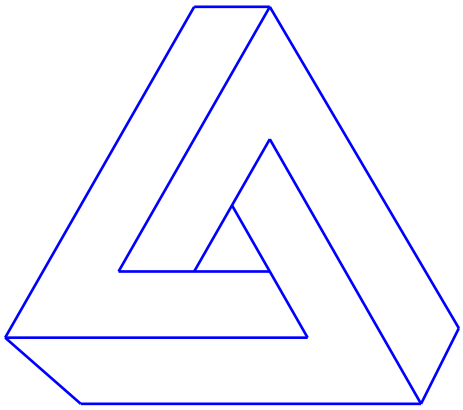
Livret de cours

Bases de données avancées

Pascal Lafourcade et Laurent Desrosiers



pascal.lafourcade@uca.fr ldesrosiers@chu-clermontferrand.fr

IUT CLERMONT - FD
UNIVERSITÉ
Clermont Auvergne

Nom :

Prénom :

Groupe :

Avant Propos

L'objectif de ce cours de base de données avancées est de présenter les fonctionnalités avancées de gestion et de compréhension des bases de données. Cela commence par la modélisation et la gestion des données redondantes grâce aux formes normales. Ensuite des fonctionnalités permettant de manipuler vos données et les utilisateurs de vos bases de données seront abordées. Puis la concurrence d'accés aux informations sera présentée afin de vous faire comprendre les différents mécanismes qui bloquent l'accès à certains utilisateurs afin de préserver l'intégrité des données. Enfin la notion d'optimisation des requêtes sera présentée grâce aux plans d'exécution et aux index.

Le cours contient de nombreux exemples illustrant les différents concepts et mécanismes abordés. Il y a aussi des exercices qui sont là pour que vous puissiez vous entraîner à assimilier le contenu du cours. Il est important que vous essayer de les faire par vous même. Attendre la solution revient à transformer un exercice en exemple, ce qui n'a plus aucun interêt pour vous. Il est crucial que vous confrontiez vos représentations mentales à des problèmes pour vous rendre compte si vous avez correctement compris une notion mais aussi pour mieux l'assimiler. Comme le disait Conficius :

"J'entends et j'oublie,
Je vois et je me souviens,
Je fais et je comprends."

Enfin de nos jours l'importance des données n'est plus à démontrer. Il y a de plus en plus d'exemples d'entreprises ayant fait fortune en sachant collecter, stocker et analyser des quantités volumineuses de données. À une plus petite échelle chaque PME possède aussi des données qu'il est important de savoir gérer. Dans cette nouvelle ère du numérique et de la digitalisation des services, la gestion des données est un élément crucial dans le développement du monde économique de demain. Ainsi il y a de fortes chances que dans vos différents métiers de futurs informaticiens et pour certains d'entre vous lors de votre stage de fin d'année, vous soyez amenés à concevoir ou à utiliser des bases de données. Il sera alors utile de se référer à ce cours pour organiser vos données lors de la conception de vos systèmes de gestion de base de données mais aussi pour optimiser vos requêtes ou encore faciliter l'interface avec l'utilisateur.

Évaluation Ls note finale NF est calculée avec la formule suivante :

$$NF = 60\%DS + 40\%CC$$

où DS correspond à la note du Devoir Surveillé de 1h00 en amphi sur feuille avec comme seul document autorisé une feuille A4 recto-verso de note personnelles manuscrites, et CC correspond au Contrôle Continu. La note de CC est calculée avec la formule suivante :

$$CC = 60\%TD + 40\%TP$$

où TD correspond à la moyenne des 3 meilleures notes obtenues lors des évaluations de début de TD et TP correspond à la note obtenue pour les Travaux Pratiques sur machine. Il y a 4 comptes rendus de TP à rendre et 1 seul sera corrigé a posteriori pour tous les groupes.

Table des matières

1 Modélisation des données **3**
1.1 Modèles de données (MLD et MCD) . . . 3
1.2 Définitions . . . 4
1.3 Associations . . . 5

2 Gestion des données redondantes **8**
2.1 Rappels . . . 8
2.2 Formes normales . . . 9
2.3 1FN - Première forme normale . . . 10
2.4 2FN - Deuxième forme normale . . . 11
2.5 3FN - Troisième forme normale . . . 11
2.6 4FN - Quatrième Forme Normale . . . 13
2.7 Du MCD au modèle relationnel . . . 14

3 Fonctionnalités avancées **24**
3.1 Types de données . . . 24
3.2 Manipulation de table . . . 25
3.3 Contraintes non référentielles . . . 25
3.4 Contraintes référentielles . . . 26
3.5 Vue . . . 29
3.6 Synonyme . . . 31
3.7 Séquence . . . 31
3.8 Utilisateur . . . 32
3.9 Privilège . . . 33

4 Fonctions **35**
4.1 Fonctions mono-lignes . . . 35
4.2 Fonctions de regroupement . . . 38
4.3 Fonctions personnalisées . . . 39
4.4 Fonctions courantes . . . 39

5 PL/SQL **41**
5.1 Instructions conditionnelles . . . 44
5.2 Boucle WHILE . . . 44
5.3 Boucle LOOP . . . 45
5.4 Boucle FOR IN . . . 45
5.5 Exceptions . . . 45
5.6 Déclencheurs (triggers) . . . 50

6 Concurrence d'accès **53**
6.1 Transaction . . . 53
6.2 ACID . . . 53
6.3 Lecture cohérente et undo . . . 54

7 Optimisation Oracle **63**
7.1 Sous-requête . . . 64
7.2 Index . . . 65
7.3 Plan d'exécution . . . 69

1 Modélisation des données

Concevoir et déployer une base de données est une tâche complexe. Cela nécessite de se poser des questions sur comment strucurer les données. Pour cela il est important d'utiliser des modèles éprouvés par la communauté et d'avoir un cadre formel pour analyser la strucure des données pour arriver à un déploiement efficace.

1.1 Modèles de données (MLD et MCD)

Dans les années 1970, le besoin d'une méthode d'analyse rigoureuse, permettant de gérer des projets de grande envergure, donna naissance à la méthode *Merise*. Cette méthode permet à partir d'un besoin réel, exprimé en langage naturel (par exemple le français ou l'anglais) de définir la structure des données à utiliser dans une base de données. Reposant sur l'analyse systémique introduite par Norbert Wiener en 1950, elle repose sur le principe de dissociation de l'analyse de données et de l'analyse des traitements. Le tableau 1 montre les 3 niveaux de l'analyse Merise et les modèles associés.

MCD : **M**odèle **C**onceptuel de **D**onnées

MCT : **M**odèle **C**onceptuel des **T**raitements

MLD : **M**odèle **L**ogique de **D**onnées

MOT : **M**odèle **O**rganisationnel des **T**raitements

La méthode Merise est particulièrement longue à mettre en œuvre lorsque l'ensemble des modèles est pris en compte. Il s'avère que les deux modèles présentant le plus d'intérêt sont le MCD et le MLD. Le MCD doit répondre à la question « *Quelles informations sont manipulées ?* ». Il contient donc l'ensemble des données et repose sur les notions d'entité et d'association. Le MLD donne la structure des données qui sont stockées en base.

Niveau / Analyse	**Données**	**Traitements**
Conceptuel	Quelles informations sont manipulées ?	Que est l'objectif ?
Logique	Comment structurer ces données ?	Qui fait quoi, où et quand ?
Physique	Où les stocker ?	Comment les stocker ?

Tableau 1 – Modèles des données et des traitements, modèles d'après la méthode Merise.

Exemple 1.1. Soit une base de données qui modélise des personnes et des projets, partant d'une personne il faut décrire son association avec les projets. Et partant du projet il faut décrire l'association vers les personnes.

Par exemple, pour modéliser les données nécessaires à une gestion de projets *où il y a toujours une personne qui est nommée chef de projet pour un projet et que plusieurs personnes travaillent sur un projet*, il n'y a pas toutes les informations nécessaires. Bien que cela pourrait suffire à un concepteur habitué à la modélisation des données pour produire un MCD possible, il manque des informations pour ne pas faire d'erreur même pour un expert. Il est donc important de se poser les bonnes questions lors de cette phase de conception. Les questions suivantes permettent de clarifier la situation :

— Est-ce qu'une personne est obligatoirement chef de projet au moins une fois ?

— Est-ce que toute personne enregistrée travaille sur au moins un projet ?

— Est-ce qu'une personne ne peut travailler que sur un seul projet à la fois ?

Muni des réponses à ces questions, il est possible d'écrire des **règles de gestion** relatives aux associations entre les personnes et les projets, sous formes de phrases décrivant comment sont associés les concepts, comme par exemple :

1. Un projet est dirigé par une seule personne (nommée « chef de projet » pour ce projet) et une personne peut diriger plusieurs projets (en étant nommée « chef de projet » pour chacun d'eux).
2. Un projet fait travailler de une à plusieurs personnes et une personne peut travailler sur plusieurs projets.

1.2 Définitions

Il y a plusieurs éléments importants à connaître pour réaliser un MCD.

Définition 1.1. Les principaux éléments permettant de réaliser un MCD sont :

Entité : Représentation d'un élément du système d'information qui est un regroupement logique de données (par exemple, individu, véhicule, commande, ordinateur, ...). Une entité est constituée d'attributs.

Attribut : Donnée élémentaire décrivant une information intrinsèque à une entité.

Domaine d'un attribut : Ensemble, fini ou infini, des valeurs possibles d'un attribut.

Identifiant : Une ou plusieurs données élémentaires regroupée permettant d'identifier de manière unique une occurrence de l'entité est appelée *identifiant* d'une identité. L'identifiant est, par convention, souligné afin de le différencier des autres attributs.

Association ou relation : C'est un sous-ensemble du produit cartésien de n domaines d'attributs ($n > 0$). Elle représente les liens existants entre les entités. Il existe plusieurs types d'associations hiérarchiques, non-hiérarchiques, réflexives.

Cardinalité : La *cardinalité* d'une association est composée de deux entiers positifs : le nombre minimum et le nombre maximum de fois où une occurrence d'une entité peut participer à une association. Si le nombre maximum exact est inconnu, la cardinalité sera notée n.

Exemple 1.2. Dans la Figure 1, l'entité INDIVIDU est constituée des attributs Numéro de SS, Nom, Prénom, Nationalité, Taille, Poids, ... l'identifiant de cette entité est le numéro de Sécurité Sociale qui est unique par individu.

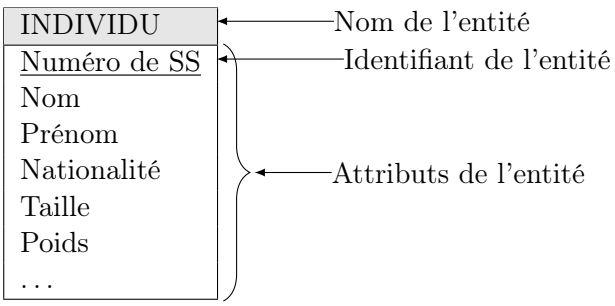


Figure 1 – Exemple d'entité.

Remarque 1.1. Le numéro de Sécurité Sociale est absolument interdit en tant qu'identifiant d'entité (voir CCNIL et 2nde GM). Le décret n° 2017-412 du 27 mars 2017 permet de l'utiliser comme identifiant dans le domaine de la santé et le médico-social [1].

1. `https://www.cnil.fr/fr/le-numero-de-securite-sociale`

Exemple 1.3. L'association « Participe à » de la Figure 2, explicite le lien entre les entités ETUDIANT et EXAMEN. Au minimum aucun étudiant participe à un examen et au maximum n étudiants participent à un examen, ce qui explique la cardinalité $0, n$. A un examen participe au minimum un étudiant et au maximum n, d'où la cardinalité $1, n$.

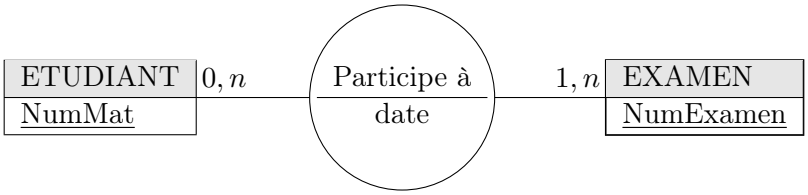


Figure 2 – Exemple d'association.

1.3 Associations

Il existe deux types d'association en fonction des cardinalités entre les entités concernées. Il est donc important de bien déterminer les cardinalités des association lors de la conception d'une base de données car cela aura un impact sur la structure des données.

Association non hiérarchique

Définition 1.2 (Association non hiérarchique)**.** Une association est dite ***non hiérarchique*** lorsque les **cardinalités maximales sont à n** pour les **deux** entités.

Exemple 1.4. Dans la relation non hiérarchique de la Figure 3, un client peut acheter 1 à n articles et un article peut être achetée par 0 ou n client.

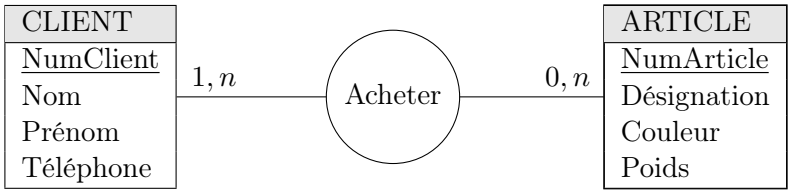


Figure 3 – Exemple de relation non hiérarchique.

Association hiérarchique ou dépendance fonctionnelle

Définition 1.3 (Association hiérarchique ou dépendance fonctionnelle)**.** Une ***association hiérarchique*** ou **dépendance fonctionnelle (DF)** est une association où pour une entité, la cardinalité **maximale est** 1 et de l'autre entité, la cardinalité **maximale est à** n.

L'entité ayant le maximum à 1 est appelée entité «***fils***». L'entité ayant un max à n est appelée entité «***père***», car «*un fils n'a qu'un seul père, et un père peut avoir plusieurs fils* ». Connaissant l'occurrence de l'entité «*père* », il est possible de retrouver l'occurrence de l'entité «*fils*».

Définition 1.4 (Association faible, forte)**.** Si la **cardinalité côté fils d'une association hiérarchique est (0,1),** alors l'association est dite ***faible***. Si la **cardinalité du côté fils est (1,1)** alors l'association est dite ***forte***. Une association hiérarchique forte est aussi appelée **Contrainte d'Intégrité Fonctionnelle (CIF)**. Il existe ainsi systématiquement un lien pour l'ensemble des occurrences de l'association.

Exemple 1.5 (CIF)**.** Dans la relation de la Figure 4, toute commande est systématiquement rattachée à un client. Il existe forcément une commande pour un client car il est considéré comme client à partir du moment où il commande.

