**PLAN DU COURS**

**BASE DE DONNEES II**

CHAP **I. IIème PARTIE ANALYSE DES DONNEES**

**CHAP II. IIème PARTIE INTRODUCTION A SQL**

**I. PARTIE ANALYSE DES DONNEES**

**1) Introduction**

Le modèle entité-association est un outil (et une technique) d'analyse permettant de construire des schémas théoriques de raisonnement sur des applications tournant avec les bases de données relationnelles. Pour mettre en œuvre cette technique, en connaître les possibilités et les limites, il n'est pas nécessaire de posséder une base de donnée.

Bien connaître les règles simples des schémas entités-associations (aussi appelé entités-relations) permet d'affiner petit à petit une application apparemment simple, sans avoir besoin de la programmer, et par conséquent d'économiser du temps de conception tout en obtenant une plus grande souplesse au niveau de l'analyse.

[**2. Du vocabulaire et comment on le représente graphiquement**](#L2)

Le modèle **entité-association** est constitué de deux éléments de base :

* Les entités, qui sont des regroupements d'informations, et possèdent des attributs (caractéristiques) ;
* Les associations qui sont les liens logiques entre les entités ;

**2.1. Les entités**

*Ce sont des regroupements d'informations*. Les informations contenues dans les entités (informations que l'on appelle "attributs") doivent être des informations variables, mais communes à une même classe d'objets.

Par exemple, si l'on considère l'entité "être humain" les informations communes aux être humains peuvent être :

* le nom
* le ou les prénoms
* la date de naissance
* le lieu de naissance
* le sexe
* l'adresse

On considère souvent qu'il s'agit de "classes" d'entités.

Une entité donnée peut être constituée de sous-classes. Par exemple, un être humain donné peut habiter au même endroit qu'un autre. Dans ce cas, l'adresse constitue une sous-classe de l'entité "être humain", c'est à dire une nouvelle entité à part entière.

D'un autre côté, il arrive souvent que plusieurs personnes résident au même endroit, sans même se connaître (cas d'un immeuble collectif par exemple). Dans ce cas on peut considérer l'adresse, comme une entité et la décrire de la manière suivante :

* Pays
* Région
* Département
* Rue

On schématise une entité par un rectangle.

**Exemple :**



**2.2. Les attributs**

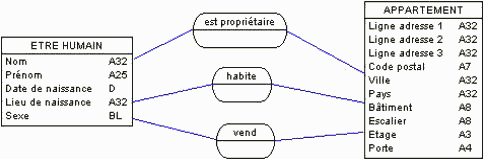
*Les attributs sont les caractéristiques décrivant les entités* et doivent être représentés comme une liste de mots, la plus simple possible, dans le cadre de l'entité correspondante. On devra préciser le type des données attendues pour chaque attribut.

**2.3. Les associations (ou relations)**

*Ce sont des liaisons logiques entre les entités.* Elles peuvent être de nature factuelle, ou de nature dynamique. Par exemple, une personne peut acheter un objet (action d'acheter), mais si l'on considère qu'une personne est propriétaire d'un objet, alors l'association entre l'objet et cette personne est purement factuelle.

Les associations se représentent dans une ellipse (ou un rectangle aux extrémités rondes), reliée par des traits aux entités qu'elles lient logiquement.

Exemple :



**2.4. Les cardinalités**

*Les cardinalités, au sens arithmétique du terme, permettent de dénombrer les éléments de l'entité d'arrivée en relation avec un élément de l'entité de départ, et vice versa.*

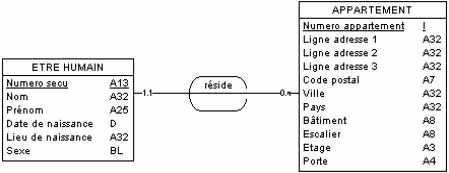
**Exemple :**

Considérons le cas de l'association "habite" et les deux entités "être humain" et "appartement" du schéma précédent, les cardinalités minimales et maximales sont les suivantes :

* sens "être humain" vers "appartement" : 1 (minimum) et 1 (maximum)
* sens "appartement" vers "être humain" : 0 (minimum) et n (maximum)

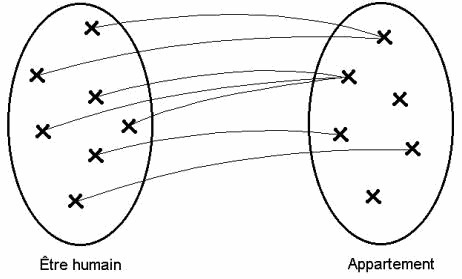
Ce qui signifie que dans cette modélisation un être humain réside dans un appartement et un seul à la fois, mais qu’un appartement peut se trouver vide ou être pourvu de plusieurs résidents.

Mais il convient de se concentrer sur ce que l’on doit modéliser et non sur l’univers entier. En outre nous avons convenu qu’un même être humain ne pouvait résider dans plusieurs appartements à la fois (notion de « résidence principale » par exemple).On note les cardinalités de chaque côté de l'association, sur les traits faisant la liaison entre l'association et l'entité.

