Oracle : SQL

Denis Roegel
roegel@loria.fr
IUT Nancy 2

1998/1999

Table des matières

Ce document est une adaptation d'un chapitre de l'ouvrage *Oracle Unleashed* (SAMS Publishing, 1996).

1 Introduction

SQL (Structured Query Language) a été introduit par IBM comme le langage d'interface de son prototype de système de gestion de base de donnée relationnelle, System-R. Le premier système SQL disponible sur le marché a été introduit en 1979 par Oracle. Aujourd'hui, SQL est devenu un standard de l'industrie et Oracle est un leader dans la technologie des systèmes de gestion de bases de données relationnelles.

Comme SQL est un langage non procédural, des ensembles d'enregistrements peuvent être manipulés à la fois. La syntaxe est naturelle et souple, ce qui permet de se concentrer sur la présentation des données. Oracle a deux optimiseurs (basés sur le coût et des règles) qui vont analyser la syntaxe et la formater en une expression efficace avant que le moteur de la base de donnée ne le reçoive pour traitement. L'administrateur de la base de données détermine quel optimiseur est sélectionné pour chaque base de données.

SQL–le standard

L'ANSI (American National Standards Institute) a déclaré SQL le langage standard pour les systèmes de gestion de bases de données relationnelles. La plupart des entreprises qui produisent des systèmes de gestion de bases de données relationnelles sont compatibles avec SQL et essaient de respecter le standard SQL89.

2 Types de données

Une règle générale pour écrire des expressions SQL valides est de ne pas mélanger des types de données. Des utilitaires de conversion sont disponibles pour passer d'un type à un autre. Ces fonctions de conversion sont décrites plus loin.

2.1 Numérique

Le type `NUMBER` est utilisé pour stocker zéro, les nombres négatifs, positifs, à virgule fixe et flottants jusqu'à 38 chiffres de précision. Les nombres peuvent s'échelonner entre 1.0×10^{-130} et 1.0×10^{126}.

Les nombres peuvent être définis de l'une des trois manières suivantes :

```
NUMBER(p,s)
```

où `p` est la précision jusqu'à 38 chiffres et `s` est l'échelle (nombre de chiffres à la droite du point décimal). L'échelle peut s'étaler de -84 à 127.

```
NUMBER (p)
```

Ceci est un nombre à virgule fixe avec une échelle de zéro et une précision de `p`.

```
NUMBER
```

Ceci est un nombre à virgule flottante avec une précision de 38.

La table suivante montre comment Oracle stocke différentes échelles et précisions :

Actual Data	123456.789	123456.789	123456.789	123456.789
Defined as	`NUMBER(6,2)`	`NUMBER(6)`	`NUMBER(6,-2)`	`NUMBER`
Stored as	123456.79	123457	123400	123456.789

2.2 Date

Au lieu de stocker la date et l'heure dans une chaîne ou sous forme numérique, IBM a créé plusieurs types séparés. Pour chaque type `DATE`, les informations suivantes sont stockées :

```
Century Year Month Day Hour Minute Second
```

Il est facile de récupérer les date et heure courantes en appelant la fonction `SYSDATE`.

L'arithmétique sur les dates est possible en utilisant des constantes numériques ou d'autres dates. Seules l'addition et la soustraction sont admises. Par exemple, `SYSDATE + 7` va rendre la date dans une semaine.

Chaque base de donnée a un format de date par défaut qui est défini par le paramètre d'initialisation `NLS_DATE_FORMAT`. Ce paramètre est généralement mis à `DD-MON-YY`, où `DD` est le jour du mois (le premier jour du mois est 01), `MON` est l'abréviation du nom du mois et `YY` est une représentation à deux chiffres de l'année.

Si une heure n'est pas spécifiée, la valeur par défaut est minuit. Si seule l'heure est saisie, la date par défaut sera le premier jour du mois courant.

2.3 Caractère

Il y a quatre types de données caractère disponibles :

1. Le type `CHAR` est utilisé quand des champs de taille fixe sont nécessaires. Toute longueur inférieure ou égale à 255 caractères peut être spécifiée. La longueur par défaut est 1. Quand des données sont entrées, tout espace résiduel est rempli de blancs. Tous les caractères alpha-numériques sont autorisés.
2. Le type `VARCHAR2` est utilisé pour des champs de longueur variable. Une longueur doit être fournie lorsque l'on utilise ce type de données. La longueur maximale est de 2000 caractères. Tous les caractères alpha-numériques sont autorisés.
3. Le type `LONG` est utilisé pour stocker de grandes quantités de texte de longueur variable. Toute longueur jusqu'à 2 gigaoctets peut être spécifiée. Ce type a des restrictions, telles que :
   - Une seule colonne d'une table peut être définie en `LONG`.
   - Une colonne de type `LONG` ne peut pas être indexée.
   - Une colonne de type `LONG` ne peut pas être passée en argument à une procédure.
   - Une fonction ne peut pas être utilisée pour rendre une colonne de type `LONG`.
   - Une colonne de type `LONG` ne peut pas être utilisée dans des clauses `where`, `order by`, `group by`, ou `connect by`.
4. Le type `VARCHAR` est synonyme de `VARCHAR2`. Oracle réserve ceci pour une utilisation future. Il ne faut pas utiliser ce type.

2.4 Binaire

Deux types de données, `RAW` et `LONGRAW`, sont disponibles pour stocker des données de type binaire telles que du son digitalisé et des images. Ces types de données ont des caractéristiques similaires aux types `VARCHAR2` et `LONG` déjà mentionnés.

Le type `RAW` est utilisé pour stocker des données binaires jusqu'à 2000 caractères et le type `LONGRAW` jusqu'à 2 gigaoctets.

Oracle ne stocke et n'extrait que des données binaires. Aucune manipulation de chaîne n'est autorisée. Les données sont extraites sous forme de valeurs de caractères hexadécimaux.

2.5 Autres

Chaque ligne de la base de donnée a une adresse. Celle-ci peut être obtenue en utilisant la fonction `ROWID`. Le format de `ROWID` est le suivant :

```
BLOCK.ROW.FILE
```

- `BLOCK` est le bloc de données des données `FILE` contenant la ligne `ROW`. Les données sont en format hexadécimal et de type `ROWID`.
- `MLSLABEL` est un type de donnée utilisé pour stocker le format binaire d'une étiquette utilisée sur un système d'exploitation sécurisé.

3 L'instruction `CREATE`

Cette instruction ouvre le monde à l'utilisateur. Seules quelques unes des instruction `CREATE` seront décrites ici.

