Introduction aux Bases de Données Relationnelles

Ce cours a pour objectif de montrer le processus qui couvre les phases de conception et de mise en œuvre de la base de données. Lors de chaque cours, les 90 premières minutes seront consacrées à la conception de la base de données à partir de schémas entités associations. Les 90 minutes restantes nous permettrons d'aborder l'implantation de la base de données en utilisant la norme SQL et le système de gestion de bases de données Oracle.

1- Qu'est-ce qu'une application bases de données

Ce sont des applications où l'aspect « données » est primordial :

- Volume de données important ;
- Très grande variété des données ;
- Persistance des données ;
- Echange de données entre applications ;
- Aspect multi-utilisateur ;
- Application à de très nombreux domaines d'activité ;
- Très grande variété d'utilisateurs autour d'une application BD ;
- …

Les réseaux informatiques (WEB), les interfaces graphiques font des bases de données un sujet d'actualité très important car il faut assurer la gestion de très gros volumes de données, l'accès aux bases de données 24 h / 24, la gestion d'un très grand nombre d'utilisateurs demandant des accès en même temps. Les applications bases de données ne sont donc plus limitées à des applications de gestion : on trouve des bases de données dans pratiquement tous les secteurs d'activités (transport, aéronautique, espace, médecine, pharmacie, …).

2- Le modèle relationnel

2.1- Définition

Le modèle relationnel est inventé par CODD à l'Université de San-José en 1970. C'est un modèle issu des Mathématiques. Malgré leurs bases mathématiques, les bases de données relationnelles n'apparaissent commercialement qu'au début des années 80.

Le modèle relationnel est basé sur la notion de relation. Une relation est un sous-ensemble du produit cartésien de différents domaines. Chaque élément de ce sous-ensemble est appelé un n-uples. Pour identifier les différents domaines, il est possible de les nommer. Ce nom est appelé l'attribut.

Exemple : Soit les deux domaines suivants :
nom = {Dupont, Durant, Dulong}
âge = {18, 19}
nom et âge sont les attributs.
On désire représenter les informations suivantes :
Dupont, 18 ans
Dulong, 19 ans
Durant, 18 ans

Soit le produit cartésien entre nom et âge :

Nom	*âge*
Dupont	18
Durant	18
Dulong	18
Dupont	19
Durant	19
Dulong	19

En gris, on a la relation. Chaque ligne grise est un n-uplet.

2.2- L'algèbre relationnelle

Afin de manipuler ces relations, il a été défini un ensemble d'opérateurs constituant une algèbre. L'application d'un opérateur produit une nouvelle relation. Cette propriété a permis de construire des langages de manipulation de données. On distingue les opérateurs suivants :

- ensemblistes : union, intersection, différence, produit cartésien ;
- spécifiques : projection, sélection, jointure, division ;
- agrégation.

Une partie du langage SQL est une mise en œuvre de ces opérateurs. Nous verrons dans la suite lors de la présentation des opérateurs SQL leur représentation en algèbre relationnelles.

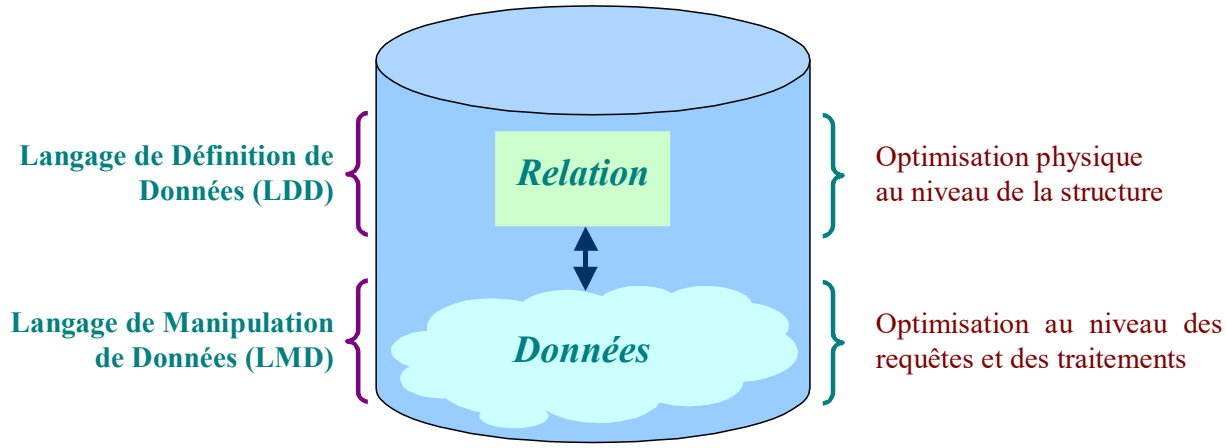
2.3- Le langage SQL (Structured Query Language)

C'est un langage créé dans les années 80 par IBM. Il est de type ensembliste mais il inclut des opérations permettant de réduire le nombre de données manipulées verticalement et/ou horizontalement. La projection permet la réduction verticale par diminution du nombre d'attributs manipulés. La sélection permet la réduction horizontale par diminution du nombre de n-uplets manipulés. La projection et la sélection sont deux éléments essentiels pour l'optimisation de l'exécution des requêtes.

Le langage SQL se décline sous la forme de deux sous-ensembles : le langage de définition de données ou LDD et le langage de manipulation de données ou LMD.

Le langage de définition de données permet de définir et de manipuler les relations au sein de la base de données.

Le langage de manipulation permet d'accéder, d'insérer, modifier et de supprimer des données dans la base.



2.4- Le Langage de Définition de Données : manipulation des relations

2.4.1- Les attributs

Chaque domaine identifié par un attribut possède un type. Quatre types sont principalement utilisés :

- ✓ char(n) : chaîne de caractères de longueur fixe n.
  <u>*Par exemple*</u> : le code postal est codé sur 5 caractères, un numéro de téléphone est composé de 10 chiffres. Attention, il ne faut pas confondre le chiffre 09000 et la chaîne de caractères '09000'. Pour un code postal, il est impensable de pouvoir faire des calculs comme on le ferait sur un nombre ;
- ✓ varchar2(n) : chaîne de caractères de longueur variable d'au maximum n ;
- ✓ number(n, m) : numérique à n chiffres ***dont*** m décimales ;
- ✓ date : comme son nom l'indique, ce type représente les dates et les heures.

Le système Oracle offre d'autres types dont nous ne parlerons pas dans ce cours. En particulier, il offre des types permettant de stocker des données binaires (images, textes utilisant des caractères nationaux, ...).

2.4.2- Création d'une relation

Pour créer une relation, il existe une seule instruction :

create table Nom_Relation **(**
nom_attribut_1 type1 [default valeur1],
nom_attribut_2 type2 [default valeur2],
...**)** ;

<u>*Exemple*</u> : créer une relation « Personne ». Une personne est caractérisée par son nom, son prénom, son adresse, sa date de naissance et son code postal.

```
create table Personne (
    nom             varchar2(20),
    prenom          varchar2(20),
    adresse         varchar2(100),
    code_postal     char(5),
    date_naissance  date);
```

<u>*Exemple*</u> : créer une relation « Voiture ». Une voiture est caractérisée par sa marque, sa couleur, son numéro d'immatriculation et son prix.

```
create table Voiture (
    marque          varchar2(10),
    couleur         varchar2(10),
    immatriculation char(8),
    prix            number(9,2));
```

2.4.3- Modification d'une relation

Il est possible de modifier la structure d'une relation. En particulier, il est possible d'ajouter ou modifier ou supprimer :

- ✓ des attributs ;
- ✓ la valeur par défaut d'un attribut ;
- ✓ des contraintes sur des attributs ;
- ✓ des contraintes sur la relation.

Pour le moment, nous ne prendrons pas en compte les contraintes.

<u>Attention</u> : toute modification d'une relation est irrémédiable et ne peut pas être annulée.

Ajout d'attributs :

alter table nom_relation
add (nom_attribut_1 type1 [default valeur1],
nom_attribut_2 type2 [default valeur2], ...) ;

Modification d'attributs :

alter table nom_relation
modify (nom_attribut_1 type1 [default valeur1],
nom_attribut_2 type2 [default valeur2], ...) ;

Exemple : ajouter à la relation « Voiture » l'attribut « prop » contenant le nom du propriétaire de la voiture

alter table Voiture
add (prop varchar2(20)) ;

Exemple : modifier l'attribut « code_postal » de la relation « Personne » pour que par défaut la valeur du code postal soit 31000.

alter table Personne
modify (code_postal char(5) default '3100');

2.4.4- Destruction d'une relation

La commande permettant de détruire une relation est :
drop table nom_relation ;

Attention : la destruction d'une relation implique de facto la destruction des données qu'elle contient. Cette opération est irréversible.

2.5- Le Langage de Manipulation de Données : manipulation des données

Attention : toute modification du contenu d'une relation est réversible et peut être annulée dans le cadre des transactions

2.5.1- Insertion de données

L'insertion de données se fait par l'instruction :

insert into nom_relation (nom_att1, nom_att2, ... , nom_attn) **values**
(val_att1, val_att2, ..., val_attn) ;

Le nom des attributs est optionnel si on fournit une valeur à chaque attribut de la relation « nom_relation ».

La valeur « null »

La valeur « ***null*** » est une valeur spécifique en SQL qui est commune à tous les types. Elle représente un attribut sans valeur connue et est manipulée avec des opérateurs spécifiques.

Exemple : insérer dans la relation Personne les n-uples suivants :

Dupont	Jean	12, rue des Lilas	31000	12/12/1941
Dulong	Louis			11/11/1960

insert into Personne
values ('Dupont', 'Jean', '12, rue des Lilas', '31000', '12/12/1941') ;

insert into Personne **values** ('Dulong', 'Louis', null, null, '11/11/1960') ;

insert into Personne (date_naissance, nom, prenom)
values ('11/11/1960', 'Dulong', 'Louis') ;

2.5.2- Consultation et accès aux données

C'est la partie la plus complexe du langage de manipulation de données. En effet, elle permet de sélectionner, projeter et regrouper des données issues d'une ou plusieurs relations. Dans ce tour d'horizon du LMD, nous ne présenterons que des cas simples de l'utilisation de SQL. Ces opérateurs SQL peuvent tous être formalisés avec l'algèbre relationnelle, formalisation que nous présenterons aussi.