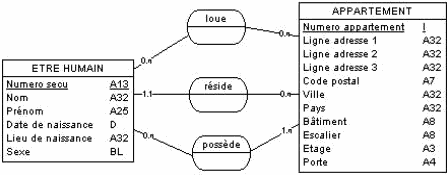


Dès cet instant, on peut en déduire le type de relation parmi les types suivants : 1/1, 1/n ou n/1 ou encore n/m. Ici c’est une relation de type 1, n. Pour déduire le type d’une relation, il suffit de récupérer les cardinalités maximales des deux côtés de l’association, sans tenir compte de leur valeur exacte.

On peut s’y aider à l’aide des diagrammes de Wen et des notions de mathématiques ensemblistes.



**Attention !** Des relations différentes entre mêmes entités peuvent posséder des cardinalités différentes; c'est même souvent le cas.



La relation « loue » est de type n:m.   
La relation « réside » est de type 1:n.  
La relation « possède » est de type n:m.

**2.5. Clef d'une entité (la base de la relation)**

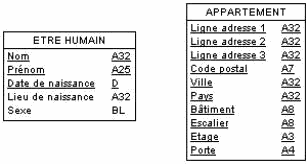
*C'est un attribut (ou un ensemble d'attributs) qui permet de distinguer un élément de l'entité de manière unique et sans aucune ambiguïté par rapport à l'ensemble des autres éléments*, et à l'univers de tous les éléments qui peuvent entrer un jour ou l'autre dans cette entité.

Par exemple la clef de l'entité "être humain" pourrait être le nom. Mais comme le cas d'homonymie est assez fréquent, surtout lorsque l'on manipule des fichiers volumineux, alors cet attribut constitue une mauvaise clef en général.

En revanche, il n'est pas impossible que la clef d'une entité soit composée de plusieurs attributs. Par exemple, la clef de l'entité "être humain" pourrait être le nom et le prénom. Cependant il n'est toujours pas impossible d'avoir deux personnes dont le nom et le prénom soient identiques...

On note qu'un attribut est une clef en le soulignant dans le schéma entité association. Si c'est une clef composée, alors plusieurs attributs seront soulignés.

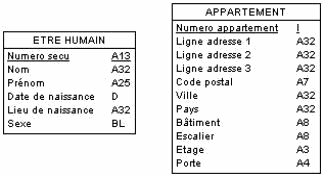
Exemple :



On voit ici que dans le cas de l'entité "appartement" tous les attributs sont utilisés pour composer la clef. Cette clef naturelle n'étant pas pratique, il est plus judicieux de créer un nouvel attribut qui servira expressément de clef à l'association.

Pour l'entité "être humain, on pourrait se servir du numéro de sécurité sociale (plus exactement du numéro INSEE), comme clef de l'entité. En revanche, pour ce qui est de l'entité "appartement" il est conseillé de créer un nouvel attribut clef qui serait, par exemple, un numéro.

Exemple :



Pour une entité de type « Voiture » il pourrait être fait usage de l’immatriculation du véhicule comme clef de l’entité.

**2.5.1. La technique de la double clef**

Une technique éprouvée consiste à introduire une double clef dans toutes les tables : la clef « informatique » et une clef « utilisateur ». La clef informatique est l’index primaire de la table et doit posséder les caractéristiques suivantes :

* purement numérique (par exemple un entier long)
* unique bien entendu
* obligatoire
* sans mise à jour en cascade
* avec intégrité référentielle
* générée automatiquement
* invisible pour l’utilisateur

La clef utilisateur doit être assez « souple », c’est à dire posséder dans la mesure du possible, les caractéristiques suivantes :

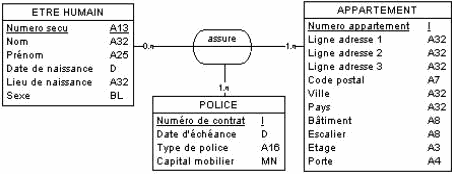
* Index secondaire unique
* Obligatoire
* Utiliser un jeu de caractère réduit s’il s’agit d’un format alpha (par exemple les 26 lettres majuscules de l’alphabet et les chiffres de 0 à 9)
* Limité à une faible taille (16/32 octets - 16 caractères maximum)

La présentation des données devant toujours s’effectuer suivant l’ordre établis par la clef utilisateur à défaut d’autre spécification.

## 3. La forme des liens

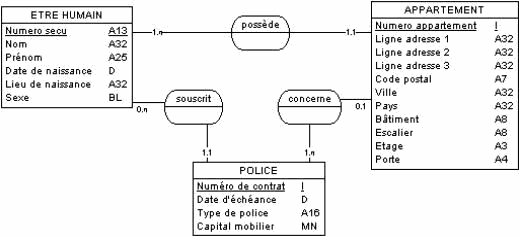
La plupart des associations sont de nature binaire, c'est à dire composées de deux entités mise en relation par une ou plusieurs associations. C'est le cas par exemple de l'association "est propriétaire" mettant en relation "être humain" et "appartement". Cependant il arrive qu'une association concerne plus de deux entités (on dit alors qu'il s'agit d'association "n-aires").