3.1 Tables

Chaque concepteur de base de donnée doit créer une table un jour ou l'autre. Il est nécessaire d'avoir un privilège système pour exécuter la commande `CREATE TABLE`. L'administrateur de la base de données gère ces privilèges. La syntaxe pour créer une table est :

```
CREATE TABLE schema.TABLE (COLUMN DATATYPE
                           default expression
                           column constraint) table constraint

PCTFREE x1 PCTUSED x2 INITRANS x3 MAXTRANS x4
TABLESPACE name STORAGE clause CLUSTER cluster clause
ENABLE clause DISABLE clause AS subquery
```

Dans cette syntaxe,

- `SCHEMA` est un paramètre optionnel pour identifier le schéma de la base de donnée dans laquelle cette table doit être placée. Par défaut, c'est celui de l'utilisateur.
- `TABLE` est obligatoire et est le nom de la table.
- `COLUMN DATATYPE` sont requis pour identifier chaque colonne dans la table. Les colonnes doivent être séparées par des virgules. Il y a au maximum 254 colonnes par table.
- L'expression `DEFAULT` est optionnelle et est utilisée pour donner une valeur par défaut à une colonne lorsque des insertions ultérieures ne réussissent pas à donner une valeur.
- `COLUMN CONSTRAINT` est optionnel. C'est utilisé pour définir une contrainte d'intégrité telle que `not null`.
- `TABLE CONSTRAINT` est optionnel et est utilisé pour définir une contrainte d'intégrité sur la table, comme par exemple la clé primaire.
- `PCTFREE` est optionnel mais a une valeur par défaut de 10. Ceci indique que 10 pour cents de chaque bloc sera réservé pour de futures mises à jour des lignes de la table. Les entiers de 1 à 99 sont autorisés.
- `PCTUSED` est optionnel mais a une valeur par défaut de 40. Ceci indique le pourcentage minimum d'espace utilisé qu'Oracle maintient avant qu'un bloc de données soit candidat pour une insertion de ligne. Les entiers de 1 à 99 sont autorisés. La somme de `PCTFREE` et `PCTUSED` doit être plus petite que 100.
- `INITRANS` est optionnel mais a une valeur par défaut de 1. Les entiers de 1 à 255 sont autorisés. Il est recommandé de ne pas modifier cette valeur. C'est une allocation du nombre d'entrées de transaction assignées au sein du bloc de données de la table.

- `MAXTRANS` est optionnel mais a une valeur par défaut qui est fonction de la taille des blocs de données. Ceci est utilisé pour identifier le nombre maximum de transactions parallèles pouvant faire des mises à jour dans un bloc de la table. Il est recommandé de ne pas modifier ce paramètre.
- `TABLESPACE` est optionnel mais a comme valeur par défaut le nom du « tablespace » du propriétaire du schéma. Un nom différent du nom par défaut peut être utilisé. Ces noms dépendent en général de l'application. L'administrateur de la base de données sera en mesure de donner des recommandations adéquates.
- `STORAGE` est optionnel et a des caractéristiques par défaut définies par l'administrateur de la base de données.
- `CLUSTER` est optionnel et spécifie qu'une table doit faire partie d'un groupe. Il faut identifier les colonnes de la table qui doivent en faire partie. Typiquement, les colonnes groupées sont des colonnes dans lesquelles se trouve la clé primaire.
- `ENABLE` est optionnel et active une contrainte d'intégrité.
- `DISABLE` est optionnel et désactive une contrainte d'intégrité.
- `AS SUBQUERY` est optionnel et insère les lignes rendues par la sous-requête dans la table à sa création.

Une fois que la table est créée, on peut utiliser la commande `ALTER TABLE` pour modifier la table. Pour modifier une contrainte d'intégrité, il faut d'abord utiliser `DROP` sur la contrainte, puis la recréer. Voyons deux exemples sur la création de tables.

```
CREATE TABLE ADDRESSES (ADRS_ID          NUMBER(6),
                        ACTIVE_DATE      DATE,
                        BOX_NUMBER       NUMBER(6),
                        ADDRS_1          VARCHAR2(40),
                        ADDRS_2          VARCHAR2(40),
                        CITY             VARCHAR2(40),
                        STATE            VARCHAR2(2),
                        ZIP              VARCHAR2(10));
```

Ceci est la forme la plus simple de création d'une table utilisant tous les paramètres par défaut. Le second exemple suit.

```
CREATE TABLE ADDRESSES  (ADRS_ID  NUMBER(6) CONSTRAINT PK_ADRS PRIMARY KEY,
                        ACTIVE_DATE   DATE DEFAULT SYSDATE,
                        BOX_NUMBER    NUMBER(6)      DEFAULT NULL,
                        ADDRS_1       VARCHAR2(40)   NOT NULL,
                        ADDRS_2       VARCHAR2(40)   DEFAULT NULL,
                        CITY          VARCHAR2(40)   DEFAULT NULL,
                        STATE         VARCHAR2(2)    DEFAULT 'NY',
                        ZIP           VARCHAR2(10))

PCTFREE 5
PCTUSED 65
TABLESPACE adrs_data
STORAGE (INITIAL 5140
         NEXT      5140
         MINEXTENTS  1
         MAXEXTENTS 10
         PCTINCREASE 10);
```

Dans cet exemple, des contraintes de données sont utilisées et certains paramètres de stockage seront actifs. L'utilisation de `PCTFREE` et `PCTUSED` est une bonne idée si les données sont relativement statiques.

3.2 Index

Les index sont utilisés pour améliorer la performance de la base de données. Un index est créé sur une ou plusieurs colonnes d'une table ou d'un groupe. On peut avoir plusieurs index par table. Le privilège système de `CREATE INDEX` est nécessaire pour exécuter cette commande. L'administrateur de la base de données est responsable de ces privilèges. La syntaxe de création d'un index est :

```
CREATE INDEX schema.index ON schema.table (COLUMN  ASC/DESC)

CLUSTER schema.cluster  INITRANS x MAXTRANS x TABLESPACE
          name STORAGE  clause PCTFREE x NOSORT
```

Dans cette syntaxe,

- `SCHEMA` est un paramètre optionnel identifiant le schéma de la base dans lequel doit se trouver l'index. Par défaut, c'est le schéma de l'utilisateur.
- `INDEX` est obligatoire et est le nom de l'index.
- `ON` est un mot réservé obligatoire.
- `TABLE` est un nom de table sur lequel est construit l'index.
- `COLUMN` est le nom de colonne à indexer. S'il y a plus d'une colonne, il faut s'assurer qu'elles sont dans l'ordre de priorité.
- `ASC/DESC` sont des paramètres optionnels. Les index sont construits en ordre croissant par défaut. `DESC` permet d'avoir l'ordre décroissant.
- `CLUSTER` est nécessaire seulement si cet index est destiné à un groupe.
- `INITRANS` est optionnel mais a la valeur par défaut de 1. Les entiers de 1 à 255 sont permis. Il est recommandé de ne pas changer ce paramètre. Il s'agit d'une allocation du nombre d'entrées de transactions assignées dans le bloc de données pour l'index.
- `MAXTRANS` est optionnel mais a une valeur par défaut qui est fonction de la taille du bloc de données. Il est utilisé pour identifier le nombre maximal de transactions qui peuvent mettre à jour en parallèle un bloc de données pour l'index. Il est recommandé de ne pas changer ce paramètre.
- `TABLESPACE` est optionnel mais a comme valeur par défaut le nom du « tablespace » du propriétaire du schéma. Un nom différent du nom par défaut peut être utilisé. L'administrateur de la base de données sera en mesure de donner des recommandations adéquates.
- `STORAGE` est optionnel et a des caractéristiques par défaut définies par l'administrateur de la base de données.
- `PCTFREE` est optionnel mais a une valeur par défaut de 10. Ceci indique que 10 pour cents de chaque bloc seront réservés pour de futures mises à jour de l'index. Les entiers de 1 à 99 sont autorisés.
- `NOSORT` est un paramètre optionnel qui permet de gagner du temps lors de la création de l'index si les données de la table sont déjà stockées dans l'ordre croissant. Ceci ne peut pas être utilisé si un index de groupe est utilisé.