Première version (selection simple) :

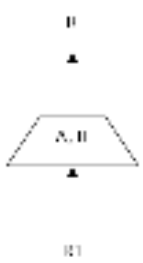
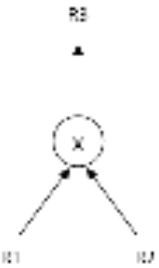
select cible **from** nom_rel_1, ..., nom_rel_n **where** condition ;

Cible : ensemble d'attributs que l'on désire extraire. La cible représente les attributs que l'on veut PROJETER.

nom_rel1, ... : ensemble des relations d'où sont extraites les données. La clause « ***from*** » permet d'effectuer le PRODUIT CARTESIEN entre les différentes relations présentes dans la clause « ***from*** ».

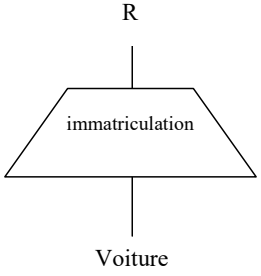
condition : expression booléenne permettant d'effectuer la SELECTION des n-uplets pertinents. Cette expression booléenne peut contenir des sous-expressions permettant de préciser les conditions selon lesquelles les différentes relations sont liées. Ces sous-expressions sont les clauses de jointure entre les différentes relations.

Formalisation avec l'algèbre relationnelle

Représentation graphique			
Représentation langage	PROJECT R1 OVER A, B GIVING R	MULTIPLY R1 BY R2 GIVING R3	SELECT R1 WHERE P GIVING R
Représentation symbolique	$R = \prod_{A,B} R1$	$R3 = R1 \times R2$	$R = \sigma_P R1$
	Projection	Produit cartésien	Sélection

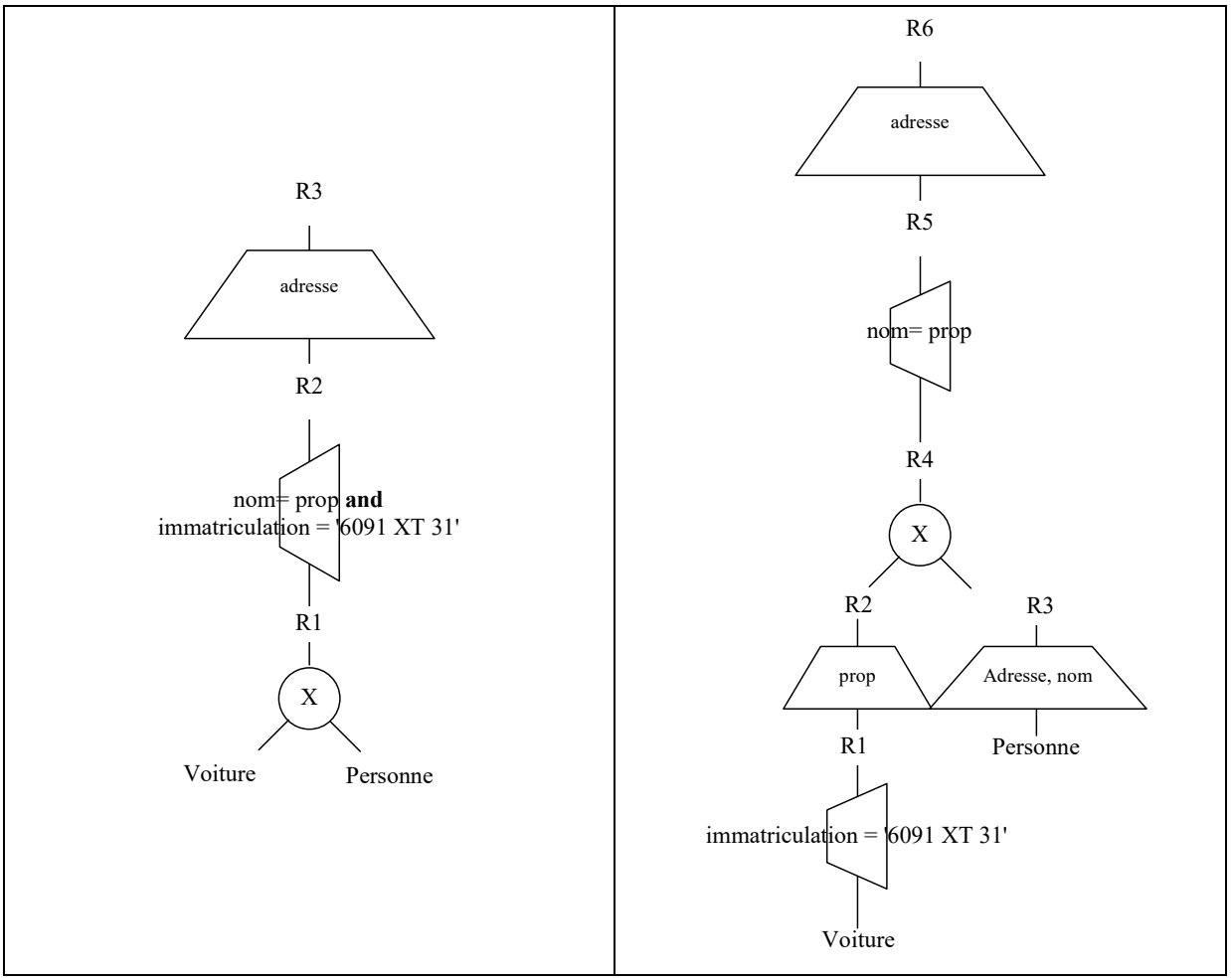
Exemple : donner le numéro d'immatriculation de toutes les voitures en SQL et en algèbre relationnelle.

select immatriculation **from** Voiture ;



donner l'adresse du propriétaire de la voiture immatriculée "6091 XT 31" en SQL et en algèbre relationnelle

select adresse **from** Voiture, Personne
where Personne.nom=Voiture.prop **and** immatriculation = '6091 XT 31' ;



Deuxième version (requête ensembliste) :

select cible1 **from** nom_rel_{11}, ..., nom_rel_{1n} **where** $condition_1$
operateur_ensembliste
select cible2 **from** nom_rel_{21}, ..., nom_rel_{2m} **where** $condition_2$;

operateur_ensembliste: Cet opérateur peut être l'union (UNION), l'intersection (INTERSECT) ou la soustraction (MINUS) de deux ensembles. Dans ce cas cible1 et cible2 doivent être compatibles, c'est-à-dire que tous les attributs constituant cible1 et cible2 doivent avoir des types compatibles deux à deux. En d'autre terme :

Soit cible1 = $\{att_{11}, \ldots, att_{1n}\}$ et cible2 = $\{at_{t21}, \ldots, att_{2n}\}$
cible1 et cible2 sont compatibles ➔
card(cible1) = card(cible2) et $\forall i$ type (att_{1i}) = type (att_{2i})

Rappel sur les opérateurs ensemblistes :

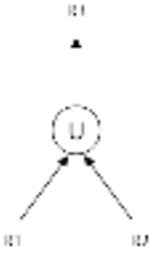
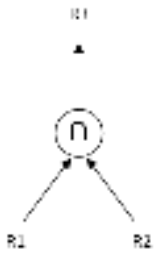
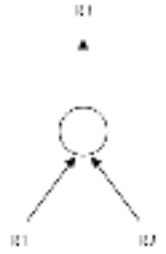
Soit deux ensembles E1={a, b, c, d} et E2={a, c, e, f} alors
E1 $\cup$ E2 = {a, b, c, d, e, f} attention les éléments en double ne sont pas dupliqués ;
E1 $\cap$ E2 = E1 – (E1 – E2) = {a, c} ;
E1 - E2 = {b, d} ;
E2 - E1 = {e, f} attention, l'opérateur minus n'est pas commutatif.

Il est possible de représenter les opérateurs d'inclusion et d'égalité en utilisant l'opérateur minus :

$E1 \subset E2 \Leftrightarrow E1 - E2 = \varnothing$

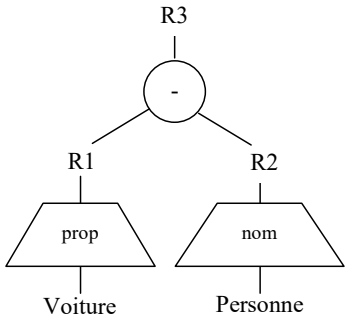
$E1 = E2 \Leftrightarrow E1 - E2 = \varnothing \wedge E2 - E1 = \varnothing$

Formalisation avec l'algèbre relationnelle

	Union	Intersect	Minus
Représentation graphique			
Représentation langage	R1 UNION R2 GIVING R3	R1 INTERSECT R2 GIVING R3	R1 MINUS R2 GIVING R3
Représentation symbolique	$R3 = R1 \cup R2$	$R3 = R1 \cap R2$	$R3 = R1 - R2$

Exemple : donner le nom de toutes les personnes ne possédant pas de voiture.

select nom **from** Personne
minus
select prop **from** Voiture ;



Troisième version (regroupement) :

select cible **from** nom_rel_{11}, ..., nom_rel_{1n} **where** condition1
goup by nom_att_1,..., nom_att_m **having** condition2 ;

group by : opérateur permettant de regrouper, en vu de leur traitement, les n-uplets ayant des valeurs identiques pour le sous-ensemble (nom_att_1, …, nom_att_n). Lors de l'utilisation d'un « group by », la cible ne peut être composée que d'attributs présents dans la clause « group by » ou d'opérations ensemblistes telles que ***sum***, ***avg***, ***max***, ***min***, ***count***, …

condition2 : expression booléenne portant uniquement sur des résultats d'opérations issus du regroupement des n-uplets.

Exemple : donner le nombre de voitures possédées par chaque personne.

select prop, count(*) **from** Voiture
group by prop;

donner le nom des personnes possédant au moins trois voitures.

select prop **from** Voiture
group by prop
having count(*) > 2 ;

donner le nombre de voitures présentes dans la relation voiture

select count(*) **from** Voiture ;

Attention, count, sum, avg, min et max ne prennent en compte que des attributs non « null ».

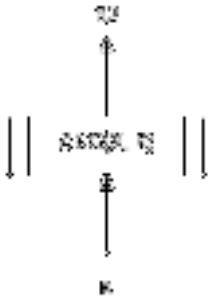
Exemple : soit la relation « Note » contenant les valeurs suivantes.