Exemple :



Mais dans ce cas il y a de grandes difficultés pour exprimer les cardinalités.

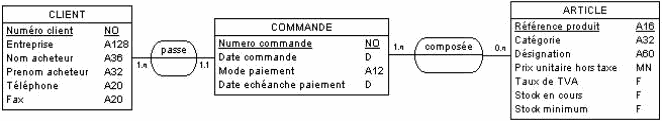
On aura tout intérêt à essayer de transformer le schéma de manière à n'obtenir que des associations binaires



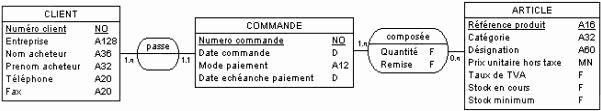
## 4. Attributs d'associations, pour aller plus loin

Il arrive parfois que l'on soit obligé de munir d'attributs des associations.  
Considérons par exemple, que nous voulons modéliser les relations existant entre les entités "client", "commande" et "article" :

Exemple :

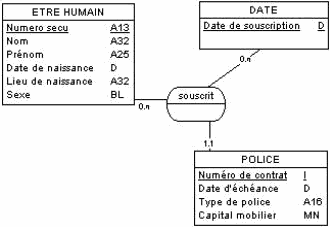


Mais comment dans ce schéma introduire l'attribut "quantité" et plus encore l'attribut "réduction" dont on voudrait qu'il puisse s'appliquer à chacun des articles d'une commande de manière différente ? En effet si l'on introduit l'attribut quantité à l'entité COMMANDE, chaque ligne de la commande se verra dotée de la même quantité... D'autre part si l'on introduit l'attribut quantité à l'entité ARTICLE alors chacun des article se verra doté de la même quantité quelque soit la commande... La solution est de pourvoir l'association "composée" des attributs "quantité" et "réduction" :



Il arrive dans certains cas que l'attribut "date" soit d'une importance capitale, notamment dans les applications SGBDR portant sur la signature de contrats à échéance ou dans la durée (assurance par exemple). Il n'est pas rare alors que le seul attribut "date" constitue à lui seul une entité.

Exemple :



On appelle alors cela une entité temporelle. Une entité temporelle possède souvent un seul attribut, mais dans le cas ou elle possède plusieurs attributs (année, mois, jour, heure, minute, seconde...), l'ensemble de ces attributs constitue alors la clef de l'entité.  
Mais dans ce cas on peut aussi retirer cette entité et introduire la date en tant qu'attribut de l'association "souscrit".

## 

## 5. De la théorie à la pratique

Ce que nous venons de voir concerne l'analyse conceptuelle des données, c'est à dire un niveau d'analyse qui s'affranchi de toutes les contraintes de la base de données sur lequel va reposer l'application. Une fois décrit sous forme graphique, ce modèle est couramment appelé MCD pour "Modèle Conceptuel des Données".

Dès lors, tout MCD peut être transformé en un MPD ("Modèle Physique des Données") c'est à dire un modèle directement exploitable par la base de données que vous voulez utiliser... Tout l'intérêt de cet outil d'analyse est de permettre de modéliser plus aisément les relations existant entre les entités et d'automatiser le passage du schéma muni d'attributs aux tables de la base de données pourvues de leurs champs.

Voici maintenant les règles de base nécessaire à une bonne automatisation du passage du MCD au MPD.

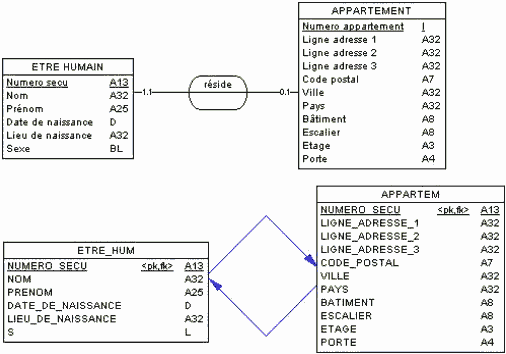
### 5.1. Transformation des entités (passer de l'entité à la table)

**Règle n°1 : toute entité doit être représentée par une table.**

#### 5.1.1. Relation de type 1:1

**Règle n°2 : Dans le cas d'entités reliées par des associations de type 1:1, les tables doivent avoir la même clef.**

Exemple :

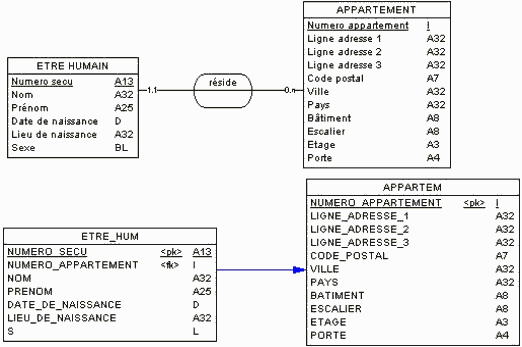


Dans ce cas une étude approfondie de la solution à adopter est nécessaire, mais ce type de relation est en général assez rare et peu performante.

