oracle permettent de mettre des selects à la place des valeurs dans un values. Chaque select ne doit renvoyer qu'une seule ligne. Il faut entourer chaque select avec des parenthèses. Cette possibilité n'est pas portable et il faut donc l'éviter dans les programmes.

Exemple :

```
insert into emp (matr, nomE, dept)
values (
  (select matr + 1 from emp
   where nomE = 'Dupond'),
  'Dupondbis', 20)
```

3.2 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 4.6).

Exemples 3.3

(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'
```

3.3 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 4.6).

Exemple 3.4
```
DELETE FROM dept
WHERE dept = 10
```

3.4 Transactions

3.4.1 Généralités sur les transactions

Définitions

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.

Exemple 3.5

Une transaction peut transférer une somme d'argent entre deux comptes d'un client d'une banque. Elle comporte deux ordres : un débit sur un compte et un crédit sur un autre compte. Si un problème empêche le crédit, le débit doit être annulé.

Une transaction est terminée :
- soit par une validation qui entérine les modifications,
- soit par une annulation qui remet la base dans son état initial.

Dans le modèle "plat" des transactions (celui utilisé par SQL), deux transactions ne peuvent se chevaucher ni être emboîtées l'une dans l'autre : dans une session de travail, la transaction en cours doit se terminer avant qu'une nouvelle transaction ne puisse démarrer.

Ce mécanisme est aussi utilisé pour assurer l'intégrité de la base en cas de fin anormale d'une tâche utilisateur : il y a automatiquement annulation des transactions non terminées.

Propriétés des transactions

Il est facile de se souvenir des propriétés essentielles des transactions : elles sont "ACID".

Atomicité : un tout indivisible ;

Cohérence : une transaction doit laisser la base dans un état cohérent ; elle ne doit pas contredire une contrainte d'intégrité ou mettre les données dans un état anormal ;

Isolation : les modifications effectuées par une transaction ne doivent être visibles par les autres transactions que lorsque la transaction est validée ;

Durabilité : le SGBD doit garantir que les modifications d'une transaction validée seront conservées, même en cas de panne.

"AID" est du ressort du système transactionnel du SGBD.
"C" est du ressort de l'utilisateur (ou du programmeur) mais il est aidé
- par "I", car ainsi il n'a pas à considérer les interactions avec les autres transactions,
- par la vérification automatique des contraintes d'intégrité par le SGBD.

On verra dans l'étude de l'utilisation et de l'implémentation des SGBD que "I" est effectué par le système de contrôle de la concurrence et "AD" sont supportés par les procédures de reprise après panne.

3.4.2 Les transactions dans SQL

Dans la norme SQL, une transaction commence au début d'une session de travail ou juste après la fin de la transaction précédente. Elle se termine par un ordre explicite de validation (commit) ou d'annulation (rollback). Certains SGBD ne respectent pas la norme et demandent une commande explicite pour démarrer une transaction.

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.

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.

IMPORTANT : cette section étudie les transactions en mode de fonctionnement "normal" (voir 6.2). Dans ce mode "`READ COMMITED`" les insertions, modifications et suppressions qu'une transaction a exécutées n'apparaissent aux autres transactions que lorsque la transaction est validée. Tous les autres utilisateurs voient la base dans l'état où elle était avant la transaction. Des compléments sur les transactions sont données dans le chapitre 6, en particulier en 6.2 et 6.5.1.

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 nécessairement de rollback automatique de la transaction.

Remarque 3.2

Certains ordres SQL, notamment ceux de définitions de données (create table...), provoquent une validation automatique de la transaction en cours.

3.4.3 Autres modèles de transactions

Dans la norme SQL, la structure des transactions est plate et les transactions sont chaînées :

- 2 transactions ne peuvent se chevaucher ;
- une transaction commence dès que la précédente se termine.

Ce modèle n'est pas toujours adapté aux situations concrètes, surtout pour les transactions longues et multi-sites :

- elles peuvent être une source de frustration pour l'utilisateur si tout le travail effectué depuis le début de la transaction est annulée ;
- le fait que les transactions gardent jusqu'à leur fin les verrous qu'elles ont posés nuit à la concurrence.

D'autres modèles ont été proposés pour assouplir ce modèle.

Transactions emboîtées

Le modèle des transactions emboîtées permet à une transaction d'avoir des sous-transactions filles, qui elles-mêmes peuvent avoir des filles.

L'annulation d'une transaction parente annule toutes les transactions filles mais l'annulation d'une transaction fille n'annule pas nécessairement la transaction mère. Si la transaction mère n'est pas annulée, les autres transactions filles ne sont pas annulées.

À l'annulation d'une transaction fille, la transaction mère peut

- décider d'un traitement substitutif ;
- reprendre la transaction annulée ;
- s'annuler (si elle ne peut pas se passer de la transaction annulée) ;
- ou même ignorer l'annulation (si le traitement que devait effectuer la transaction annulée n'est pas indispensable).

Ainsi, un traitement effectué dans une transaction fille peut être repris ou être remplacé par un traitement alternatif pour arriver au bout du traitement complet. Ce modèle est bien adapté aux transactions longues et multi-sites (une transaction fille par site sur le réseau) qui doivent faire face à des problèmes de rupture de réseau.

SQL ne supporte pas ce modèle de transaction.

Points de reprise

Sans passer au modèle des transactions emboîtées, les dernières versions des principaux SGBD (et la norme SQL3) ont assoupli le modèle des transactions plates avec les points de reprise (appelés aussi points de sauvegarde ; *savepoint* en anglais).

On peut désigner des points de reprise dans une transaction :

```
savepoint nomPoint
```

On peut ensuite annuler toutes les modifications effectuées depuis un point de reprise s'il y a eu des problèmes :

```
rollback to nomPoint
```

On évite ainsi d'annuler toute la transaction et on peut essayer de pallier le problème au lieu d'annuler la transaction globale.

Chapitre 4

Interrogations

4.1 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 ...
```

4.2 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 ou par le mot clé AS (optionnel) ; il constituera le titre de la colonne dans l'affichage du résultat de la sélection. Ces noms ne peuvent être utilisés dans les autres clauses (where par exemple).

"*" 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 4.1

(a) `SELECT * FROM DEPT`

(b) `SELECT DISTINCT POSTE FROM EMP`

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

(d) La requête suivante va provoquer une erreur car on utilise le nom Salaire dans la clause where :

```
SELECT NOME, SAL + NVL(COMM,0) AS Salaire FROM EMP
WHERE Salaire > 1000
```

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 4.2

```
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 4.5).

4.2.1 select comme expression

La norme SQL-2, mais pas tous les SGBD, permet d'avoir des select emboîtés parmi les expressions. Il faut éviter cette facilité dans les programmes si on veut qu'ils soient portables. Rien n'empêche de l'utiliser en interactif si elle existe.

C'est possible avec Oracle :

```
select nomE, sal/(select sum(sal) from emp)*100
from emp
```

Un select "expression" doit ramener une valeur au plus. S'il ne ramène aucune valeur, l'expression est égale à la valeur `null`.

Si cette possibilité existe, le select emboîté peut même alors être synchronisé avec le select principal (le vérifier sur le SGBD que vous utilisez). La requête suivante affiche les noms des employés avec le nombre d'employés qui gagnent plus :

```
select nomE,
       (select count(*) from emp where sal > e1.sal) as rang
from emp e1
```

Attention, avec Oracle on ne peut utiliser rang dans la clause where (à vérifier sur votre SGBD). Par exemple, la requête suivante renverra une erreur :

```
select nomE,
       (select count(*) from emp where sal > e1.sal) as rang
from emp e1
where rang < 8
```

Si on veut contourner cette erreur, on peut le faire ainsi (mais il y a un moyen plus simple pour obtenir les 8 plus gros salaires) :

```
select nome, rang + 1
from (select nome,
             (select count(*) from emp
              where sal > e1.sal) as rang
      from emp e1)
where rang < 5
order by rang;
```

4.2.2 Pseudo-colonnes

Les SGBD fournissent souvent des pseudo-colonnes qui facilitent l'écriture de certaines requêtes.

Elles sont à éviter si on veut écrire des programmes portables.

Voici les pseudo-colonnes les plus utilisées d'Oracle :

rownum est le numéro des lignes sélectionnées par une clause where : le rownum de la première ligne a la valeur 1, celui de la deuxième la valeur 2, etc.

rownum permet de numéroter les lignes :

```
select rownum, nome from emp
where sal > 2000
```

Elle permet de restreindre le nombre de lignes renvoyées :

```
select rownum, nome from emp
where sal > 2000
  and rownum < 5
```

Attention,

```
select rownum, nome from emp
```

```
where sal > 2000
  and rownum > 5
```

ne renvoie aucune ligne puisque la première ligne sélectionnée par la clause where a un rownum égal à 1 et elle n'est donc pas sélectionnée. Cette condition "`rownum > 5`" ne peut donc être satisfaite.

On peut s'en sortir avec une sous-interrogation :

```
select nome
from (select rownum RANG, nome from emp where sal > 2000)
where RANG > 5
```

On peut ainsi récupérer les lignes numérotées entre deux nombres :

```
select nome from
  (select rownum RANG, nome from emp where sal > 2000)
where RANG between 3 and 8
```

Attention, le rang ne dépend pas d'une éventuelle clause `order by` ; la requête suivante ne donnera pas des lignes numérotées dans l'ordre :

```
select rownum, nome from emp
order by sal
```

rowid est une valeur qui permet de repérer une ligne parmi toutes les autres d'une table. Cette pseudo-colonne peut servir, par exemple, à ne garder qu'une ligne parmi plusieurs lignes qui ont des clés primaires identiques (ce qui ne devrait jamais arriver).

```
DELETE FROM emp EMP1
WHERE ROWID > (
  SELECT min(rowid) FROM EMP EMP2
  WHERE EMP1.matr = EMP2.matr)
```

sysdate est la date actuelle.

user est le nom de l'utilisateur courant.

uid est le numéro affecté par Oracle à l'utilisateur courant.

4.3 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 4.5.2 ou 4.6.3 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 schéma selon la norme SQL-2 : voir 5.1), 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 plus loin (remarque 4.5 page 34) une syntaxe spéciale pour les produits cartésiens.

Exemple 4.3

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 SQL2 (et les dernières versions d'Oracle) permet d'avoir un SELECT à la place d'un nom de table.

Exemples 4.4

(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 5.3). 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 donner un synonyme comme nom de table au select pour l'utiliser par ailleurs dans le select principal :

```
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 4.1

Ce select n'est pas une sous-interrogation ; on ne peut synchroniser le select avec les autres tables du from (voir 4.6.3).

4.4 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.

4.4.1 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.

Remarques 4.2

(a) L'utilisation des jokers ne fonctionne qu'avec LIKE ; elle ne fonctionne pas avec "=".

(b) Si on veut faire considérer "_" ou "%" comme des caractères normaux, il faut les faire précéder d'un caractère d'échappement que l'on indique par la clause ESCAPE. Par exemple, la requête suivante affiche les noms qui contiennent le caractère "%" :

```
select nome from emp
where nome like '%\%%' escape '\'
```

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 4.3
Le prédicat "`expr = NULL`" n'est jamais vrai, et ne permet donc pas de tester si l'expression est NULL.