Nom	***Note***
Dupin	12,50
Dulong	8,00
Durant	
Martin	14,00
Durant	

count(*)=6
count(Note)=3
sum(Note)=34,50
avg(Note)=11,5 et non 5,750

Formalisation avec l'algèbre relationnelle

Représentation graphique



Représentation langage

AGG (R, X, B) GIVING R2

Représentation symbolique

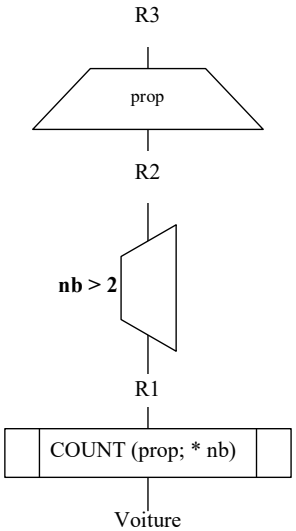
R2 = AGG (R; X; B)

Agrégation

R : relation sur laquelle s'applique la fonction d'agrégation AGG
X : liste des attributs de R servant à définir un critère de regroupement.
On regroupe dans R les lignes possédant la même valeur par rapport à X.
B : attribut agrégé
AGG : fonction d'agrégation

- COUNT dénombrement
- SUM cumul
- MIN plus petite valeur
- MAX plus grande valeur
- AVG moyenne

Exemple : donner le nom des personnes possédant au moins trois voitures.



2.5.3- Modification d'une donnée

update nom_rel **set** nom_att$_1$=val_att$_1$, ..., nom_att$_n$=val_att$_n$
where condition ;

Exemple : modifier l'immatriculation de la voitures immatriculée '6091XT 31' par '333AZE75'.

update Voiture
set immatriculation ='333AZE75'
where immatriculation='6091XT31';

2.5.4- Destruction d'une donnée

delete from nom_rel **where** condition ;

Exemple : détruire le n-uplet correspondant à la voiture immatriculée '333AZE75'.

delete from Voiture
where immatriculation='333AZE75';

Exemple : détruire tous les n-uplet de la relation voiture.

delete from Voiture;

A la différence de la commande permettant la destruction d'une relation, la destruction des données contenues dans une relation est réversible.

2.5.5- Exercice récapitulatif

On désire créer une relation contenant les informations inhérentes aux salariés de la société "TOUT VA BIEN". Un salarié est caractérisé par son code (unique, différent pour chaque salarié et composé de trois caractères), son nom, son prénom, son ancienneté (en mois) et son salaire.
Ecrire en SQL, les requêtes suivantes :

1. Créer la relation SALARIE ;

2. Ajouter l'attribut contenant le code du supérieur direct de chaque salarié. Un supérieur est un salarié comme les autres et les informations le concernant se trouvent aussi dans la relation ;

3. Insérer dans la relation « ***SALARIE*** » les données suivantes :

S01	DUPONT	Louis	36	1220,00	
S02	DULONG	Pierre	40	1400,00	S01

4. Mettre à jour le code du supérieur du salarié « S01 » sachant que ce supérieur à le numéro de code « S10 » ;

5. Extraire tous les salariés dont le supérieur a pour code « S12 » ;

6. Calculer la masse salariale totale de l'entreprise ;

7. Pour chaque salarié qui dirige des collaborateurs, calculer le nombre de collaborateur qu'il dirige ;

8. Extraire le code du supérieur qui a le plus grand nombre de collaborateurs directement sous ses ordres ;

9. Extraire le nom du supérieur qui à le plus grand nombre de collaborateurs sous ses ordres ;

10. Calculer le salaire moyen dans l'entreprise ;

11. Donner l'ensemble des salariés sous la direction du salarié de code « S02 » et ceux sous la direction du salarié de code « S03 » (utilisation de l'union) ;

12. Donner l'ensemble des collaborateurs sous la direction du salarié de code « S02 » et ceux sous la direction du salarié de code « S03 » (utilisation d'une requête simple) ;

13. Donner nom des salariés qui ont le plus petit salaire de la société ;

14. Donner le nombre de salariés ayant un salaire strictement supérieur au salaire moyen de l'entreprise ;

15. Ajouter 10% d'augmentation aux salariés dont le salaire est inférieur ou égal au salaire moyen de l'entreprise

16. Donner le code des salariés qui ne sont supérieurs d'aucun autre salarié

3- Clés primaires et clés étrangères

3.1- Clé primaire

Une relation est un ensemble de n-uplets. Afin de pouvoir les manipuler, il est nécessaire de pouvoir les distinguer au sein de la relation. Un attribut ou groupe d'attributs va jouer le rôle d'identifiant de la relation : ***c'est la clé primaire***. **Une valeur de clé primaire permet d'identifier de manière *unique* un n-uplet d'une relation.**

3.1.1- Définitions

Une clé primaire est un ensemble d'attributs, K, vérifiant la double propriété :
Unicité : les valeurs de clés primaires sont uniques et non nulles ;
Minimalité : aucun attribut composant K ne peut être enlevé sans perdre la propriété d'unicité.

Remarques :
1. lorsque la clef primaire est composée de plusieurs attributs, il peut être pertinent, pour simplifier la manipulation des données, d'introduire un nouvel attribut faisant office de clef primaire pour cette relation ;
2. Une relation ne peut avoir qu'une clef primaire. Si plusieurs clefs sont possibles, il est nécessaire d'en fixer une arbitrairement. Les clefs potentielles qui ne sont pas primaires sont appelées clefs candidates.

3.1.2- Exemple

Soit la relation « Voiture » définie précédemment, quel ou quels attributs peuvent être considérés comme une clef primaire ?

Voiture = {marque, couleur, immatriculation, prix} ;

Soit la relation « Personne » définie précédemment, quel ou quels attributs peuvent être considérés comme une clef primaire ?

Personne = {nom, prenom, date_naissance, adresse, code_postal };

Il n'y a pas de clef primaire évidente. On peut considérer que pour une entreprise de petite taille les trois attributs nom, prénom et date_naissance définissent de manière unique l'adresse et le code postal. Cette clef étant composée de trois attributs, il peut être intéressant d'introduire un attribut *id_personne* dont on assure l'unicité et l'existence pour tous les n-uplets de la relation. Dans ce cas, la définition de la relation devient :

Personne = {<u>id_personne</u>, nom, prenom, date_naissance, adresse, code_postal };

3.1.3- exercice

Soit la relation R= {A, B, C, D} et les n-uplets suivants :

A	B	C	D
a_1	b_1	c_1	d_1
a_2	b_2		d_2
a_2	b_3	c_2	d_3
a_3	b_4	c_3	d_1
a_3	b_5	c_1	
a_3	b_6	c_2	d_5
a_1	b_2	c_3	d_4

$\forall i,j \ / \ i \neq j \Rightarrow a_i \neq a_j$
$\forall i,j \ / \ i \neq j \Rightarrow b_i \neq b_j$
$\forall i,j \ / \ i \neq j \Rightarrow c_i \neq c_j$
$\forall i,j \ / \ i \neq j \Rightarrow d_i \neq d_j$

Déterminer la ou les clés potentielles de la relation R.

3.2- clef étrangère

3.2.1- Définitions

3.2.1.1- <u>Domaine primaire</u>

Un domaine primaire est un domaine sur lequel une clef primaire est définie.

3.2.1.2- <u>clef étrangère</u>

Un attribut qui n'est pas clef primaire mais qui est défini sur un domaine primaire est appelé une clef étrangère.

<u>Remarque</u> : une clef étrangère est clef primaire dans une relation.

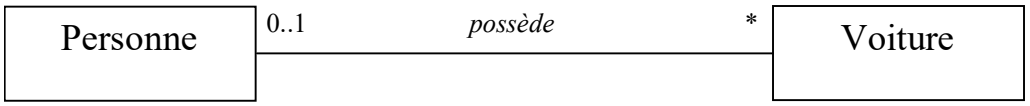
3.2.1.3- <u>Intégrité référentielle</u>

C'est une conséquence de la définition d'une clef primaire et des clefs étrangères. Soit un ensemble d'attributs A d'une relation R_1 définit sur le domaine primaire D d'une relation R_2. Alors à chaque valeur v de A dans R_1 on doit avoir, soit la valeur ***v non renseignée***, soit la valeur v est une des valeurs définie sur le domaine D de la relation R_2.

3.2.2- Exemple

On désire introduire dans la relation « Voiture » l'information suivante :

Toute voiture, lorsqu'elle est vendue, est possédée par un propriétaire qui est une personne.



Personne = {<u>id_personne</u>, nom, prenom, date_naissance, adresse, code_postal };

Voiture = { <u>immatriculation</u>, marque, couleur, prix, #id_personne} ;

Personne	id_personne	nom	prenom	date_naissance	adresse	Code_Postal
	1	Millan	Thierry	17/04/1980	Toulouse	31200
	2	Dupont	Pierre	12/12/1960	Paris	75002
	3	Durant	Louis	01/12/1943	Foix	09000

Voiture	Immatriculation	marque	couleur	prix	id_personne	
	6091XT31	Citroen	Gris	15000	1	
	453ABS31	Peugeot	Blanc	10000		
	234XZY75	Fiat	Bleu	9000	2	
	1234ZE09	Citroen	Rouge	16000	5	IMPOSSIBLE

car la personne ayant comme ***id_personne*** « 5 » n'existe pas dans la relation « Personne », il sera donc impossible d'insérer ce n-uplet dans la relation « Voiture » car il y a violation de la contrainte d'intégrité référentielle.

Si l'on désire supprimer la personne ayant comme ***id_personne*** « 2 », il y a violation de la contrainte d'intégrité référentielle car si ce n-uplet est supprimé alors le propriétaire de la voiture « 234XZY75 » n'a plus de propriétaire.

Si l'on désire modifier l'identifiant de « ***Thierry Millan*** », alors le propriétaire de la voiture « 6091XT31 » n'a plus de propriétaire, il y a donc une violation de la contrainte d'intégrité référentielle.

3.3- Mise en œuvre en SQL

Il existe deux techniques pour créer les clefs primaires et les clefs étrangères associées à une relation : soit directement lors de la création de la relation, soit par ajout de ces contraintes une fois la relation créée.

Il est préférable d'utiliser la seconde solution car lorsque la relation est créée à partir d'une relation existante, les contraintes et en particulier la clef primaire et les clés étrangères ne sont pas créées par cette recopie. La seconde solution permet de les ajouter indépendamment de la création de la relation. De plus, certaines contraintes comme les contraintes de clefs étrangères doivent être créées une fois les clefs primaires créées. Cet ordonnancement n'est pas garanti si les contraintes sont implantées en même temps que la structure.