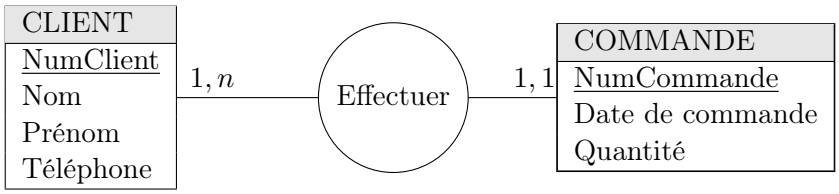


Figure 4 – Exemple d'association hiérarchique.

Exemple 1.6 (Dépendence fonctionelle faible)**.** La Figure 5 montre une relation pour une base de données pour un assembleur d'ordinateur. Un ordinateur est composé de 1 à N composant. Un composant peut, par contre, n'être utilisé dans aucun ordinateur.

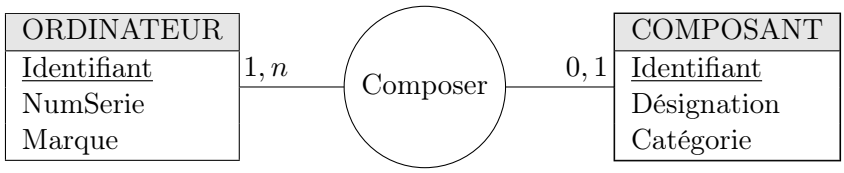


Figure 5 – Exemple de dépendance fonctionnelle faible.

Association réflexive

Définition 1.5 (Association réflexive)**.** Une association ***réflexive*** est une association hiérarchique reliant des occurrences de la même entité. Pour lire une association réflexive, il faut connaître le rôle attribué à chaque branche de l'association.

Association réflexive non hiérarchique

Définition 1.6 (Association réflexive non hiérarchique)**.** Une association *réflexive non hiérarchique* est une association reliant des occurrences de la même entité, pour lesquelles il n'existe pas de considération hiérarchique, c'est à dire l'association est non hiérarchique.

Exemple 1.7 (Association réflexive non hiérarchique)**.** Dans l'association réflexive non hiérarchique de la Figure 6, une molécule peut être composée de plusieurs molécules.

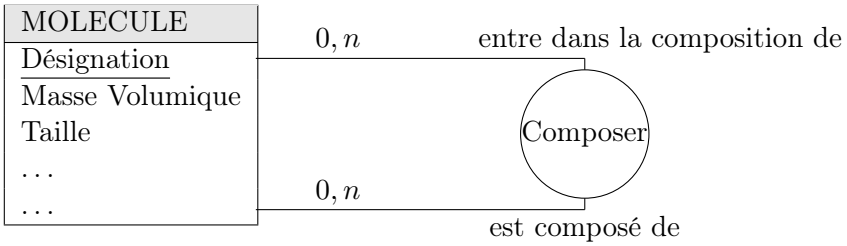


Figure 6 – Exemple d'association réflexive non hiérarchique.

Association réflexive hiérarchique

Définition 1.7 (Association réflexive hiérarchique)**.** Une association *réflexive hiérarchique* est une association hiérarchique reliant des occurrences de la même entité.

Exemple 1.8. Dans la relation de la Figure 7, un salarié a pour supérieur direct un autre salarié.

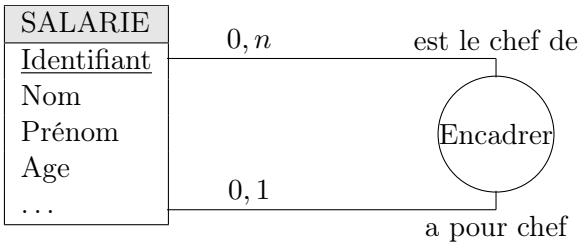


Figure 7 – Exemple d'association réflexive hiérarchique.

Associations de dimension 3 ou plus (ternaires ou plus)

Une association peut relier 3 entités (ou plus) autour d'une même association. Une telle association n'a que pour but d'alléger le MCD mais n'est jamais obligatoire. Il faut donc faire très attention car il est très rare de faire une association ternaire (de dimension 3) sans se tromper.

Exemple 1.9. Dans la Figure 8, pour connaître la pièce fabriquée par un ouvrier, il faut savoir sur quelle machine il a travaillé.

Cette association ternaire n'est possible que si les trois entités sont *indissociables*. C'est-à-dire qu'il n'est pas possible d'utiliser cette relation pour définir "*quel ouvrier travaille sur une machine ?*". En effet, dans cet exemple, tant qu'il n'y a de fabrication d'une pièce par un ouvrier (matérialisée à gauche par une relation et à droite par une entité), il n'est pas possible de faire le lien entre machine et ouvrier.

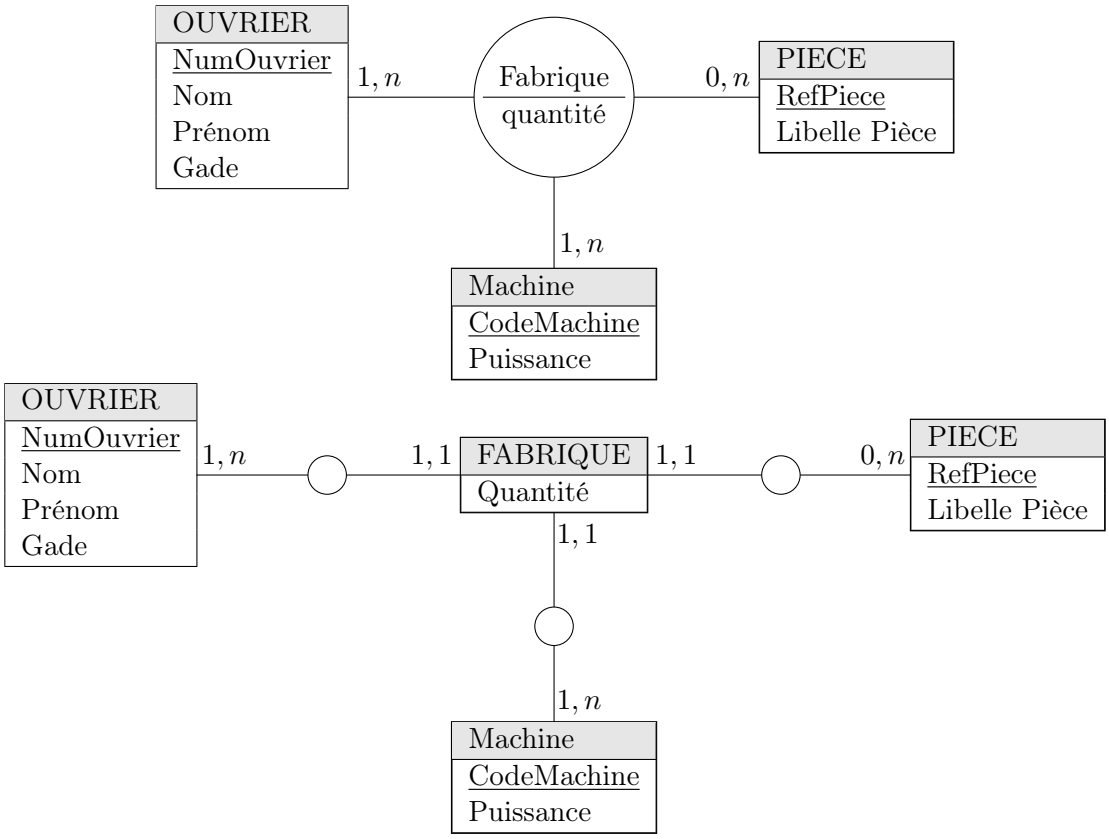


Figure 8 – Exemple de relation ternaire qui sont équivalentes.

2 Gestion des données redondantes

2.1 Rappels

Les entités contiennent des données, chaque ligne d'information est appelée un **tuple**. Une **clé** d'une relation est un ensemble minimal d'attributs dont chaque valeur détermine un tuple unique dans la relation. Une **référence** ou **clé étrangère** est un attribut ou un groupe d'attributs dont les valeurs sont celles d'une clé d'une autre relation. La *clé étrangère* permet de trouver des données dans une entité à partir d'une information extraite d'une autre entité.

Exemple 2.1. À un client correspond un numéro de client. Pour chaque facture, il est indispensable d'avoir l'information indiquant à quel client elle correspond.

— Clé primaire de la table client : numéro de client.

— Clé étrangère dans la table facture : numéro de client.

Remarque 2.1. Les données basées sur les clés étrangères sont redondantes mais indispensables pour faire le lien entre les différentes entités.

Hormis dans le cas de clés étrangères, les données peuvent être dupliquées, mais cela pose des problèmes de cohérence. En effet, si une information est présente dans plusieurs entités, en cas de mise à jour ou de suppression de l'information, il faut être certain que la donnée a été modifiée partout où elle est présente.

Exemple 2.2. Cet exemple a pour but de mettre en évidence les anomalies qu'il est possible de rencontrer et qu'il est important d'éviter. Soit la table COMMANDE décrivant les commandes et contenant

les champs et les données suivants.

COMMANDE

N ° piece	Qté	Nom fournisseur	Adresse fournisseur	N° com	N° Client
101	10	DURAND	10, Rue des gras 63000 Clermont-Ferrand	C54	15
102	20	DUPONT	86 rue de la république 03200 Vichy	C54	15
101	30	BRUNEAU	26 rue des Dômes 03200 Vichy	C17	18
103	10	DURAND	10, Rue des gras 63000 Clermont-Ferrand	C574	85

Cette table possède les anomalies suivantes lors des actions suivantes :

Modification : pour mettre à jour l'adresse d'un fournisseur, il faut le faire pour toutes les occurrences de la table concernant ce fournisseur.

Insertion : pour introduire le nom et l'adresse d'un fournisseur, il faut également fournir une valeur pour chacun des attributs « nopiece » et « quantité » ou introduire des valeurs nulles (ce qui pose d'autres problèmes).

Suppression : la suppression de la pièce n° l02 fait perdre toute information concernant le fournisseur DUPONT.

Ces différents cas peuvent conduire à des problèmes d'intégrité des données, par exemple, un même fournisseur se retrouve référencé avec plusieurs adresses, quelle est la bonne ? Ou encore la perte de données.

Ces problèmes dans la relation COMMANDE viennent du fait que la relation contient des attributs qui ne sont pas caractéristiques d'une même entité. Ils concernent à la fois la commande et le fournisseur. La solution pour pallier à ces problèmes est de décomposer cette entité en deux entités distinctes. Il est souvent relativement compliqué de trouver la bonne décomposition. Les règles à respecter pour réaliser un modèle sans redondances sont appelées ***Formes Normales (FN).***

Objectif des formes normales : éviter les redondances, pour éviter des incohérences, et l'existence de valeurs nulles.

2.2 Formes normales

La normalisation peut se faire au niveau conceptuel (pas obligatoire mais fortement conseillé). Sinon elle doit être faite au niveau logique. Il existe plusieurs règles de normalisation. Seules les 4 premières sont présentées dans ce document. Le but d'utiliser ces principes de normalisation est d'obtenir des tables organisées, sans redondance de données et sans structure répétitive. Dans certains cas, par exemple pour une table de trace qui enregistre les différentes actions faites, il est possible de ne pas respecter la norme, le modèle est dit "*dégradé*".

Pour faciliter l'explication des FN quelques notions sur les dépendances fonctionnelles sont introduites.

Définition 2.1 (Dépendance fonctionnelle)**.** Un groupe d'attributs Y dépend fonctionnellement d'un groupe d'attributs X si, étant donnée une valeur de X, il lui correspond une valeur unique de Y. Cette dépendance est notée $X \to Y$.

C'est à dire des valeurs identiques de X impliquent des valeurs identiques de Y, ce qui est aussi équivalent à si connaissant une occurrence de X il n'est pas possible de lui associer qu'une seule occurrence de Y.

Définition 2.2 (Dépendance fonctionnelle élémentaire)**.** Une dépendance fonctionnelle $X \to A$ est *élémentaire* si :

1. A est un attribut unique.
2. A n'appartient pas à X.
3. il n'existe pas X' inclus au sens strict dans X tel que $X' \to A$.

Une dépendance fonctionnelle est élémentaire si la cible est un attribut unique et si la source ne comporte pas d'attributs superflus. Ceci n'a du sens que lorsqu'il y a plusieurs attributs sur la partie de gauche de la dépendance fonctionnelle.

Définition 2.3 (Dépendance fonctionnelle directe)**.** $X \to A$ est une dépendance fonctionnelle *directe* s'il n'existe aucun ensemble d'attributs Y tel que $X \to Y$ et $Y \to A$.

C'est à dire la dépendance entre X et A ne peut être obtenue par *transitivité*.

2.3 1FN - Première forme normale

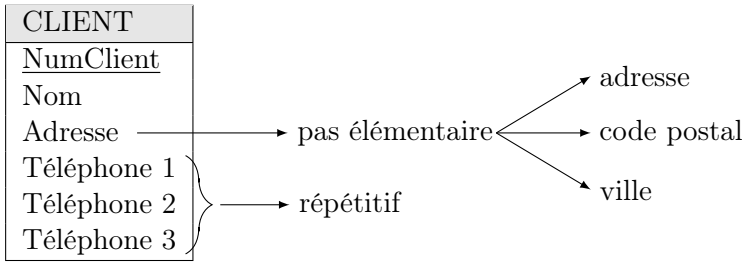
L'objectif de la première forme normale est d'éviter les répétitions dans les données et d'éviter les attributs qui contiennent plusieurs informations pour faciliter leurs gestions.

Définition 2.4 (1FN)**.** Une relation (une entité) est en 1FN si et seulement si tout attribut contient une valeur atomique.

C'est-à-dire :

1. Tous les attributs sont élémentaires (ou atomiques). C'est-à-dire qu'un attribut contient une seule information.
2. Elle ne contienne pas de structures répétitives.

Exemple 2.3. La relation CLIENT n'est pas en 1FN car elle n'a pas de clé, l'adresse n'est pas un attribut élémentaire et les téléphones sont répétés.



Exercice 2.1. Est-ce que la relation **COMMANDE V0** ci-dessous est en 1FN ?

COMMANDE V0

Nom fournisseur	Adresse fournisseur	N° com	N° client	Date com
DURAND	10, Rue des gras 63000 Clermont	C54	15	12/01/2016
DUPONT	86 rue de la république 03200 Vichy	C54	15	10/05/2016
BRUNEAU	26 rue des Dômes 03200 Vichy	C17	18	01/08/2016
DURAND	10, Rue des gras 63000 Clermont	C574	85	30/12/2015

Écrire la relation **COMMANDE V1** avec les données de COMMANDE V0 qui soit en 1FN.

2.4 2FN - Deuxième forme normale

Le but de la deuxième forme normale est d'avoir les tuples dans les relations uniques en introduisant des clés et de s'assurer que les données dépendent de la clé primaire.