4.4.2 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 4.5

(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.

4.5 Jointure

Quand on précise plusieurs tables dans la clause FROM, on obtient le produit cartésien des tables. Ce produit cartésien offre en général peu d'intérêt.

Ce qui est normalement souhaité, c'est de joindre les informations de diverses tables, en "recollant" les lignes des tables suivant les valeurs qu'elles ont dans certaines colonnes. Une variante de la clause FROM permet de préciser les colonnes qui servent au recollement.

On retrouve l'opération de jointure du modèle relationnel.

Exemple 4.6

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

```
SELECT NOME, NOMD
FROM EMP JOIN DEPT ON EMP.DEPT = DEPT.DEPT
```

La clause FROM 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.

Par opposition aux jointures externes que l'on va bientôt étudier, on peut ajouter le mot clé INNER :

```
SELECT NOME, NOMD
FROM EMP INNER JOIN DEPT ON EMP.DEPT = DEPT.DEPT
```

Remarque 4.4

Cette syntaxe SQL2 n'est pas supportée par tous les SGBD (par exemple, les versions d'Oracle antérieures à la version 9 ne la supportaient pas). La jointure peut aussi être traduite par la clause WHERE :

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

Cette façon de faire est encore très souvent utilisée, même avec les SGBD qui supportent la syntaxe SQL2.

Remarque 4.5

La norme SQL2 a aussi une syntaxe spéciale pour le produit cartésien de deux tables :

```
SELECT NOME, NOMD
FROM EMP CROSS JOIN DEPT
```

4.5.1 Jointure naturelle

Lorsque l'on a une équi-jointure (c'est-à-dire quand la condition de jointure est une égalité de valeurs entre une colonne de la première table et une colonne de la deuxième), il est le plus souvent inutile d'avoir les deux colonnes de jointure dans le résultat, puisque les valeurs sont égales. On peut préférer la jointure naturelle qui ne garde dans la jointure qu'une colonne pour les deux colonnes qui ont servi à la jointure. Évidemment, à moins d'utiliser "*" dans la clause du select, on ne voit pas la différence à l'affichage. Voici l'exemple précédent avec une jointure naturelle :

```
SELECT NOME, NOMD
FROM EMP NATURAL JOIN DEPT
```

On remarque qu'on n'a pas besoin dans cet exemple d'indiquer les colonnes de jointure car la clause "`natural join`" joint les deux tables sur toutes les colonnes qui ont le même nom dans les deux tables.

Remarque 4.6

Si on utilise une jointure naturelle, il est interdit de préfixer une colonne utilisée pour la jointure par un nom de table. La requête suivante provoque une erreur :

```
SELECT NOME, NOMD, DEPT.DEPT
FROM EMP NATURAL JOIN DEPT
```

Il faut écrire :

```
SELECT NOME, NOMD, DEPT
FROM EMP NATURAL JOIN DEPT
```

Pour obtenir une jointure naturelle sur une partie seulement des colonnes qui ont le même nom, il faut utiliser la clause "`join using`" (s'il y a plusieurs colonnes, le séparateur de colonnes est la virgule). La requête suivante est équivalente à la précédente :

```
SELECT NOME, NOMD
FROM EMP JOIN DEPT USING (DEPT)
```

4.5.2 Jointure d'une table avec 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 4.3), afin de pouvoir préfixer sans ambiguïté chaque nom de colonne.

Exemple 4.7

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

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

ou

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

4.5.3 Jointure externe

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

```
SELECT DEPT.DEPT, NOMD, NOME
FROM DEPT JOIN EMP ON 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 sans employés. On écrira alors :

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

La jointure externe ajoute des lignes fictives dans une des tables pour faire la correspondance avec les lignes de l'autre table. Dans l'exemple précédent, une ligne fictive (un employé fictif) est ajoutée dans la table des employés si un département n'a pas d'employé. Cette ligne aura tous ses attributs `null`, sauf celui des colonnes de jointure.

`RIGHT` indique que la table dans laquelle on veut afficher toutes les lignes (la table `dept`) est à droite de `RIGHT OUTER JOIN`. C'est dans l'autre table (celle de gauche) dans laquelle on ajoute des lignes fictives. De même, il existe `LEFT OUTER JOIN` qui est utilisé si on veut afficher toutes les lignes de la table de gauche (avant le "`LEFT OUTER JOIN`") et `FULL OUTER JOIN` si on veut afficher toutes les lignes des deux tables.

Remarque 4.7

Oracle n'accepte cette syntaxe que depuis la version 9. Pour les versions plus anciennes, la requête doit s'écrire :

```
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 fictive 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.

On peut aussi dire ça autrement : la table qui n'a pas le (+) correspond à la table qui aura chacune de ses lignes affichée, même si l'autre table n'a pas de ligne correspondante. On l'appellera la table dominante. Cette interprétation convient mieux pour retenir la syntaxe SQL2.

On ne peut ajouter de (+) des deux côtés ; on doit utiliser une union (voir 4.12.1) si on veut faire afficher les lignes qui n'ont pas de lignes correspondantes pour les deux tables.

4.5.4 Jointure "non équi"

Les jointures autres que les équi-jointures peuvent être représentées en remplaçant dans la clause ON ou la clause WHERE le signe "=" par un des opérateurs de comparaison (`< <= > >=`), ou encore `between` et `in`.

Exemples 4.8

(a) Liste des employés, avec tous les employés qui gagnent plus :

```
select emp.nome, emp.sal, empplus.nome, empplus.sal
from emp join emp empplus on emp.sal < empplus.sal
order by emp.sal
```

(b) 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
```

(c) Dans les anciennes versions d'Oracle, on écrit :

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

4.6 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 EMP.NOME
FROM EMP JOIN EMP MARTIN ON EMP.POSTE = MARTIN.POSTE
WHERE 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.

4.6.1 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 4.9

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 4.10

(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 4.11

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

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

On peut aussi plus simplement utiliser la jointure naturelle puisque les noms des colonnes de jointures sont les mêmes :

```
SELECT NOME, POSTE
FROM EMP NATURAL JOIN DEPT
WHERE LIEU = 'LYON'
  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.

4.6.2 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 4.12

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 4.8

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

4.6.3 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 4.13

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.

4.6.4 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 4.14

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 dernier exemple des exemples 4.18 page 46).

4.6.5 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 4.15

```
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 4.9

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 4.16. 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, le plus souvent en simplifiant le select externe en remplaçant une jointure de tables par une seule table.

Exemple 4.16

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_N{Matr} EMP) [Dept, CodeP] ("J_N{Matr}" indique une jointure naturelle sur l'attribut Matr) et S = PROJET [CodeP]

Il reste à faire la traduction "mot à mot" en SQL :

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

Remarque 4.10

Il faudrait ajouter DISTINCT dans le premier select pour éviter les doublons.

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 NATURAL JOIN EMP
      WHERE DEPT = DEPT.DEPT
        AND CODEP = PROJET.CODEP))
```

Remarque 4.11

Dans le cas où il est certain que la table dividende (celle qui est divisée)

ne contient dans la colonne qui sert pour la division que des valeurs qui existent dans la table diviseur, on peut exprimer la division en utilisant les regroupements (étudiés dans la prochaine section) et en comptant les lignes regroupées. Pour l'exemple des départements qui participent à tous les projets, on obtient :

```
select dept
from emp natural join participation
group by dept
having count(distinct codeP) =
  (select count(distinct codeP) from projet)
```

Traduction : si le nombre des `codeP` associés à un département donné est égal au nombre des tous les `codeP` possibles, ça signifie que ce département est associé à tous les départements. Ici on a bien le résultat cherché car les `codeP` du select sont nécessairement des `codeP` de la table des projets (clé étrangère de `PARTICIPATION` qui référence la clé primaire de la table `PROJET`). Si un ensemble A est inclus dans un ensemble B et si A a le même nombre d'éléments que B, c'est que A = B.

Mais il ne faut pas oublier que dans des requêtes complexes les données qui interviennent dans les divisions peuvent provenir de requêtes emboîtées. Il n'y a alors pas nécessairement de contraintes de référence comme dans l'exemple traité ici (contrainte qui impose que `codeP` doit nécessairement correspondre à un `codeP` dans la table Projet). Si on n'a pas $A \subseteq B$, le fait que A et B aient le même nombre d'éléments ne signifie pas que A = B.

S'il peut y avoir dans la colonne qui sert pour la division des valeurs qui n'existent pas dans la table diviseur, la requête est légèrement plus complexe :

```
select dept
from emp natural join participation
where codeP in
  (select codeP from projet)
group by dept
having count(distinct codeP) =
  (select count(distinct codeP) from projet)
```

4.7 Fonctions de groupes

Les fonctions de groupes peuvent apparaître dans le Select ou le Having (voir 4.9) ; 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 4.17

(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)
```

4.8 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 4.18

(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) IN
  (SELECT DEPT, MAX(SAL) FROM EMP
   GROUP BY DEPT)
```

RESTRICTION :

Une expression d'un SELECT avec clause GROUP BY ne peut évidemment que correspondre à une caractéristique de groupe. SQL n'est pas très "intelligent" pour comprendre ce qu'est une caractéristique de groupe ; une expression du SELECT ne peut être que :

- soit une fonction de groupe,
- soit une expression figurant dans le GROUP BY.

L'ordre suivant est invalide car NOMD n'est pas une expression du GROUP BY :

```
SELECT NOMD, SUM(SAL)
FROM EMP NATURAL JOIN DEPT
GROUP BY DEPT
```

Il faut, soit se contenter du numéro de département au lieu du nom :

```
SELECT DEPT, SUM(SAL)
FROM EMP NATURAL JOIN DEPT
GROUP BY DEPT
```

Soit modifier le GROUP BY pour avoir le nom du département :

```
SELECT NOMD, SUM(SAL)
FROM EMP NATURAL JOIN DEPT
GROUP BY NOMD
```

4.9 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 4.19

```
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 NATURAL JOIN DEPT
WHERE 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.

4.10 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 ; elles peuvent aussi avoir une syntaxe différente,ou même un autre nom.

4.10.1 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 4.10.2)
TO_NUMBER(*chaîne*)	convertit la chaîne de caractères en numérique

Exemple 4.20

Calcul du salaire journalier :

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

4.10.2 Fonctions chaînes de caractères

Rappel de 1.8.1 : il est possible de concaténer des chaînes avec l'opérateur "||" ;

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 4.21

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". Si l'ensemble des caractères n'est pas donné, ce sont les espaces qui sont enlevés.

RTRIM(*chaîne*, *car*)
a une fonction analogue, les caractères étant supprimés à l'extrémité droite de la chaîne. Si l'ensemble des caractères n'est pas donné, ce sont les espaces qui sont enlevés.

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 4.22

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 (1 pour lundi, 7 pour dimanche)
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 4.23

```
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.