Il est important de nommer les contraintes afin de pouvoir les manipuler. Par exemple, il est possible de détruire une contrainte ou de la désactiver momentanément

3.3.1- Contrainte de relation ou contrainte d'attribut

La plupart des contraintes peuvent être positionnées soit au niveau des attributs, soit au niveau des relations. La principale différence se situe dans le fait qu'une contrainte au niveau d'un attribut ne peut concerner qu'un seul attribut. Une contrainte de niveau relation peut concerner un ensemble d'attributs.

3.3.1.1- Création d'une contrainte d'attribut durant la création d'une relation

create table Nom_Relation **(**
nom_attribut$_1$ type$_1$ [default valeur$_1$],
nom_attribut$_i$ type$_i$ [default valeur$_i$]
constraint nom_contrainte définition_contrainte,
nom_attribut$_n$ type$_n$ [default valeur$_n$]
...**)** ;

3.3.1.2- Création d'une contrainte de relation durant la création d'une relation

create table Nom_Relation **(**
nom_attribut$_1$ type$_1$ [default valeur$_1$],

nom_attribut$_2$ type$_2$ [default valeur$_2$],
...
nom_attribut$_n$ type$_n$ [default valeur$_n$],
constraint nom_contrainte définition_contrainte
) ;

Remarque :
1. **Les noms des différentes contraintes doivent être différents**.

3.3.1.3- Création d'une contrainte d'attribut après la création d'une relation

alter table nom_relation
modify nom_attribut **constraint** nom_contrainte définition_contrainte;

Remarque : si la contrainte concerne un attribut existant, il s'agit alors d'une modification de l'attribut et il faut utiliser un « **alter table** … **modify** … ».

3.3.1.4- Création d'une contrainte de relation après la création d'une relation

alter table nom_relation
add constraint nom_contrainte définition_contrainte ;

Remarque : les contraintes de relation n'existent pas lors de la création de la relation. Il faut donc les ajouter en utilisant « **alter table** … **add** … ».

Dans la suite de ce cours, nous ne présenterons plus que les contraintes de relation hormis si la contrainte n'existe qu'en tant que contrainte d'attribut. De plus, nous dissocierons la création de la structure de la relation de la création des contraintes.

3.3.2- Définition de la clé primaire

alter table nom_relation
add constraint contrainte **primary key** (nom_att$_i$, [nom_att$_j$, …]) ;

Exemples :

alter table Voiture
modify immatriculation **constraint** CP_Voiture **primary key** ;
Contrainte d'attribut positionnée une fois la relation créée

alter table Personne
add constraint CP_Personne **primary key** (nom, prenom, date_naissance));
Contrainte de relation positionnée une fois la relation créée

ou

alter table Personne
add id_personne char(4) **constraint** CP_Personne **primary key**;
Ajout d'un nouvel attribut avec positionnement d'une contrainte

3.3.3- Définition de clés étrangères

alter table nom_relation
add [**constraint** nom_contrainte] **foreign key** (nom_att$_i$, [nom_att$_j$, …])
references nom_relation$_1$(nom_attribut$_x$, [,nom_attribut$_y$] …);

Remarque :

Si les attributs suivants « foreign key » ont le même nom que ceux suivants « *nom_relation* » alors la liste des attributs suivants « *nom_relation$_1$* » peut être omise.

Exemple :

alter table Voiture
add constraint CE_Personne_id_personne **foreign key (**Prop)
references Personne**(**id_personne**);**

Attention :

foreign key (nom_attribut$_i$,nom_attribut$_j$) **references** relation(nom_attribut$_x$, nom_attribut$_y$) ***est différent de*** **foreign key** (nom_attribut$_i$) **references** relation(nom_attribut$_x$) et **foreign key** (nom_attribut$_j$) **references** relation(nom_attribut$_y$)

3.4- Contraintes liées à l'utilisation des clés

Soit deux relations R_a et R_b avec R_b contenant une clef étrangère référençant la clef primaire de R_a.

3.4.1- Insertion des données

Lors de l'insertion d'un tuple dans $\mathbf{R_b}$, deux cas sont possibles :

1. Soit les valeurs prévues pour la clef étrangère sont présentes comme clef primaire de $\mathbf{R_a}$ et l'insertion est possible ;
2. Soit les valeurs prévues pour la clef étrangère ne sont pas présentes comme clef primaire de $\mathbf{R_a}$ et l'insertion est alors refusée.

3.4.2- Suppression et mise à jour de la clé primaire

Lors de la suppression ou de la mise à jour d'une donnée de $\mathbf{R_a}$ deux cas sont possibles :

1. Soit les valeurs de la clef primaire de $\mathbf{R_a}$ ne sont pas utilisées comme clef étrangère dans $\mathbf{R_b}$ et la suppression ou la mise à jour de $\mathbf{R_a}$ est possible ;
2. Soit les valeurs de la clef primaire de $\mathbf{R_a}$ sont utilisées comme clef étrangère dans R_b et la suppression ou la mise à jour de $\mathbf{R_a}$ dépend de la déclaration de la clef étrangère de $\mathbf{R_b}$:
   i. Soit aucune option, lors de la définition, n'est précisée et dans ce cas la suppression ou la modification du tuple de $\mathbf{R_a}$ est impossible ;
   ii. Soit une option est précisée :

   Foreign key (liste_de_colonne) **references** nom_relation(liste_de_colonne**)** [**on delete** {**no action** | **cascade** | **set default** | **set null**}] [**on update** {**no action | cascade | set default | set null**}]
   No action est équivalent à l'omission des clauses **on delete** ou **on update** ;
   Cascade indique qu'il faut :
   - Détruire les n-uples de la relation $\mathbf{R_b}$ dont la clef étrangère référence la clef primaire de $\mathbf{R_a}$
   - Modifier la valeur de la clef étrangère de manière à refléter les modifications de la valeur de la clef primaire de $\mathbf{R_a}$ correspondante ;

   set default indique que le tuple de $\mathbf{R_a}$ est supprimé (resp. modifié) et que les valeurs de la clef étrangère des tuples de $\mathbf{R_b}$ sont initialisés à leur valeur par défaut. Cette option n'est possible que s'il existe des valeurs par défaut pour les valeurs de la clef étrangère ;
   set null indique que le tuple de $\mathbf{R_a}$ est supprimé (resp. modifié) et que les valeurs de la clef étrangère des tuples de $\mathbf{R_b}$ sont initialisés à « ***null*** ». Cette option n'est possible que si la valeur « ***null*** » est possible pour la clef étrangère ;

3.4.3- Exemple

Reprenons la relation « Voiture » et la clef étrangère « prop » et les tuples suivants :

alter table Voiture
add constraint CE_Personne_id_personne **foreign key (**Prop)
references Personne(id_personne**) on delete cascade**
on update cascade ;

Personne	id_personne	Nom	prenom	date_naissance	adresse	Code_Postal
	1	Millan	Thierry	17/04/1980	Toulouse	31200
	2	Dupont	Pierre	12/12/1960	Paris	75002
	3	Durant	Louis	01/12/1943	Foix	09000

Voiture	Immatriculation	Marque	couleur	Prix	prop
	6091XT31	Citroen	Gris	15000	1
	453ABS31	Peugeot	Blanc	10000	3
	234XZY75	Fiat	Bleu	9000	2
	1234ZE09	Citroen	Rouge	16000	2

delete from Personne **where** id_personne=2 ;
update Persone set id_personne=2 **where** id_personne =3 ;

Personne	id_personne	nom	prenom	date_naissance	adresse	Code_Postal
	1	Millan	Thierry	17/04/1980	Toulouse	31200
	~~3~~ **2**	Durant	Louis	01/12/1943	Foix	09000

Voiture	Immatriculation	marque	couleur	Prix	prop
	6091XT31	Citroen	Gris	15000	1
	453ABS31	Peugeot	Blanc	10000	~~3~~ 2

alter table Voiture
modify constraint CE_Personne_id_personne **foreign key (**Prop)
references Personne(id_personne**) on delete set null**;

delete from Personne **where** id_personne=2;

Personne	id_personne	nom	prenom	date_naissance	adresse	Code_Postal
	1	Millan	Thierry	17/04/1980	Toulouse	31200

Voiture	Immatriculation	marque	couleur	Prix	prop
	6091XT31	Citroen	Gris	15000	1
	453ABS31	Peugeot	Blanc	10000	

4- Requêtes complexes

4.1- Fonctions supplémentaires

least : ***Syntaxe*** : least(exp[, exp]…)
Retourne la plus petite valeur de la liste
greatest : ***Syntaxe*** : greatest(exp[, exp]…)
Retourne la plus grande valeur de la liste
nvl : ***Syntaxe :*** nvl(exp1, exp2)
Retourne exp2 si exp1 est nulle, si non retourne exp1

substr : ***Syntaxe*** : substr(char, m, [,n])
Retourne une portion de ***n*** caractères de la chaîne ***char*** commençant à la position ***m***

4.2- Prédicat "like"

Le prédicat « ***like*** » permet de réaliser une comparaison entre la valeur de la colonne et celle de la chaîne en utilisant des caractères génériques de substitution.

Syntaxe : colonne [not] like char [escape 'c']

Les caractères génériques de substitution offerts par SQL sont :

- % substitue zéro ou plusieurs caractères
- _ substitue un seul caractère

L'option « ***escape*** » permet d'identifier un caractère comme une séquence d'échappement. Ceci est surtout utilisé pour permettre l'utilisation des caractères génériques comme des caractères littéraux. Si la séquence d'échappement est définie par le caractère '\' alors '_' signifie que le souligné est une donnée et non pas un caractère générique.

Exemple : lister tous les clients dont le nom se termine par la chaîne 'nd' et tous les articles dont la désignation comprend la chaîne 'N_D' à partir de la deuxième position.

select * from Client **where** nom **like '%**nd**';**
select * from article **where** designation **like '**_N_D**%' escape '\';**

4.3- Requête multi-tables

Les requêtes multi-tables utilisent autant de relations que le programmeur le désire. Pour être efficace, ces requêtes doivent exploiter autant que faire se peut la relation existant entre les relations c'est-à-dire la relation entre les clefs étrangères et les clefs primaires. En effet, la définition des clefs primaires et des clefs étrangères entraîne la mise en place par le système de gestion de bases de données de mécanismes d'optimisation pour améliorer les performances en consultation.