En utilisant la table `ADRESSES` définie dans l'exemple du `CREATE TABLE`, deux index vont être créés dans le prochain exemple :

```
CREATE INDEX x_adrs_id ON ADDRESSES (ADRS_ID);
```

Ceci créera un index sur la colonne `adrs_id` seulement.

```
CREATE INDEX x_city_state ON ADDRESSES (CITY,STATE)
TABLESPACE application_indexes;
```

Cet index a deux colonnes ; `CITY` est la colonne primaire. Pour que les requêtes puissent utiliser un index, les noms des colonnes doivent faire partie de l'instruction `SELECT`. Si une instruction `SELECT` inclut `STATE` mais non `CITY`, l'index ne sera pas utilisé. Toutefois, si l'instruction `SELECT` contient une référence à `CITY` mais pas à `STATE`, une partie de l'index sera utilisée car `CITY` est la première colonne de l'index.

3.3 Séries

Les séries (*sequences*) sont un excellent moyen d'avoir une base de données qui génère automatiquement des clés primaires entières uniques. Le privilège système `CREATE SEQUENCE` est nécessaire pour exécuter cette commande. L'administrateur de la base de données est responsable de l'administration de ces privilèges. La syntaxe pour créer une série est

```
CREATE SEQUENCE schema.name
   INCREMENT BY x1
   START WITH x2
   MAXVALUE x3    NOMAXVALUE
   MINVALUE x4    NOMINVALUE
   CYCLE          NOCYCLE
   CACHE x5       NOCACHE
   ORDER          NOORDER
```

Dans cette syntaxe,

- `SCHEMA` est un paramètre optionnel qui identifie le schéma de base de données dans lequel se place cette série. Par défaut, c'est celui de l'utilisateur.
- `NAME` est obligatoire car c'est le nom de la série.
- `INCREMENT BY` est optionnel. La valeur par défaut est 1. 0 n'est pas autorisé. Si un entier négatif est spécifié, la série décroîtra dans l'ordre. Un entier positif fera croître en ordre.
- `START WITH` est un entier optionnel qui permet à la série de commencer avec n'importe quelle valeur.
- `MAXVALUE` est un entier optionnel qui définit une limite pour la série.
- `NOMAXVALUE` est optionnel. Ceci a pour effet de définir la valeur maximale croissante à 10^{27} et la valeur maximale décroissante à -1. Cette option est l'option par défaut.
- `MINVALUE` est un entier optionnel qui détermine le minimum d'une série.
- `NOMINVALUE` est optionnel. Ceci a pour effet de définir la valeur minimale croissante à 1 et la valeur minimale décroissante à -10^{26}. Ceci est l'option par défaut.
- `CYCLE` est une option qui permet à la série de continuer même lorsque le maximum a été atteint. Dans ce cas, la série suivante qui sera générée est celle correspondant à la valeur minimale.
- `NOCYCLE` est une option qui interdit à la série de produire des valeurs au-delà des maximum ou minimum définis. C'est la valeur par défaut.
- `CACHE` est une option qui permet à des numéros de série d'être préalloués et stockés en mémoire pour un accès plus rapide. La valeur minimale est 2.
- `NOCACHE` est une option qui n'autorise pas la préallocation de numéros de série.
- `ORDER` est une option qui assure que les numéros de série seront générés dans l'ordre des demandes.
- `NOORDER` est une option qui n'assure pas que les numéros de série seront générés dans l'ordre où ils sont demandés.

Si l'on souhaite créer une série pour la colonne `adrs_id` dans la table `ADDRESSES`, cela pourrait se faire de la manière suivante :

```
CREATE SEQUENCE adrs_seq
     INCREMENT BY 5
     START WITH 100;
```

Pour générer un nouveau numéro de série, on peut utiliser la pseudo-colonne `NEXTVAL`. Ceci doit être précédé du nom de la série. Par exemple, `adrs_seq.nextval` rendrait 100 pour le premier accès et 105 pour le second. Si la détermination du numéro de la série courante est nécessaire, on utilise `CURRVAL`. Par conséquent, `adrs_seq.currval` renvoie la valeur courante de la série.

3.4 Autres objets

Le but de ce chapitre n'est pas de détailler chaque instruction SQL. Ceux qui ont été donnés ont été décrits pour donner un survol des instructions les plus courantes de création. Nous donnons maintenant une liste alphabétique de tous les objets qui peuvent être créés avec une instruction `CREATE`.

```
CREATE xxx
```

où `xxx` est l'un des suivants :

CLUSTER	CONTROLFILE	DATABASE
DATABASE LINK	DATAFILE	FUNCTION
INDEX	PACKAGE BODY	PACKAGE
PROCEDURE	PROFILE	ROLE
ROLLBACK SEGMENT	SCHEMA	SEQUENCE
SNAPSHOT	SNAPSHOT LONG	SYNONYM
TABLE	TABLESPACE	TRIGGER
USER	VIEW	

4 Écriture de requêtes

Pour extraire des données de la base de données, on utilise l'instruction `SELECT`. Une fois de plus, des privilèges convenables sont nécessaires et maintenus par l'administrateur de la base de données. Le format de `SELECT` est le suivant :

```
SELECT column(s)
FROM tables(s)
WHERE conditions are met
GROUP BY selected columns
ORDER BY column(s);
```

Chaque instruction SQL s'achève par un point-virgule (`;`). Lors de l'écriture de scripts qui seront exécutés, on peut aussi utiliser « \ » pour terminer l'instruction.

Lorsque `SELECT column(s)` est utilisé, on suppose que toutes les colonnes satisfaisant la clause `WHERE` seront extraites. Il est quelquefois nécessaire de ne conserver que les colonnes qui sont distinctes les unes des autres. Pour ce faire, le mot clé `DISTINCT` doit être utilisé devant les descriptions de colonnes. Dans l'exemple suivant, une instruction `SELECT` est utilisée pour extraire toutes les villes et états de la table `ADRESSES` (définie précédemment).

```
SELECT city, state
FROM addresses;
```

Quand ce code est exécuté, chaque ville et état seront extraits de la table. Si 30 personnes vivent à Rochester, NY, ces données seront affichées 30 fois. Pour voir seulement une occurence pour chaque ville et état, on utilise `DISTINCT`, comme montré ci-dessous :

```
SELECT DISTINCT city, state
FROM addresses;
```

Ceci causera l'extraction d'une seule ligne des entrées avec Rochester, NY.

La clause `FROM` est une liste de toutes les tables nécessaires pour la requête. Des alias peuvent être utilisés pour simplifier les requêtes, comme montré dans l'exemple ci-dessous :

```
SELECT adrs.city, adrs.state
FROM addresses adrs;
```

Dans cet exemple, l'alias `adrs` a été donné à la table `addresses`. L'alias sera utilisé pour distinguer des colonnes avec des noms identiques dans des tables différentes.