Remarque 4.12

Attention à l'interprétation par Oracle des dates ne contenant que 2 chiffres pour l'année. Toute année inférieure à 50 est considérée comme une année du 21ème siècle ; les autres années sont considérées comme étant des années du 20ème siècle. Par exemple, 38 correspond à 2038 et 50 correspond à 1950.

Par défaut Oracle affiche les années avec 2 chiffres. Pour savoir à coup sûr de quelle année il s'agit, il faut utiliser la fonction `TO_CHAR` :

```
select to_char(date_naissance, 'DD/MM/YYYY') from personne
```

4.10.3 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 ou **CURRENT_DATE**
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
```

4.10.4 Fonction de choix (CASE)

Une "fonction" de choix existe dans la norme SQL2 (et dans Oracle depuis la version 9i). Elle correspond à la structure `switch` du langage C (ou `case` de Pascal ou Ada).

Elle remplace avantageusement la fonction `decode` d'Oracle (qui n'est pas dans la norme SQL2).

Il existe deux syntaxes pour `CASE` : une qui donne une valeur suivant des conditions quelconques et une qui donne une valeur suivant la valeur d'une expression.

```
CASE
  WHEN condition1 THEN expression1
  [WHEN condition2 THEN expression2]
  . . .
  [ELSE expression_défaut]
END
```

Remarques 4.13

(a) La condition qui suit un `WHEN` peut être n'importe quelle expression booléenne.

(b) Si aucune condition n'est remplie et s'il n'y a pas de ELSE, `null` est retourné.

(c) Les différentes expressions renvoyées doivent être de même type.

```
CASE expression
  WHEN valeur1 THEN expression1
  [WHEN valeur2 THEN expression2]
  . . .
  [ELSE expression_défaut]
END
```

Remarques 4.14

(a) Attention, ne fonctionne pas pour le cas où une des valeurs n'est pas renseignée (`when null then`). Pour ce cas, il faut utiliser l'autre syntaxe de `case` : `when colonne is null then`.

(b) Si l'expression n'est égale à aucune valeur et s'il n'y a pas de ELSE, `null` est retourné.

(c) Les différentes expressions renvoyées doivent être de même type.

Exemple 4.24

Liste des employés avec leur catégorie (président = 1, directeur = 2, autre = 3), en appelant la colonne "Niveau" :

```
SELECT nome,
  CASE
    WHEN (poste  = 'Président') THEN 1
    WHEN (poste  = 'Directeur') THEN 2
    ELSE 3
  END Niveau
FROM emp;
```

pourrait aussi s'écrire plus simplement

```
SELECT nome,
  CASE poste
    WHEN 'Président' THEN 1
    WHEN 'Directeur' THEN 2
    ELSE 3
  END Niveau
FROM emp;
```

Pour les versions d'Oracle antérieures à la version 9i, il est possible d'utiliser la fonction `decode`.

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 4.25

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

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

4.10.5 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`

4.11 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 4.2).

Exemple 4.26

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 4.27

(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"
```

4.12 Opérateurs ensemblistes

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

4.12.1 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.

Si on veut conserver les doublons, on peut utiliser la variante `UNION ALL`.

Exemple 4.28

Liste des ingénieurs des deux filiales :

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

4.12.2 Opérateur INTERSECT

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

Exemple 4.29

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

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

4.12.3 Opérateur EXCEPT

L'opérateur EXCEPT de SQL2 (ou MINUS pour Oracle) permet d'ôter d'une sélection les lignes obtenues dans une deuxième sélection.

Exemple 4.30

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

```
SELECT DEPT FROM EMP1
EXCEPT
SELECT DEPT FROM EMP2
```

4.12.4 Clause ORDER BY

On ne peut ajouter une clause `ORDER BY` que sur le dernier select.

4.13 Limiter le nombre de lignes renvoyées

Aucune méthode standardisée ne permet de limiter le nombre de lignes renvoyées par un select mais la plupart des SGBDs offre cette facilité.

Voici quelques exemples qui montrent comment quelques SGBDs limitent à 10 le nombre de lignes renvoyées :

Avec MySQL et Postgresql :

```
SELECT matr, nomE FROM emp
LIMIT 10
```

Avec Oracle :

```
SELECT matr, nomE FROM emp
WHERE ROWNUM <= 10
```

Avec SQL Server :

```
SELECT TOP 10 matr, nomE FROM emp
```

La solution d'Oracle n'est pas facile à manipuler car `rownum` numérote avant un éventuel tri (ce qui n'est pas le cas avec limit ou top) comme on l'a vu en 4.2.2.

Si un select comporte une clause order by il faut donc ruser avec Oracle en utilisant une sous-requête qui trie, alors que la requête principale limite le nombre de lignes :

```
select * from
  (select nomE, sal
   from emp
   order by sal)
where rownum <= 10
```

4.14 Injection de code SQL

Tous les langages de programmation permettent de lancer des requêtes SQL pour interagir avec une base de données. Il est le plus souvent possible de construire une requête SQL dont le texte contient une partie entrée par l'utilisateur (par concaténation de chaînes de caractères). Il est alors indispensable de se prémunir contre ce qui est appelé "l'injection de code SQL". Ce problème arrive lorsque l'utilisateur s'arrange pour changer le sens de la requête SQL qui va être construite avec la valeur qu'on lui demande de saisir.

Exemple 4.31

Un programme demande son nom et son mot de passe à un utilisateur, et les range dans 2 variables `nom` et `mdp`.

Il lance la requête suivante et accepte l'utilisateur si elle renvoie bien une ligne (on suppose que "+" effectue une concaténation de chaînes de caractères dans le langage utilisé) :

```
select * from utilisateur"
+ " where nom = '" + nom + "' and mdp ='" + mdp + "'"
```

Un utilisateur malhonnête sait que le SGBD utilisé par le programme est Oracle et que Dupond est un utilisateur, mais il ne connaît pas son mot de passe. Il peut contourner cette vérification en saisissant la valeur "`Dupond' --`" pour son nom, et n'importe quoi pour son mot de passe, par exemple "`xxx`".
En effet, la requête générée par la concaténation est :

```
select * from utilisateur
where nom = 'Dupond' --' and mdp ='xxx'
```

Or, avec Oracle, "`--`" introduit un commentaire en fin de ligne. Le select qui sera exécuté par Oracle sera donc le suivant :

```
select * from utilisateur
where nom = 'Dupond'
```

Ce select renverra bien une ligne et le programme laissera entrer cet utilisateur sous le nom "Dupond".

Il existe de nombreuses variantes pour tromper un programme comme dans l'exemple précédent.

La parade est de toujours vérifier la saisie d'un utilisateur avant de s'en servir pour construire une requête SQL.

Pour l'exemple précédent, il aurait suffit d'interdire le caractère "'" ou de le doubler pour qu'il perde son sens de délimiteur de chaîne de caractères.

La saisie de l'utilisateur aurait été transformée en "`Dupond'' --`" et le select aurait été

```
select * from utilisateur
where nom = 'Dupond'' --' and mdp ='xxx'"
```

et n'aurait renvoyé aucune ligne.

Le plus souvent il suffit de s'appuyer sur les API fournies avec les langages pour accéder à une base de données. Ces API offrent souvent des facilités pour se protéger contre l'injection de code SQL. Avec JDBC par exemple (API Java pour accéder à une base de données), il suffit d'utiliser des objets de type `PreparedStatement` et des paramètres pour contenir les noms et mots de passe donnés par l'utilisateur, au lieu de créer un `Statement` et de construire la requête en concaténant des chaînes de caractères. Les caractères spéciaux comme "'" sont alors traités spécialement et inclus comme tous les

autres caractères dans la valeur des paramètres (pour l'exemple précédent, cela revient à doubler l'apostrophe).

Chapitre 5

Langage de définition des données

Le langage de définition des données (LDD est la partie de SQL qui permet de décrire les tables et autres objets manipulés par le SGBD.

5.1 Schéma

Un schéma est un ensemble d'objets (tables, vues, index, autorisations, etc...) gérés 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 SQL2 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 SQL2, le catalogue, est un ensemble de schémas. Un catalogue doit nécessairement comprendre un schéma particulier qui correspond au dictionnaire des données (voir 5.8).

5.2 Tables

5.2.1 CREATE TABLE AS

La commande CREATE TABLE a déjà été vue au chapitre 2.1. Une variante (d'Oracle mais pas de la norme SQL2) 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 5.1

```
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 ne pas donner les noms et type des colonnes de la table créée. Dans ce cas les colonnes de cette table auront les mêmes noms, types et tailles que celles de l'interrogation :

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

5.2.2 ALTER TABLE

Les sections 2.2.2 et 2.2.3 montrent comment gérer les contraintes d'intégrité avec la commande ALTER TABLE. Cette commande permet aussi de gérer les colonnes d'une table : ajout d'une colonne (après toutes les autres colonnes), suppression et modification d'une colonne existante.

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).

Il est possible de définir des contraintes de colonne.

Exemple 5.2

```
alter table personne
add (email_valide char(1)
constraint personne_email_valide check(email_valide in ('o', 'n')))
```

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 contienne 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.

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.

Exemple 5.3

```
alter table personne
modify (
  prenoms null,
  nom varchar(50))
```

On peut donner une contrainte de colonne dans la nouvelle définition de la colonne.

Exemple 5.4

```
alter table personne modify (
  sexe char(1)
  constraint personne_sexe_ck check(sexe in ('m', 'f')))
```

Suppression d'une colonne - DROP COLUMN

Oracle n'a ajouté cette possibilité que depuis la version 8i. Auparavant, il fallait se contenter de mettre toutes les valeurs de la colonne à NULL (si on voulait récupérer de la place). On pouvait aussi se débarrasser de la colonne en créant une nouvelle table sans la colonne en copiant les données par "create table as", et en renommant la table du nom de l'ancienne table.

```
ALTER TABLE table
DROP COLUMN col
```

La colonne supprimée ne doit pas être référencée par une clé étrangère ou être utilisée par un index.

Renommer une colonne - RENAME COLUMN

```
ALTER TABLE table
RENAME COLUMN ancien_nom TO nouveau_nom
```

permet de renommer une table.

Renommer une table - RENAME TO

```
ALTER TABLE ancien_nom
RENAME TO nouveau_nom
```

permet de renommer une table.

Oracle offre une commande équivalente pour renommer une table :

```
RENAME ancien_nom TO nouveau_nom
```

5.2.3 Supprimer une table - 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.

Il est impossible de supprimer une table si la table est référencée par une contrainte d'intégrité référentielle. Une variante Oracle (pas SQL2) de la commande permet de supprimer les contraintes d'intégrité et la table :

```
DROP TABLE table CASCADE CONSTRAINTS
```

5.2.4 Synonyme public de table ou de vue

Si une table ou une vue (étudié en 5.3) doit être utilisée par plusieurs utilisateurs, il peut être intéressant de lui donner un synonyme public pour que les utilisateurs ne soient pas obligés de préfixer le nom de la table ou de la vue par le nom de son créateur. Oracle offre la commande `create public synonym` pour cela.

Exemple 5.5

```
CREATE PUBLIC SYNONYM employe FOR toto.emp
```

Pour chercher des synonymes dans le dictionnaire des données :

```
select synonym_name, table_name, owner
from all_synonyms
where table_owner = 'TOTO'
```

5.3 Vues

Une vue est une vision partielle ou particulière des données d'une ou plusieurs tables de la base.

La définition d'une vue est donnée par un SELECT qui indique les données de la base qui seront vues.

Les utilisateurs pourront consulter la base, ou modifier la base (avec certaines restrictions) à travers la vue, c'est-à-dire manipuler les données renvoyées par la vue comme si c'était des données d'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.

5.3.1 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 ...
```

Exemples 5.6

(a) Vue ne comportant que le matricule, le nom et le département des employés :

```
CREATE VIEW EMP2 (MATR, NOM, DEPT)
AS SELECT MATR, NOM, DEPT FROM EMP
```

(b) Il est possible de créer un vue de vue :

```
CREATE VIEW EMP3 (MATR, NOM, DEPT)
AS SELECT MATR, NOM FROM EMP2
```

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 résultats du SELECT. Si certaines colonnes résultats 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 5.7

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

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

Remarque 5.1

Si l'ordre create de l'exemple ci-dessus était inséré dans un programme, il serait 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.