4.4- prédicats "in" et "exists"

4.4.1- Prédicat "in"

4.4.1.1- Définition

Le prédicat « **in** » permet de comparer la valeur de l'expression située à gauche du mot clef « **in** » à la liste des valeurs comprises entre parenthèses.

Syntaxe : {expression [**not**] **in** {liste_exp | requête} |
liste_exp [**not**] **in** {(liste_exp[, liste_exp]…) | requête}

Le prédicat '**in**' retourne vrai si l'expression de gauche coïncide avec au moins une valeur de la liste de droite ou une valeur retournée par la requête.

Le prédicat '**not in**' retourne vrai si l'expression de gauche ne coïncide avec aucune valeur de la liste de droite ou avec aucune valeur retournée par la requête.

Attention : il faut veiller à ce que la liste des valeurs de droites ne contienne pas de valeur « ***null*** ». En effet, lorsque la valeur « ***null*** » apparaît dans la liste de droite le prédicat « ***in*** » retourne faux.

4.4.1.2- Exemple

Reprenons la relation « Voiture », on désire connaître le prix des Citroen blanches, des Peugeot rouges et des Renault bleus.

select PRIX **from** VOITURE
where (MARQUE, COULEUR) **in** (('Citroen', 'Blanc'), ('Renault', 'bleu'), ('Peugeot', 'rouge'));

Reprenons la relation « ***SALARIE*** », on désire connaître le nom des salariés qui ne sont supérieurs de personne.

select NOM **from** SALARIE
where CODSAL **not in (** **select** CODSUP **from** SALARIE
where CODSUP **is not null);**

4.4.2- Prédicat "exists"

4.4.2.1- Définition

Le prédicat "***exists***" renvoie la valeur vrai ou faux selon le résultat de la sous-requête. Si l'évaluation de la sous-requête donne lieu à un ou plusieurs résultats, la valeur retournée est vraie. Cette valeur est fausse dans le cas contraire.

Syntaxe : **[not] exists** (sous-requête)

Attention : Pour que ce prédicat ait un sens, il faut déterminer un lien entre la requête initiale et la sous-requête : c'est la synchronisation.

4.4.2.2- Exemple

Reprenons la relation « SALARIE », on désire connaître le nom des salariés qui ne sont supérieur de personne.

select NOM **from** SALARIE S1
where not exists (select * from SALARIE **where** CODSUP=S1.CODSAL**);**

4.5- Opérateur de jointure

4.5.1- Produit cartésien

Le produit cartésien consiste à croiser toutes les données d'une relation avec celles d'une autre.

Syntaxe : **select** att_1, att_2, …
from $relation_1$ [$alias_1$], $relation_2$ [$alias_2$], …;

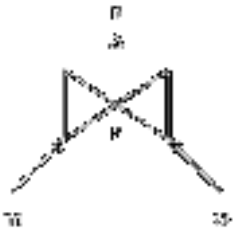
4.5.2- Jointure

Une jointure de deux relations permet de faire un rapprochement entre ces deux relations par comparaison des valeurs des attributs des deux relations. Les attributs utilisés pour la comparaison sont appelés attributs de jointure. Le résultat d'une jointure est une troisième relation virtuelle dont les attributs proviennent des deux relations et les n-uplets sont ceux des deux relations vérifiant la condition.

Syntaxe : **select** att_1, att_2, …
from $relation_1$ [$alias_1$], $relation_2$ [$alias_2$], …
where $relation_1.att_{1i} = relation_2.att_{2j}$
and condition ;

Formalisation avec l'algèbre relationnelle

Représentation graphique



Représentation langage JOIN R1, R2 WHERE P GIVING R

Représentation symbolique $R = R1 \underset{P}{\bowtie} R2 = \sigma_P(R1 \times R2)$

Jointure

4.5.3- Autojointure

Une autojointure est une jointure d'une relation avec elle-même. Dans ce cas, l'utilisation des alias est obligatoire pour différentier les différentes occurrences de la relation.

Exemple : soit la relation « ***Parent*** » contenant les trois attributs « id_mère », « id_enfant » et « id_père » correspondant respectivement à l'identifiant de la mère, de l'enfant et du père. Ecrire la requête permettant de calculer pour tous les enfants ses grands-parents maternels.

```
select P.id_enfant, GP.id_père, GP.id_mère
from Parent P, Parent GP
where P.id_mère = GP.id_enfant ;
```

4.5.4- Jointure externe ou demi-jointure

Une jointure externe est une jointure qui favorise une relation par rapport à une autre. Ainsi, les n-uplets de la relation dominante seront affichés même si la condition n'est pas vérifiée.

Une jointure externe est explicitée par l'opérateur (+) qui est placé dans la clause where après l'attribut de la relation subordonnée.

4.5.5- Syntaxe SQL2 pour la jointure et la jointure externe

	Syntaxe Oracle	**Syntaxe SQL2**
Jointure	**select** att_1, att_2, … **from** rel_1, rel_2 **where** $rel_1.att_{1i} = rel_2.att_{2j}$;	**select** att_1, att_2, … **from** rel_1 [**inner**] **join** rel_2 **on** $rel_1.att_{1i} = rel_2.att_{2j}$;
Jointure externe droite	**select** att_1, att_2, … **from** rel_1, rel_2 **where** $rel_1.att_{1i}$ (+) = $rel_2.att_{2j}$;	**select** att_1, att_2, … **from** rel_1 **right outer join** rel_2 **on** $rel_1.att_{1i} = rel_2.att_{2j}$;
Jointure externe gauche	**select** att_1, att_2, … **from** rel_1, rel_2 **where** $rel_1.att_{1i} = rel_2.att_{2j}$ (+) ;	**select** att_1, att_2, … **from** rel_1 **left outer join** rel_2 **on** $rel_1.att_{1i} = rel_2.att_{2j}$;

4.5.6- Exemples

- Soit les deux relations $R_a(A, B)$ et $R_b(A, C)$.

R_a	A	B
	a_1	b_1
	a_2	b_1
	a_3	b_2

R_b	A	C
	a_1	c_1
	a_4	c_2

Soit les requêtes suivantes :

1. select * from R_a, R_b;
2. select * from Ra, Rb where $R_a.A=R_b.A$;
3. select * from Ra, Rb where $R_a.A(+)=R_b.A$;
4. select * from Ra, Rb where $R_a.A=R_b.A(+)$;

select * from R_a, R_b	R_a.A	B	R_b.A	C
	a_1	b_1	a_1	c_1
	a_1	b_1	a_4	c_2
	a_2	b_1	a_1	c_1
	a_2	b_1	a_4	c_2
	a_3	b_2	a_1	c_1
	a_3	b_2	a_4	c_2

nb tuples créés = nb_tuples(R_a)*nb_tuples(R_b)

select * from R_a, R_b where $R_a.A=R_b.A$;	R_a.A	B	R_b.A	C
	a_1	b_1	a_1	c_1

nb tuples créés <= nb_tuples(R_a)*nb_tuples(R_b)

select * from R_a, R_b where $R_a.A(+)=R_b.A$;	R_a.A	B	R_b.A	C
	a_1	b_1	a_1	c_1
			a_4	c_2

nb tuples créés <= nb_tuples(R_a)*nb_tuples(R_b)

select * from R_a, R_b where $R_a.A=R_b.A(+)$;	R_a.A	B	R_b.A	C
	a_1	b_1	a_1	c_1
	a_2	b_1		
	a_3	b_2		

nb tuples créés <= nb_tuples(R_a)*nb_tuples(R_b)

- Reprenons les relations « Voiture » et « Personnes »,

Personne	id_personne	nom	prenom	date_naissance	adresse	Code_Postal
	1	Millan	Thierry	17/04/1980	Toulouse	31200
	2	Dupont	Pierre	12/12/1960	Paris	75002
	3	Durant	Louis	01/12/1943	Foix	09000

Voiture	Immatriculation	marque	couleur	prix	prop
	6091XT31	Citroen	Gris	15000	1
	453ABS31	Peugeot	Blanc	10000	
	234XZY75	Fiat	Bleu	9000	2

1. Donner le nom des personnes et éventuellement le numéro de leur voiture.
   select nom, immatriculation **from** Personne, Voiture
   where id_personne=prop **(+)** ;
2. Donner le numéro des voitures avec éventuellement le nom de leur propriétaire.
   select immatriculation, nom **from** Personne, Voiture
   where id_personne (+)=prop ;
3. Donner le nom des propriétaires et le numéro d'immatriculation de leur voiture.
   select nom, immatriculation **from** Personne, Voiture
   where id_personne=prop ;

- Reprenons la relation « Salarié », donnez le nom des employés qui ne sont supérieurs de personne (demi-jointure) :
  select sup.nomsal
  from Salarie sal, Salarie sup
  where sup.codsal = sal.codsup **(+)**
  and sal.codsal **is null** ;

Considérons les données suivantes dans la table SALARIE :

CODSAL	NOMSAL	PRENOMSAL	ANCIENETE	SAL	COSUP
S01	DUPONT	Louis	36	1220,00	
S02	DULONG	Pierre	40	1400,00	S01
S03	DUPIN	Jean	45	1500,00	S02
S04	DURANT	Laure	40	1440,00	S02
S05	DUVANT	Jeanne	36	1100,00	S04

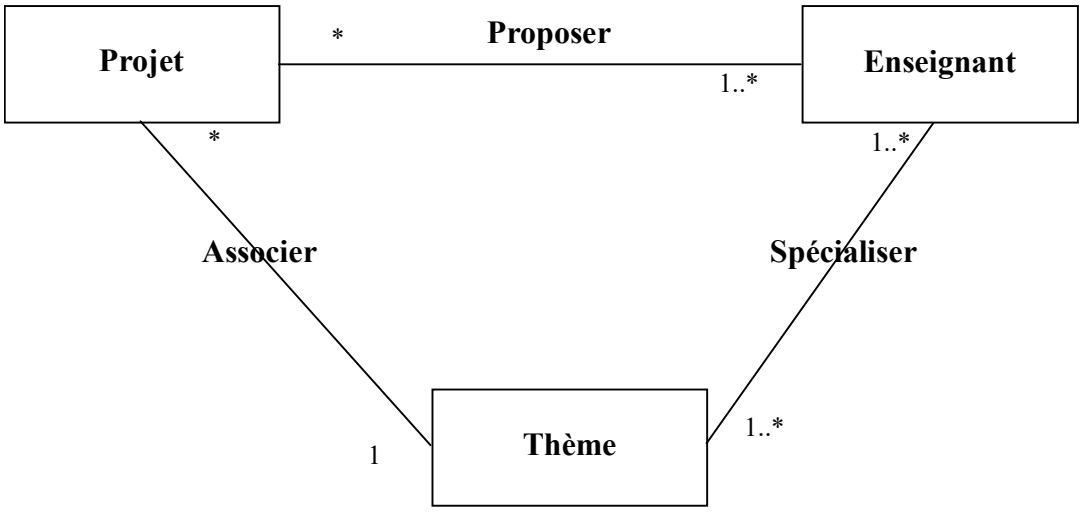
Soit le résultat de la demi- jointure :

SAL		SUP	
CODSAL	COSUP	CODSAL	CODSUP
S01	***null***	***null***	***null***
S02	S01	S01	null
S03	S02	S02	S01
S04	S02	S02	S01
S05	S04	S04	S02
null	***null***	S03	S02
null	***null***	S05	S04

Le résultat de la requête est donc {‘DUPIN’, ‘DUVANT’}

5- Exercice récapitulatif

On se propose d'implanter le diagramme de classes suivant pour gérer les projets proposés par les enseignants.