La clause `WHERE` est utilisée pour donner la liste des critères nécessaires pour restreindre la sortie de la requête ou pour joindre des tables dans la clause `FROM`. Cf. exemple ci-dessous :

```
SELECT DISTINCT city, state
FROM addresses
WHERE state in ('CA','NY','CT')
  AND city is NOT NULL;
```

Cet exemple va extraire les villes et états qui se trouvent dans les états de Californie, New York et Connecticut. Le test pour des villes non nulles (`NOT NULL`) ne renvoiera pas des données si le champ `city` n'a pas été rempli.

La clause `GROUP BY` dit à Oracle comment grouper des enregistrements lorsque certaines fonctions sont utilisées.

```
SELECT dept_no, SUM(emp_salary)
FROM emp
GROUP BY dept_no;
```

L'exemple `GROUP BY` va donner la liste de tous les numéros de département une fois avec la somme des salaires des employés pour ce département particulier.

4.1 Fonctions de base

Les fonctions forment une part intrinsèque de toute instruction SQL. La table suivante donne une liste alphabétique de fonctions SQL.

Nom	Type	Syntaxe	Retour
`ABS`	Nombre	`ABS(n)`	Valeur absolue de `n`.
`ADD_MONTHS`	Date	`ADD_MONTHS(a,b)`	Date `a` plus `b` mois.
`ASCII`	Caractère	`ASCII(c)`	Représentation décimale de `c`.
`AVG`	Groupement	`AVG(DISTINCT|ALL n)`	Moyenne de `n`. `ALL` est la valeur par défaut.
`CEIL`	Nombre	`CEIL(n)`	Plus petit entier égal ou supérieur à `n`.
`CHARTOROWID`	Conversion	`CHARTOROWID(c)`	Convertit un caractère en un type `rowid`.
`CHR`	Caractère	`CHR(n)`	Caractère dont l'équivalent est `n`.
`CONCAT`	Caractère	`CONCAT(1,2)`	Caractère 1 concaténé avec le caractère 2.
`CONVERT`	Conversion	`CONVERT (a,dest_c [,source_c])`	Convertit une chaîne de caractères `a` d'un ensemble de caractères vers un autre.
`COS`	Nombre	`COS(n)`	Cosinus de `n`.
`COSH`	Nombre	`COSH(n)`	Cosinus hyperbolique de `n`.
`COUNT`	Groupement	`COUNT(DISTINCT|ALL e)`	Nombre de lignes dans une requête. `ALL` est la valeur par défaut. `e` peut être représenté par `*` pour indiquer toutes les colonnes.
`EXP`	Nombre	`EXP(n)`	e à la puissance `n`.

FLOOR	Nombre	FLOOR(n)	Plus grand entier égal ou inférieur à n.
GREATEST	Autre	GREATEST(e [,e]...)	Le plus grand de la liste des expressions e.
HEXTORAW	Conversion	HEXTORAW(c)	Convertit un caractère hexadécimal c en raw.
INITCAP	Caractère	INITCAP(c)	c avec la première lettre de chaque mot en majuscule.
INSTR	Caractère	INSTR(1, 2 [, n [, m]])	Recherche dans 1 à partir du caractère n une m-ième occurence de 2 et renvoie la position de cette occurence.
INSTRB	Caractère	INSTRB(1,2[,n[,m]])	Identique à INSTR, mais les paramètres numériques sont exprimés en octets.
LAST_DAY	Date	LAST_DAY(a)	Dernier jour du mois contenant a.
LEAST	Autre	LEAST(e [,e]...)	La plus petite des expressions e.
LENGTH	Caractère	LENGTH(c)	Nombre de caractères dans c. Si c est une donnée de longueur fixe (char), tous les blancs de fin sont inclus.
LENGTHB	Caractère	LENGTHB(c)	Identique à LENGTH sauf en octets.
LN	Nombre	LN(n)	Logarithme népérien de n, si $t > 0$.
LOG	Nombre	LOG(b,n)	Logarithme de n en base b.
LOWER	Caractère	LOWER(c)	c avec toutes les lettres en minuscule.
LPAD	Caractère	LPAD(1,n [,2])	Caractère 1 rempli à gauche pour obtenir une longueur de n. Si le caractère 2 n'est pas omis, il est utilisé pour le remplissage à la place des blancs par défaut.
LTRIM	Caractère	LTRIM(c [,set])	Retire des caractères de la gauche de c. Si l'ensemble s est défini, retire les caractères initiaux jusqu'au premier ne se trouvant pas dans l'ensemble.
MAX	Autre	MAX(DISTINCT|ALL e)	Maximum de l'expression e. ALL est la valeur par défaut.
MIN	Autre	MIN(DISTINCT|ALL e)	Minimum de l'expression e. ALL est la valeur par défaut.
MOD	Nombre	MOD(r,n)	Reste de r divisé par n.
MONTHS_BETWEEN	Date	MONTHS_BETWEEN(a,b)	Nombre de jours entre les dates a et b.
NEW_TIME	Date	NEW_TIME(a, z1, z2)	Date et heure dans le fuseau horaire z2 lorsque la date et l'heure exprimés dans le fuseau z1 est a.

NEXT_DAY	Date	NEXT_DAY(a, c)	Date du premier jour de la semaine identifié par c qui est postérieur à la date a.
NLSSORT	Caractère	NLSSORT(c [,parm])	Chaîne d'octets pour trier c.
NLS_INITCAP	Caractère	NLS_INITCAP(c [,parm])	c avec la première lettre de chaque mot en majuscules. parm a la forme de NLS_SORT = s où s est une sorte linguistique ou binaire.
NLS_LOWER	Caractère	NLS_LOWER(c [,parm])	c avec toutes les lettres en minuscule. Pour parm, voir plus haut.
NLS_UPPER	Caractère	NLS_UPPER(c [,parm])	c avec toutes les lettres en majuscule. Pour parm, voir plus haut.
NVL	Autre	NVL(e1, e2)	Si e1 est nul, retourne e2. Sinon, retourne e1.
POWER	Nombre	POWER(m,n)	m élevé à la puissance n.
RAWTOHEX	Conversion	RAWTOHEX(raw)	Convertit une valeur raw en son équivalent hexadécimal.
REPLACE	Caractère	REPLACE(c, s1 [, r2])	Remplace chaque occurence de la chaîne s1 dans c par r2. Si r2 est omis, toutes les occurences de s1 sont supprimées.
ROUND	Date	ROUND(n [,f])	Date arrondie au format modèle f. Si f est omis, n sera arrondi au jour le plus proche.
ROUND	Nombre	ROUND(n[,m])	n arrondi à m chiffres à droite de la virgule. Si m est omis, on arrondi à 0 chiffres.
ROWIDTOCHAR	Conversion	ROWIDTOCHAR(rowid)	Convertit le format rowid au format varchar2 avec une longueur de 18.
RPAD	Caractère	RPAD(1, n [, 2])	1 est complété à droite pour obtenir la longueur n. Le remplissage se fait avec 2.
RTRIM	Caractère	RTRIM(c [, s])	c avec des caractères retirés après le dernier caractère qui ne se trouve pas dans l'ensemble s. Si s est omis, sa valeur par défaut est ''.
SIGN	Nombre	SIGN(n)	-1 si $n < 0$, 0 si $n = 0$, 1 si $n > 0$.
SIN	Nombre	SIN(n)	Sinus de n.
SINH	Nombre	SINH(n)	Sinus hyperbolique de n.
SOUNDEX	Caractère	SOUNDEX(c)	une chaîne avec la représentation phonétique « soundex » de c.