5.3.2 DROP VIEW

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

5.3.3 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 5.8

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

Remarque 5.2

Les restrictions données ci-dessus sont les restrictions de la norme SQL2. Elles sont parfois trop strictes et les SGBD peuvent assouplir ces

restrictions. C'est le cas d'Oracle qui permet de modifier les données d'une table sous-jacente par l'intermédiaire d'une vue qui comporte une jointure lorsque la vue "préserve" la clé de la table. Sans entrer dans les détails, voici un exemple du vue qui permet, sous Oracle, de modifier la table EMP :

```
CREATE VIEW EMP2 AS
SELECT matr, nomE, nomD
FROM EMP NATURAL JOIN DEPT
```

L'instruction suivante est permise sous Oracle :

```
UPDATE EMP
SET nomE = 'Dupont'
where nomE = 'Dupond'
```

5.3.4 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. En fait, si les données ne sont pas modifiables à travers la vue (voir 5.3.3, il faudra utiliser des trigger "instead of" (voir **??**) pour utiliser pleinement la vue comme une table ; sinon la vue ne pourra être utilisée que pour récupérer des données mais pas pour les modifier.

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 5.9.1.

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 5.9

En créant la vue

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

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

```
SELECT * FROM EMPSAL
```

5.4 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.

5.4.1 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 5.3

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).

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.

5.4.2 Utilité des index

Les requêtes SQL sont transparentes au fait qu'il existe un index ou non. C'est l'optimiseur de requêtes 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 (avec plusieurs colonnes) 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".

Un autre problème est que la modification des données provoque la position de verrou sur une partie de l'index pour éviter les problèmes liés aux accès multiples aux données. Les accès aux données en peuvent donc être ralentis.

On peut dire qu'il n'est pas toujours facile de savoir si un index doit être créé ou non. Si la table est rarement modifiée un index occupe seulement de la place et c'est l'optimiseur de requêtes qui choisira s'il l'utilise ou non. Les inconvénients sont donc minimes en ce cas. Il n'en est pas de même si la table est souvent modifiée. Il ne faudra alors créer un index que si on pense qu'il améliorera vraiment les performances pour les requêtes les plus courantes ou les plus critiques. On a vu ci-dessus des règles générales pouvant aider à faire un choix, par exemple, un index ne donnera pas d'amélioration pour une requête si plus de 20 % des lignes sont récupérées. Ces règles ne sont pas à prendre au pied de la lettre et il faut souvent les adapter aux

situations particulières. Il faut aussi consulter les notes de version des SGBD qui donnent d'autres règles qui dépendent des versions des optimiseurs de requêtes. Il peut aussi se poser le choix du type d'index (voir 5.4.4).

5.4.3 DROP INDEX

Un index peut être supprimé par la commande DROP INDEX :

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

Le nom de la table ne doit être précisé que si vous voulez supprimer un index d'une table d'un autre utilisateur alors que vous possédez un index du même nom.

Remarque 5.4

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

5.4.4 Types d'index

Le principe de tous les index est le même : un algorithme permet une recherche rapide dans l'index de la clé d'une ligne que l'on cherche et l'index fournit alors un "pointeur" vers la ligne correspondante. Ce pointeur est une valeur qui détermine la position de la ligne dans les fichiers utilisés par le SGBD ; pour Oracle cette valeur se nomme le ROWID.

La plupart des index sont implémentés avec des B+-arbres. Les B+-arbres sont une variante de B-arbres dont toutes les clés sont dans les feuilles et dont les feuilles forment une liste chaînée. Ce chaînage permet un parcours rapide des clés dans l'ordre des clés enregistrées dans l'index.

Les B-arbres sont des arbres équilibrés : les algorithmes d'ajout et de suppression assurent que toutes les branches auront une même profondeur. Il est théoriquement inutile de les réorganiser périodiquement (mais de nombreux administrateurs de SGBD estiment que la reconstruction des index améliorent les performances). On sait que des arbres déséquilibrés ont des mauvaises performances pour effectuer des recherches (au pire, un arbre peut n'avoir qu'une seule branche et avoir les performances d'une simple liste).

Pour un SGBD, l'avantage essentiel de ces arbres est qu'ils réduisent énormément le nombre d'accès disque. Chaque nœud de l'arbre contient de nombreuses clés et à chaque clé d'un nœud on associe un pointeur vers une nœud qui contient les clés qui suivent cette clé. On s'arrange pour avoir des nœuds de la bonne taille pour qu'une lecture disque récupère un nœud entier. On arrive ainsi en très peu de lectures du disque à localiser les lignes que

l'on cherche. On rappelle que les accès disques sont environ un million de fois plus lents que les accès en mémoire centrale.

Un autre avantage est que l'on peut très facilement lire les clés dans l'ordre. L'application d'une clause "order by" peut ainsi bénéficier de ces index.

Il existe aussi d'autres types d'index qui peuvent être utilisés dans des circonstances particulières ; les plus courants sont les suivants :

Index bitmap : ils ne sont utiles que lorsque les données de la table ne sont presque jamais modifiées. Ils sont le plus souvent utilisés dans les applications décisionnelles OLAP (*On Line Analytical Processing*) qui facilitent les prises de décisions liées à l'analyse des données conservées par une entreprise. L'implémentation est très différente de celle des B-arbres. Pour chaque valeur distincte de la colonne indexée l'index contient un tableau de bits (autant de bits que de lignes dans la table) indiquant si chaque ligne de la table a cette valeur. Il est facile de combiner différents tableaux de bits pour répondre aux sélections de type "`col1=valeur1 and col2=valeur2`" où `col1` et `col2` sont indexées par un index bitmap. Lorsque les index bitmap sont utilisés par l'optimiseur ils peuvent fournir de très bonnes performances. Oracle a aussi introduit un nouveau type d'index bitmap pour accélérer les jointures.

Index de type table de hachage : ils ne sont plus vraiment utilisés car ils n'offrent pas d'avantages conséquents par rapport aux B-arbres. Ils offrent un accès quasi instantané à une ligne dont on donne la clé mais ils ne permettent pas le parcours dans l'ordre des clés et ne sont donc d'aucune utilité pour les sélections du type "`col1 > valeur1`". On trouve ce type d'index dans PostgreSQL mais pas dans Oracle.

Index sur des valeurs de fonction : ils peuvent être utilisés pour accélérer les recherches du type "`col1 > fonction(...)`". Ce sont les résultats des valeurs d'une fonction sur toutes les lignes de la table qui sont indexés.

5.4.5 Dictionnaire des données

L'utilisateur peut avoir des informations sur les index qu'il a créés en consultant la table `USER_INDEX` du dictionnaire des données. La table `ALL_INDEX` lui donne les index qu'il peut utiliser (même s'il ne les a pas créées).

5.5 Génération de clés primaires

On sait qu'il n'est pas recommandé d'avoir des identificateurs significatifs. Par exemple, si on identifie les lignes d'une table d'employés par les noms des employés, toute modification du nom (erreur dans le nom, mariage,...) entraînera des traitements pour modifier les clés étrangères qui se réfèrent à un employé. Il est préférable d'avoir des identificateurs non significatifs, par exemple un nombre entier, dont la valeur n'est pas liée aux propriétés des employés.

5.5.1 Utilisation de tables de la base

Dans les premiers temps des SGBD, les développeurs utilisaient des valeurs enregistrées dans des tables de la base pour créer les identificateurs dont ils avaient besoin. L'idée est de créer une suite de nombres entiers de telle sorte que jamais le même nombre n'est utilisé deux fois, même si deux transactions distinctes ont besoin d'un identificateur en même temps. Voici les procédés les plus courants :

- Le premier identificateur des lignes d'une table est égal à 1. Pour trouver les identificateurs suivants il suffit d'ajouter 1 à la plus grande valeur de la table ; par exemple,
  `select max(matr) + 1 from employe`
  Cette requête a déjà un coût non négligeable. Mais pour éviter les problèmes bien connus des accès concurrents (cf. 6) il est en plus nécessaire de bloquer toute la table pendant la récupération du maximum.
  Cette technique est beaucoup trop coûteuse dans les environnements à forte concurrence. De plus, si on veut garder un historique des données enregistrées dans la base, on peut se retrouver avec des lignes qui ont eu le même identificateur (il suffit que la ligne de plus grand identificateur soit supprimée à un moment donné).
- Pour éviter de bloquer la table des employés, une autre technique est de créer une table qui ne contient qu'une seule ligne d'une seule colonne. Cette colonne contient la valeur du prochain identificateur à utiliser. Pour les mêmes raisons que pour la solution précédente on est obligé de bloquer la table pendant l'attribution d'un identificateur. Mais l'avantage est que l'on ne bloque pas toute la table des employés, que l'on n'a pas à effectuer une recherche de maximum, que cette table est petite et qu'elle sera donc le plus souvent entièrement en mémoire cache du SGBD. Une telle table peut fournir les identificateurs à toutes les tables (ou à plusieurs tables) ou elle peut ne fournir les identificateurs qu'à une seule table. Dans le premier cas, les attentes (dues au blocage de

la table à chaque attribution d'un identificateur) sont accentuées.
Une variante est d'avoir une seule table pour fournir les identificateurs à plusieurs table, mais cette table contient autant de lignes que de table à qui elle fournit un identificateur. Cette table a deux colonnes : une pour le nom de la table à qui l'identificateur est fourni et l'autre qui contient la valeur courante de l'identificateur. En ce cas, seulement une ligne est bloquée au moment de l'attribution d'un identificateur (par un select for update).

5.5.2 Autres solutions

Tous les SGBD offrent désormais une facilité pour obtenir des identificateurs sans avoir à accéder à une table. Cette facilité permet d'obtenir des valeurs qui sont générées automatiquement par le SGBD. Cette facilité n'est malheureusement pas standardisée ; par exemple,

- MySQL permet d'ajouter la clause `AUTO_INCREMENT` à une colonne ;
- DB2 et SQL Server ont une clause `IDENTITY` pour dire qu'une colonne est un identifiant ;
- Oracle, DB2 et PostgreSQL utilisent des séquences.