Le schéma ci-dessus montre que les projets sont proposés par des enseignants et sont rattachés à un seul thème. Chaque thème possède au moins un spécialiste qui est un enseignant.

Le modèle logique correspondant au schéma ci-dessus est le suivant :

- Projet (IDP, Nom, Désignation, #IDT, Complexite)
- Thème (IDT, Nom, Nombre_Heures_Enseignement)
- Enseignant (IDE, Nom, E-Mail, Téléphone)
- Proposer (#IDP, #IDE)
- Spécialiser (#IDT, #IDE, Niveau_Compétence)

1. Implanter le schéma ci-dessus en utilisant le langage SQL. Vous proposerez les types les mieux adaptés pour chaque attribut sachant que les identifiants seront constitués de trois caractères alphanumériques.

2. Implanter les contraintes de clefs primaires et de clefs étrangères

3. Donner les thèmes qui ne sont associés à aucun projet (not in, not exists, demi_jointure)

4. Donner la liste de tous les thèmes classés par ordre alphabétique.

5. Donner l'identifiant des enseignants qui n'ont pas proposé de projet (not in, not exists, demi-jointure, minus).

6. Donner l'identifiant des enseignants qui ont proposé au moins un projet.

7. Donner le nom des enseignants qui ont proposé au moins un projet (in, exists, jointure).

8. Donner le nombre de projets proposés par chaque enseignant (identifiant et nombre).

9. Donner le nombre de projet par thème (identifiant et nombre).

10. Donner le nom du ou des thèmes qui regroupe(nt) le plus de projet.

11. Donner le nom du ou des thèmes qui ont le plus de spécialistes.

12. Donner le nom du ou des thèmes dont les spécialistes ont le niveau de compétence moyen le plus haut.

13. Donner le nom du ou des projets proposés par au moins un spécialiste de niveau 1 du thème auquel le projet est rattaché (Jointure, exists).

14. Donner le nom des projets proposés uniquement par des non spécialistes du thème auquel le projet est rattaché

15. Donner le nom des projets proposés par des non spécialistes du thème auquel le projet est rattaché

16. Donner le nom des projets proposés uniquement par des spécialistes du thème auquel le projet est rattaché.

17. Donner pour chaque projet, le nom du projet, le nom du thème ou il est rattaché et le nom des personnes qui l'ont proposé

18. Donner le nom du ou des projets proposés par au moins tous les spécialistes du thème

19. Donner le nom du ou des projets proposés par tous les spécialistes du thème et uniquement eux

20. Donner le nom du ou des projets proposés par au moins les mêmes enseignants que le projet 'P01'

6- Les contraintes

6.1- contrainte "not null"

C'est la seule contrainte qui ne s'applique qu'aux attributs. En effet, il n'est pas possible de définir la contrainte "**not null**" sur une relation. Cela n'a pas de sens.

alter table nom_relation
modify $nom\_attribut_i$ **constraint** contrainte **not null**;

6.2- Contrainte unique

Cette contrainte permet d'exprimer le fait qu'un attribut ou un ensemble d'attributs est unique pour l'ensemble des n-uplets de la relation.

alter table nom_relation
add constraint contrainte **unique** ($nom\_attribut_i$ [,$nom\_attribut_j$] …);

Remarque :

Soit la relation R={A, B}, les deux définitions ne sont pas équivalentes ;

1. create table R (
   A char(5) constraint UNIQ_R_A unique,
   B char(5) constraint UNIQ_R_B unique) ;

et

2. create table R (
   A char(5),
   B char(5),
   constraint UNIQ_R unique(A, B)) ;

Par exemple, considérons les n-uplets suivants :

R	A	B
	a1	b1
	a1	b2
	a2	b1
	a2	b2

Dans le cas 1, l'insertion est impossible car il y a violation des contraintes UNIQ_R_A, UNIQ_R_B tandis que le cas 2 n'entraîne aucune violation.

6.3- Les clefs candidates

Une clef primaire se caractérise par :

- L'unicité de chacune de ses valeurs ;
- L'impossibilité d'avoir des valeurs nulles pour les valeurs de clef ou ses composantes.

Il est donc possible de définir une clef candidate comme un attribut ou un ensemble d'attributs ayant des valeurs distinctes et des valeurs non nulles.

6.4- Contraintes sur les valeurs attributs

Il est possible de définir des contraintes sur les valeurs des attributs. Les contraintes définies au niveau des attributs ne peuvent pas porter sur les valeurs des autres attributs du n-uplet. Pour faire porter une contrainte sur plusieurs attributs d'un même n-uplet, il faut définir la contrainte au niveau de la relation.

alter table nom_relation
add constraint contrainte **check** condition;

Syntaxe d'une condition :

La syntaxe d'une condition « check » est similaire à la syntaxe d'une condition d'une clause « where » pour une relation. Toutefois, la syntaxe d'une condition pour une clause check ne peut en aucun cas utiliser les opérateurs de la clause having. En effet, il est impossible d'utiliser les opérateurs portant sur des regroupements : count, sum, avg, …

Exemples :

- L'age d'un salarié est compris entre 16 ans et 65 ans.
  alter table SALARIE
  modify age **constraint** CHK_AGE **check** (age **between** 16 **and** 65) ;
- Seul les salariés de type « COMMERCIAL » peuvent avoir une partie variable dans le salaire.

Considérons l'algorithme de vérification de cette contrainte :

si TYPE = 'COMMERCIAL' alors
  contrainte vérifiée
sinon si PARVAR est non renseigné alors
  contrainte vérifiée
sinon
  contrainte non validée
fin

Cet algorithme peut se traduire en logique du premier ordre par l'expression suivante :

TYPE ≠ 'COMMERCIAL' ⇒ PARVAR est non renseignée

En logique du premier ordre $(A \Rightarrow B) \Leftrightarrow (\neg A \vee B)$

Dans notre exemple, cela se traduit par :

$\neg$(TYPE ≠ 'COMMERCIAL') $\vee$ PARVAR est non renseignée ⇔

TYPE = 'COMMERCIAL' $\vee$ PARVAR est non renseignée

Cette dernière expression se traduit aisément en SQL :

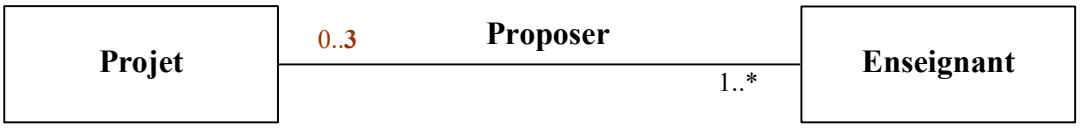
TYPE = 'COMMERCIAL' or PARVAR is null

D'où l'expression de la contrainte :

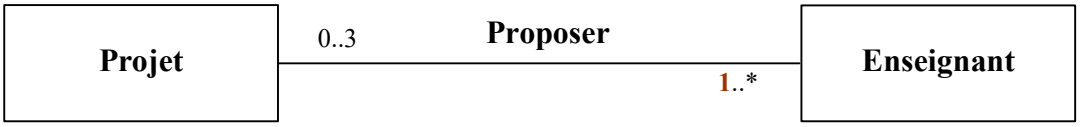
```
alter table SALARIE
add constraint CHK_PARVAR check (
        TYPE = 'COMMERCIAL' or  PARVAR is null) ;
```

6.5- Contraintes *non exprimables en SQL 2*

1. Il est impossible d'exprimer des contraintes portant sur plusieurs n-uplets.
   Par exemple, il est impossible d'exprimer :
   - Un enseignant ne peut être spécialiste de niveau 1 que d'un seul thème ;
   - Un enseignant ne peut pas proposer plus de trois projets.



2. Il est aussi impossible d'exprimer des contraintes nécessitant la connaissance des attributs d'une autre relation. Par exemple, il est impossible d'exprimer la règle qui dit :
   Tout projet doit être proposé par au moins un enseignant



Pour ces types de contrainte, il est nécessaire d'utiliser un autre formalisme tel que les déclencheurs (triggers) Pl/SQL dans le cas d'Oracle ou les gestionnaires d'évènements VBA pour SQL-Server. Attention toutefois à n'utiliser les déclencheurs qu'à bon escient car leur coût d'exécution est loin d'être négligeable. Il est pénalisant d'utiliser un déclencheur pour mettre en œuvre une contrainte de type « 1..1 ».

7- Architecture générale d'un système de bases de données relationnel

7.1- Objectifs

Les objectifs principaux des systèmes de gestion de bases de données sont :

- L'indépendance physique : réalisation de l'indépendance des structures de stockage aux structures de données. Il faut donc pouvoir définir l'assemblage des données élémentaires entre elles dans le système informatique indépendamment de l'assemblage réalisé dans le monde réel, en tenant compte seulement des critères de performances et de flexibilités d'accès ;
- L'indépendance logique : c'est permettre à chaque groupe de travail de voir les données comme il le souhaite. Il en résulte la possibilité de faire évoluer la vue de chaque groupe de travail ;
- *La manipulation des données par des non-informaticiens ;*
- L'efficacité des accès aux données ;
- L'administration centralisée des données ;
- La non-redondance des données ;
- La cohérence des données ;
- La partageabilités des données ;
- La sécurité des données : les données doivent être protégées contre les accès non autorisés ou mal intentionnés.