#### 5.1.2. Relation de type 1:n

**Règle n°3 : Dans le cas d'entités reliées par des associations de type 1:n, chaque table possède sa propre clef, mais la clef de l'entité côté 0,n (ou 1,n) migre vers la table côté 0,1 (ou 1,1) et devient une clef étrangère (index secondaire).**

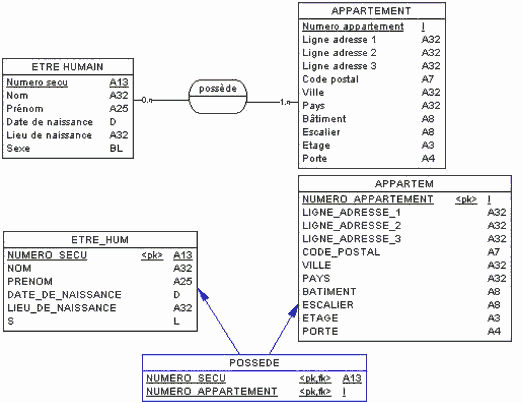
Exemple :



#### 5.1.3. Relation de type n:m

**Règle n°4 : Dans le cas d'entités reliées par des associations de type n:m, une table intermédiaire dite table de jointure, doit être créée, et doit posséder comme clef primaire une conjonction des clefs primaires des deux tables pour lesquelles elle sert de jointure.**

Exemple :



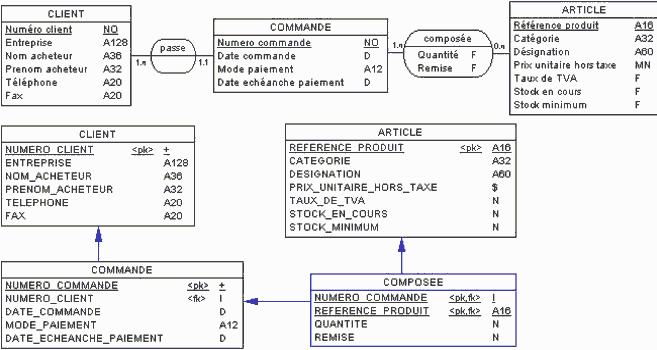
**5.2. Ou placer les attributs d'association ?**

**Règle n°5 : Cas des associations pourvues d'au moins un attribut :**

* **si le type de relation est n:m, alors les attributs de l'association deviennent des attributs de la table de jointure.**
* **si le type de relation est 1:n, il convient de faire glisser les attributs vers l’entités pourvue des cardinalités 1:1.**
* **si le type de relation est 1:1, il convient de faire glisser les attributs vers l’une ou l’autre des entités.**

Pour synthétiser toutes ces règles, voici un exemple de modélisation d'une application. En l'occurrence il s'agit d'un service commercial désirant modéliser les commandes de ses clients.

Exemple :



**IIème PARTIE INTRODUCTION A SQL**

**1 Types de données**

**1.1 Types numériques**

Les types numériques de la norme SQL 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).

**1.2 Type numérique d'Oracle**

Oracle n'a qu'un seul type numérique NUMBER. Par soucis de compatibilité, Oracle permet d'utiliser les types SQL-2 mais ils sont ramenés au type NUMBER. Lors de la définition d'une colonne de type numérique, on peut préciser le nombre maximum de chiffres et de décimales qu'une valeur de cette colonne pourra contenir :

NUMBER

NUMBER(taille\_maxi)

NUMBER(taille\_maxi, décimales)

Si le paramètre décimales n'est pas spécifié, 0 est pris par défaut. La valeur absolue du nombre doit être inférieure à 10128. 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.

1.3 Types chaîne de caractères

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).

**1.4 Types temporels**

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) ;

**1.5 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.

**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 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.

Syntaxe générale :

CREATE TABLE nom\_table

(colonne1 type1,

colonne2 type2,

) ;

nom\_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

Exemple

CREATE TABLE article (

ref VARCHAR(10),

nom VARCHAR(30) NOT NULL,

prix DECIMAL(9,2),

datemaj DATE,

constraint pk\_article primary key(ref)

) ;

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

CREATE TABLE article (

Ref VARCHAR(10),

nom VARCHAR(30) NOT NULL,

prix DECIMAL(9,2),

datemaj DATE DEFAULT CURRENT\_DATE,

constraint pk\_article primary key(Ref)

) ;

**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 définitions 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 :**

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 update 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 l6a 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**

* **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))

) ;=

* **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)

);

* **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.

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

**II.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,SGBD 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**

* INSERT INTO dept VALUES (10, 'FINANCES', 'PARIS')
* 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.

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 :**

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 nom\_table SET (col1, col2,...) = (SELECT ...) WHERE prédicat;

table est le nom de la table à mettre 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.

**Remarque :**

La clause WHERE est facultative. Si elle est absente, toutes les lignes sont mises à jour.

**Exemples :**

* Faire passer MARTIN dans le département 10 :

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

* Augmenter de 10 % les commerciaux :

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

* 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.

* 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!).***

**Exemple :**

DELETE FROM dept WHERE dept = 10;

**II. 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:**

* SELECT \* FROM DEPT
* SELECT DISTINCT POSTE FROM EMP
* SELECT NOME, SAL + NVL(COMM,0) AS Salaire FROM EMP
* 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 :**

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.