Définition 2.5 (2FN)**.** Une relation est en 2FN si :

1. Elle respecte la 1FN.
2. Il existe un groupe d'attributs constituant la clé (et un groupe d'attributs non clé).
3. Tout attribut non clé est en dépendance fonctionnelle directe avec **l'ensemble** du groupe définissant la clé.

Exercice 2.2. Expliquer pourquoi la relation **COMMANDE V1** n'est pas en 2FN.

2.5 3FN - Troisième forme normale

L'objectif de la troisième forme normale est que tous les attributs d'une relation dépendent uniquement de la clé.

Définition 2.6 (3FN)**.** Une relation est en 3FN si :

1. Elle respecte la 2FN.
2. Tout attribut n'appartenant pas à la clé n'est pas en dépendance fonctionnelle directe avec un ensemble d'attributs non clé (un ensemble d'attributs qui ne constitue pas une clé candidate pour la relation).

C'est-à-dire que tout attribut non clé ne dépend pas d'un ou plusieurs attributs ne participant pas à la clé.

Exercice 2.3. Pourquoi la troisième forme normale n'est pas respectée dans la relation **COMMANDE V3** et proposer une solution pour que la base de données soit en 3FN ?

COMMANDE V3

Nom four	Adresse fournisseur	CP four	Ville four	N° com	N° client	Date com
DURAND	10, Rue des gras	63000	Clermont	C54	15	12/01/2016
DUPONT	86 rue de la république	03200	Vichy	C54	15	10/05/2016
BRUNEAU	26 rue des Dômes	03200	Vichy	C17	18	01/08/2016
DURAND	10, Rue des gras	63000	Clermont	C574	85	30/12/2015

2.6 4FN - Quatrième Forme Normale

Le but de la quatrième forme normale est de s'assurer que les attributs de la clé ne sont pas liés entre eux.

Définition 2.7 (4FN)**.** Une relation est en 4FN si :

1. Elle respecte la 3FN.
2. Les seules dépendances fonctionnelles élémentaires sont celles dans lesquelles une clé détermine un attribut non-clé.

Dans le cas d'une clé composée de plusieurs attributs, il ne doit pas y avoir de dépendance fonctionnelle à l'intérieur de la clé.

Exercice 2.4. Pourquoi la relation **COMMANDE V4** n'est pas en 4FN ? Proposer une solution pour que la relation soit en 4FN.

COMMANDE V4 :

# Nom four	N° com	N° client	Date com
DURAND	C54	15	12/01/2016
DUPONT	C54	15	10/05/2016
BRUNEAU	C17	18	01/08/2016
DURAND	C574	85	30/12/2015

2.7 Du MCD au modèle relationnel

Rappel sur le modèle relationnel

Le modèle relationnel est un modèle logique de données, correspondant à l'organisation des données dans les bases de données relationnelles. Le modèle relationnel reste le plus répandu actuellement (Oracle, SQLServer, MySQL, PostGreSQL,...) mais il existe d'autres organisations : hiérarchiques, réseau, objet.

Un modèle relationnel est composé de relations caractérisées par des attributs (ce qui correspond à une table et à ces colonnes), noté comme suit : < NOM DE LA RELATION > (attribut$_1$, attribut$_2$, ..., attribut$_n$).

Le NOM de la relation est par convention en MAJUSCULES. *L'identifiant* d'une relation, appelé « clé primaire » est composé d'un ou plusieurs attributs. La « clé étrangère » est un attribut de la relation faisant référence à la clé primaire d'une autre relation. Il existe plusieurs manières de noter les clés primaires et étrangères, la **première** est choisie dans ce cours :

1. Souligner les attributs de clé primaire et faire précéder les clés étrangères du symbole #
   Exemple : COMMANDE (numérocommande, datecommande, # numéroclient)
2. Souligner d'un trait continu les attributs de clé primaire et en pointillés les clés étrangères
   Exemple : COMMANDE (numérocommande, datecommande, numéroclient)
3. Noter sous la notation de la relation, les éléments constituant les clés primaires et étrangères.

Exemple 2.4. COMMANDE (numérocommande, datecommande, # numéroclient)

— Clé primaire : numérocommande

— Clé étrangère : numéroclient référençant CLIENT

Transformation du MCD au MLD

Afin de faciliter le passage du MCD au MLD il existe trois règles de transformation en fonction des cardinalités afin de créer un MLD correspondant au MCD de départ.

Règle 1 **:** Toute entité devient une relation ayant pour clé primaire son identifiant. Chaque propriété devient un attribut. Les espaces pouvant être contenus dans les propriétés sont remplacés par le symbole souligné « _ ». Le nom de l'attribut est modifié pour qu'il reste explicite sans accents, ni mots inutiles.

Exemple 2.5. Dans la Figure 9, l'entité COMMANDE avec comme identifiant NumCommande devient la relation COMMANDE avec comme clef primaire NumCommande.

Figure 9 – Exemple de l'application de la règle 1.

Règle 2 **:** Toute association hiérarchique (de cardinalité $[1, n]$) devient une clé étrangère. La clé primaire correspondante à l'entité père (côté n) devient alors une clé étrangère dans l'entité fils (côté 1) associée.

Exemple 2.6. Dans la Figure 10, l'association hiérarchique CLIENT impose que le champs NumClient soit une clé étrangère dans la relation COMMANDE.

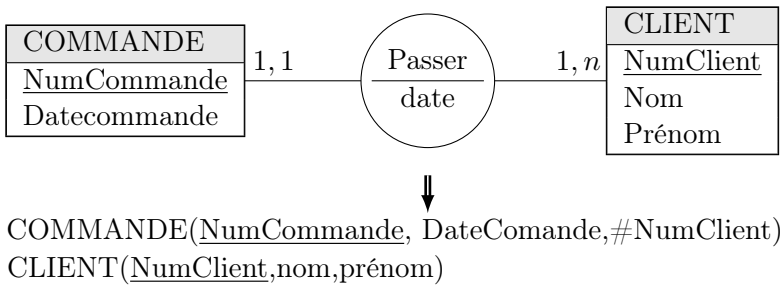


Figure 10 – Exemple d'application de la règle 2.

Règle 3 **:** Toute association non hiérarchique (de type $[n, n]$ ou de dimension > 2) devient une relation. La clé primaire est formée par la concaténation (juxtaposition) de l'ensemble des identifiants des entités reliées. Toutes les propriétés éventuelles existantes dans la relation du MCD deviennent des attributs qui ne peuvent pas faire partie de la clé dans le modèle relationnel.

Exemple 2.7. Dans la Figure 11, il faut créer 3 relations car les valeurs des cardinalités de l'entité COMMANDE et de l'entité ARTICLE sont n et n. Il faut alors créer une clef primaire dans la relation CONCERNER composé des deux clés primaires des relations COMMANDE et ARTICLE. Par construction ces champs sont aussi des clés étrangères.

Remarque 2.2. Il est souvent préférable de changer le nom de la relation ainsi générée pour éviter des doublons. Au lieu de CONCERNER, il faudrait mettre « RELATION_COMMANDE_ARTICLE », « COMMANDE_ARTICLE » ou bien « LIEN_COMMANDE_ARTICLE ».

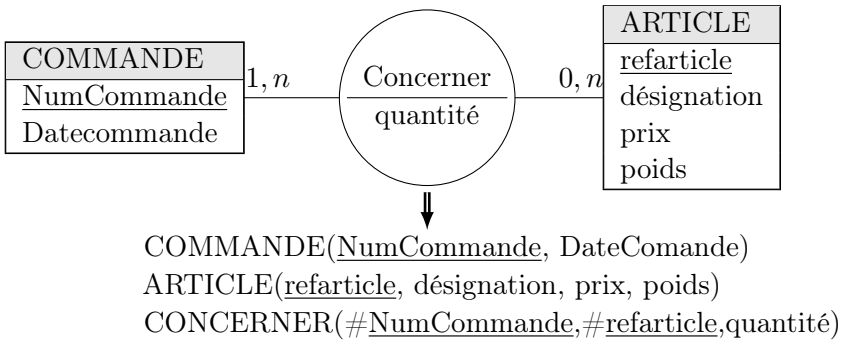


Figure 11 – Exemple d'application de la règle 3.

Associations réflexives

1. Les règles de transformation s'appliquent aussi aux associations réflexives **non hiérarchiques** comme les autres associations.

   Exemple 2.8. Dans la Figure 12, une molécule entre dans la composition de 0 ou plusieurs autres molécules, une ou plusieurs fois (exemple : 1 molécule d'amidon = plusieurs molécules de glucose).

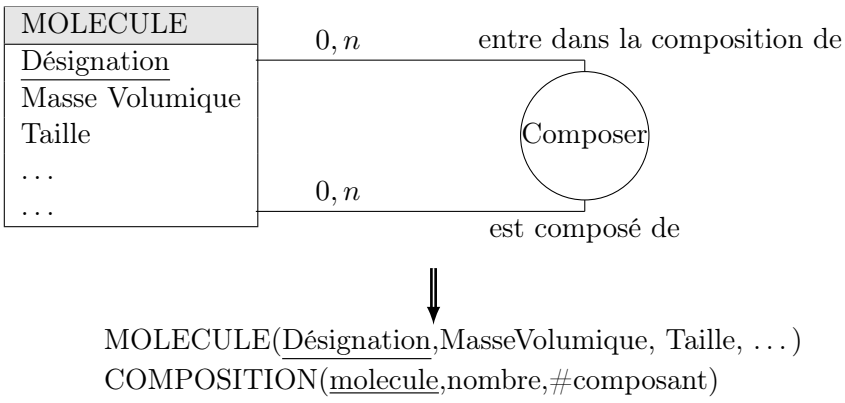


Figure 12 – Exemple de transformation d'une association réflexive non hiérarchique, où les attributs suivants sont égaux : composant = Désignation.

   Il est donc possible d'appliquer la règle 1 : l'identifiant de l'entité devient clé primaire. Il est aussi possible d'appliquer la règle 3 : l'identifiant de l'entité MOLECULE (côté composant) rentre comme attribut dans une autre relation. Il est ainsi possible d'indiquer le nombre de molécules entrant dans la composition.

2. Les associations réflexives suivent les règles 1, 2 et 3 selon les cardinalités mais le problème est que, dans une même relation, une propriété se retrouve 2 fois. Il faut alors donner des noms différents et significatifs aux attributs concernés.

   Exemple 2.9. Dans la Figure 13, la relation SALARIE possède une clé primaire qui correspond à l'identifiant des salariés. Elle possède également une clé étrangère nommée identifiant_chef qui prend comme valeurs celles de la clé primaire afin d'assurer le lien réflexive qui existe entre ces deux relations.

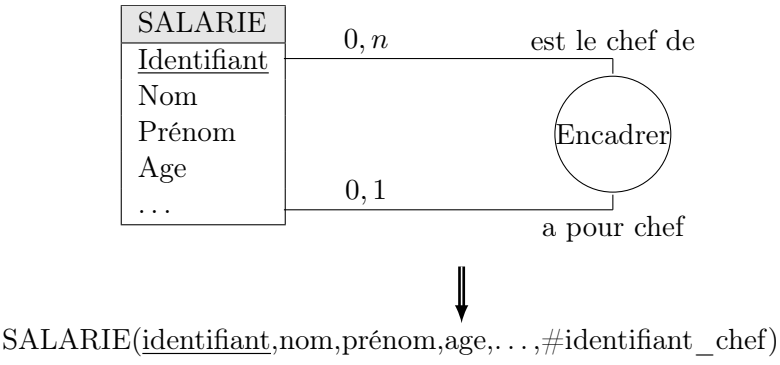
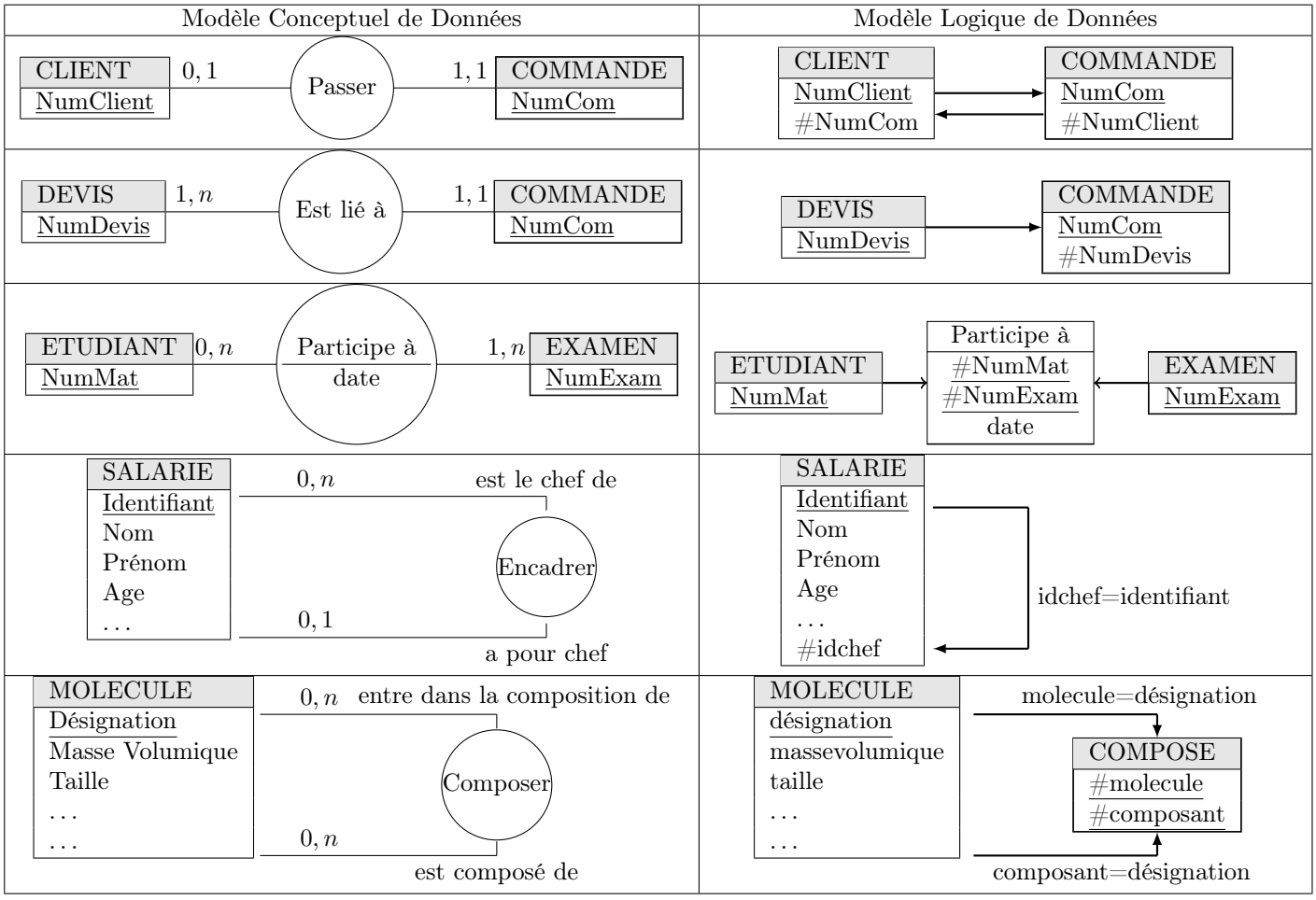


Figure 13 – Exemple de transformation d'une association réflexive, où les attributs suivants sont égaux identifiant = identifiant_chef.