7.2- Les différents niveaux de description de données

Selon l'architecture ANSI/SPARC, la description des données dans un SGBD se fait à trois niveaux. A chacun de ces niveaux correspond un ou plusieurs schémas.

7.2.1- Le niveau conceptuel ou logique

C'est l'univers réel à modéliser. Il décrit l'univers réel à l'aide des concepts du modèle utilisé. Cette description concerne les entités avec leurs caractéristiques, les liens entre les entités et éventuellement des règles de gestion appelées contraintes d'intégrité. A ce niveau, on fait abstraction de l'utilisation des données ainsi que de leur mise en œuvre physique. Cette description est représentée dans un schéma dit schéma conceptuel.

7.2.2- Le niveau interne ou physique

Un schéma dit schéma interne décrit la manière dont les objets conceptuels sont stockés sur la mémoire secondaire et la correspondance entre structures logiques de données et structures physiques. Les choix des structures de stockage doivent se faire en tenant compte des contraintes de mise en œuvre et de l'utilisation qui sera faite des données de façon à optimiser les accès à la base.

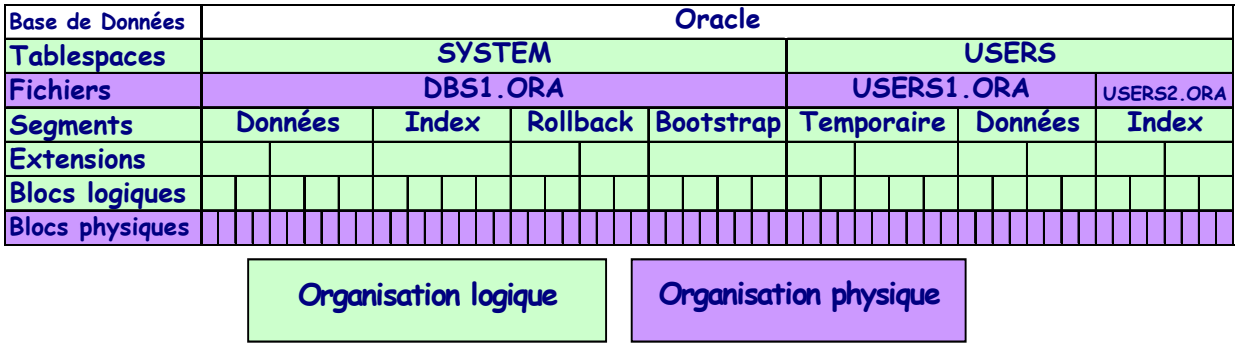
7.2.3- Le niveau externe

Le niveau externe correspond aux vues que vont avoir les utilisateurs, par l'intermédiaire des applications, des entités du schéma conceptuel. Ces différentes vues sont décrites à l'aide de schémas externes ou sous-schémas. Chaque schéma externe traduit un type d'utilisation de la base de données.

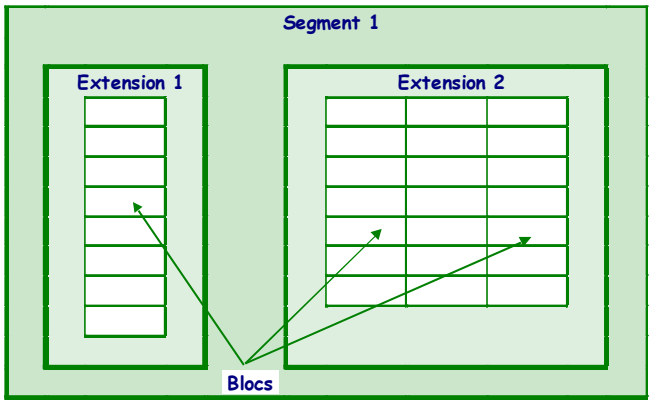
Ce choix d'architecture a pour objectif d'accroître le niveau d'indépendance entre les données et les traitements à savoir, l'indépendance physique, l'indépendance logique et l'indépendance des vis-à-vis des stratégies d'accès (un programme d'application n'a pas à préciser comment accéder à telle ou telle donnée mais uniquement ce qu'il désire).

7.3- Le stockage physique des données avec Oracle

7.3.1- Correspondance entre Structure Physique et Logique

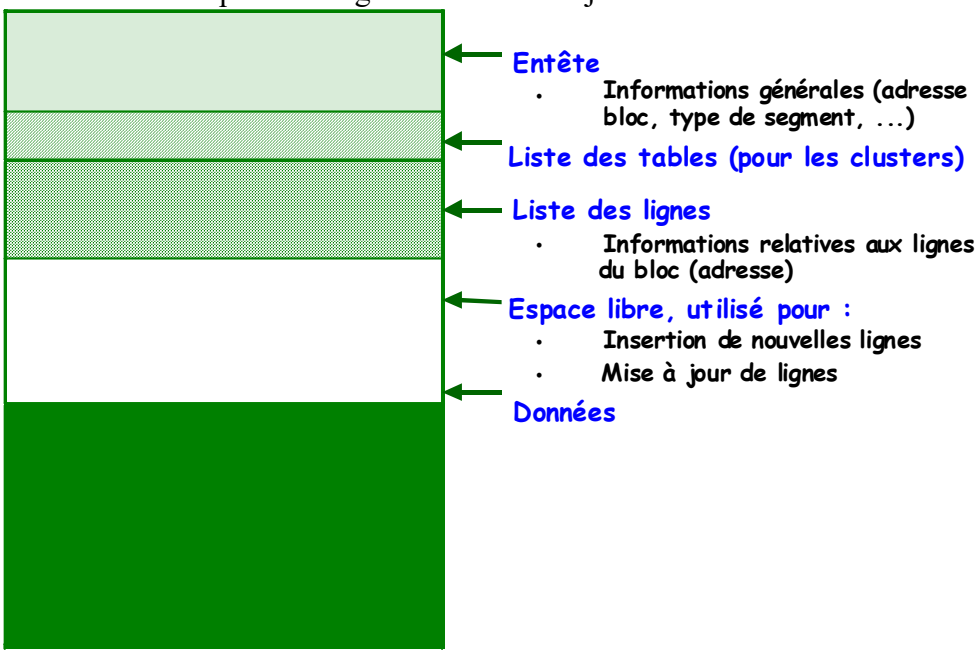


7.3.2- Correspondance entre les Trois Niveaux Logiques : Segment, Extension et Blocs



7.3.3- Format d'un Bloc de données

Cluster : c'est un regroupement physique d'une ou plusieurs tables autours d'une ou plusieurs colonnes communes. Un cluster est une structuration de données dans une ou plusieurs tables pour permettre un accès rapide aux lignes issues d'une jointure.



7.4- Index

Le but principal d'un index est d'éviter de parcourir une table séquentiellement du premier enregistrement jusqu'à celui visé. Le principe d'un index est l'association de l'adresse de chaque enregistrement avec la valeur des colonnes indexées. Un index permet d'améliorer les performances lors des consultations lorsque les relations contiennent un grand nombre de données. Un index peut être soit unique, soit non unique. Il existe trois types d'index sous Oracle : les index basés sur les B-tree, les index basés sur des fonctions et ceux appelé bitmap.

7.4.1- Les différents types d'index

7.4.1.1- Les index basés sur les B-tree

La particularité de ce type d'index est qu'il conserve en permanence une arborescence symétrique (balancé). Toutes les feuilles sont à la même profondeur. Le temps de recherche est ainsi à peu près constant quel que soit l'enregistrement cherché. Le plus bas niveau de l'index contient les valeurs des colonnes indexées et l'identifiant de l'enregistrement. Toutes les feuilles de l'index sont chaînées entre elles. Il n'y a pas de dégradation des performances lors de la monté en charge.

7.4.1.2- Les index bitmap

Alors qu'un index de type B-tree permet de stocker une liste d'identifiants pour chaque valeur de la colonne indexée, un bitmap ne stocke qu'une chaîne de bits. Chaque bit correspond à une valeur possible de la colonne indexée. Si le bit est positionné à 1, pour une valeur donnée de l'index, cela signifie que le tuple courant contient une valeur. Une fonction de transformation convertit la position du bit en identifiant. Si le nombre de valeur de la colonne indexée est faible, l'index bitmap sera très peu gourmand en occupation de l'espace disque.

7.4.1.3- Les index basés sur des fonctions

Une fonction de calcul, autres que les fonctions de regroupement (SUM, COUNT, MAX, etc.) peut définir un index. Il est dit basé sur une fonction. Ces index servent à accélérer les requêtes contenant un calcul pénalisant s'il est effectué sur de gros volumes de données. Un index basé sur une fonction peut-être de type B-tree ou bitmap mais dans ce cas l'index bitmap ne peut pas être de type unique.

7.4.1.4- Quel index utiliser ?

Le tableau ci-dessous résume l'utilisation des index en fonction de l'application visée.

	B-tree	Bitmap	fonction
Application décisionnelles		+++	
Applications transactionnelles	+++		
Applications avec de gros calculs			+++

7.4.2- Création et mise en place des index

7.4.2.1- Index mis en place automatiquement

Lors de la déclaration d'une contrainte de clef primaire ou d'une contrainte de type unique, un index de type unique est automatiquement créé.

7.4.2.2- Index mis en place par l'utilisateur

7.4.2.2.1 Création d'index

Syntaxe : **create index** nom_index
on nom_relation(att_1, …, att_n) ;

7.4.2.2.2 Index conseillés

1. Il est important d'associer pour chaque clef étrangère un index. En effet, les clefs étrangères sont souvent utilisées lors de jointure et une optimisation est alors la bienvenue.

   Remarque :
   Un problème se pose pour les attributs participant à la fois à une clef primaire et étant individuellement des clefs étrangères. Par exemple si on reprend la relation "***Proposer***" de l'exercice précédent, les attributs IDE et IDP constituent la clef primaire de la relation. Il y a donc un index unique automatiquement créé sur la relation "***Proposer***".

   Question : faut-il créer un index sur les attributs IDE et IDP ?
   Pour IDE ce n'est pas la peine car celui existant sur la clef primaire est suffisant. Pour IDP cette création est essentielle car l'index sur la clef primaire est trié sur l'attribut IDE et n'a que peut d'effet sur l'attribut IDP.

2. On peut aussi créer des index sur des attributs souvent accédés seuls et ayant une grande plage de valeur possible. Par exemple, le nom dans la relation "Personne".