La section suivante présente les séquences.

5.5.3 Séquences

Création d'une séquence

La syntaxe Oracle propose de nombreuses options. Voici les plus utilisées :

```
CREATE SEQUENCE nom_séquence
[INCREMENT BY entier1]
[START WITH entier2]
```

Par défaut, la séquence commence à 1 et augmente de 1 à chaque fois.

D'autres options permettent de choisir des séquences décroissantes, de choisir une limite supérieure ou inférieure, de recycler les valeurs, de respecter l'ordre des requêtes (utile pour les traitements optimistes des accès concurrents), etc.

Exemple 5.10

```
create sequence seqdept
increment by 10
start with 10
```

Remarque 5.5

Utiliser un incrément supérieur à 1 permet de disposer de plusieurs identificateurs en un seul appel, pour améliorer les performances.

Utilisation d'une séquence

Deux pseudo-colonnes permettent d'utiliser les séquences :

CURRVAL retourne la valeur courante de la séquence ;

NEXTVAL incrémente la séquence et retourne la nouvelle valeur.

Exemple 5.11

```
insert into dept(dept, nomd, lieu)
values (seqdept.nextval, 'Finances', 'Nice')
```

Pour voir la valeur d'une séquence, on utilise `currval` avec la table `dual` (voir 1.5.2) : `select seqdept.currval from dual`.

Remarques 5.6

(a) `currval` n'est pas défini tant qu'on n'a pas appelé au moins une fois `nextval` dans la session de travail.

(b) La valeur de `currval` ne dépend que des `nextval` lancés dans la même transaction.

(c) `nextval` modifie immédiatement les valeurs futures renvoyées par les `nextval` des autres sessions, même s'il est lancé dans une transaction non validée.

Modification d'une séquence

```
ALTER SEQUENCE nom_séquence
INCREMENT BY entier
```

Exemple 5.12

```
alter sequence seqdept increment by 5
```

On ne peut pas modifier le numéro de démarrage de la séquence.

Suppression d'une séquence

```
DROP SEQUENCE nom_séquence
```

Dictionnaire des données

L'utilisateur peut avoir des informations sur les séquences qu'il a créées en consultant la table `USER_SEQUENCES` du dictionnaire des données. La table `ALL_SEQUENCES` lui donne les séquences qu'il peut utiliser (même s'il ne les a pas créées).

5.6 Procédure et fonction stockée

5.6.1 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 5.8).

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 standardisé et ils sont donc liés à chacun des SGBD. Les procédures stockées d'Oracle peuvent aussi être écrites en Java et en langage C. Le langage utilisé dans les exemples donnés dans cette section est le langage PL/SQL d'Oracle.

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. De plus les erreurs sont repérées dès la compilation et pas à l'exécution.
- Les développeurs n'ont pas à connaître les détails de l'exécution des procédures. Une procédure fonctionne en "boîte noire". L'écriture et la maintenance des applications sont donc facilitées.
- 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.

Le principal inconvénient des procédures stockées est qu'elles impliquent une dépendance forte vis-à-vis du SGBD car chaque SGBD a sa propre syntaxe et son propre langage de programmation.

En Oracle, on peut créer une nouvelle procédure stockée par la commande CREATE PROCEDURE (CREATE OR REPLACE PROCEDURE si on veut écraser une éventuelle précédente définition).

Exemple 5.13

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 decimal,
   cout out decimal) is
begin
  select sum(sal) * pourcentage / 100
  into cout
  from emp
  where dept = unDept;

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

Remarques 5.7

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

(b) Les erreurs de compilation des procédures stockées peuvent être vues sous Oracle par la commande SQL*PLUS "`show errors`". On peut aussi voir ces erreurs en utilisant le dictionnaire des données (voir 5.6.3).

5.6.2 Fonction stockée

La différence entre une fonction et une procédure est qu'une fonction renvoie une valeur.

Le code est semblable au code d'une procédure stockée mais la déclaration de la fonction indique le type de la valeur retournée et seuls les paramètres de type `IN` sont acceptés.

Exemple 5.14

Le code suivant crée une fonction qui permet le calcul en franc d'un montant en euros :

```
create or replace function euro_to_fr(somme in number) return number
is
  taux constant numeric(6,5) := 6.55957;
begin
  return somme * taux;
end;
```

Ces fonctions peuvent alors être utilisées comme les fonctions prédéfinies dans les requêtes SQL.

On peut ensuite utiliser la fonction :

```
select nome, sal, round(euro_to_fr(sal), 2) from emp
```

5.6.3 Tables du dictionnaire

Pour avoir les noms des procédures on peut utiliser les vues `USER_PROCEDURES` ou `ALL_OBJECTS`. La vue `USER_SOURCE` donne les sources des procédures (chaque ligne de la table contient une ligne de code).

Exemples 5.15

(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) Le code de la procédure `augmentation` est donné par (Oracle stocke les noms des procédures et fonctions en majuscules) :

```
select text from user_source
where name = 'AUGMENTATION'
order by line
```

(c) Pour voir les messages d'erreur de la compilation d'une procédure ou fonction nommée `euro_to_fr` :

```
select * from user_errors
where name = 'EURO_TO_FR'
order by line
```

5.7 Triggers

Les *triggers* (déclencheurs en français) ressemblent aux procédures stockées car ils sont eux aussi compilés et enregistrés dans le dictionnaire des données de la base et ils sont le plus souvent écrits dans le même langage. La différence est que leur exécution est déclenchée automatiquement par des événements liés à des actions sur la base. Les événements déclencheurs peuvent être les commandes LMD insert, update, delete ou les commandes LDD create, alter, drop.

Les *triggers* complètent les contraintes d'intégrité en permettant des contrôles et des traitements plus complexes. Par exemple, on peut implanter la règle qu'il est interdit de baisser le salaire d'un employé (on peut toujours rêver). Pour des contraintes très complexes, des procédures stockées des procédures stockées peuvent encapsuler des requêtes SQL.

Ils peuvent aussi être utilisés pour d'autres usages comme de mettre à jour des données de la base suite à une modification d'une donnée.

Si l'exécution d'un trigger provoque une erreur, par exemple s'il viole une contrainte d'intégrité, la requête qui l'a déclenché est annulée (mais pas la transaction en cours). Les actions effectuées par un trigger font partie de la même transaction que l'action qui les a déclenchés. Un rollback va donc annuler aussi tout ce qu'ils ont exécuté.

Les *triggers* sont normalisés dans la norme SQL3.

Ils se créent avec la commande `create or replace trigger` dont des exemples sont donnés ci-dessous.

La syntaxe :

```
CREATE [OR REPLACE] TRIGGER nom-trigger
{BEFORE | AFTER } {INSERT | DELETE | UPDATE [OF col1, col2,...]}
ON {nom-table | nom-vue}
[REFERENCING ...]
[FOR EACH ROW]
[WHEN condition]
bloc PL/SQL
```

Pour supprimer un trigger :

```
drop trigger nomTrigger;
```

Sous Oracle, comme pour les procédures stockées, les erreurs de compilation des triggers s'affichent avec la commande SQL*PLUS "`show errors`".

Remarque 5.8

Toutes les possibilités offertes par les triggers ne sont pas décrites ici et chaque SGBD peut ajouter des fonctionnalités. Pour plus d'informations consultez la documentation de votre SGBD.

Remarque 5.9

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, qui ne sont 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).

5.7.1 Types de triggers

Il peut y avoir des triggers de 3 types
- INSERT
- DELETE
- UPDATE

Le type indique la requête SQL qui va déclencher l'exécution du trigger. Un trigger peut être lancé avant (`BEFORE`) ou après (`AFTER`) la requête déclenchante.

Pour UPDATE, on peut aussi optionnellement préciser une colonne. Le trigger ne sera déclenché que si la valeur de cette colonne a été modifié.

L'option "for each row" est facultative ; elle indique que le traitement du trigger doit être exécutée pour chaque ligne concernée par la requête déclenchante. Sinon, cette commande n'est exécutée qu'une seule fois pour chaque requête déclenchante.

5.7.2 Exemple

Une table cumul sert à enregistrer le cumul des augmentations dont ont bénéficié les employés d'une entreprise.

Exemple 5.16

Le trigger suivant met à jour automatiquement une table cumul qui totalise les augmentations de salaire de chaque employé.

```
CREATE OR REPLACE TRIGGER totalAugmentation
AFTER UPDATE OF sal ON emp
FOR EACH ROW
begin
  update cumul
  set augmentation = augmentation + :NEW.sal - :OLD.sal
  where matricule = :OLD.matr;
end;
```

Il faudra aussi créer un autre trigger qui ajoute une ligne dans la table cumul quand un employé est créé :

```
CREATE OR REPLACE TRIGGER creetotal
AFTER INSERT ON emp
for each row
begin
  insert into cumul (matricule, augmentation)
  values (:NEW.matr, 0);
end;
```

Les pseudo-variables `:NEW` et `:OLD` permettent de se référer aux anciennes et nouvelles valeurs des lignes. `:NEW` a la valeur `NULL` après une commande delete et `:OLD` a la valeur `NULL` après une commande insert. Elles ne sont évidemment utilisables qu'avec l'option "for each row".

Une option "`REFERENCING`" permet de donner un alias aux variables préfixées par `:NEW` et `:OLD` :

```
CREATE OR REPLACE TRIGGER totalAugmentation
AFTER UPDATE OF sal ON emp
REFERENCING old as ancien, new as nouveau
FOR EACH ROW
update cumul
set augmentation = augmentation + nouveau.sal - ancien.sal
where matricule = ancien.matr
```

5.7.3 Restriction sur le code des triggers

Le code d'un trigger ne peut contenir de `commit` ou de `rollback`.

Un trigger ne peut modifier par une commande `update`, `insert` ou `delete` la table indiquée dans la définition du trigger (celle sur laquelle a eu lieu l'action qui a déclenché l'exécution du trigger). Mais un trigger « for each row » peut modifier les lignes concernées par la requête SQL qui a déclenché le trigger, en utilisant l'ancienne et la nouvelle valeur.

Exemple 5.17

```
create or replace trigger incremente_version
after update on emp
for each row
begin
  -- version est une colonne de la table emp
  :new.version := :old.version + 1;
end;
```

5.7.4 Clause WHEN

Il est possible d'ajouter une clause `WHEN` pour restreindre les cas où le trigger est exécuté. `WHEN` est suivi de la condition nécessaire à l'exécution du trigger.

Cette condition peut référencer la nouvelle et l'ancienne valeur d'une colonne de la table (`new` et `old` ne doivent pas être préfixés par " :" comme à l'intérieur du code du trigger.

Exemple 5.18

```
create or replace trigger modif_salaire_trigger
before update of sal on emp
for each row
when (new.sal < old.sal)
begin
  raise_application_error(-20001,
                          'Interdit de baisser le salaire ! ('
                          || :old.nome || ')');
end;
```

`raise_application_error` est une instruction Oracle qui permet de déclencher une erreur en lui associant un message et un numéro (compris entre -20000 et -20999).