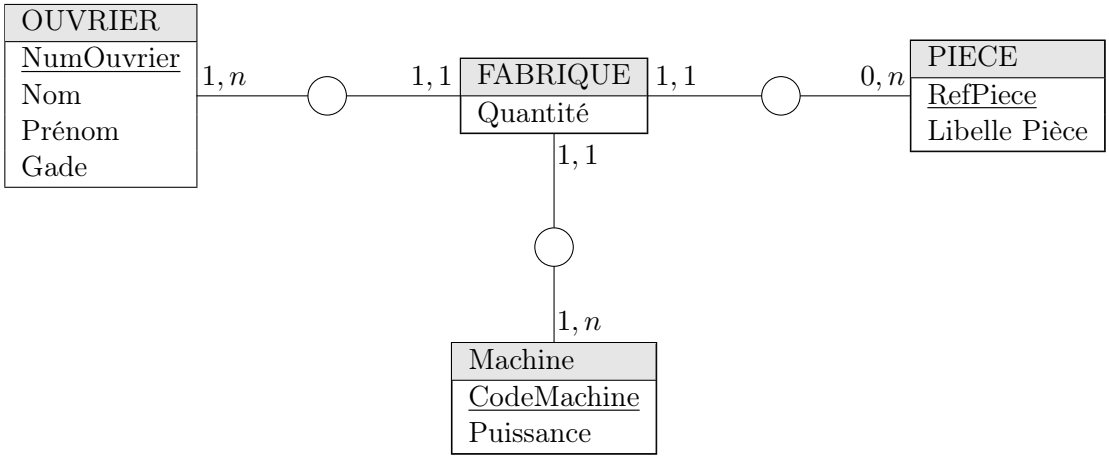
Passage du MCD au MLD



Remarque 2.3. Une association peut être porteuse d'informations. Naturellement, ce n'est valable que pour une association de cardinalités x, n - x, n car c'est la seule qui, en passant du MCD au MLD génère une table qui sera alors utilisée pour stocker les informations relatives à la relation.

Remarque 2.4. Dans la Figure 14, le résultat pour le passage du MCD au MLD est donné pour l'association ternaire indiquée dans la Figure 8. Les relations ternaires ne sont pas obligatoires au niveau du MCD mais cela aide pour la construction du MLD, cela ne change pas le résultat du MLD.

MCD



MLD

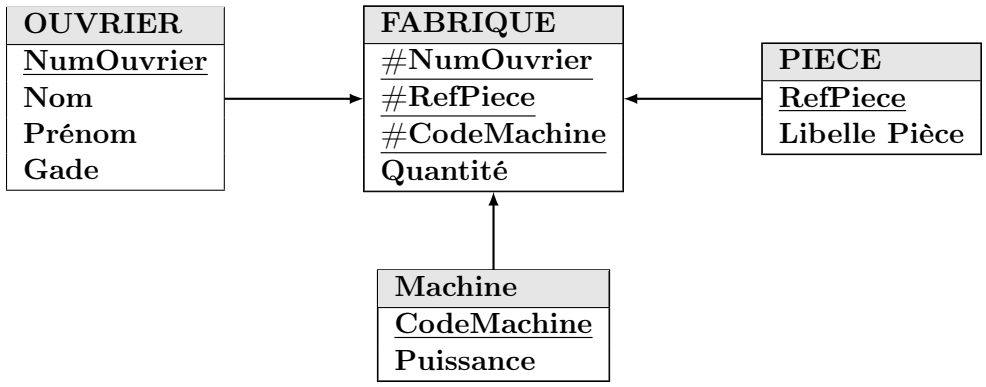


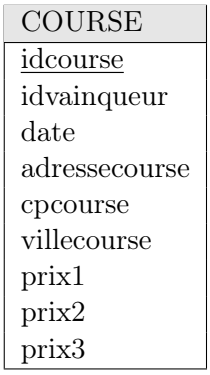
Figure 14 – Exemple de transformation d'association ternaire.

Exercice 2.5. Pour les 4 MCD ci-dessous :

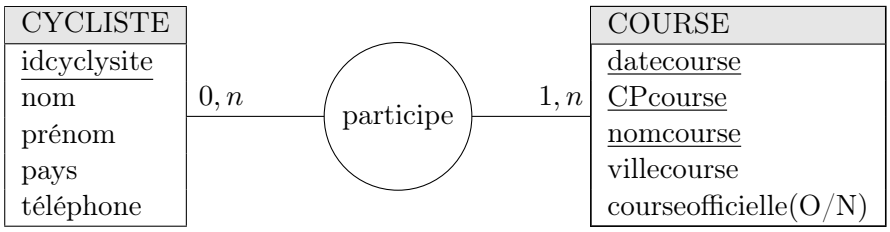
— Indiquer quelle FN respecte le MCD.

— Indiquer les modifications à effectuer pour que le MCD respecte la FN supérieure.

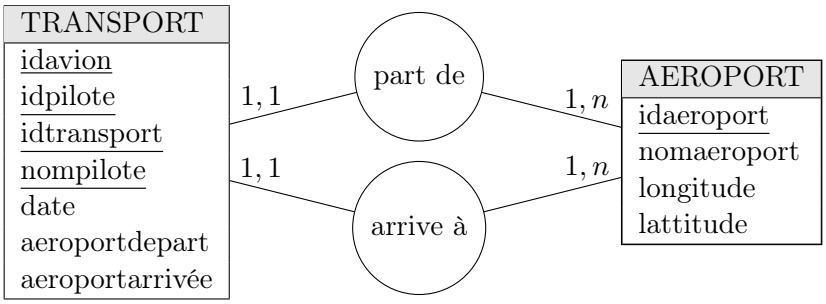
MCD 1



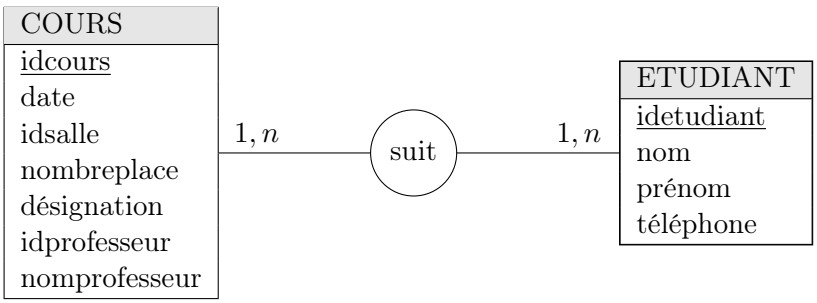
MCD 2

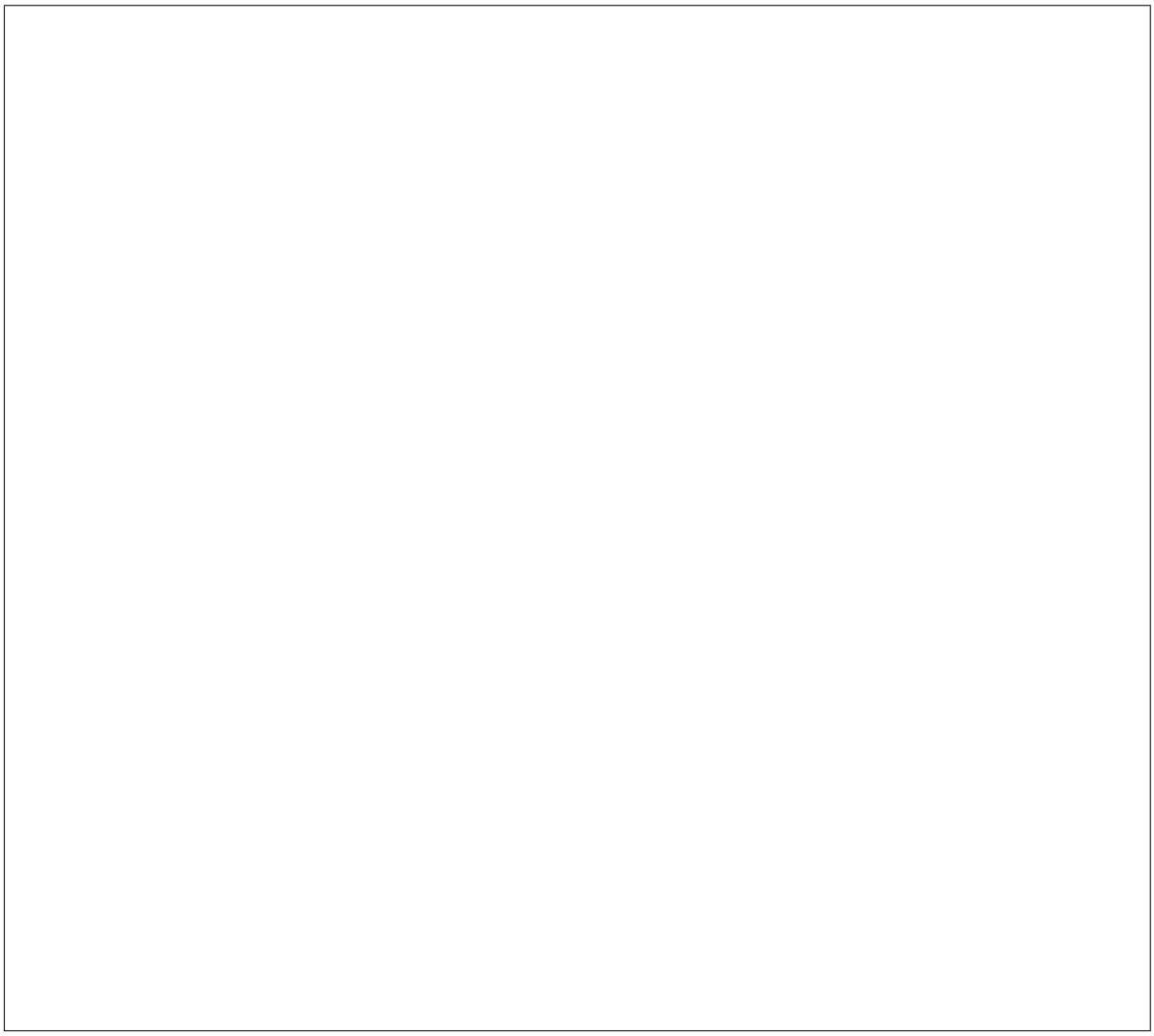

CYCLISTE
idcyclysite
nom
prénom
pays
téléphone
0, n
participe
1, n
COURSE
datecourse
CPcourse
nomcourse
villecourse
courseofficielle(O/N)

MCD 3


TRANSPORT
idavion
idpilote
idtransport
nompilote
date
aeroportdepart
aeroportarrivée
1, 1
part de
1, n
AEROPORT
idaeroport
nomaeroport
longitude
lattitude
1, 1
arrive à
1, n

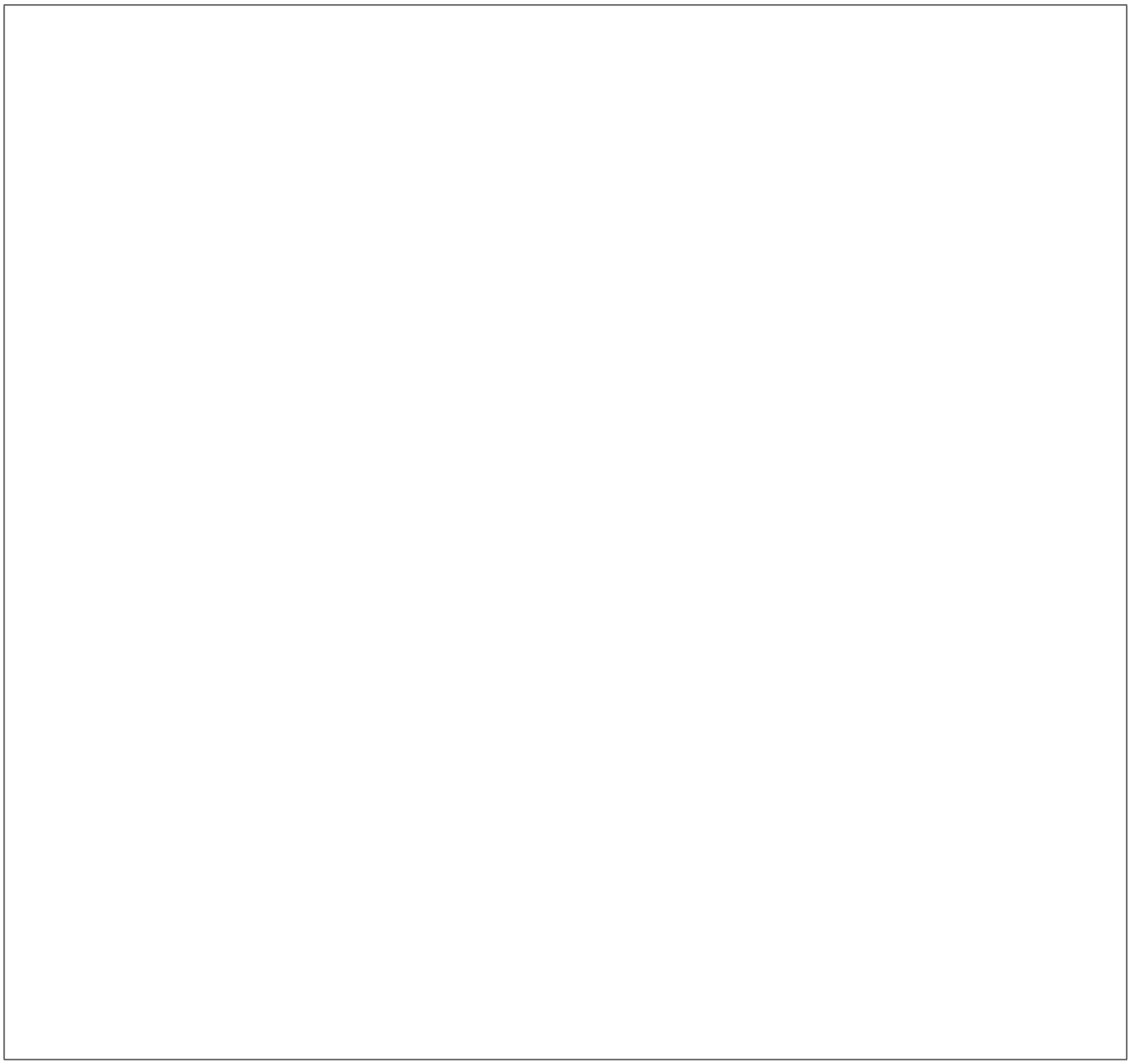
MCD 4


COURS
idcours
date
idsalle
nombreplace
désignation
idprofesseur
nomprofesseur
1, n
suit
1, n
ETUDIANT
idetudiant
nom
prénom
téléphone



Exercice 2.6 (Arbre généalogique)**.** Pour s'intéresser aux ancêtres et regrouper différentes informations il faut pouvoir garder une trace des membres d'une famille (passé, présent, futur). Pour chaque individu, il faut connaître les informations suivantes : nom, prénom, date et lieu de naissance, numéro de sécurité social mais aussi ses différentes coordonnées (adresse postal, mail, téléphone, ...). Il faut aussi pouvoir connaître l'historique des coordonnées des membres de la famille.