SUBSTR	Caractère	SUBSTR(c, m [,n])	Une portion de c commençant au caractère numéro m et s'étendant sur n caractères. Si m est négatif, Oracle compte en arrière à partir de la fin de c. Si n est omis, tous les caractères jusqu'au dernier sont retournés.
SUBSTRB	Caractère	SUBSTRB(c, m [,n])	Comme SUBSTR sauf que m et n sont des nombres d'octets.
SQRT	Nombre	SQRT(n)	Racine carrée de n.
STDDEV	Groupement	STDDEV(DISTINCT|ALL n)	Écart-type du nombre n.
SUM	Groupement	SUM(DISTINCT|ALL n)	Somme des nombres n.
SYSDATE	Date	SYSDATE	Date et heure courante.
TAN	Nombre	TAN(n)	Tangente de n.
TANH	Nombre	TANH(n)	Tangente hyperbolique de n.
TO_CHAR	Conversion	TO_CHAR(d [,f [,parm])	Convertit la date d dans le type varchar2 avec le format f et parm de type nls_date_language.
TO_CHAR	Conversion	TO_CHAR (n [,f [,parm])	Convertit le nombre n en son équivalent varchar2 avec le format f et parm.
TO_DATE	Conversion	TO_DATE (c [, f [, parm])	Convertit c de type varchar2 en une date avec le format f et parm de type nls_date_language.
TO_MULTI_BYTE	Conversion	TO_MULTI_BYTE(c)	Convertit c en son équivalent multioctet.
TO_NUMBER	Conversion	TO_NUMBER (c [,f [, parm]])	Convertit le caractère c en un nombre en utilisant le format f et parm de type nls_date_language.
TO_SINGLE_BYTE	Conversion	TO_SINGLE_BYTE(c)	Convertit un caractère multioctet c en son équivalent sur un octet.
TRANSLATE	Caractère	TRANSLATE(c, f, t)	c où chaque occurence de f est remplacée par l'occurence correspondante dans t.
TRUNC	Date	TRUNC(c [,f])	c avec la partie heure tronquée au format f.
TRUNC	Nombre	TRUNC(n[,m])	n tronqué à m décimales. Si m est omis, on tronque à 0 décimales.
UID	Autre	UID	Un entier qui identifie de manière unique l'utilisateur.
USER	Autre	USER	Utilisateur courant sous la forme varchar2.
UPPER	Caractère	UPPER(c)	c avec toutes les lettres en majuscule.
VARIANCE	Groupement	VARIANCE(DISTINCT|ALL n)	Variance du nombre n.
VSIZE	Autre	VSIZE(e)	Nombre d'octets de la représentation interne de e.

Voyons maintenant quelques exemples utilisant des fonctions.

```
SELECT SUBSTR(addrs_1,1,30),city, state, zip
FROM addresses
WHERE addrs_1 is not null
  AND UPPER(city)  =  'ROCHESTER'
  AND TO_NUMBER(SUBSTR(zip,1,5)) > 14525
  AND NVL(active_date,SYSDATE)  >  TO_DATE('01-JAN-90');
```

On notera l'emploi de la fonction `UPPER`. Lorsqu'Oracle compare des chaînes de caractères, la casse (majuscule ou minuscule) doit correspondre exactement. Ainsi, `'Rochester'` n'est pas égal à `'ROCHESTER'`. La fonction `UPPER` va assurer que la colonne de la ville sera convertie en majuscule avant la comparaison avec le litéral `'ROCHESTER'`.

La fonction `SUBSTR` est aussi utilisée pour extraire les caractères 1 à 30 de la colonne `addrs_1`. Tous les caractères restants ne seront pas visibles. Cette fonction est aussi utilisée dans la clause `WHERE` pour extraire les cinq premiers caractères de la colonne du code postale (`zip`) avant de la convertir en une valeur numérique. La comparaison est faite après que la conversion soit faite.

Si la colonne `active_date` contient des valeurs nulles (pas de valeur), elles seront incluses dans l'ensemble des données en raison de la fonction `NVL`. Si `active_date` est nul, la date courante sera renvoyée avant que la comparaison avec la constante `'01-JAN-90'` soit faite. La constante `'01-JAN-90'` est convertie dans une date pour assurer une compatibilité avec le format. Pour une liste de tous les formats, voir le manuel de référence d'Oracle.

```
SELECT dept_no,SUM(emp_salary),AVG(emp_salary)
FROM emp
WHERE dept_no = dept_no
GROUP BY dept_no;
```

Cet exemple montre l'emploi des fonctions `SUM` et `AVG`. Les données extraites vont montrer la somme des salaires des employés et le salaire moyen par département. Il faut noter que la clause `GROUP BY` doit être utilisée dans cette requête.

4.2 Connaître ses tables et vues

Pour assurer que les données contiennent toutes les colonnes et restrictions nécessaires, il faut être familier avec le schéma de la base de données. Si un diagramme du schéma n'est pas disponible, il y a plusieurs moyens de trouver quelles tables ou vues peuvent être nécessaires pour écrire des requêtes. Une manière est de regarder certaines des tables du dictionnaire de données.

Pour voir tous les noms de tables du dictionnaire des données, on peut faire la requête suivante :

```
SELECT table_name
FROM dictionary
ORDER BY table_name;
```

Certaines des tables utiles devraient être `all_tables`, `all_columns`, `all_views` et `all_constraints`.

Pour voir les noms des colonnes de ces tables, il faut indiquer '`DESC table_name`'. `DESC` correspond à `DESCribe` et '`table_name`' est le nom de la table concernée, comme par exemple '`all_tables`'. Par conséquent, '`DESC all_tables`' va rendre toutes les colonnes et tous les types de données pour la table '`all_tables`'.

À l'aide des tables du dictionnaire de données, il est possible de déterminer quelles tables, vues et contraintes sont effectives por l'application considérée.

4.3 Jointures de tables

Les tables sont physiquement jointes dans la clause `FROM` de la requête. Elles sont logiquement jointes dans la clause `WHERE`. Les colonnes des tables qui apparaissent dans la clause `WHERE` doivent avoir leur nom de table dans la clause `FROM`. La clause `WHERE` est l'endroit où les tables sont liées.

La manière avec laquelle la clause `WHERE` est construite a une forte incidence sur l'efficacité de la requête. Une jointure de deux tables n'est pas forcément plus rapide qu'une jointure de dix tables.

S'il y a beaucoup de requêtes qui ont un grand nombre de tables jointes ensemble (par exemple plus que sept tables), il peut être nécessaire d'envisager la dénormalisation de certains éléments de donnée pour réduire le nombre de jointures de table. Ce type de dénormalisation peut être requis lorsque la productivité de l'utilisateur ou la performance du système ont chuté de manière significative.

La table suivante montre trois tables que les exemples suivant utiliseront.