7.4.2.2.3 Limitation

Il faut éviter :

- De positionner plus de trois index sur une même relation ;
- De positionner des index sur des attributs souvent mis à jour ;
- De créer des index pour une petite relation.

7.4.3- Exercice

- En reprenant l'exercice récapitulatif, indiquez les index créés automatiquement lors de la mise en place de relations ;
- Créez les index qui vous semblent essentiels à des performances optimales du système.

8- Les plans d'exécution

Oracle possède un outil permettant de décrire le plan d'exécution d'une requête. Cette description comprend les chemins d'accès utilisés, les opérations physiques tels que par exemple le tri ou la fusion, et l'ordre des opérations représenté par un arbre.

8.1- Principe

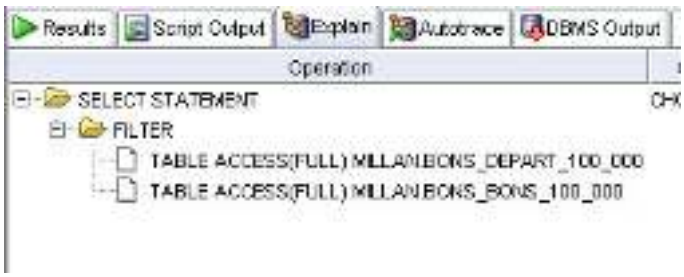
Le SGBD analyse la requête et transforme celle-ci en un arbre d'exécution. Celui-ci est optimisé en fonction des index et des optimisations mises en œuvre sur les relations manipulées par la requête. Cet arbre optimisé est enfin évalué pour obtenir le résultat.

8.2- Interprétation

Soit deux relations BONS_DEPART et BONS_BONS qui contiennent respectivement les résultats d'un concours et les résultats une fois les bons erronés supprimés. Ces relations contiennent dans l'ordre le numéro du bon de participation, la réponse à la première et à la seconde question.

Dans un premier temps, nous étudierons les quatre types de requêtes vues en cours (ensembliste, jointure, exists, in) lorsqu'aucun index n'est positionné. Dans un second temps, nous étudierons les mêmes requêtes lorsque des index sont positionnés.

8.2.1- Requête sans optimisation

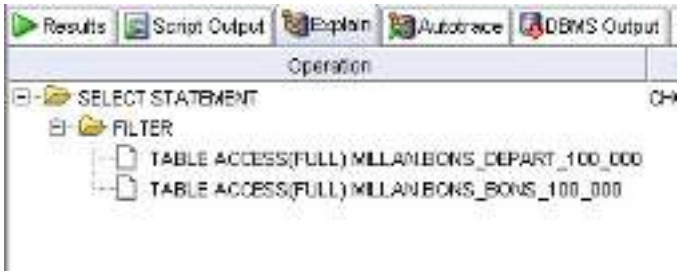



```
select * from BONS_DEPART
where NUMBON not in (select NUMBON from BONS_BONS) ;
```

Temps d'exécution pour 100 000 n-uplets : ∞

Complexité : n^2
Soit dans notre cas $1e^{+10}$ données traitées

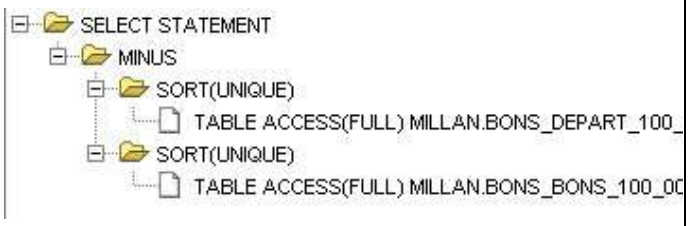
filter : Accepte un ensemble de lignes, applique un filtre pour en éliminer quelques unes et retourne le reste ;
table access full : Obtension de toutes les lignes d'une table ;




```
select * from BONS_DEPART
where not exists (select NUMBON from BONS_BONS
where BONS_BONS.NUMBON = BONS_DEPART.NUMBON) ;
```

Temps d'exécution pour 100 000 n-uplets : ∞

Complexité : n^2
Soit dans notre cas $1e^{+10}$ données traitées

<table>
<tr>
<td> 
</td>
<td>select NUMBON, REP1, REP2 from BONS_DEPART
minus
select NUMBON, REP1, REP2 from BONS_BONS ;

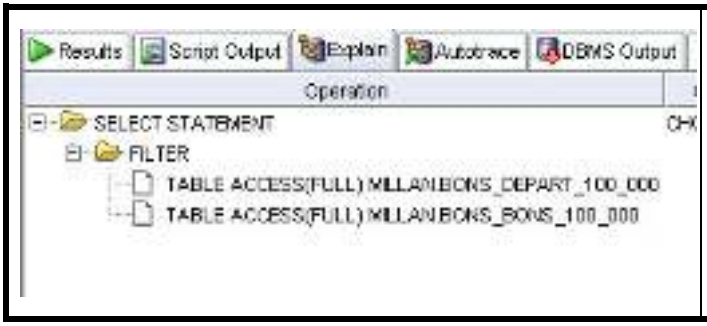
Temps d'exécution pour 100 000 n-uplets : 2,43 s

Complexité : 2*n*log(n)+2*n=2*n*log(n)
Soit dans notre cas 2,5e^{+6} données traitées</td>
</tr>
<tr>
<td colspan="2">minus : Différence de deux ensembles de lignes ;
sort unique : Tri d'un ensemble de lignes pour éliminer les doublons.</td>
</tr>
<tr>
<td> 
</td>
<td>select distinct BONS_DEPART.NUMBON, BONS_DEPART.REP1, BONS_DEPART.REP2
from BONS_BONS, BONS_DEPART
where BONS_BONS .NUMBON (+) = BONS_DEPART.NUMBON
and BONS_BONS .NUMBON is null ;

Temps d'exécution pour 100 000 n-uplets : 2,377 s

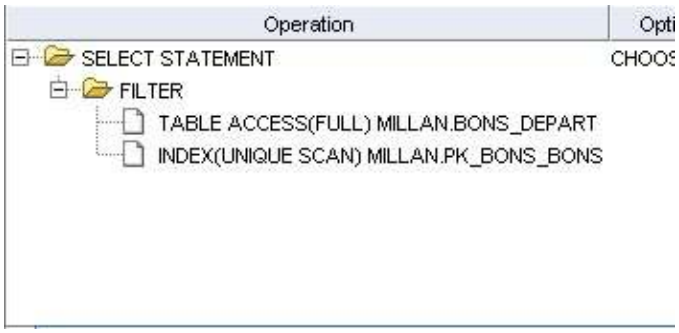
Complexité : 2*n*log(n)+2*n=2*n*log(n)
Soit dans notre cas 2,5e^{+6} données traitées</td>
</tr>
<tr>
<td colspan="2">merge join outer : Accepte deux ensembles de lignes (chacun trié selon un critère), combine chaque ligne du premier ensemble avec ses correspondants du second et retourne le résultat. Le « outer » indique une jointure externe ;
sort join : Tri avant la jointure (merge join).</td>
</tr>
</table>

8.2.2- Requête avec optimisation

<table>
<tr>
<td> 
</td>
<td>select * from BONS_DEPART
where NUMBON not in (select NUMBON from BONS_BONS) ;

Temps d'exécution pour 100 000 n-uplets : ∞

Complexité : n^2
Soit dans notre cas 1e^{+10} données traitées</td>
</tr>
</table>

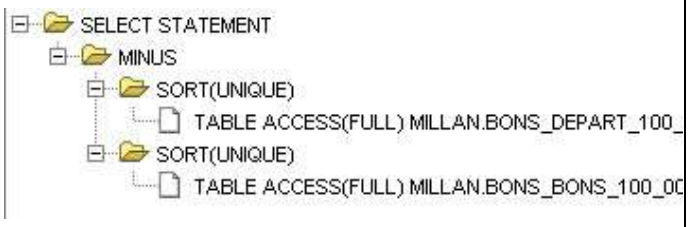



```
select * from BONS_DEPART
where not exists (select NUMBON from BONS_BONS
where BONS_BONS.NUMBON = BONS_DEPART.NUMBON) ;
```

Temps d'exécution pour 100 000 n-uplets : 1,422 s

Complexité :2*n
Soit dans notre cas $2e^{+5}$ données traitées

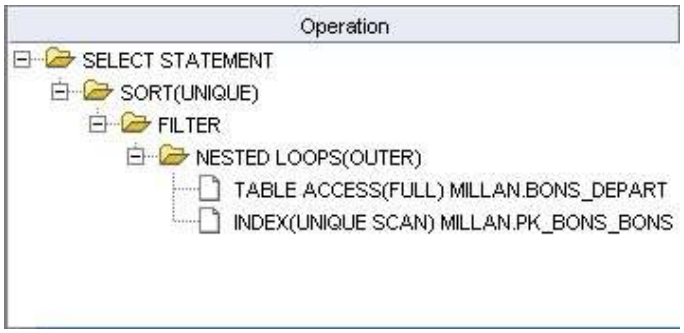
index unique scan : Recherche d'une seule valeur Rowid d'un index. Un rowid est une chaîne hexadécimale représentant l'adresse unique d'une ligne de la table. Sa valeur est retournée par la pseudo-colonne de même nom




```
select NUMBON, REP1, REP2 from BONS_DEPART
minus
select NUMBON, REP1, REP2 from BONS_BONS ;
```

Temps d'exécution pour 100 000 n-uplets : 2,43 s

Complexité : 2*n*log(n)+2*n=2*n*log(n)
Soit dans notre cas $2,5e^{+6}$ données traitées




```
select distinct BONS_DEPART.NUMBON, BONS_DEPART.REP1, BONS_DEPART.REP2
from BONS_BONS, BONS_DEPART
where BONS_BONS .NUMBON (+) = BONS_DEPART.NUMBON
and BONS_BONS .NUMBON is null ;
```

Temps d'exécution pour 100 000 n-uplets : 0,953 s

Complexité : ?

nested loops outer : Accepte deux ensembles de lignes (chacun trié selon un critère), combine chaque ligne du premier ensemble avec ses correspondants du second et retourne le résultat. Le « outer » indique une jointure externe ;

Remarque : les exemples ci-dessus ne présentent pas la totalité des opérateurs possibles. L'objectif est ici de présenter succinctement les principes qui régissent les plans d'exécution, et au de là l'optimisation des requêtes.