**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 . 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.

Exemple :

* SELECT B.dept, A.nomd 2 FROM dept A,dept B;

Pour obtenir la liste des employés avec le pourcentage de leur salaire

par rapport au total des salaires, Il est 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);

**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

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.

**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

* Sélectionner les employés du département 30 ayant un salaire supérieur à 1500 frs.

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

* 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 ;

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 :**

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.

**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 de 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.

**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 :**

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 :**

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'));

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.

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 =, !=, <, >, <=, >=.

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) ;

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 :**

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 :**

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.

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 :**

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

**Remarque :**

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.

**4.7 Fonctions de groupes**

Les fonctions de groupes peuvent apparaître dans le Select ou le Having

; 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 :**

SELECT COUNT(\*) FROM EMP

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 :**

SELECT DEPT, COUNT(\*) FROM EMP GROUP BY DEPT

SELECT DEPT, COUNT(\*) FROM EMP WHERE POSTE = 'SECRETAIRE'

GROUP BY DEPT

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

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:**

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 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

**Exemple :**

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 :**

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

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

ORDER BY 2;

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

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

**4.11 Opérateurs ensemblistes**

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

**4.11.1 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 :**

Liste des ingénieurs des deux filiales :

SELECT \* FROM EMP1 WHERE POSTE='INGENIEUR' UNION SELECT \* FROM EMP WHERE POSTE='INGENIEUR' ;

**4.11.2 Opérateur INTERSECT**

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

Exemples :

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

SELECT DEPT FROM EMP1 INTERSECT SELECT DEPT FROM EMP2;

**4.11.3 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 :**

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).

.

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

**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 Tables**

**5.1.1 CREATE TABLE AS**

La commande CREATE TABLE a déjà été vue. 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**

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

**5.1.2 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 nom\_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 :**

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 sûres 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.

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

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

**Exemple :**

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 collone d’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

**5.1.3 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 de la commande permet de supprimer les contraintes d'intégrité et la table :

DROP TABLE table CONSTRAINTS ;

* ***Quelques fonctions utiles en MySQL:***
* STR\_TO\_DATE('00/00/0000', '%m/%d/%Y');
* **SELECT**TIMEDIFF('2000:01:01 00:00:00', '2000:01:01 00:00:00.000001');
* NOW() ;
* SYSDATE() ;

**5.2 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.

5.2.1 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 :**

* 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;

* Il est possible de créer une vue de vue :

CREATE VIEW EMP3 (MATR, NOM, DEPT) AS SELECT MATR, NOM FROM MP2;

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 :**

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

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

**Remarque :**

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.

**5.2.3 DROP VIEW**

DROP VIEW vue;

**5.2.4 Syntaxe ALTER VIEW**

ALTER view view\_name [(colums\_list)] AS select\_statement . Cette commande modifie la définition d’une vue. Select\_stament est le même que pour CREATE VIEW sauf qu’on doit omettre **OR REPLACE**.

**5.2.3 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 :**

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

**Remarque :**

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 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';

**5.4 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. 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 TRIGGER *trigger\_name* *trigger\_time* *trigger\_event*

ON *tbl\_name* FOR EACH ROW *trigger\_stmt*

Pour supprimer un trigger : **drop trigger nomTrigger;**

**Remarque :**

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.

**5.4.0 Instructions aux notions des procédures stockées**

Réf http://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx

Les instructions IF, CASE, LOOP, WHILE, ITERATE et LEAVE sont toutes supportées. Ces instructions peuvent contenir des commandes simples, ou des blocs de commandes BEGIN ... END. Les instructions peuvent être imbriquées.

* **La commande DECLARE**

La commande DECLARE sert à définir différents objets locaux dans une routine : variables locales.

* **DECLARE var\_name[,...] type [DEFAULT value];**

Cette commande sert à déclarer des variables globales. Le scope des variables est le bloc BEGIN ... END.

* **Commande d'affectation de variables SET**

La commande SET des procédures stockées est une version étendue de la commande SET classique. Les variables référencées peuvent être déclarées dans le contexte de la routine ou comme variables globales. La commande SET des procédures stockées est implémentée comme une sous-partie de la syntaxe SET. Cela permet la syntaxe étendue SET a=x, b=y, ..., où plusieurs types de variables (locales, serveur, globale ou session) sont mélangées. Cela permet aussi la combinaison de variables locales et d'option système qui n'ont de sens qu'au niveau global du serveur : dans ce cas, les options sont acceptées mais ignorées.

* **SET variable = expression [,...]Syntaxe de SELECT ... INTO**

SELECT column[,...] INTO variable[,...] table\_expression

Cette syntaxe de SELECT stocke les colonnes sélectionnées dans des variables. Par conséquent, une seule ligne doit être lue. Cette commande est aussi extrêmement utile lorsqu'elle est utilisée avec des curseurs.

* **Commande IF**

IF search\_condition THEN statement(s)

[ELSEIF search\_condition THEN statement(s)]

...