5.7.5 Triggers INSTEAD OF

Ces triggers servent à utiliser les vues non modifiables pleinement comme des tables.

Ils existent dans la plupart des SGBD (Oracle, DB2, SQL Server par exemple), mais pas dans tous (pas dans MySQL en particulier).

Dans Oracle, un tel trigger ne peut porter que sur une vue (pas sur une table).

Syntaxe :

```
CREATE [OR REPLACE] TRIGGER nom-trigger
INSTEAD OF {INSERT | DELETE | UPDATE} ON nom-vue
[REFERENCING ...]
[FOR EACH ROW]
bloc PL/SQL
```

Exemple 5.19

Soit une vue qui joint les tables EMP et DEPT :

```
create view emp_dept(matr, nom_emp, dept, nom_dept) as
select matr, nome, dept, nomd
from emp natural join dept
```

On souhaite pouvoir insérer des nouveaux employés et leur département en utilisant cette vue. Pour cela on crée le trigger suivant :

```
create or replace trigger trig_insert_emp_dept
instead of insert on emp_dept
begin
  insert into dept(dept, nomd)
```

```
    values(:new.dept, :new.nom_dept);
    insert into emp(matr, nome, dept)
    values(:new.matr, :new.nom_emp, :new.dept);
end;
```

Il est alors possible d'insérer un nouvel employé et un nouveau département avec la commande

```
insert into emp_dept(matr, nom_emp, dept, nom_dept)
values(1238, 'Bernard Paul', 38, 'Direction du personnel');
```

Évidemment, une erreur sera levée si, par exemple, il existe déjà un employé avec ce matricule ou un département avec ce numéro. Il est possible d'améliorer le code PL/SQL pour prendre en compte les différentes erreurs avec une section "EXCEPTION" (voir cours sur PL/SQL).

5.7.6 Dictionnaire de données

La vue `USER_TRIGGERS` du dictionnaire des données (voir 5.8) donne des informations sur les triggers. Par exemple le trigger "creetotal" de l'exemple 5.16 de la page 77 sera décrit ainsi par les divers colonnes :

```
TRIGGER_NAME  TRIGGER_TYPE     TRIGGERING_EVENT
-----------   --------------   ----------------
CREETOTAL     AFTER EACH ROW   INSERT

BASE_OBJECT_TYPE TABLE_NAME
---------------- ----------
TABLE            EMP

COLUMN_NAME   REFERENCING_NAMES
-----------   -------------------------------
              REFERENCING NEW AS NEW OLD AS OLD

WHEN_CLAUSE   STATUS    DESCRIPTION
-----------   --------  -----------
              ENABLED   creetotal AFTER INSERT ON emp
                        for each row

ACTION_TYPE TRIGGER_BODY
----------- ---------------------------------------------
PL/SQL      BEGIN insert into cumul (matricule, augmentation)
            values (:NEW.matr, 0); END;
```

La vue `USER_TRIGGER_COLS` donne des informations complémentaires.

5.8 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_PRIVS	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 5.20

(a) Colonnes de la table EMP :

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

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

```
SELECT TABLE_NAME, CONSTRAINT_NAME, T.CONSTRAINT_TYPE, COLUMN_NAME
FROM USER_CONSTRAINTS T NATURAL JOIN USER_CONS_COLUMNS C
WHERE 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'
```

5.9 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.

5.9.1 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 5.21

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
```

5.9.2 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 5.10

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

5.9.3 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
```

5.9.4 Synonyme

Oracle (et d'autres SGBD comme SQL Server) permet de donner des synonymes pour les tables et vues.

Les synonymes permettent le plus souvent des noms simplifiés. Les synonymes peuvent être publics s'ils sont connus de tous les utilisateurs de la base, ou privés s'ils ne sont connus que de celui qui a créé le synonyme.

`CREATE [ PUBLIC | PRIVATE ] SYNONYM` *nom_synonyme* `FOR` *objet*

Les synonymes publics sont particulièrement intéressants quand ils permettent de ne pas préfixer une table ou une vue par le nom du propriétaire lorsque l'utilisateur du synonyme n'est pas le propriétaire de la table ou de la vue.

Exemple 5.22

```
CREATE PUBLIC SYNONYM departement FOR toto.dept
```

Il est possible de supprimer un synonyme par la commande

`DROP SYNONYM` *nom_synonyme*

La table du dictionnaire `all_synonym` contient les définitions des synonymes.

La commande suivante permet de rechercher dans le dictionnaire des données les synonymes, la table ou vue renommée, et le propriétaire des synonymes pour les tables ou vues possédées par l'utilisateur toto :

```
select synonym_name, table_name, owner
from all_synonyms
where table_owner = 'TOTO'
```

5.9.5 Création d'un utilisateur, rôle

Bien que ce cours n'aborde pas l'administration d'une base de données, voici, sans explication, comment on peut créer des utilisateurs sous Oracle.

```
CREATE USER machin IDENTIFIED BY 123adc
DEFAULT TABLESPACE t2
TEMPORARY TABLESPACE temp_t2
QUOTA 1M ON t2;

GRANT etudiant TO machin;
```

"`etudiant`" est un rôle, c'est-à-dire un ensemble de droits que l'on peut donner par GRANT. L'administrateur peut créer plusieurs rôles correspondants à différents types standard d'utilisateurs pour les affecter aux utilisateurs qu'il crée.

On peut créer un rôle par la commande CREATE ROLE nom-rôle. On peut ensuite lui attribuer des droits comme on le fait avec un utilisateur par la commande GRANT.

Des rôles sont prédéfinis par Oracle : CONNECT permet à un utilisateur de se connecter et de créer des tables. RESOURCE permet en plus à un utilisateur de créer des procédures stockées, des types, des séquences,....

Chapitre 6

Gestion des accès concurrents

Les problèmes liés aux accès concurrents aux données sont présentés dans leur généralité. Les solutions apportées par les différents SGBDs, et plus particulièrement Oracle, sont ensuite étudiées.

6.1 Problèmes liés aux accès concurrents

Il peut arriver que plusieurs transactions veuillent modifier en même temps les mêmes données. Dans ce cas, il peut se produire des pertes de données si l'on ne prend pas certaines précautions.

Étudions quelques cas d'école des problèmes liés aux accès concurrents.

6.1.1 Mises à jour perdues

Si deux processus mettent à jour en même temps la somme S contenue dans un compte client d'une banque en enregistrant deux versements de 1000 et de 2000 francs, on peut avoir

Temps	Transaction T1	Transaction T2
t1	Lire S	
t2		Lire S
t3	S = S + 1000	
t4		S = S + 2000
t5	Enregistrer S	
t6		Enregistrer S

Finalement, la valeur du compte sera augmentée de 2000 francs au lieu de 3000 francs.

Pour éviter ce genre de situation, les SGBD bloquent l'accès aux tables (ou parties de tables : blocs mémoire ou lignes par exemple) lorsque de nouveaux accès pourraient occasionner des problèmes. Les processus qui veulent accéder aux tables bloquées sont mis en attente jusqu'à ce que les tables soient débloquées.

Cependant ceci peut aussi amener à l'interblocage de processus (*deadlock* en anglais) comme dans l'exemple suivant :

Temps	Transaction T1	Transaction T2
t1	Verrouiller la table A	
t2		Verrouiller la table B
t3	Verrouiller la table B	
t4	*Attente*	Verrouiller la table A
t5		*Attente*

On remarquera que ce type d'interblocage ne peut arriver si tous les utilisateurs bloquent les tables dans le même ordre.

6.1.2 Lecture inconsistante ou lecture impropre

Une transaction T2 lit une valeur V donnée par une autre transaction T1. Ensuite la transaction T1 annule son affectation de V et la valeur lue par T2 est donc fausse.

Temps	Transaction T1	Transaction T2
t1	V = 100	
t2		Lire V
t3	ROLLBACK	

Ce cas ne peut arriver si les modifications effectuées par une transaction ne sont visibles par les autres qu'après validation (Commit) de la transaction. C'est le plus souvent le cas.

6.1.3 Lecture non répétitive, ou non reproductible, ou incohérente

Une transaction lit deux fois une même valeur et ne trouve pas la même valeur les deux fois.

Temps	Transaction T1	Transaction T2
t1	Lire V	
t2		V = V + 100
t3		COMMIT
t4	Lire V	

Pour éviter ceci, T1 devra bloquer les données qu'elle veut modifier entre les temps t1 et t3, pour empêcher les autres transactions de les modifier (voir, par exemple, les blocages explicites d'Oracle en 6.5).

6.1.4 Lignes fantômes

Le phénomène des "lignes fantômes" peut arriver si une transaction ajoute des données sans qu'une autre transaction ne s'en aperçoive. Cette dernière transaction peut posséder des informations contradictoires sur la base.

Par exemple, une transaction récupère tous les t-uplets qui vérifient un certain critère. Une autre transaction ajoute des lignes qui vérifient ce critère. La même requête relancée une deuxième fois ne retrouve plus le même nombre de lignes.

Temps	Transaction T1	Transaction T2
t1	Sélectionne les lignes avec V = 10	
t2		Créer des lignes avec V = 10
t3	Sélectionne les lignes avec V = 10	

Une variante que l'on peut rencontrer si on veut accéder aux données depuis un langage de programmation tel que Java : on commence par lire le nombre de lignes qui vérifient un critère pour dimensionner un tableau Java qui les contiendra ; on relance ensuite la même sélection pour ranger les lignes dans le tableau ; si entre temps un autre utilisateur a rajouté des lignes qui vérifient ce critère, on provoque une erreur (le tableau n'est plus assez grand pour contenir toutes les lignes).

Un blocage de ligne n'est plus la solution ici car ces lignes n'existent pas à la première lecture. On sera souvent amené à effectuer un blocage explicite de table, ou à choisir le degré d'isolation "serializable" (décrit dans la section suivante).

6.2 Isolation des transactions

6.2.1 Sérialisation des transactions

Pour éviter tous les problèmes décrits dans la section précédente, la gestion des transactions doit, si possible, rendre les transactions "*sérialisable*" :

les exécutions simultanées de plusieurs transactions doivent donner le même résultat qu'une exécution séquentielle de ces transactions.

Il existe plusieurs moyens de forcer une exécution de transactions concurrentes de telle sorte qu'elles soient sérialisables. La stratégie la plus utilisée utilise le protocole de verrouillage à 2 phases :

- on doit bloquer un objet avant d'agir sur lui (lecture ou écriture) ;
- on ne doit plus acquérir de verrou après avoir relâché un verrou.

On a donc 2 phases :

1. acquisition de tous les verrous pour les objets sur lesquels on va agir ;
2. abandon de tous les verrous (en pratique, souvent sur COMMIT et ROLLBACK).

Avec ce protocole les situations qui auraient pu conduire à des problèmes de concurrence provoquent des blocages ou des inter-blocages (avec redémarrage possible des transactions invalidées pour défaire l'inter-blocage).