1. Réaliser le Modèle Conceptuel de Données correspondant.
2. A partir du MCD obtenu, réaliser le Modèle Logique de Données.



3 Fonctionnalités avancées

Il est important de connaître comment mettre en œuvre le MLD obtenu lors de la conception du MLD. Pour cela il faut savoir comment respecter les types des données qui seront stockées, être capable de créer les tables avec les contraintes associées. Une fois le système mis en place, afin de faciliter l'accès au données aux utilisateurs les notions de vue, synonyme et séquence vont s'avérer utiles. Enfin il est aussi important de savoir gérer les utilisateurs et leurs donner les droits appropriés à leurs rôles.

3.1 Types de données

Les principaux types de données d'une base Oracle sont les suivants.

Type de données	**Description**	**Commentaire**
BFILE	Données binaires, enregistrées dans un fichier externe.	Jusqu'à 4 Go.
BLOB	Données **binaires**, conservées dans la base de données.	Jusqu'à 4 Go.
CLOB	Données de jeux de caractères (single-byte).	Jusqu'à 4 Go.
LONG	Données de jeux de caractères de longueur variable (forme ancienne de CLOB, mais toujours utile car elle peut être mise en œuvre directement).	Jusqu'à 2 Go.
CHAR (n)	Chaîne de caractères de longueur **fixe** *n*. A n'utiliser que lorsque la taille des données est constante (exemple : numéro de sécurité sociale).	Jusqu'à 2Go.
VARCHAR2(n)	Chaîne de caractères de longueur **variable** *n*. C'est-à-dire que si ce champ est *null*, la taille sur disque sera nulle. Ce qui prend moins de place. Par contre, lors d'un `update`, ce champ va se remplir et il est possible que les données ne puissent tenir dans le bloc de données Oracle donc elles seront écrites dans un autre bloc. Il y aura un chaînage de bloc, ce qui est pénalisant au niveau des performances.	4000 caractères maximum.
DATE	Date et heure.	Stockées en nombre de secondes depuis la date de référence 01/01/1970.
NUMBER (v, n)	Données numériques de longueur variable, dont les positions avant la virgule sont définies avec v et celles après la virgule, avec n.	Il est possible d'utiliser number sans v et n.

Pseudo-colonnes : Les pseudo-colonnes `rowid` et `rownum` ne sont pas des colonnes d'une table, mais des valeurs qu'il est possible d'appeler dans une commande.

Exemple 3.1. Les pseudo-colonnes `rowid` et `rownum` s'utilisent comme ceci :

— `SELECT rowid, désignation FROM voiture;`
donne l'identifiant unique de la ligne dans la base Oracle.

— `UPDATE livre SET id_livre=rownum;`
remplace la valeur de `id_livre` par le numéro de ligne (qui est forcément unique).

3.2 Manipulation de table

La table est l'objet principal permettant de stocker des données.

Remarque 3.1. Il ne faut jamais mettre les noms de table entre guillemets. Cela est autorisé mais complexifie énormément les commandes SQL.

Exemple 3.2 (Création et suppression d'une table)**.**

```
CREATE TABLE mon_schema.ma_table(
 DBINC_KEY      NUMBER NOT NULL,
 CREATE_TIME    DATE,
 TS             NUMBER NOT NULL,
 CLONE_FNAME    VARCHAR2(100),
 DROP_SCN       NUMBER,
 DROP_TIME      DATE );

DROP TABLE mon_schema.ma_table [ CASCADE CONSTRAINTS ];
```

L'option "`cascade constraint`" permet supprimer en cascade lorsqu'il y a une contrainte référentielle, par contre toutes les données seront supprimées.

La commande `ALTER TABLE` permet de modifier une contrainte sur une table déjà créée.

```
ALTER TABLE RMAN.DF ADD (NEW_COLUMN  VARCHAR2(10));
```

3.3 Contraintes non référentielles

Les contraintes non référentielles permettent de préciser les données d'un attribut. Les contraintes peuvent être gérées à la création de la table ou lors de modifications, par exemple :

```
CREATE TABLE matable (nombre number PRIMARY KEY);
ALTER TABLE matable ADD CONSTRAINT check_nombre CHECK (nombre < 10);
ALTER TABLE matable DROP CONSTRAINT check_nombre;
```

Syntaxe : `CREATE TABLE ma_table`
`<nom_colonne 1> <type de donnée> [contrainte], <nom_colonne 2> <type de donnée> [contrainte]` : contrainte pouvant concerner plusieurs colonnes, comme par exemple : `primary key`. Sinon les contraintes possibles sont les suivantes :

— `DEFAULT` : indique la valeur par défaut à mettre dans la colonne en cas d'insertion, sans précision de valeur pour la colonne concernée.

— `NULL/NOT NULL` : forcer le fait qu'une donnée puisse être nulle ou pas.

— `CHECK` : effectue une vérification de la donnée (par exemple : <10).

— `UNIQUE` : force l'unicité de la valeur.

— `PRIMARY KEY` : indique la clé primaire de la table. Une clé primaire est *non nulle et unique.*

Exemple 3.3.

```
CREATE TABLE test(
 NUCL varchar2(6) CONSTRAINT loue_pk1 PRIMARY KEY,
 NUMAT varchar2(6) CONSTRAINT loue_numat NOT NULL,
 NUEX varchar2(3) CONSTRAINT loue_nuex CHECK (nuex BETWEEN 1 AND 20),
 NUCONFIRM char(3) CONSTRAINT loue_confirm CHECK (nuconfirm IN('OUI','NON')),
 DEBLOC DATE CONSTRAINT loue_debloc CHECK (debloc > '15/01/2002'),
 JLOCPREV NUMBER(2) DEFAULT 1,
 CONSTRAINT new_cons CHECK (NUBOUT >0);
```

Il est préférable de nommer la contrainte. En effet, lorsqu'un ordre SQL engendre une violation de contrainte, un message d'erreur apparaît en indiquant la contrainte concernée. Si le nom est clairement défini, l'erreur est plus facilement compréhensible.

Il est possible de supprimer, invalider, ou modifier une contrainte en utilisant la commande `ALTER TABLE`.

```
ALTER TABLE <nom_table> DROP CONSTRAINT <nom_contrainte> ;
ALTER TABLE <nom_table> ADD nom_colonne type_donnees
ALTER TABLE <nom_table> ADD CONSTRAINT <nom_contrainte> PRIMARY KEY (champs1,champ2) ;
ALTER TABLE <nom_table> MODIFY nuex CHAR(3) CHECK (nuex BETWEEN 1 AND 50) ;
ALTER TABLE <nom_table> DISABLE CONSTRAINT <nom_contrainte> ;
ALTER TABLE <nom_table> ENABLE CONSTRAINT <nom_contrainte> [EXCEPTIONS INTO ...] ;
ALTER TABLE <nom_table> DROP unique (colonne);
```

A la mise en place d'une contrainte, il est possible que certaines lignes de la table ne respectent pas cette contrainte. Dans ce cas, un message d'erreur apparaît. Avec la clause `EXCEPTION`, il est possible de stocker dans une table (à définir), l'ensemble des lignes (`rowid`) posant problème.

Exemple 3.4. `CREATE TABLE new (nu number CHECK (nu in (1,2,3,4,5)));`

3.4 Contraintes référentielles

Une contrainte *référentielle* permet de matérialiser, au niveau de la base de données, les relations indiquées dans le MCD. Pour cela, il faut que la table mère ait une clé primaire définie, et que la table fille fasse référence à cette clé primaire. Il ne peut y avoir d'exception à une contrainte référentielle. Les valeurs dans la table colonne de la table fille doivent forcément exister dans la table mère.

`CONSTRAINT <nom> FOREIGN KEY (colonne) REFERENCES nom_table(nom colonne) [ON DELETE CASCADE]`

- `FOREIGN KEY` : indique que c'est une contrainte référentielle. Cette clause n'est pas obligatoire si utilisée directement lors de la définition de la colonne mais il est beaucoup plus propre de l'utiliser.
- `REFERENCES` : définit une contrainte d'intégrité référentielle par rapport à une clé unique ou primaire.
- `ON DELETE CASCADE` : Option permettant de conserver l'intégrité référentielle en supprimant automatiquement les enregistrements d'une table fille dépendant des enregistrements supprimés par un ordre `DELETE`.

Exemple 3.5.

```
CREATE TABLE Histo (
 numat char(6) CONSTRAINT ref_mat REFERENCES materiel,
 nubout char (6) REFERENCES boutique(nubout),
```

```
nucl char (6) REFERENCES client,
datret date,
jlocreel number(2),
PRIMARY KEY (numat, nuex, nubout, nucl)
CONSTRAINT fk_client FOREIGN KEY (nucl) REFERENCES client(nucl));
```

Exercice 3.1. Soit le MLD de la Figure 15 et le modèle relationnel correspondant :
INDIDIVU(idind, nom, prenom, datenaissance, lieunaissance)
COORDONNEES (idcoord, valeur, # idind, # idtype)
TYPE(idtype, designation)
PARENT (#idparent, #idenfant)

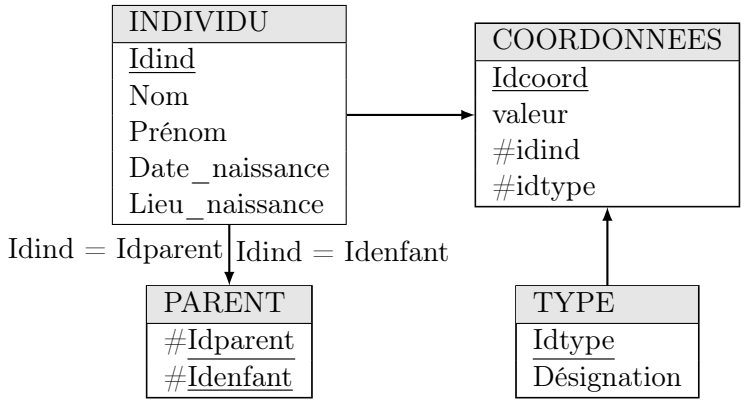


Figure 15 – MLD de gestion d'arbre généalogique.

à partir du MLD ci-dessus, lister les contraintes référentielles puis écrire les commandes de création des tables et des contraintes référentielles, dans un premier temps, sans commande ALTER et dans un second temps avec les commandes ALTER.

sans commandes ALTER :

Avec commandes ALTER :

3.5 Vue

Une vue correspond à une requête stockée en base de données. Aucune donnée n'est stockée dans une vue. Une des utilités est de ne pas réécrire les requêtes souvent utilisées. Lors de l'exécution de la requête, le moteur Oracle remplace la vue par le code SQL correspondant.

— Création d'une vue : `CREATE VIEW OR REPLACE total_salaire AS SELECT sum(salaire) FROM salaire;`

— Interrogation d'une vue : `SELECT * FROM total_salaire;`

— Suppression d'une vue : `DROP VIEW total_salaire;`

Vue matérialisée

La vue matérialisée est l'équivalent d'une vue, mais les données sont stockées. Ceci implique que les données soient rafraîchies, soit automatiquement, soit manuellement. Les vues matérialisées sont utilisées lorsque l'obtention des données nécessite un temps de traitement important et que les données sont souvent accédées. C'est souvent le cas de données statistiques.

```
CREATE MATERIALIZED VIEW mon_schema.SCRIBE_NO_CATPER
 REFRESH COMPLETE
 START WITH TO_DATE('12/09/2016 12:00:00','DD/MM/YYYY hh24:mi:ss')
 NEXT TRUNC(sysdate + 1)
 WITH PRIMARY KEY
 AS
  SELECT   DISTINCT code_objet, total
  FROM   vente
  WHERE   date > trunc(sysdate-1);

DROP MATERIALIZED VIEW mon_schema.SCRIBE_NO_CATPER;
```

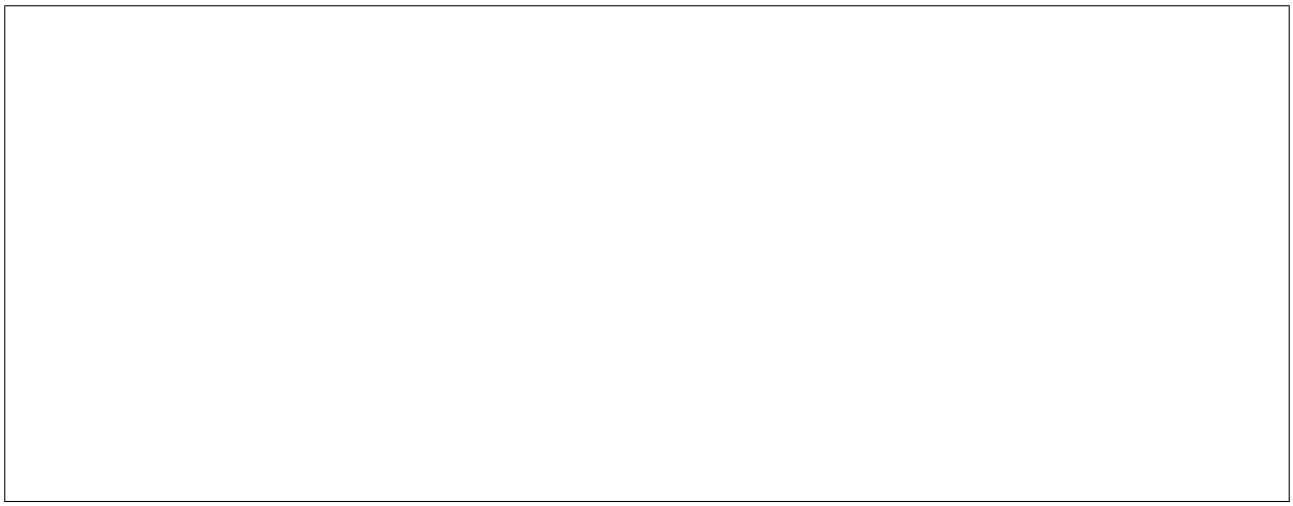
Exercice 3.2. Créer la vue "PARENTS" permettant de voir les parents d'une personne donnée. Les données doivent s'afficher en lançant la requête suivante :

```
set lines 150
col parents for a80
SELECT * FROM parents WHERE idind=36;
```