[ELSE statement(s)]

END IF ;

IF implémente une instruction de condition simple. Si search\_condition est vrai, la commande SQL correspondante est exécutée. Si search\_condition est faux, la commande dans la clause ELSE est exécutée.

* **Commande CASE**

CASE case\_value

WHEN when\_value THEN statement

[WHEN when\_value THEN statement ...]

[ELSE statement]

END CASE

ou:

CASE

WHEN search\_condition THEN statement

[WHEN search\_condition THEN statement ...]

[ELSE statement]

END CASE

CASE implémente une structure conditionnelle complexe. Si un des conditions search\_condition est vraie, la commande SQL correspondante est exécutée. Si aucune condition n'est vérifiée, la commande SQL de la clause ELSE est exécutée.

**Note :** la syntaxe de la commande CASE à l'intérieure d'une procédure stockée diffère légèrement de l'expression SQL CASE. La commande CASE ne peut pas avoir de clause ELSE NULL, et l'instruction se termine avec END CASE au lieu de END.

* **Commande LOOP**

[begin\_label:] LOOP

statement(s)

END LOOP [end\_label]

LOOP implémente une boucle, permettant l'exécution répétée d'un groupe de commande. Les commandes à l'intérieure de la boucle sont exécutée jusqu'à ce que la boucle se termine, généralement lorsqu'elle atteint la commande LEAVE. begin\_label et end\_label doivent être identiques, si les deux sont spécifiés.

* **Commande LEAVE**

LEAVE label Cette commande sert à sortir d'une instruction de contrôle.

* **Commande ITERATE**

ITERATE label

ITERATE ne peut être utilisée qu'à l'intérieur d'une boucle LOOP, REPEAT ou WHILE. ITERATE signifie ``exécute encore une fois la boucle.''

Par exemple :

CREATE PROCEDURE doiterate(p1 INT)

BEGIN

label1: LOOP

SET p1 = p1 + 1;

IF p1 < 10 THEN

ITERATE label1;

END IF;

LEAVE label1;

END LOOP label1;

SET @x = p1;

END

* **Commande REPEAT**

[begin\_label:] REPEAT

statement(s)

UNTIL search\_condition

END REPEAT [end\_label]

Les commandes à l'intérieur d'une commande REPEAT sont répétées jusqu'à ce que la condition search\_condition soit vraie. begin\_label et end\_label doivent être identiques, s'ils sont fournis.

Par exemple :

mysql> delimiter |

mysql> CREATE PROCEDURE dorepeat(p1 INT)

-> BEGIN

-> SET @x = 0;

-> REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;

-> END

-> |

mysql> CALL dorepeat(1000)|

mysql> SELECT @x|

| @x |

| 1001 |

* **Syntaxe WHILE**

[begin\_label:] WHILE search\_condition DO

statement(s)

END WHILE [end\_label]

Les commandes dans l'instruction WHILE sont répétées tant que la condition search\_condition est vraie. begin\_label et end\_label doivent être identiques, s'ils sont spécifiés.

Par exemple :

CREATE PROCEDURE dowhile()

BEGIN

DECLARE v1 INT DEFAULT 5;

WHILE v1 > 0 DO

SET v1 = v1 - 1;

END WHILE;

END

* **CURSOR**

Pour parcourir un result set dans une procedure stockée mysql, il faut utiliser les curseurs et les boucles.

Commençons par la théorie:

- tout d'abord définir un curseur qui est lié à une requête avec **DECLARE CURSOR**

- puis on ouvre le curseur, ce qui va exécuter la requête avec **OPEN CURSOR**

Maintenant le curseur pointe sur le résultat de la requête.

- on utilise une boucle pour récupérer chaque ligne du curseur avec **FETCH CURSOR**

- on ferme le curseur avec **CLOSE CURSOR**

Maintenant passons au concret:

**DECLARE** country\_name **VARCHAR**(255);  
**DECLARE** country\_size **INT**;  
**DECLARE** country\_cursor **CURSOR FOR**  
   **SELECT** name, size **FROM** countries **ORDER BY** name;  
   
**DECLARE** done **INT DEFAULT** 0;  
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;  
   
**OPEN** country\_cursor;  
   
country\_loop: **LOOP**  
   **FETCH** country\_cursor **INTO** country\_name, country\_size;  
   **IF** done **THEN**  
      **LEAVE** country\_loop;  
   **END IF**;  
   **SELECT** concat(country\_name, country\_size,’-');  
**END LOOP**;  
    
**CLOSE** country\_cursor;

Note: la variable 'done' est une petite astuce qui permet de sortir de la boucle lorsqu'il n'y a plus de lignes à parcourir.

La variable passe automatiquement à 1 si FETCH remonte une erreur NOT FOUND.

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

Un déclencheur est un objet de base de données associé à une table, qui s'active lorsqu'un événement particulier survient. Le déclencheur est associé à la table appelée *tbl\_name*. *tbl\_name* doit faire référence à une table permanente. Vous ne pouvez pas associer un déclencheur avec une table TEMPORARY ou une vue.