On peut utiliser d'autres stratégies pour forcer une exécution sérialisable. On peut, par exemple, estampiller les transactions : toutes les transactions reçoivent une estampille qui indique le moment où elles ont démarré. En cas d'accès concurrent, les lectures/écritures peuvent être mises en attente en se basant sur cette estampille. Ce type de traitement est souvent trop restrictif car il effectue dans certains cas des blocages même si la situation ne va pas conduire à des problèmes, et il nuit donc à la concurrence.

6.2.2 Niveaux d'isolation

Cette contrainte de rendre les exécutions sérialisables est souvent trop forte et peut dégrader sérieusement les performances en limitant les accès simultanés aux données. Les SGBD permettent de choisir d'autres niveaux d'isolation qui ne donnent pas une garantie aussi forte que la sérialisation complète mais offrent tout de même une bonne sécurité contre les problèmes d'accès multiples, tout en offrant de bonnes performances.

Voici tous les niveaux définis par SQL2 :

SERIALIZABLE : la transaction s'exécute comme si elle était la seule transaction en cours. Elle peut être annulée si le SGBD s'aperçoit qu'une autre transaction a modifié des données utilisées par cette transaction. On évite ainsi le problème dit de lecture non répétitive et les lignes fantômes (voir 6.1). Ce mode est cependant coûteux puisqu'il limite le fonctionnement en parallèle des transactions.

REPEATABLE READ : il empêche les lectures non répétitives mais pas les lignes fantômes.

READ COMMITTED : c'est le mode de fonctionnement de la plupart des SGBD (d'Oracle en particulier). Les modifications effectuées par une transaction ne sont connues des autres transactions que lorsque la transaction a été confirmée (ordre COMMIT). Il empêche les principaux problèmes de concurrence mais pas les lectures non répétitives.

READ UNCOMMITED : les autres transactions voient les modifications d'une transaction avant même le COMMIT ; c'est le niveau d'isolation le plus bas qui n'empêche aucun des problèmes d'accès concurrents cités ci-dessus et il n'est donc utilisé que dans des cas exceptionnels.

La commande SQL suivante fixe le niveau d'isolation de la transaction qui vient de commencer (doit être la première instruction de la transaction) :

```
SET TRANSACTION ISOLATION LEVEL {SERIALIZABLE | READ COMMITTED}
```

Tous les SGBD n'offrent pas toutes les possibilités ; Oracle, par exemple, ne permet que les niveaux *Serializable* et *Read Committed*. DB2 offre les niveaux *Uncommitted Read* (correspond au niveau *Read Uncommitted*), *Cursor Stability* (correspond au niveau *Read Committed* ; le niveau par défaut), *Read Stability* (correspond au niveau *Repeatable Read*) et *Repeatable Read* (correspond au niveau Serializable ; ne permet pas les lignes fantômes).

Sous Oracle on peut simuler le mode "*repeatable read*" en bloquant les lignes lues par un "*select for update*" (voir 6.5.1) les tables des données lues (avec le plus souvent une importante perte de performances).

6.3 Types de traitement pour les accès concurrents

La sérialisation des transactions étant très coûteuse, il n'est souvent pas possible de l'assurer au niveau du SGBD car les performances seraient désastreuses. On peut choisir plusieurs stratégies pour offrir aux utilisateurs des performances acceptables, tout en assurant l'intégrité des données.

Les développeurs d'applications peuvent essentiellement traiter de deux façons les problèmes liés aux accès concurrents : en étant pessimiste ou optimiste.

6.3.1 Traitement pessimiste

Si l'on est pessimiste, on pense qu'il y aura forcément des problèmes d'accès concurrent. Pour pouvoir faire tranquillement son travail, on (les transactions qu'on lance) bloque donc les données sur lesquelles on veut travailler. Un tel blocage durera le temps de la transaction en cours.

Le coût de positionnement des verrous n'est pas toujours négligeable. Même si on peut espérer qu'un utilisateur ne va pas poser un verrou et aller prendre un café avant de valider la transaction, la pose de verrou peut aussi provoquer des interblocages. De plus, ces blocages risquent de nuire beaucoup au fonctionnement des autres transactions qui sont mises en attente si elles veulent accéder à une partie de la base bloquée. Le blocage doit donc être le plus court possible et il faut donc terminer rapidement une transaction qui a posé des verrous.

Ce mode pessimiste est à éviter si possible dans les traitements longs, par exemple ceux qui comportent une intervention de l'utilisateur ou des accès distants avec risque d'attente au cours de la transaction.

Le mode optimiste a actuellement la faveur des programmeurs.

6.3.2 Traitement optimiste

Si l'on est optimiste, on pense que tout va bien se passer et que les autres transactions ne vont pas modifier les données sur lesquelles on va travailler. On n'effectue donc aucun blocage.

Pour ne pas être d'un optimisme béat, au moment où on veut valider le traitement effectué (et seulement à ce moment), on contrôle si on a eu raison d'être optimiste. Pour cela on démarre une transaction courte qui bloque les données et vérifie que les données que l'on a utilisées n'ont pas été modifiées par une autre transaction.

Si elles n'ont pas été modifiées, on valide la transaction.

Si malheureusement les données ont été modifiées c'est que l'optimisme n'était pas justifié. On agit en conséquence pour éviter les problèmes des accès concurrents décrits précédemment : soit on invalide tout le travail effectué dans la transaction, soit on fait un travail correcteur ou alternatif. Par exemple, si le traitement comporte une interaction avec l'utilisateur, on peut lui proposer plusieurs choix pour terminer au mieux la transaction.

Il y a plusieurs façons de repérer qu'une autre transaction a modifié les valeurs. La façon la plus simple est de comparer tout simplement les valeurs lues au moment où on en a eu besoin et au moment où on va valider la transaction. Mais ça ne convient pas aux données de grandes tailles. On peut aussi ajouter à chaque ligne des tables une information liée à la dernière modification de la ligne. Par exemple on peut ajouter un numéro de version incrémenté à chaque modification de la ligne ; on peut utiliser pour cela une séquence ; voir 5.5.3, ou un "*timestamp*" qui indique le moment exact de la dernière modification (à la microseconde près par exemple).

Remarque 6.1

Le traitement optimiste permet une granularité très fine (au niveau des attributs et pas des lignes) dans le cas où la validation de la transaction utilise la comparaison des valeurs des données utilisées. En effet, rien n'empêche une autre transaction de modifier des données des lignes que l'on utilise à partir du moment où elle ne modifie pas les attributs qui sont intervenus dans le traitement.

Le choix entre un traitement pessimiste ou optimiste dépend du contexte et il n'est pas toujours évident. Comme il a été dit, si le traitement comporte une intervention de l'utilisateur placée entre la lecture des données et l'écriture des nouvelles données, le traitement optimiste est sans doute le meilleur. En revanche, si le risque de problèmes de concurrence est trop grand (s'il y a peu de chance que l'optimisme soit justifié), le choix pessimiste est sans doute le meilleur pour éviter trop de calculs inutiles et d'invalidation au dernier moment de la procédure.

6.4 Traitement par défaut des accès concurrents par les SGBDs

Pour mettre en œuvre le type de traitement des accès concurrents qu'ils ont choisis, les développeurs peuvent s'appuyer sur les possibilités offertes par les SGBDs.

Les SGBDs gèrent automatiquement les accès concurrents mais permettent de choisir d'autres modes de fonctionnement en posant des verrous explicites. Nous allons décrire le comportement d'Oracle.

Le mode de fonctionnement par défaut de la plupart des SGBDs est le mode "*read committed*" : les modifications effectuées par une transaction ne sont connues des autres transactions que lorsque la transaction a été confirmée (ordre COMMIT).

Une opération effectuée par un SGBD doit s'efforcer d'effectuer une "lecture consistante" des données qu'elle manipule : toutes ces données doivent avoir les valeurs qu'elles avaient à un même moment précis. La lecture de tous les salaires des employés d'une entreprise par une requête SQL peut prendre un certain temps. Cependant si le premier salaire est lu au moment t1, en tenant compte des modifications effectuées par toutes les transactions validées à ce moment, il faut que les valeurs lues pour tous les autres salaires soient celles qu'ils avaient au même moment t1, même si une autre transaction validée les a modifiés depuis le moment t1.

Certains SGBDs, Oracle en particulier, assurent une lecture consistante

des données pendant l'exécution d'un seul 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. Les SGBDs comme DB2 qui ne conservent pas plusieurs versions des données récemment modifiées n'offrent pas cette lecture consistante.

Aucun SGBD n'assure automatiquement une lecture consistante pendant toute une transaction. Si une transaction est validée, toutes les autres transactions voient ensuite les modifications qu'elle a effectuée. Ce comportement n'est pas souhaité pour des traitements, tels que les bilans, qui souhaitent voir toutes les données comme elles étaient au moment du début de la transaction. Il est cependant possible d'obtenir une lecture consistante pendant toute une transaction. On peut

- soit indiquer que la transaction est en lecture seule par "`set transaction read only`" (voir 6.5.1), mais, dans ce cas, la transaction ne pourra pas modifier des données ;
- soit passer au niveau d'isolation serializable (voir 6.2.2), ce qui est pénalisant au niveau des performances ;
- soit effectuer des blocages explicites (voir 6.5).

Les écritures bloquent les autres écritures : si une transaction tente de modifier une ligne déjà modifiée par une autre transaction non validée, elle est mise en attente. Les mises à jour perdues sont ainsi évitées.

Certains SGBDs bloquent les lignes lues. Dans son mode de fonctionnement par défaut DB2 bloque chaque ligne lue du résultat d'un select, jusqu'à la lecture de la prochaine ligne (ou la fin de la transaction). Oracle ne bloque jamais les lignes lues.

Une particularité d'Oracle est que les lectures ne bloquent jamais les modifications et qu'une modification ne bloque jamais les lectures :

- Une lecture n'effectue aucune blocage. La lecture d'une donnée ne pose aucun verrou sur la donnée lue et une autre transaction peut donc modifier la donnée. Pour une transaction en mode read committed, cela revient à avoir un comportement optimiste (voir 6.3.2). Si ce comportement optimiste n'est pas souhaité, il est possible d'utiliser "select for update" (voir 6.5.1) pour bloquer explicitement les données lues.
- Si une donnée a été modifiée par une transaction en cours non encore validée, les autres transactions peuvent tout de même lire cette donnée mais puisque la transaction n'est pas encore validée, les autres transactions lisent la valeur que la donnée avait avant la modification de la

transaction.

Ces comportements sont mis en œuvre par les SGBDs en provoquant implicitement des blocages sur les lignes ou les tables impliquées lors de certains traitements : 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. Certains SGBDs (mais pas Oracle) posent aussi un verrou avec la commande SELECT.

Voici les blocages effectués implicitement par Oracle :

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

Les blocages explicites ou implicites sont relâchés à la fin des transactions.

6.5 Blocages explicites

Dans de rares cas, l'utilisateur (ou le programmeur) peut ne pas se satisfaire des blocages effectués automatiquement par son SGBD.