Le résultat attendu est le suivant :

```
    IDIND parents
---------- --------------------------------------------------------------------
        36 Adam MERCIER est l'enfant de Alain MERCIER et Helene SOLLIER

1 row selected.
```



3.6 Synonyme

Un *synonyme* permet d'utiliser un objet en le nommant différemment. Imaginons qu'un développement soit déjà fait en utilisant une table qui s'appelle salaire. Une modification de la structure des données a été faite et la table a été renommée `SLR`. Dans ce cas, toutes les commandes SQL permettant d'accéder aux données échouent. En créant un synonyme `SALAIRE` pour la table `SLR`, les requêtes fonctionnent à nouveau.

```
CREATE SYNONYM salaire FOR slr;
DROP synonym DRH.salaire;
```

3.7 Séquence

Une séquence est un compteur qui s'incrémente dès qu'il est consulté. Dans les bases de données telles que MariaDB ou MySQL, il est possible de créer une table avec une colonne de type "*autoincrement*". Ainsi, à l'insertion d'une nouvelle ligne dans la table, la valeur qui est insérée dans cette colonne est une incrémentation de la valeur précédente. Cette colonne sert alors de clé primaire. Dans une base de données Oracle, ce type de colonne n'existe pas. Il existe un mécanisme plus complet, la **séquence**, qui permet d'avoir le même fonctionnement. Pour cela il faut créer une séquence. Lors de l'insertion d'une ligne dans la table, il faudra lire la valeur suivante de la séquence pour l'affecter à la colonne identifiant. Dans les bases de données Oracle à partir de la version 11, il est possible de créer une table dont une colonne est concernée par une clause "Generated as identity" afin de permettre la gestion d'une auto incrémentation. En réalité, derrière ce mécanisme, se cache la gestion de séquence.

Lors de la création d'une séquence il faut définir :

— `START WITH` : valeur de départ.

— `INCREMENT BY` : incrémentation, augmente par exemple de 1, 10, ou -1, -5.

— `MAXVALUE` : valeur maximale.

— `MINVALUE` : valeur minimale.

— `CYCLE/NOCYCLE` : cyclique ou non cyclique.

Exemple 3.6 (Utilisation d'une séquence)**.**

```
CREATE SEQUENCE ma_sequence START WITH 100 INCREMENT BY 1 NOCYCLE;
SELECT ma_sequence.NEXTVAL FROM dual;  -- valeur suivante de la séquence.
SELECT ma_sequence.CURRVAL FROM dual;  -- valeur courante de la séquence.
DROP SEQUENCE ma_sequence;
```

Pour définir une colonne en primary key :

```
INSERT INTO ma_table VALUES (ma_sequence.NEXTVAL,'champ1');
CREATE SEQUENCE ma_sequence START WITH 100 INCREMENT BY 1 NOCYCLE;
SELECT ma_sequence.NEXTVAL FROM dual; -- valeur suivante de la séquence.
SELECT ma_sequence.CURRVAL FROM dual; -- valeur courante de la séquence.
DROP SEQUENCE ma_sequence;
```

Pour définir une colonne en primary key :

```
INSERT INTO ma_table VALUES (ma_sequence.NEXTVAL,'champ1');
```

Exercice 3.3 (Séquence)**.** Créer la table `trace(id_trace number, trace varchar2(50))` dont la colonne `id_trace` est un numéro incrémenté automatiquement. Indiquer la commande permettant d'insérer une nouvelle ligne dans cette table de trace.

3.8 Utilisateur

Les utilisateurs (users) permettent d'établir une connexion au SGBDR, de plus, il faut avoir des privilèges d'administration sur la base pour pouvoir créer un utilisateur. La syntaxe minimale lors de la création d'un utilisateur est la suivante :

`create user <utilisateur> identified by <mdp>;`

Les options utilisables sont :

- `DEFAULT TABLESPACE` : espace de stockage utilisé par défaut lors des commandes de création de tables, indexes, ...
- `TEMPORARY TABLESPACE` : espace temporaire, utilisé pour faire les tris lors des tris de données.
- `QUOTA` : limitation de l'espace sur un ou plusieurs TABLESPACES.
- `PROFILE` : définition des limites d'utilisation des ressources, de gestion des mots de passe, ...
- `PASSWORD EXPIRE` : cette option force l'expiration du mot de passe et obligeant à en mettre un nouveau dès la première connexion.
- `ACCOUNT LOCK/UNLOCK` : option pour verrouiller, déverrouiller le compte (empêche la connexion mais ne gêne pas l'utilisation des données)

Exemple 3.7 (Création et suppression d'utilisateur)**.**

```
CREATE USER cnam
   IDENTIFIED BY mot_de_passe
   DEFAULT TABLESPACE tbs_cnam
   QUOTA 5M ON tbs_cnam;

CREATE USER cnam IDENTIFIED BY pwdcnam;
DROP USER cnam;
DROP USER cnam CASCADE;
```

Cette dernière commande supprime un utilisateur même s'il a des objets (tables, index, ...). Tout est supprimé dans ce cas. **Attention**, cela supprime aussi les lignes concernées dans les autres tables qui seraient liées par les contraintes référentielles.

3.9 Privilège

Les privilèges permettent d'autoriser ou de limiter les actions des comptes ou des rôles. C'est un élément important dans la gestion d'une base Oracle. Il est possible de donner des droits via l'utilisation de rôles. Il existe 2 types de privilèges :

- **privilège objet** permet d'attribuer un **droit sur un objet.**
  - `ALTER` : permet de modifier un objet.
  - `DELETE` : permet de supprimer des lignes.
  - `INDEX` : permet de créer un index.
  - `INSERT` : permet d'insérer des lignes.
  - `REFERENCES` : permet d'utiliser la table dans une contrainte de clé externe (forein key).
  - `SELECT` : permet de lire la table.
  - `UPDATE` : permet de mettre à jour la table.
  - `EXECUTE` : permet d'exécuter un code (procédure, fonction, ...).
  - `READ` : permet de lire un fichier présent dans un objet de type "directory".
  - `WRITE` : permet d'écrire dans un fichier présent dans un objet de type "directory".
  - `FLASHBACK` : permet d'utiliser la corbeille pour un objet, si la corbeille est activée
- les privilèges système permettent **d'autoriser une action**, ici seuls quelques exemples sont décrits.
  - `CREATE SESSION` : permet de se connecter à la base de données.
  - `CREATE TABLE` : permet de créer une table. Dans votre propre schéma, vous avez alors la possibilité de modifier sa définition et de la supprimer.
  - `CREATE ANY TABLE` : permet de créer une table dans un autre schéma.
  - `ALTER ANY TABLE` : permet de modifier la définition de n'importe quelle table.
  - `BACKUP ANY TABLE` : permet de faire un backup de n'importe quelle table.
  - `DELETE ANY TABLE` : permet de supprimer des lignes dans n'importe quelle table
  - `DROP ANY TABLE` : permet de supprimer n'importe quelle table.
  - `INSERT ANY TABLE` : permet d'insérer des lignes dans n'importe quelle table.
  - `LOCK ANY TABLE` : permet de verrouiller n'importe quelle table.
  - `FLASHBACK ANY TABLE` : permet d'utiliser la corbeille sur n'importe quelle table.
  - `UPDATE ANY TABLE` : permet de mettre à jour n'importe quelle table.
  - `CREATE TABLESPACE` : permet de créer un tablespace.
  - `ALTER TABLESPACE` : permet de modifier la définition d'un tablespace.
  - `EXP_FULL_DATABASE` : permet de faire un export full de la base.
  - `IMP_FULL_DATABASE` : permet de faire un import full dans la base.

Il est possible de donner un droit à un utilisateur défini, ou à l'ensemble des utilisateurs d'une base grâce au mot clé : `public`.

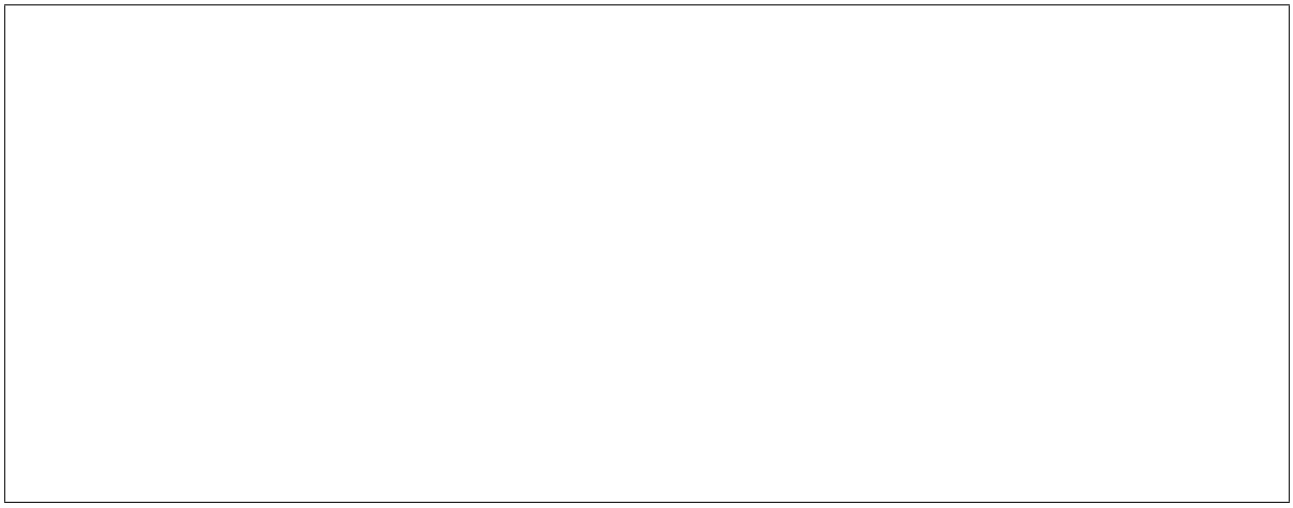
Attribuer et révoquer un privilège objet

```
GRANT SELECT ON ma_table TO public; -- autorise la lecture de ma_table à tout le monde.
REVOKE SELECT ON ma_table FROM public; -- supprime le droit de lire ma_table à tout le monde.
```

Attribuer et révoquer un privilège système

```
GRANT CREATE SESSION TO newuser; -- permet à "newuser" d'ouvrir une session.
GRANT dba FROM newuser; -- L'administrateur donne le rôle dba "newuser". C'est-à-dire
                        -- que tous les droits attribués au rôle dba sont donnés à
                        -- newuser.
```

Exercice 3.4 (Priviléges)**.** A partir des tables créées dans le TD, créer un second environnement (compte PALAFOUR2) qui va travailler sur les mêmes données que le compte principal (compte PALAFOUR).



4 Fonctions

Il existe plusieurs type de fonctions les fonctions mono-lignes, les fonctions de regroupement et les fonctions personnalisées

4.1 Fonctions mono-lignes

Les fonctions mono-lignes agissent sur chaque ligne indépendamment. Les fonctions peuvent s'imbriquer.

Fonctions numériques

NOM	DÉFINITION	EXEMPLE	RÉSULTAT
`ABS(x)`	Valeur absolue	`Select abs(-1.70) from dual;`	1.7
`CEIL(x)`	Plus petit entier sup ou égal à x	`Select ceil(8.044) from dual;`	9
`FLOOR(x)`	Partie entière de x	`Select floor(4.244) from dual;`	4
`SIGN(x)`	Signe de x (-1,0,+1)	`Select sign(-3) from dual;` `Select sign(0) from dual;`	-1 0
`MOD(x,y)`	Reste de la division de x par y	`Select mod(7,2) from dual;`	1
`POWER(x,y)`	x à la puissance y	`Select power(2,3) from dual;`	8
`ROUND(x,y)`	Arrondi à y décimal	`Select round(1250,-3) from dual;` `Select round(5.132,3) from dual;`	1000 5,13
`TRUNC(x,y)`	Tronque à y décimales	`Select trunc(7487.74177,2) from dual;`	7487,74

Fonctions de mises en forme

NOM	DEFINITION	EXEMPLE	RÉSULTAT
`UPPER/LOWER`	Majuscules/minuscules	`Select UPPER(hello') from dual;`	HELLO
`INITCAP(c1)`	Première lettre de chaque mot en majuscule	`Select INITCAP('comment ça va ?') from dual;`	Comment Ça Va ?
`CONCAT(c1,c2)`	Concaténation (équivalent à ||) Uniquement avec 2 arguments	`Select CONCAT('Bon','jour') from dual;` `Select 'bon' || 'jour' from dual;`	Bonjour Bonjour
`SUBSTR(c,n,m)`	Sous chaîne : m caractères à partir du n ième caractère	`Select SUBSTR('bonjour',3,2) from dual;`	nj
`LENGTH(c1)`	Longueur de c1	`Select length('bonjour') from dual;`	7
`INSTR(c1,c2,n,m)`	Position de la `m` ième fois que `c2` se trouve dans `c1` en commençant à la position `n`. Renvoie 0 si échec	`Select instr('Le SQL est un langage puissant','an',5,2) from dual;`	28

Fonctions de remplacement et de remplissage

NOM	DEFINITION	EXEMPLE	RÉSULTAT
LPAD(c1,n,c2)	Chaîne complétée par la gauche par c2 sur n caractères. Si n est inférieur à la taille de la chaîne, la chaîne est tronquée	Select LPAD('DURAND',8,'$ ') from dual;	$ $ DURAND
RPAD(c1,n,c2)	Chaîne complétée par la droite par c2	Select RPAD('DURAND',8,'$ ') from dual;	DURAND$ $
LTRIM(c1,c2)	Suppression des caractères de c1 (**de gauche à droite**) appartenant à l'ensemble c2 tant qu'un caractère de c1 fait partie de c2	Select LTRIM('DURAND','DA') from dual; Select ltrim('* * * * * * * TOTO','* ' FROM dual;	URAND TOTO
RTRIM(c1,c2)	Suppression des caractères de c1 (**de droite à gauche**) appartenant à l'ensemble c2 tant qu'un caractère de c1 fait partie de c2	Select RTRIM('DURAND','NAD') from dual; Select rtrim(machaine,' ') FROM ma_table;	DUR Supprime les blancs en fin de machaine
REPLACE(c1, c2,c3)	Remplacement des occurrences de c2 par c3 dans c1	Select Replace('DURAND','D','TRI') from dual;	TRIURANTRI

Fonctions dates

NOM	DEFINITION	EXEMPLE	RÉSULTAT
SYSDATE	Indique la date système	Select sysdate from dual	
LAST_DAY(d)	Dernier jour du mois de d	Select LAST_DAY('10-OCT-05') from dual; select LAST_DAY('10/10/05') from dual;	31/10/05 31/10/05
ADD_MONTHS(d,n)	Ajoute n mois à la date d	Select ADD_MONTHS(to_date('10/10/2005','DD/MM/YYYY'),3) from dual; Select add_months('31-JAN-16',1) from dual;	10/01/06 29-FEB-16
MONTHS_BETWEEN(d1,d2)	Nombre de mois entre d1 – d2	Select MONTHS_BETWEEN(to_date('1/1/16','DD/MM/YY'),to_date('10/10/16','DD/MM/YY')) from dual ;	-9.2903226
NEXT_DAY (d,jour)	Date du prochain jour suivant la date d. Le 6/08/16 est un dimanche et donc le 9/08/16 est un mardi.	Select NEXT_DAY(to_date('06/08/16','DD/MM/YY'), 'tuesday') from dual	09/08/16
ROUND (d,format)	Date d, arrondie au format	Select ROUND(to_date('20-AVR-05'),'MONTH') from dual;	01-MAI-05
TRUNC (d,format)	Date d, tronquée au format	Select TRUNC(to_date('20-AVR-05','DD-MON-YY'), 'MONTH') from dual;	01-AVR-05

Format de date Par défaut = DD-MON-YY ou DD/MM/YY

D : Jour de la semaine de 1 à 7.