Nom de la table	Nom de la colonne	Type de donnée
emp	emp_id	number(6)
emp	adrs_id	number(6)
emp	first_name	varchar2(40)
emp	last_name	varchar2(40)
emp	dept_no	number(3)
emp	hire_date	date
emp	job_title	varchar2(40)
emp	salary	number(6)
emp	manager_id	number(6)
dept	dept_no	number(3)
dept	name	varchar(40)
dept	adrs_id	number(6)
addresses	adrs_id	number(6)
addresses	active_date	date
addresses	box_number	number(6)
addresses	adrs_1	varchar2(40)
addresses	adrs_2	varchar2(40)
addresses	city	varchar2(40)
addresses	state	varchar2(2)
addresses	zip	varchar2(10)

Dans l'exemple suivant, une requête est écrite pour donner la liste de tous les départements avec les employés correspondants et la ville dans laquelle le département se trouve.

```
SELECT d.name,e.last_name,e.first_name,a.city
FROM emp e,dept d,addresses  a
WHERE d.dept_no    = e.dept_no
  AND a.adrs_id    = d.adrs_id
ORDER BY d.name,e.last_name,e.first_name;
```

Si la ville de l'employée doit aussi être extraite, la requête pourrait être formulée de la manière suivante :

```
SELECT d.name,a.city  dept_city,e.last_ name,
       e.first_name,z.city  emp_city
FROM emp e,dept d,addresses  a,addresses  z
WHERE  d.dept_no = e.dept_no
   AND  a.adrs_id      = d.adrs_id
   AND  z.adrs_id      = e.adrs_id
ORDER BY d.name,e.last_name,e.first_name;
```

Dans cet exemple, la table `addresses` a été jointe deux fois, permettant ainsi d'obtenir à la fois la ville correspondant au département et celle correspondant à l'employé. Afin de clarifier la sortie, des alias ont été donnés aux différentes colonnes de villes dans la partie `SELECT` de la requête.

L'exemple suivant ajoute le nom du manager de l'employé à la requête.

```
SELECT d.name,a.city  dept_city,e.last_name,
       e.first_name,z.city  emp_city,m.first_name || m.last_name manager
FROM emp e,dept d,addresses  a,addresses  z,emp m
WHERE  d.dept_no = e.dept_no
   AND  a.adrs_id      = d.adrs_id
   AND  z.adrs_id      = e.adrs_id
   AND  m.emp_id       = e.manager_id
ORDER BY    d.name,e.last_name,e.first_name;
```

La sortie de cette requête va faire apparaître la colonne du manager (alias) comme une seule colonne, bien qu'elle soit constituée de deux colonnes. Le symbole « `||` » est utilisé pour concaténer des colonnes.

4.4 Éviter les jointures cartésiennes

Une jointure cartésienne se produit lorsque la clause `WHERE` n'est pas correctement construite. Un enregistrement est alors renvoyé pour toutes les occurences dans les tables `Z` et `X`. C'est le cas dans l'exemple ci-dessous.

```
SELECT X.name,Z.last_name,Z.first_name
FROM emp Z,dept X
ORDER BY X.name, Z.last_name;
```

Si la table `emp` a 10 employés et que la table `dept` contient trois départements, cette requête retourne 30 lignes. Pour chaque nom de département, tous les employés sont listés parce que les tables ne sont pas jointes correctement (pas du tout dans cet exemple). Avec la condition de jointure `WHERE X.dept_no = Z.dept_no`, seules 10 lignes sont rendues.

4.5 Jointures externes

Lorsque les colonnes d'une table sont jointes de manière externe, ceci indique à la base de donnée d'extraire des lignes même lorsque des données ne sont pas trouvées. Le symbole « `+` » est utilisé pour dénoter une condition de jointure externe, comme illustré dans l'exemple suivant :

```
SELECT d.name,a.city,e.last_name,e.first_name
FROM emp e,dept d,addresses a
WHERE  d.dept_no(+)   = e.dept_no
   AND  a.adrs_id      = d.adrs_id
ORDER BY d.name,e.last_name,e.first_name;
```

Si le président de l'entreprise n'a jamais fait partie d'un département, son nom n'aurait jamais été extrait dans les exemples précédents car son numéro de département aurait été nul. La jointure externe conduit à l'extraction de toutes les lignes même lorsqu'il n'y a pas de correspondance pour `dept_no`.

Les jointures externes sont effectives mais rendent la requête plus lente. Il peut être nécessaire de récrire la requête pour en améliorer l'efficacité.

4.6 Sous-requêtes

Les sous-requêtes, ou requêtes imbriquées, sont utilisées pour récupérer un ensemble de lignes afin de les utiliser par la requête père. Suivant la manière avec laquelle la requête est écrite, elle peut être exécutée une fois pour la requête père ou bien elle peut être exécutée une fois pour chaque ligne rendue par la requête père. Si la sous-requête est exécutée pour chaque ligne du père, on parle de sous-requête corrélée.

Une sous-requête corrélée peut être aisément identifiée si elle contient des références aux colonnes du parent dans sa clause `WHERE`. Les colonnes de la sous-requête ne peuvent pas être référencées ailleurs que dans la requête parent. L'exemple suivant montre une sous-requête non corrélée.

```
SELECT e.first_name,e.last_name,e.job_title
FROM emp  e
WHERE e.dept_no in (SELECT dept_no
                    FROM dept
                    WHERE name = 'ADMIN');
```

Dans cet exemple, tous les noms d'employés et intitulés de professions (`job_title`) vont être extraits pour le département `'ADMIN'`. On remarque l'emploi de l'opérateur `IN` en référence à la sous-requête. L'opérateur `IN` est utilisé lorsqu'une ou plusieurs lignes peuvent être retournées par une sous-requête. Si l'opérateur d'égalité (`=`) est utilisé, cela suppose qu'une seule ligne est retournée. Dans le cas contraire, Oracle renvoie une erreur.

Cette instruction aurait pu être écrite en joignant directement la table `dept` avec la table `emp` dans la requête principale. Les sous-requêtes sont quelquefois utilisées pour obtenir une meilleure performance. Si la requête parent comprend de nombreuses tables, il peut être avantageux de scinder la clause `WHERE` en sous-requêtes.

```
SELECT d.name,e.first_name,e.last_name,e.job_title
FROM emp  e,dept d
WHERE e.dept_no = d.dept_no
AND d.adrs_id = (SELECT adrs_id
                 FROM ADDRESSES
                 WHERE adrs_id = d.adrs_id)
ORDER BY d.name, e.job_title, e.last_name;
```

Dans cet exemple, tous les employés et leur département respectif vont être extraits uniquement pour les départements qui ont une valeur `adrs_id` valide dans la table `addresse`. Il s'agit d'une sous-requête corrélée car la sous-requête fait référence à une colonne de la requête principale.

```
SELECT d.name,e.first_name,e.last_name,e.job_title
FROM emp  e,dept d
WHERE e.dept_no = d.dept_no
      AND not exists (SELECT 'X'
                      FROM ADDRESSES
                      WHERE city in ('ROCHESTER','NEW YORK')
                        AND adrs_id = d.adrs_id)
ORDER BY d.name, e.job_title, e.last_name;
```

Cet exemple va rendre tous les départements et employés, sauf lorsque les départements sont situés à `'ROCHESTER'` et `'NEW YORK'`. `SELECT 'X'` va rendre une réponse de type `true` ou `false` qui sera évaluée par l'opérateur `not exists`. N'importe quelle constante peut être utilisée ici ; `'X'` est juste un exemple.