Il peut effectuer des blocages explicites au niveau des lignes ou des tables. Par exemple, si l'utilisateur veut que les lignes sur lesquelles il va travailler ne soient pas modifiées par une autre transaction pendant toute la durée d'un traitement, il ne peut se fier aux blocages implicites de Oracle.

Un blocage ne sert pas seulement à bloquer des tables ou des lignes. Les blocages comportent une partie "préventive" qui empêche les autres transactions de bloquer les mêmes données avec certains types de blocage (voir 6.5.1).

Tous les SGBDs offrent plusieurs types de blocages qui sont divisés en deux grands groupes :

blocages SHARE (partagé) : plusieurs transactions peuvent obtenir ce blocage en même temps sur les mêmes données. Ce sont des blocages préventifs pour empêcher d'autres transactions de bloquer en mode exclusif; si une autre transaction veut effectuer un blocage interdit, elle est mise en attente.

blocages EXCLUSIVE (exclusif) : une seule transaction peut obtenir ce type de blocage à un moment donné sur les mêmes données.

Tous les SGBDs n'offrent pas exactement les mêmes types de blocages. Nous allons étudier dans ce cours les blocages offerts par Oracle. La plupart des notions décrites dans cette section se retrouvent dans les autres SGBD,

associées à des noms de commandes différentes (consultez la documentation de votre SGBD).

Il faut cependant savoir qu'Oracle (et PostGreSQL) ne bloque pas les données en lecture, ce qui le différencie des autres SGBD comme DB2 ou SQL Server. Plusieurs versions des données peuvent être utilisées. En effet, si une transaction non encore validée a modifié une donnée, les autres transactions qui veulent lire cette donnée ne sont pas bloquées mais lisent l'ancienne valeur de la donnée.

6.5.1 Blocages explicites d'Oracle

Les commandes qui provoquent des blocages explicites sont "lock table" et "select for update".

Voici les cinq variantes de la commande **LOCK TABLE**, de la plus restrictive à la moins restrictive :

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

Il s'agit d'un blocage de table, sauf si le mot ROW apparaît. Tous les blocages de lignes impliquent un autre blocage de table (vous comprendrez cette phrase en lisant les détails de chaque commande ci-dessous).

Au niveau des lignes il n'existe que le blocage exclusif : même si plusieurs transactions peuvent bloquer en même temps des lignes, elles ne peuvent bloquer les mêmes lignes.

Les blocages explicites peuvent conduire à des interblocages. Oracle détecte et résout les interblocages (en annulant certaines des requêtes bloquées).

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

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 6.2

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 programme (ou à l'utilisateur) de relancer plus tard la commande LOCK.

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 Share Row. Il empêche donc toute modification sur les tables par les autres transactions mais celles-ci peuvent se réserver des lignes par un select for update ; elles pourront modifier ces lignes après le relâchement du verrou "share row exclusive". On comprend pourquoi ce mode comporte à la fois les mots EXCLUSIVE et SHARE : EXCLUSIVE car une seule transaction peut effectuer ce blocage sur une table, SHARE (ROW) car ça n'empêche pas d'autres transactions de bloquer des lignes par un select for update.

L'avantage par rapport au verrouillage share est qu'aucun blocage d'une autre transaction ne pourra empêcher de modifier tout de suite des lignes (sauf celles éventuellement bloquées par un *select for update*) par un blocage en mode SHARE. C'est la différence avec le blocage suivant (SHARE).

Verrouillage d'une table en mode partagé (Share)

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

Ce mode interdit toute modification de la table par les autres transactions.

Il permet aux autres transactions de bloquer aussi la table en mode partagé. En ce cas la transaction ne pourra pas non plus modifier tout de suite les données de la table ; elle devra attendre la fin des autres transactions.

Ce blocage permet, par exemple, d'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.

Si on veut enregistrer les résultats du bilan, ce mode ne convient pas car une autre transaction peut empêcher cet enregistrement. Il faudrait plutôt choisir le niveau d'isolation SERIALIZABLE pour la transaction (voir 6.2).

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 sur la table modifiée par un ordre INSERT, DELETE ou UPDATE. En plus, les lignes concernées par ces ordres ne peuvent être modifiées par les autres transactions.

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.

Verrouillage d'une table 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 de blocage le moins restrictif pour une table.

Le mode "Row Share" est aussi le mode implicite de blocage de la table de la commande SELECT FOR UPDATE

SELECT FOR UPDATE

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

En plus du blocage de la table en mode "Row Share", cette commande réserve les lignes sélectionnée pour une modification ultérieure (les lignes sont bloquées en mode exclusif) et les fournit à la transaction. On peut ainsi utiliser les valeurs actuellement enregistrées dans les lignes pour préparer tranquillement les modifications à apporter à ces lignes ; on est certain qu'elles ne seront pas modifiées 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.

Remarque 6.3

Si on n'effectue pas ce type de blocage sur les données que l'on veut modifier, on risque d'avoir un problème de "mise à jour perdue" : les données que l'on a lues par un SELECT pour les modifier ont été modifiées par une autre transaction avant qu'on ait le temps d'enregistrer leur nouvelle valeur. On peut choisir de ne pas effectuer ce blocage et faire un traitement "optimiste" (on espère qu'aucune transaction n'a travaillé sur les mêmes données) et des vérifications au moment du COMMIT pour voir si on avait effectivement raison d'être optimiste (voir 6.3.2 page 91).

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.

Si on veut travailler pendant toute la transaction sur des données inchangées mais si on veut en plus modifier des données (par exemple, si on veut faire un bilan et écrire dans la base le résultat du bilan), il faudra travailler avec une transaction SERIALIZABLE (voir 6.2) ou faire des blocages explicites.

Tableau récapitulatif

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

Modes / **Commandes**	X	SRX	S	RX	RS
LOCK EXCLUSIVE (X)	non	non	non	non	non
LOCK ROW SHARE EXCLUSIVE (SRX)	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.

Index

ABS, *48*
accès concurrents, 86
ACID, 22
ajouter une colonne, 60
ALTER TABLE, *60*
AND, *33*
annuler une transaction, 23
ASCII, *50*
AVG, *45*

BETWEEN, *32*
bloquer des données, 98

case, 51
catalogue, 59
changer son mot de passe, 83
CHAR, *6*
CHR, *50*
COALESCE, *9*
colonne, 4
COMMIT, *23*, 95
concaténer des chaînes, 9
CONSTRAINT, *12*
contrainte d'intégrité, 12, 65, 82
COUNT, *45*
création d'un utilisateur, 84
créer un index, 66
créer une table, 11, 59
créer une vue, 63
CREATE INDEX, *66*
CREATE TABLE, *11*
CREATE TABLE AS, *59*
CREATE VIEW, *63*
CURRENT_DATE, *51*
DATE, *7*
date du jour, 29
DECODE, *53*
DEFAULT, 12
DELETE, *22*
DESC, *54*
dictionnaire de données, 81
différer des contraintes, 16
DISTINCT, *45*
division, 42
donner des droits d'accès, 82
DROP INDEX, *68*
DROP VIEW, *64*
DUAL, 4

enlever des droits d'accès, 83
EXCEPT, *55*
except, 55
EXISTS, *41*
EXIT, *3*

fonction de choix, 51
fonction stockée, 73
fonction utilisateur, 74
fonctions arithmétiques, 48
fonctions chaînes de caractères, 48
fonctions de groupes, 45
FOREIGN KEY, *13*
FROM, *29*

génération de clés primaires, 70
GRANT, *82*
GREATEST, *48*
GROUP BY, *45*

HAVING, *47*

identificateur, 3
IN, *32*
index, 66
injection de code SQL, 56
insérer des lignes, 19
INSERT, *19*
INSTEAD OF, 79
INSTR, *48*
interblocage, 87, 95
interroger la base, 26
INTERSECT, *55*
intersect, 55
IS NULL, *33*

jointure, 33
jointure externe, *36*
jointure naturelle, 35
jointure non équi, *37*
joker dans une chaîne, 32

LCD, 1, 94
LDD, 1, 59, 94
LEAST, *48*
lecture consistante, 92, 98
LENGTH, *48*
limiter le nombre de lignes renvoyées, 56
LMD, 1, 19
LOCK TABLE, *95*
LOWER, *49*
LPAD, *49*
LTRIM, *49*

MAX, *45*
MIN, *45*
MINUS, *55*
minus, 55
mise à jour avec une vue, 64
MOD, *48*
modifier des lignes, 20
modifier une colonne, 60
mot de passe, 83

nom de l'utilisateur, 29
norme SQL, 1
NOT, *33*
NULL, *8*
NULLIF, *10*
NUMBER, *5*
NVL, *10*

OLAP, 69
ON DELETE CASCADE, *13*
ON DELETE SET NULL, *13*
opérateur de concaténation de chaînes, 9
opérateur ensembliste, 54
opérateur logique, 33
opérateurs, 9
opérateurs arithmétiques, 9
opérateurs sur les dates, 9
optimiste, 91
OR, *33*
ORDER BY, *53*

pessimiste, 90
point de reprise, 25
POWER, *48*
PRIMARY KEY, *12*
privilèges d'accès, 82
procédure stockée, 73
pseudo-colonne, 28
pseudo-table, 4
PUBLIC, *83*

rôle, 84
REFERENCES, *13*
renommer une colonne, 61
renommer une table, 62
REPLACE, *49*
REVOKE, *83*
ROLLBACK, *23*, 95

ROUND, *48*, *51*
rowid, 29
rownum, 28
RPAD, *49*
RTRIM, *49*

savepoint, 25
schéma, 59
SELECT FOR UPDATE, *97*
séquence, 71
sérialisable, 88
sérialisation, 88
SET TRANSACTION READ ONLY, *98*
SGBDR, 1
SIGN, *48*
sous-interrogation, 38
SQLFORMS, 2
SQRT, *48*
STDDEV, *45*
SUBSTR, *48*
SUM, *45*
supprimer des lignes, 21
supprimer un index, 68
supprimer une colonne, 61
supprimer une table, 62
supprimer une vue, 64
synonyme, 84
synonyme public, 62
SYSDATE, *51*
sysdate, 29

table, 3
table DUAL, 4
TO_CHAR, *48*, *49*
TO_DATE, *51*
TO_NUMBER, *48*, *50*
traitement optimiste, 91
traitement pessimiste, 90
transaction, 22
TRANSLATE, *49*
trigger, 75
trigger INSTEAD OF, 79
TRUNC, *48*, *51*
type chaîne, 6
type numérique, 5, 7
type numérique d'Oracle, 5
type numérique SQL-2, 5
type temporels, 7
types d'index, 68
types de contraintes, 12
types temporels Oracle, 7
types temporels SQL-2, 7

UNION, *55*
union, 55
UNIQUE, *13*
UPDATE, *20*
UPPER, *49*
USER, *53*
user, 29
utilisation d'une vue, 64
utilité des vues, 65

valider une transaction, 23
VARCHAR, *6*
VARCHAR2, *6*
VARIANCE, *45*
vue, 62

WHERE, *31*
WITH CHECK OPTION, *64*
WITH GRANT OPTION, *83*