*trigger\_time* est le moment d'action du déclencheur. Il peut être BEFORE (avant) ou AFTER (après), pour indiquer que le déclencheur s'active avant ou après la commande qui le déclenche. *trigger\_event* indique le type de commande qui active le déclencheur. Il peut valoir INSERT, UPDATE ou DELETE. Par exemple, un déclencheur BEFORE pour une commande INSERT peut être utilisé pour vérifier les valeurs avant leur insertion dans la table.

Il ne peut pas y avoir deux déclencheurs pour une même table avec les mêmes configurations de moment et de commande. Par exemple, vous ne pouvez pas avoir deux déclencheurs BEFORE UPDATE pour la même table. Mais vous pouvez avoir un déclencheur BEFORE UPDATE et un déclencheur BEFORE INSERT, ou un déclencheur BEFORE UPDATE et un déclencheur AFTER UPDATE.

*trigger\_stmt* est la commande a exécuter lorsque le déclencheur s'active. Si vous voulez utiliser plusieurs commandes, utilisez les agrégateurs BEGIN ... END. Cela vous permet aussi d'utiliser les mêmes codes que ceux utilisés dans des procédures stockées.

La commande suivante crée la table et le déclencheur :

mysql> CREATE TABLE account (acct\_num INT, amount DECIMAL(10,2));

mysql> CREATE TRIGGER ins\_sum BEFORE INSERT ON account

FOR EACH ROW Begin Set @su=@su+ NEW.amount; END

La commande CREATE TRIGGER crée un déclencheur appelé ins\_sum qui est associé avec la table account. Il inclut aussi des clauses pour spécifier le moment d'activation, l'événement et l'action du déclencheur :

* Le mot réservé BEFORE (avant) indique le moment d'activation. Dans ce cas, le déclencheur sera activé avant l'insertion des lignes dans la table. L'autre mot réservé est AFTER (Après, en anglais).
* Le mot réservé INSERT indique l'événement qui active le déclencheur. Dans l'exemple, le déclencheur s'active lors des commandes INSERT. Vous pouvez créer des déclencheur pour les commandes DELETE et UPDATE.
* La commande qui suit le mot clé FOR EACH ROW définit la commande à exécuter à chaque fois que le déclencheur s'active, ce qui arrive dès qu'une ligne est insérée. Dans l'exemple, la commande du déclencheur est un simple SET qui accumule la somme des valeurs insérées dans les colonnes amount. La commande utilise la valeur de la colonne avec la syntaxe NEW.amount (en anglais, nouvelle.montant) ce qui signifie ``la valeur de la colonne amount qui va être insérée''.

Pour utiliser le déclencheur, initialisé l'accumulateur à zéro, puis exécutez une commande INSERT et voyez la valeur finale de l'accumulateur :

mysql> **SET @sum = 0;**

mysql> **INSERT INTO account VALUES(137,14.98),(141,1937.50),(97,-100.00);**

mysql> **SELECT @sum AS 'Total amount inserted';**

Pour détruire un déclencheur, utilisez la commande DROP TRIGGER. Le nom du déclencheur doit inclure le nom de la table :

mysql> **DROP TRIGGER account.ins\_sum;**

Comme le déclencheur est associé avec une table, vous ne pouvez pas avoir plusieurs déclencheurs sur une même table qui portent le même nom. Pour faciliter la compatibilité ascendante, essayez d'utiliser des noms de déclencheurs qui soient uniques dans toute la base.

En plus du fait que les noms de déclencheurs doivent être uniques pour une table, il y a d'autres limitations sur le type de déclencheurs que vous pouvez mettre en place. En particulier, vous ne pouvez pas avoir deux déclencheurs qui ont le même moment d'activation et le même événement d'activation. Par exemple, vous ne pouvez pas définir deux déclencheurs BEFORE INSERT et deux déclencheurs AFTER UPDATE pour la même table. Ce n'est probablement pas une limitation importate, car il est possible de définir un déclencheur qui exécute plusieurs commandes en utilisant une commande complexe, encadrée par les mots BEGIN … END, après le mot clé FOR EACH ROW. Un exemple vous est présenté ultérieurement dans cette section.

Il y a aussi des limitations dans ce qui peut apparaître dans la commande que le déclencheur peut éxecuter lorsqu'il est activé :

* Le déclencheur ne peut pas faire référence directe aux tables par leur nom, y compris la table à laquelle il est associé. Par contre, vous pouvez utiliser les mots clés OLD (ancien en anglais) et NEW (nouveau en anglais). OLD fait référence à la ligne existante avant la modification ou l'effacement. NEW faire référence à la nouvelle ligne insérée ou à la ligne modifiée.
* Le déclencheur ne peut pas exécuter de procédures avec la commande CALL. Cela signifie que vous ne pouvez pas contourner le problème des noms de tables en appelant une procédure stockée qui utilise les noms de tables.
* Le déclencheur ne peut pas utiliser de commande qui ouvre ou ferme une transaction avec START TRANSACTION, COMMIT ou ROLLBACK.