5 L'instruction `DECODE`

Une des instructions SQL les plus puissantes et négligées est l'instruction `DECODE`. Cette instruction a la syntaxe suivante :

```
DECODE(val, exp1, exp2, exp3, exp4, ..., def);
```

`DECODE` va d'abord évaluer la valeur ou expression `val` puis comparer l'expression `exp1` à `val`. Si `val` est égal à `exp1`, l'expression `exp2` sera retournée. Si `val` n'est pas égal à `exp1`, l'expression `exp3` sera évaluée et

retournera l'expression `exp4` si `val` est égal à `exp3`. Ce processus continue jusqu'à ce que toutes les expressions aient été évaluées. S'il n'y a pas de correspondance, la valeur par défaut `def` est renvoyée.

```
SELECT e.first_name,e.last_name,e.job_title,
       DECODE(e.job_title, 'President', '******', e.salary)
FROM emp e
WHERE e.emp_id in (SELECT NVL(z.manager_id, e.emp_id)
                   FROM emp z);
```

Dans cet exemple, tous les noms de manager sont extraits avec leur salaire. Lorsque la ligne identifiant le président est affichée, on affiche `'******'` à la place du salaire. On notera aussi l'emploi de la fonction `NVL` pour évaluer un manager ayant une valeur d'identificateur nulle. Seul le président aura une valeur d'identificateur nulle, et il n'aurait pas été extrait sans `NVL`.

Il faut aussi remarquer que `DECODE` évalue `job_title` et renvoie le salaire, ce qui aurait normalement été la cause d'une erreur de type, mais cela ne pose pas de problèmes ici.

```
SELECT e.first_name, e.last_name,e.job_title,e.salary
FROM emp e
WHERE DECODE(USER,'PRES',e.emp_id,UPPER(e.last_name),e.emp_id,  0) = e.emp_id ;
```

Dans cet exemple, si l'utilisateur est le président, tous les employés seront retournés avec leur salaire correspondant. Pour tous les autres utilisateurs, seule une ligne sera extraite, permettant à l'utilisateur de voir son salaire uniquement.

```
SELECT e.first_name,e.last_name,e.job_title,
  DECODE(USER,'ADMIN',DECODE(e.job_title, 'PRESEDENT', '*****', e.salary),
  'PRES', e.salary, '******')
FROM emp e
WHERE e.emp_id in (SELECT NVL(z.manager_id, e.emp_id)
                   FROM emp z);
```

Dans cet exemple, l'instruction `DECODE` est imbriquée dans une autre instruction `DECODE`. Si l'utilisateur Oracle est `'ADMIN'`, on montre les salaires sauf celui du président. Si l'utilisateur est `'PRES'`, on montre tous les salaires et si l'utilisateur est toute autre personne, on renvoie `'******'`.

Il est aussi possible d'utiliser l'instruction `DECODE` dans la clause `ORDER BY`. L'exemple suivant va trier la sortie de telle manière que le président est sur la première ligne, suivie par les départements `'SALES'`, `'ADMIN'` puis `'IS'` avec leurs employés correspondants.

```
SELECT d.name, e.job_title, e.first_name, e.last_name
FROM emp e, dept d
WHERE d.dept_no = e.dept_no
ORDER BY DECODE(e.job_title,'PRESIDENT', 0,
           DECODE(d.name,'SALES',   1,
            'ADMIN',   2, 3)), e.last_name;
```

Cet exemple ne fait pas de `ORDER BY e.job_title` mais utilise cette colonne pour chercher le titre `'PRESIDENT'` et renvoie un 0. Pour toutes les autres lignes, un autre `DECODE` est utilisé pour évaluer le nom du département et rendre les nombres 1, 2 ou 3 suivant le nom du département. Après que les `DECODE`s soient finis, les données sont encore triées par le nom de l'employé `e.last_name`.

6 INSERT, UPDATE et DELETE

L'instruction `INSERT` est utilisée pour ajouter de nouvelles lignes dans la base de données. Ceci peut être fait à raison d'une ligne à la fois en utilisant l'expression `VALUES`, ou avec un ensemble d'enregistrements en utilisant une sous-requête. La syntaxe de l'instruction `INSERT` est :

```
INSERT INTO schema.table column(s) VALUES subquery
```

où

- `schema` est un paramètre optionnel pour identifier le schéma de base de donnée utilisé pour l'insertion. Par défaut, c'est le schéma de l'utilisateur.
- `table` est obligatoire et est le nom de la table.
- `column` est la liste des colonnes qui vont recevoir les valeurs insérées.
- `VALUES` est utilisé lorsqu'une ligne de données est insérée. Les valeurs sont représentées comme des constantes.
- `subquery` est utilisé lorsque l'option `VALUES` n'est pas utilisée. Les colonnes de la sous-requête doivent correspondre à l'ordre et aux types des données des colonnes dans la liste de la commande `INSERT`.

```
INSERT INTO dept  (dept_no, name, adrs_id)
     VALUES (dept_seq.NEXTVAL,'CUSTOMER SERVICE',adrs_seq.NEXTVAL);
```

Cet exemple insère une ligne dans la table `dept`. Les séries `dept_seq` et `adrs_seq` sont utilisées pour extraire les valeurs numériques suivantes de `dept_no` et `adrs_id`.

Si plusieurs lignes doivent être insérées, l'instruction `INSERT` aurait dû être exécuté pour chaque ligne individuelle. Si une sous-requête peut être utilisée, plusieurs lignes seraient insérées pour chaque ligne rendue par la sous-requête.

```
INSERT INTO emp (emp_id, first_name,
       last_name,dept_no,hire_date,job_title,salary,manager_id)
SELECT emp_seq.NEXTVAL,new.first_name,
    new.last_name,30,SYSDATE,
   'CUSTOMER REPRESENTATIVE',new.salary,220
FROM candidates new
WHERE new.accept      = 'YES'
   AND new.dept_no = 30;
```

Cet exemple va insérer toutes les lignes de la table `candidates` correspondant au numéro de département 30. Comme le numéro de département et l'identification du manager sont connues, ces informations sont utilisées en tant que constantes dans la sous-requête.

L'instruction `UPDATE` est utilisée pour changer des lignes existantes dans la base de données. La syntaxe de l'instruction `UPDATE` est

```
UPDATE  schema.table SET column(s) = expr sub query  WHERE condition
```

où

- `schema` est un paramètre optionnel pour identifier le schéma de base de données utilisé pour la mise à jour. Par défaut, il s'agit du schéma de l'utilisateur.
- `table` est obligatoire et est le nom de la table.
- `SET` est un mot clé obligatoire réservé.
- `column` est une liste de colonnes qui vont recevoir les valeurs mises à jour.
- `expr` est la nouvelle valeur à affecter.
- `sub query` est une instruction `SELECT` qui va extraire les nouvelles valeurs des données.
- `WHERE` est optionnel et est utilisé pour restreindre les lignes qui vont être mises à jour.

```
UPDATE emp
SET dept_no = 30
WHERE last_name = 'DOE'
  AND first_name = 'JOHN';
```

Cet exemple va transférer un employé nommé `JOHN DOE` dans le département 30. S'il y a plus d'un `JOHN DOE`, d'autres restrictions devront être apportées à la clause `WHERE`.

```
UPDATE emp
SET salary = salary + (salary * .05);
```

Cet exemple de mise à jour va donner à toutes les personnes de la table `emp` une augmentation de salaire de 5 pour cents.

```
UPDATE emp a
SET a.salary = (SELECT a.salary
        + (a.salary * DECODE(d.name, 'SALES', .1,
                                     'ADMIN', .07,
                                              .06))
FROM dept d
WHERE d.dept_no = a.dept_no)
WHERE a.dept_no = (SELECT dept_no
                   FROM dept y, addresses z
                   WHERE y.adrs_id = z.adrs_id
                   AND z.city    = 'ROCHESTER');
```

Cet exemple va donner des augmentations aux employés localisés à Rochester. Le montant de l'augmentation est traité par l'instruction `DECODE` qui évalue le nom du département. Les employés du département des ventes (`Sales`) vont recevoir une augmentation de 10 pour cents, ceux du département administratif (`Admin`) sept pour cents et tous les autres six pour cents.