DD : Jour du mois de 1 à 31.

DDD : Jour de l'année de 1 à 366.

DAY : Nom du jour en entier ("lundi", "mardi", "mercredi", ...).

WW : Numéro de la semaine de 1 à 52.

W : Numéro de la semaine dans le mois de 1 à 4.

MON : Nom du mois sur 3 lettres.

MONTH : Mois écrit en entier.

MM : Numéro du mois (de 1 à 12).

YY : Année sur 2 chiffres.

YYYY : Année sur 4 chiffres.

HH : Heure de 1 à 12.

HH24 : Heure de 0 à 23.

MI : Minute.

SS : Seconde.

Conversion

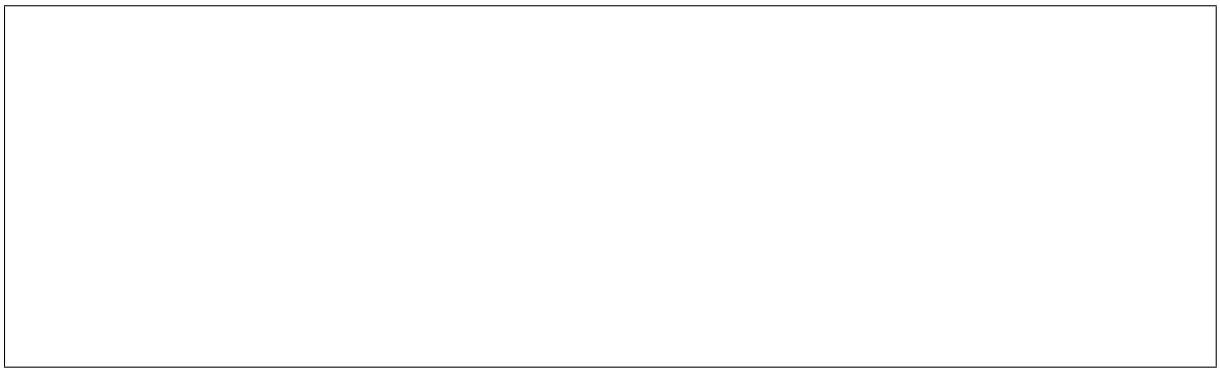
```
TO_CHAR(nombre, format)
TO_CHAR(chaîne, format)
TO_CHAR(date, format)
TO_DATE(chaîne, format)
TO_NUMBER(chaîne)

select substr( to_char( sysdate,'dd/mm/yy'),1,10) from dual;  -- 05/09/2016
select to_date('15/12/05','dd/mm/yy') from dual;              -- 15/12/05
select to_number('50') from dual;                            -- 50
```

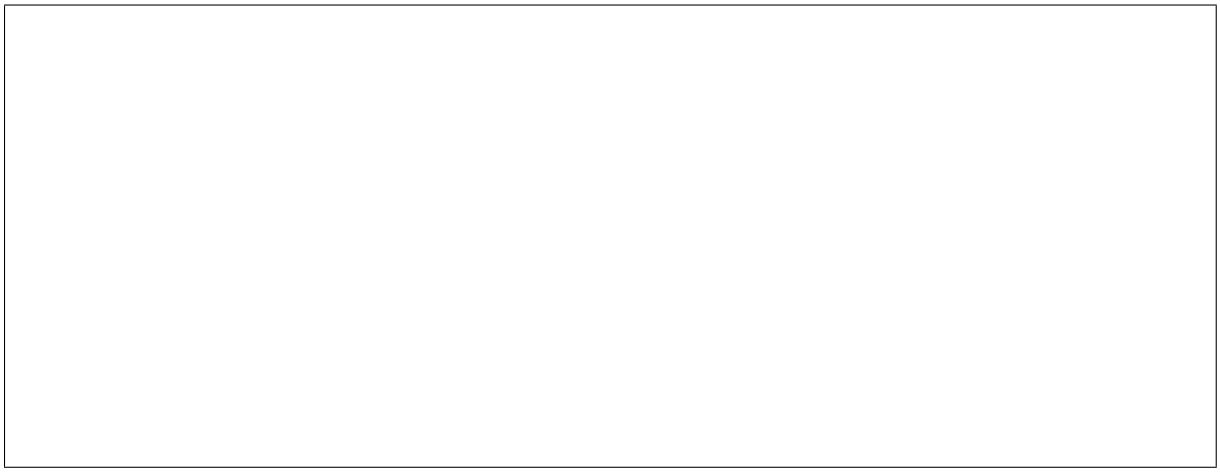
Exercice 4.1 (Dates)**.** Écrire les requêtes suivantes

1. Afficher la date sous cette forme : "nous sommes le mardi 5 septembre 2016. Il est 10 :59"

2. Afficher la date de demain, mais avec heure, minutes et secondes à 0



3. Pour un jour donné, il faut écrire la date et une autre colonne : Si c'est samedi : "samedi, c'est le début du we" Si c'est dimanche : "dimanche, c'est repos" Sinon : "il faut bosser"



4.2 Fonctions de regroupement

Les fonctions de regroupement permettent d'effectuer des calculs sur plusieurs lignes.

— `AVG` : moyenne d'une donnée
— `MIN` : valeur minimale d'une donnée
— `MAX` : valeur maximale d'une donnée
— `COUNT` : compte le nombre de lignes
— `SUM` : somme des valeurs

Syntaxe :

```
SELECT <colonnes indiquant le regroupement>, fonction
FROM ...
WHERE <conditions>
GROUP BY <colonnes indiquant le regroupement>
HAVING <condition suite au regroupement>;
```

Exemple 4.1. Calculer la note minimale, la note maximale et la moyenne du 1er septembre 2017 au 30 juin 2017.

```
SELECT min(note), max(note), avg(note)
FROM notation
WHERE date_note between '01/09/2017' and 30/06/2017';
```

Remarque 4.1. Comme l'ensemble des lignes de la table notation correspondant à la condition représentent un seul group, il n'est pas nécessaire d'utiliser la clause "GROUP BY". C'est le seul cas dans lequel il n'est pas nécessaire.

Exemple 4.2. Calculer la note minimale, la note maximale et la moyenne pour les étudiants du 1er septembre 2017 au 30 juin 2017.

```
SELECT id_etudiant, min(note), max(note), avg(note)
FROM notation
WHERE date_note between '01/09/2017' and 30/06/2017'
GROUP BY id_etudiant;
```

Remarque 4.2. la clause "GROUP BY" doit référencer toutes les colonnes indiquée dans la clause SELECT, hormis les fonctions de regroupement, donc dans notre cas : id_etudiant.

Exemple 4.3. Calculer la note minimale, la note maximale et la moyenne pour les étudiants, par matière, du 1er septembre 2017 au 30 juin 2017.

```
SELECT id_etudiant, id_matiere, min(note), max(note), avg(note)
FROM notation
WHERE date_note between '01/09/2017' and 30/06/2017'
GROUP BY id_etudiant, id_matiere;
```

Remarque 4.3. vous avez compris, la clause "GROUP BY" fait référence à id_etudiant et id_matiere.

4.3 Fonctions personnalisées

Une fonction est un code (PL/SQL) qui peut être appelée directement dans une requête.

```
CREATE FUNCTION moyenne(id IN NUMBER)
   RETURN NUMBER
   IS moy NUMBER(11,2);
BEGIN
   SELECT avg(note) INTO moy FROM Tresultat WHERE id_etudiant = id;
   RETURN (moy);
END;
/
SELECT moyenne(123456) FROM DUAL;
DROP FUNCTION moyenne;
```

4.4 Fonctions courantes

Traitement des valeurs nulles

NVL2 (x, y, z) : si x n'est pas nul, renvoie y sinon renvoi z.

Decode

La fonction `DECODE` permet d'interpréter les différentes valeurs d'une colonne d'une table. En fonction des valeurs correspondant aux indications le résultat est affecté. Si aucune valeur ne correspond alors une valeur par défaut est affectée. Cette fonction perd de son intérêt au regard de la fonctionnalité `CASE`.

```
DECODE (colonne, val1, retour1, val2, retour2, val3, retour3, ... valdefaut)
```

CASE

`Case` permet d'évaluer un champ et de modifier la valeur retournée en fonction du champ.

```
SELECT numero_client,
   CASE decouvert_autorise
        WHEN 0 THEN 'pas de découvert'
    WHEN 2000 THEN 'Elevé'
    WHEN 5000 THEN 'Privilégié'
    ELSE 'Normal' END
   FROM client;
```

Cette requête remplace la valeur retournée (numérique) par une chaîne de caractère explicite, en fonction de la valeur du champ "decouvert_autorise".

5 PL/SQL

Le langage procédural proposé par Oracle s'appelle le PL/SQL - Procédural Language / Structured Query Language). Les principaux avantages d'un langage procédural intégré à la base de données sont les suivants :

— Peu importe le langage de programmation utilisé pour coder une application, le code PL/SQL est le même.

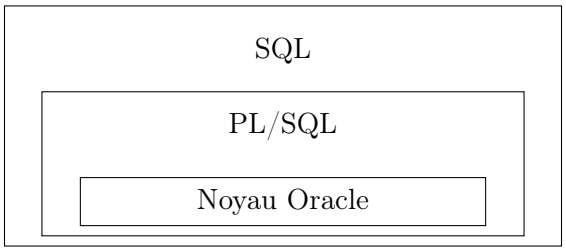
— Le PL/SQL peut être appelé depuis n'importe quel client de la base de données.

— Il peut être déclenché directement par des mécanismes internes (triggers, jobs, ...).

— Il est plus performant qu'un traitement externe.

— Il intègre nativement le SQL.

L'inconvénient évident est qu'il est lié au moteur de base de données. Donc une application développée dont les données sont stockées dans Oracle avec du PL/SQL ne peut pas fonctionner sur MySQL sans redévelopper la partie procédural interne à la base de données.



Structure et syntaxe de PL/SQL

Déclarations des variables	DECLARE
Traitements	BEGIN
Gestion des exceptions (facultatif)	EXCEPTION
Fin du programme	END ;
Demande d'exécution	/

— " ;" à la fin de chaque instruction PL/SQL sauf pour DECLARE, BEGIN, EXCEPTION

— Les lignes commentaire sont décrites soit :

  — /∗ et ∗ / pour un ensemble de lignes.

  — "–" pour une ligne de commentaire.

— **Pas de différence entre les minuscules et les majuscules pour l'écriture des programmes.**

Le programme suivant est le plus simple possible et il ne fait absolument rien, mais permet de vérifier que l'environnement fonctionne correctement. Le message en retour est alors : **Procédure PL/SQL terminée avec succès.**

```
BEGIN
Null;
END;
/
```

Déclaration des variables

Variables de bases `nom_de_variable [CONSTANT] TYPE [not null] [DEFAULT] [:=valeur];`

Le type de données peut être un type de données d'une une table ou un type spécifique parmi ceux-ci :

Type de données	Explication	Exemple
`BOOLEAN`	Donnée booléenne (`NULL`, `FALSE`, `TRUE`)	`Drapeau BOOLEAN;`
`EXCEPTION`	Variable de type exception	`Err_cpt EXCEPTION;`
`BINARY_INTEGER`	Donnée binaire entière (de -2^{31} à 2^{31})	`Var1 BINARY_INTEGER;`
`NATURAL`	Sous ensemble de `BINARY_INTEGER` (0 à 2^{31})	`Var2 NATURAL;`
`POSITIVE`	Sous ensemble de `BINARY_INTEGER` (1 à 2^{31})	`Var3 POSITIVE;`

Variables faisant référence à des objets de la base

Il est possible de définir un type de variable directement en faisant référence à une donnée déjà existante dans la base avec la commande % `TYPE`est que si le type de donnée change dans la table, il n'est pas nécessaire de modifier le code PL/SQL y faisant référence.

```
nom_variable table.colonne % TYPE;
nom client.nomcl % TYPE;
prix materiel.plocje % TYPE;
```

Les variables faisant référence à la structure d'une table

De la même manière, il est possible de récupérer la définition complète d'une table et de l'affecter à une variable PL/SQL qui contiendra un tuple de la table, pour cela, il faut utiliser % ROWTYPE.

```
nom_variable  table% ROWTYPE;
client2 client % ROWTYPE;
histobis  histo % ROWTYPE;
```

Affectations

L'affectation au moment de la déclaration de valeurs à des variables :

```
DECLARE
var  date := sysdate ;
tva  constant number(4,2) := 19.6;
test  varchar2(20);
test2  char(20) := 'valeur indiquee';
BEGIN
Null;
END;
/
```

Affectation au cours du traitement :

```
DECLARE
N  number;
C  char(20);
D date;
E varchar2(12);
F e%TYPE;
BEGIN
N := 0;
N := N + 1 ;
C := 'Durand' ;
d := sysdate + 3 ;
E := null ;
F := 'Aujourd''hui' ; -- afin d'avoir une côte dans une chaîne, il faut la doubler (côte côte)
F := E;
END;
/
```

Affectation de variables faisant référence à une table

```
DECLARE
nom  client.nomcl%TYPE := 'Degrier';
nom2  client.nomcl%TYPE;
prix  materiel.plocj%TYPE ;    -- NUMBER
client2   client%ROWTYPE;
histobis  histo%ROWTYPE ;
cout  number(7,2):=2;
coutTTC cout%TYPE;
BEGIN
nom2 := nom;
 -- Prix := nom ;  -- impossible car types différents
client2.nucl := histobis.nucl;
-- histobis.datret := client2.nucl;  -- impossible car types différents DATE et NUMBER
coutTTC := cout * 1.96;
-- HISTObis := CLIENT2 ;  -- impossible car structure différents
END;
/
```

SQL/Plus

Variables saisies par l'utilisateur

Il existe 2 ordres spécifiques SQL/Plus pour afficher du texte et de faire saisir des données par l'utilisateur :

```
PROMPT [texte]
ACCEPT <nom de variable> [number |char]
```

Exemple 5.1. PROMPT Introduire un numero de client :

```
PROMPT Par exemple '010', '020'
ACCEPT nclient CHAR PROMPT 'Numéro client : '
DECLARE
ncli varchar2(6) := &nclient;
```

```
numclient varchar2(6);
BEGIN
numclient := ncli;
END;
/
```

Si le code ci-dessus est copié dans un fichier nommé "`test.sql`", il est possible de l'exécuter avec la commande sqlplus suivante :

```
SQL> @test
Introduire un numero de client :
Par exemple '010', '020'
Numéro client : '010'
ancien 2 : NCLI varchar2(6) := &nclient;
nouveau 2 : NCLI varchar2(6) := '010';
```

Procédure PL/SQL terminée avec succès.