Les mots clé OLD et NEW vous permettent d'accéder aux colonnes dans les lignes affectées par le déclencheur. OLD et NEW ne sont pas sensibles à la casse. Dans un déclencheur INSERT, seul NEW.*col\_name* peut être utilisée : il n'y a pas d'ancienne ligne. Dans un déclencheur DELETE, seul la valeur OLD.*col\_name* peut être utilisée : il n'y a pas de nouvelle ligne. Dans un déclencheur UPDATE, vous pouvez utiliser OLD.*col\_name* pour faire référence aux colonnes dans leur état avant la modification, et NEW.*col\_name* pour faire référence à la valeur après la modifcation.

Une colonne identifiée par OLD est en lecture seule. Vous pouvez lire sa valeur mais vous ne pouvez pas la modifier. Une colonne identifiée avec la valeur NEW peut être lue si vous avez les droits de SELECT dessus. Dans un déclencheur BEFORE, vous pouvez aussi changer la valeur avec la commande SET NEW.*col\_name* = *value* si vous avez les droits de UPDATE. Cela signifie que vous pouvez utiliser un déclencheur pour modifier les valeurs insérées dans une nouvelle ligne ou les valeurs modifiées.

Dans un déclencheur BEFORE, la valeur NEW d'une colonne AUTO\_INCREMENT vaut 0, et non pas le nombre séquentiel automatiquement généré car ce nombre sera généré lorsque la ligne sera réellement insérée.

OLD et NEW sont des extensions de MySQL aux déclencheurs.

En utilisant la syntaxe BEGIN … END, vous pouvez définir un déclencheur qui exécute plusieurs commandes. À l'intérieur d'un bloc BEGIN, vous pouvez aussi utiliser les autres syntaxes autorisées dans les routines stockées, telles que les conditions et les boucles. Cependant, tout comme pour les procédures stockées, lorsque vous définissez un déclencheur qui s'exécute sur plusieurs commandes, il est nécessaire de redéfinir le délimiteur de commande si vous saisissez le déclencheur à l'aide d'un utilisateur en ligne de commande tel que **mysql** pour que vous puissiez utiliser le caractère ‘;’ à l'intérieur de la définition. L'exemple ci-dessous illustre ces points. Il définit un déclencheur UPDATE qui vérifie la valeur d'une ligne avant sa modification, et s'arrange pour que les valeurs soient dans l'intervalle de 0 à 100. Cela doit être fait avant (BEFORE) la modification, pour que la valeur soit vérifiée avant d'être utilisée :

mysql> **delimiter //**

mysql> **CREATE TRIGGER upd\_check BEFORE UPDATE ON account**

-> **FOR EACH ROW**

-> **BEGIN**

-> **IF NEW.amount < 0 THEN**

-> **SET NEW.amount = 0;**

-> **ELSEIF NEW.amount > 100 THEN**

-> **SET NEW.amount = 100;**

-> **END IF;**

-> **END//**

mysql> **delimiter ;**

Il vous viendra surement à l'esprit qu'il serait plus facile de définir une procédure stockée séparément, pour l'invoquer depuis le déclencheur grâce à un simple appel à CALL. Cela serait surement avantageux si vous voulez appeler la même routine depuis plusieurs déclencheurs. Cependant, les déclencheurs ne peuvent pas utiliser la commande CALL. Vous devez absolument réécrire les commandes composées de chaque commande CREATE TRIGGER que vous voulez utiliser.

**5.4.2 Exemple**

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

**Exemple :**

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

CREATE 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 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

**5.4.3 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:**

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;

5.4.4 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 :**

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).

**5.5 Procédure et fonction stockée**

**5.5.1 Procédure stockée**

Une procédure stockée est un programme qui comprend des instructions SQL précompilées et qui est enregistré dans la base de données. 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,...).

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. De plus ce système des procédures stockées augmente la charge du serveur, car ce dernier doit réaliser plus de travail. Souvent, il y a de nombreux clients, mais peut de serveurs.

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 :**

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 procedure augmentation (in unDept integer, in pourcentage decimal, out cout decimal)

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;

Sous MySQL ou Oracle, on peut avoir une description des paramètres d'une procédure stockée par la commande DESC nom\_procédure.

**5.5.2 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 :**

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

CREATE FUNCTION factorial (n DECIMAL(3,0))

RETURNS DECIMAL(20,0)

DETERMINISTIC

BEGIN

DECLARE factorial DECIMAL(20,0) DEFAULT 1;

DECLARE counter DECIMAL(3,0);

SET counter = n;

factorial\_loop: REPEAT

SET factorial = factorial \* counter;

SET counter = counter - 1;

UNTIL counter = 1

END REPEAT;

RETURN factorial;

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;

Exercise

CREATE TRIGGER gestionVente BEFORE INSERT ON vente

**FOR EACH ROW**

**BEGIN**

**DECLARE unAccountTicket int(4) default 0;**

**select count(ticketID) uAT into unAccountTicket from vente where compte='0' and CLIENTid=NEW.CLIENTid ;**

**IF unAccountTicket=4 THEN**

update vente set compte='1' where CLIENTid=nouveau.CLIENTid and compte='0' ;

NEW.compte= '1' ;

NEW.etat\_paiement='0' ;

ELSE

NEW.compte='0' ;

NEW.etat\_paiement='1' ;

**END IF;**

**END; |**