L'instruction `DELETE` est utilisée pour retirer des lignes de la base de données. La syntaxe de `DELETE` est :

```
DELETE FROM schema.table WHERE condition
```

où

- `SCHEMA` est un paramètre optionnel pour identifier le schéma de base de donnée utilisé pour le `DELETE`. Par défaut il s'agit du schéma de l'utilisateur.
- `TABLE` est obligatoire et est le nom de la table.
- `WHERE` restreint l'opération `DELETE`.

```
DELETE FROM addresses
WHERE adrs_id = (SELECT e.adrs_id
FROM emp e
WHERE e.last_name   = 'DOE'
  AND e.first_name  = 'JOHN');

DELETE FROM emp  e
WHERE e.last_name   = 'DOE'
  AND e.first_name  = 'JOHN';
```

Si l'employé `John Doe` quitte l'entreprise, on va probablement vouloir le supprimer de la base de donnée. Une manière de faire ceci est de supprimer la ligne contenant son nom des tables `addresses` et `emp`. Afin de trouver `John Doe` dans la table `addresses`, il faut exécuter une sous-requête en utilisant la table `emp`. Par conséquent, l'entrée de la table `emp` doit être la dernière ligne à être supprimée, sans quoi il y aurait une ligne orpheline dans la table `addresses`.

```
DELETE FROM dept
WHERE adrs_id is null;
```

Dans cet exemple, toutes les lignes de la table `dept` vont être supprimées si la valeur correspondante `adrs_id` est nulle.

Une opération de suppression est permanente ! Une fois faite, il est impossible de récupérer la ou les ligne(s) autrement que par une opération `INSERT`. Il n'y a pas de commande `undo`.

7 SQL parent/enfant

Chaque fois qu'une instruction SQL est construite avec plusieurs tables, on a généralement une relation parent/enfant qui est effective.

L'utilisateur doit être familier avec le schéma de la base utilisée et des contraintes correspondantes afin de pouvoir convenablement faire des jointures. L'écriture d'instructions `SELECT` négligeamment construites ne va pas endommager la base de données mais pourrait nuire aux performances du système et éventuellement donner aux utilisateurs une fausse représentation des relations. Avec des `INSERT`, `UPDATE` ou `DELETE` mal construits, l'effet peut être désastreux.

Avant de voir des exemples, quelques hypothèses sont de rigueur :

1. Un employé ne peut pas être entré sans un numéro de département. Ceci indique que la table `emp` est fille de la table `dept`.
2. Les adresses n'ont pas besoin d'être entrées lors de la création d'un nouvel employé ou département. Par conséquent, la table `addresses` est optionnelle et est une fille de la table `emp` et une fille de la table `dept`.

Si ces contraintes sont inscrites dans la base de données, une protection sera procurée lorsqu'une ligne parent est supprimée mais ne conduit pas à la suppression de l'enfant correspondant.

```
SELECT d.name  dept_name,
       d.dept_no                        dept_number,
       e.first_name || e.last_name      emp_name,
       e.job_title                      title,
       e.hire_date                      start_date
FROM dept    d, emp     e
WHERE d.dept_no   =  e.dept_no
ORDER BY d.name, e.last_name;
```

Dans cet exemple, tous les noms et numéros de département seront affichés (parent) avec tous les employés correspondant (enfants) des départements.

```
SELECT d.name dept_name, d.dept_no   dept_number,
       e.first_name || e.last_name    emp_name,
       e.job_title    title,
       e.hire_date   start_date,
       DECODE(a.box_number, NULL, a.adrs_1, a.box_number)  address,
       DECODE(a.adrs_2, NULL, NULL, a.adrs_2)  address_2,
       a.city || ', '||a.state ||'   '||a.zip     city_stat_zip
FROM dept  d, emp    e, addresses  a
WHERE d.dept_no   =  e.dept_no
   AND e.adrs_id   =  a.adrs_id (+)
ORDER BY d.name, e.last_name;
```

Cet exemple montre l'addition de la table optionnelle fille appelée `addresses`. Une jointure externe (`+`) est utilisée de telle sorte que la ligne de l'employé va être extraite même s'il n'y a pas encore d'information d'adresse disponible. Les `DECODE`s vont extraire le numéro de boîte (`box_number`) ou l'adresse 1 (`adrs_1`) suivant que le numéro de boîte existe ou non.

Lors de l'écriture d'`INSERT`s, `UPDATE`s ou de `DELETE`s, il faut faire attention et être sûr que les relations convenables existent au sein des sous-requêtes. Si chaque ligne de la requête doit être manipulée, il faut utiliser un curseur qui fait partie du langage PL/SQL.

8 Quelques trucs et astuces

Voici un petit résumé de cette partie avec quelques nouvelles idées.

1. Lors de la comparaison de types de données `date`, il peut être sage de tronquer les dates (`TRUNC hire_date`) pour être sûr que la partie horaire ne cause pas de résultats erronés. Si l'application autorise l'insertion de l'heure dans le type de données `date`, alors les heures insérées seront aussi prises en compte en manipulant des dates.
2. En écrivant des instructions SQL, les valeurs nulles ne seront pas considérées par la base de données, à moins qu'elles soient expressement demandées.

```
SELECT e.first_name || e.last_name      emp_name,
       z.first_name || z.last_name      manager
FROM emp e,emp z
WHERE  z.emp_id = e.manager_id;
```

Dans cet exemple, toutes les lignes de la table des employés vont être extraites sauf celle concernant le président. Il en est ainsi parce que pour le président `manager_id` est nul.

L'exemple suivant montre comment récupérer le président en même temps que les autres employés.

```
SELECT e.first_name || e.last_name      emp_name,
       z.first_name || z.last_name      manager
FROM emp e,emp z
WHERE  z.emp_id = NVL(e.manager_id, e.emp_id);
```

Ce code teste si l'identificateur du manager (`manager_id`) est nul et si oui, la base de donnée va rendre l'identificateur de l'employé (`emp_id`), ce qui fonctionnera et rendra une ligne pour le président.

9 Résumé

Ce chapitre a couvert d'importants aspects du langage SQL en insistant sur des fonctionnalités qui sont souvent utilisées ou mal comprises (comme avec l'instruction `DECODE`). Le lecteur devrait avoir suffisamment d'information pour commencer l'aventure et écrire du bon code SQL.