En cas d'erreur Au moment de l'exécution, si un programme contient une ou plusieurs erreurs, il existe sous SQL*Plus un ordre spécifique pour les afficher : `show errors` ou `select * from user_errors;`

5.1 Instructions conditionnelles

```
IF ..... THEN ..... END IF ;
IF condition THEN
Action;
ELSIF condition2 THEN
Action;
END IF;
ELSE
Action;
END IF;

DECLARE
NUM number;
BEGIN
NUM := 99;
IF num = 99 THEN
Num := 100;
ELSE
Num := 98;
END IF;
END;
/
```

5.2 Boucle WHILE

```
WHILE <expression booléenne>
LOOP
<instructions>
END LOOP ;
```

```
DECLARE
I number(2) := 1;
BEGIN
WHILE I < 50 LOOP
I := I + 1;
END LOOP;
END;
/
```

5.3 Boucle LOOP

```
LOOP
<instructions>
EXIT WHEN <expression booléenne> ;
END LOOP ;
```

```
DECLARE
I number(2) := 1;
BEGIN
LOOP
I := I + 1;
EXIT WHEN I = 50;
END LOOP;
END;
/
```

5.4 Boucle FOR IN

La boucle s'exécute tant que la variable de boucle reste dans les bornes `MINI` et `MAXI`.

```
FOR <variable de boucle> IN [REVERSE] <min> ..<max>
LOOP
<instructions>
END LOOP ;
```

```
BEGIN
FOR I IN 1 .. 50 LOOP
NULL;
END LOOP;
END;
/
```

La variable de boucle, `I` dans l'exemple, commence à `MIN` et s'incrémente automatiquement de 1 en 1 jusqu'à `MAX`.

5.5 Exceptions

La clause `EXCEPTION` vue dans la structure générale d'un programme PL/SQL permet d'une part de traiter les erreurs internes à Oracle, mais aussi de traiter les erreurs orientées application. La section `EXCEPTION` est facultative, mais sa présence est fortement conseillée dans tout programme PL/SQL

digne de ce nom, et pouvant ainsi se parer à toutes éventualités de "dysfonctionnement". Lorsque qu'une exception survient le programme PL/SQL est stoppé. Le code contenu dans la clause exception correspondante est exécuté.

Syntaxe générale :

```
WHEN <nom d'exception> [OR <nom d'exception> ....] THEN <ensemble d'instructions>
```

Les exceptions sont réparties en deux grandes familles :

— Les exceptions **prédéfinies** et disponibles dès l'installation d'Oracle.

— Les exceptions **utilisateurs** orientés applications.

L'exception `OTHERS` permet de traiter toutes les exceptions non traitées explicitement dans la section exception.

Exceptions prédéfinies

Les utilisateurs ont horreur de trouver sur leurs écrans un message d'erreur généré par Oracle ou le système d'exploitation. Ces messages sont souvent peu explicites, peu compréhensibles et peuvent aider à monter des attaques. C'est pour cela que vous devez traiter un maximum d'erreurs possibles, et donner à vos utilisateurs un message d'erreur plus proche de leur langage quotidien. Au minimum il faut toujours traiter l'exception `OTHERS`, et récupérer le message d'erreur Oracle correspondant avec son libellé. Vous accompagnez l'ensemble d'un message d'erreur sympathique du style :

"Erreur non prévue n° : xxxxx, contacter le service informatique"

Pour pouvoir récupérer le code et le libellé d'erreur, PL/SQL dispose de 2 fonctions prédéfinies `SQLCODE` et `SQLERRM`.

— `SQLCODE` correspond au code d'erreur du système.

— `SQLERRM` contenant le code plus le message d'erreur et ayant une longueur de 2000 caractères, il est préférable d'utiliser une variable intermédiaire. Cette dernière récupérant les 100 premiers caractères par exemple.

Liste des exceptions prédéfinies sous Oracle avec leur code d'erreur

Nom de l'exception	Code	Description
`CURSOR_ALREADY_OPEN`	-6511	Ouverture d'un curseur déjà ouvert. Il faut le fermer et le ré-ouvrir.
`DUP_VAL_ON_INDEX`	-1	Un ordre `INSERT` ou `UPDATE` a tenté d'insérer un doublon dans une colonne ayant un index unique.
`INVALID_CURSOR`	-1001	Arrive lorsqu'un `FETCH` ou un `CLOSE` est exécuté sur un curseur non ouvert.
`INVALID_NUMBER`	-1722	Conversion d'une chaîne en nombre impossible.
`LOGIN_DENIED`	-1017	Utilisateur ou mot de passe incorrect.
`NO_DATA_FOUND`	+100	`Select into` infructueux (fin de recherche ou mauvaise sélection).
`PROGRAM_ERROR`	-6501	Erreur interne. Difficile à solutionner (voir syntaxe sur la structure, ...).
`TOO_MANY_ROWS`	-1422	Un select into ramène plus d'une ligne, ce qui n'est pas possible. Vous devez utiliser une boucle avec `FETCH`.
`VALUE_ERROR`	-6502	Erreur de conversion, de troncation, de bornes sur données, ...
`ZERO_DIVIDE`	-1476	Division par 0.
`OTHERS`		Toutes les autres exceptions.

Exemple 5.2 (Division par zéro)**.** Le code ci-dessous provoque volontairement une division par zéro. Elle est traitée par l'exception `ZERO_DIVIDE`. Sans les lignes du traitement du `ZERO_DIVIDE`, la clause `WHEN OTHERS` prend le relais et traite quand même cette erreur comme les autres.

```
DECLARE
z number(2) := 50;
mess1 char(100);
code1 char(10);
i number(2) := -10;
BEGIN
WHILE i < 10 LOOP
z := z / I;
i := i + 1;
END LOOP;
EXCEPTION
WHEN zero_devide THEN insert into result values ('except', division par zero interdite !!!');
WHEN others THEN
mess1 := substr (sqlerrm, 1,100);
code1 := sqlcode;
END ;
/
```

Exceptions utilisateur

La mise en œuvre de ce type d'exceptions est fonction de votre analyse. Les exceptions utilisateur doivent être déclarées dans la section `DECLARE` et être de type `EXCEPTION`. Elles sont déclenchées par l'instruction `RAISE` nom d'exception dans le corps du programme PL/SQL. Si l'instruction `RAISE` est exécutée, elle provoque le débranchement du programme dans la section `EXCEPTION` et les ordres contenus dans la clause `WHEN` nom d'exception seront exécutés.

```
ACCEPT age number PROMPT 'introduire un age'
DECLARE
mess1 char(100);
code1 char(10);
wage integer := &AGE;
err_age EXCEPTION;
BEGIN
IF wage > 130 then RAISE err_age;
END IF;
EXCEPTION
WHEN err_age THEN
Insert into result values
('ERR_AGE','Age doit etre inférieur a 130 ans !!!');
WHEN others THEN
Mess1 := substr (SQLERRM, 1,100);
Code1 := SQLCODE;
Insert into result values ('ERREUR','Erreur interne: '||mess1 ||
                           ' contacter l''informatique');
END;
/
```

Utilisation des accès aux tables

L'utilisation la plus courante du PL/SQL est le traitement des données contenues dans la base. Tous les ordres SQL peuvent être utilisés directement sauf les ordres LDD (Langage de Définition des Données : `CREATE`, `DROP`, ...). Ces derniers nécessitent la mise en œuvre de package spécifiques.

```
SELECT <liste select>INTO <liste de variables>
FROM <tables>
WHERE ...;

DECLARE
  datejour date ;
BEGIN
SELECT sysdate INTO datejour FROM dual ;
END ;
/
```

Curseurs implicites

Certains ordres SQL d'accès aux données peuvent être utilisés directement sans déclarer de curseurs. Dans le langage PL/SQL, l'utilisation de curseurs est obligatoire dès que le programme contient des ordres SQL. Dans le cas des curseurs implicites c'est Oracle qui prend en charge la gestion des curseurs (s'il n'y a qu'une valeur retournée).

```
DECLARE
Wnomcl varchar2(50);
BEGIN
SELECT nomcl INTO wnomcl
FROM client
WHERE nucl = '010' ;
END ;
/
```

Curseurs explicites

Ceci correspond au traitement d'un ordre SQL **ramenant plusieurs enregistrements**. Pour cela, il faut obligatoirement gérer de façon explicite (et manuelle) le ou les curseurs, afin de traiter les lignes retournées par l'ordre SQL une à une.

- Définir le curseur (`CURSOR ... IS`)
- Ouvrir le curseur (`OPEN ...`)
- Lire chaque occurrence dans une boucle (`FETCH ... INTO ...`) en définissant la condition de sortie (`EXIT ...`)
- Fermer le curseur (`CLOSE ...`)

```
DECLARE
CURSOR c_loc IS
SELECT * FROM  commande WHERE  commande.age > 18;
loc_rec c_loc%ROWTYPE ;
BEGIN
OPEN c_loc ;
```

```
LOOP
FETCH c_loc INTO loc_rec ;
EXIT WHEN c_loc%NOTFOUND ;
Insert into result values (...);
END LOOP ;
CLOSE c_loc ;
END ;
/
```

Boucle FOR sur curseur

Reprenant le principe de la boucle FOR . . . IN, la boucle FOR sur curseur facilite la vie du développeur. En effet comme la boucle FOR .. IN, vous n'avez pas besoin de : déclarer le curseur, ouvrir le curseur, tester la fin du curseur, faire des FETCH, et fermer le curseur.

```
FOR <nom d'enregistrement> IN {<nom de curseur> | (<ordre select>) } LOOP
<instructions>
END LOOP ;
```

```
BEGIN
FOR loc_rec IN (
SELECT nomcl, prenomcl, c.nucl, debloc
FROM client C, loue L
WHERE c.nucl = L.nucl) LOOP
Insert into result values (loc_rec.nucl || ' ' || loc_rec.debloc);
END LOOP ;
END ;
/
DECLARE
CURSOR c_loc IS
SELECT nomcl, prenomcl, c.nucl, debloc
FROM client C, loue L
WHERE c.nucl = L.nucl;
BEGIN
      FOR loc_rec IN c_loc  LOOP
Insert into result values (loc_rec.nomcl|| ' ' || loc_rec.nucl || ' ' || loc_rec.debloc);
END LOOP ;
END ;
/
```

Avec l'injection d'une variable

```
DECLARE
CURSOR cur1 (DEP varchar2) IS
SELECT nucl, nomcl, telcl
FROM client
WHERE substr(postalcl,1,2) = DEP;
BEGIN
FOR cur1_rec IN cur1('&1') LOOP
```

```
insert into result values ('test',cur1_rec.nucl);
END LOOP ;
END ;
/
```

5.6 Déclencheurs (triggers)

Les triggers permettent d'exécuter du code PL/SQL lorsqu'un événement arrive. La majorité des triggers concernent des actions sur une table. Il est par exemple possible de définir le déclenchement d'un code PL/SQL à chaque insertion d'une nouvelle donnée dans une table. Un déclencheur s'exécute dans le cadre d'une transaction. Il existe plusieurs options concernant les triggers, dans la suite une syntaxe simplifiée est présentée.

Syntaxe simplifiée : Ce qui est entre crochets est facultatif.

```
CREATE TRIGGER <nom_du_trigger> BEFORE INSERT ON <table> [ FOR  EACH ROW ]
                                AFTER  UPDATE
                        DELETE
[ REFERENCING OLD as ancien NEW as nouveau ]
[ WHEN <condition> ]

<commande SQL / appel à procedure / bloc PL/SQL>
```

`BEFORE/AFTER` **:** Le bloc PL/SQL peut être exécuté **avant** ou **après** la vérification des contraintes d'intégrité.

`INSERT, UPDATE` **ou** `DELETE` **:** Action pour laquelle le code PL/SQL est déclenché. Il est possible de mettre plusieurs actions séparées par `OR`, puis de décomposer le code en fonction de l'action qui a déclenché le PL/SQL.

`ON` **:** Nom de la table concernée par le trigger.

`FOR EACH ROW` **:** Permet d'exécuter le code PL/SQL pour chaque ligne concernée par l'action SQL. Sans cette option, le code PL/SQL est exécuté une fois.

`REFERENCING OLD` **as ancien** `NEW` **as nouveau :** Lors d'un trigger exécuté sur une commande update ou delete, les anciennes valeurs sont accessibles via la variable `OLD`.

Lors d'un trigger exécuté sur une commande insert out update, les nouvelles valeurs sont accessibles via la variable `NEW`. Il est possible de changer ces noms de variables par l'option indiquée.

`WHEN CONDITION` **:** Permet de restreindre l'exécution du PL/SQL.

COMMANDE SQL / APPEL A PROCEDURE / BLOC PL/SQL : Code à exécuter.

Exemple 5.3. `CREATE OR REPLACE TRIGGER trace_delete_commande`

```
BEFORE DELETE ON commande
FOR EACH ROW
BEGIN
Insert into trace values (sysdate,'suppression de la commande'||:OLD.nocmd ) ;
END;
/
```

Exemple 5.4. `CREATE OR REPLACE TRIGGER TRG_commande`

```
BEFORE INSERT OR UPDATE OR DELETE ON commande
FOR EACH ROW
```

```
BEGIN
IF INSERTING THEN
dbms_output.put_line( 'Insertion dans la table EMP' ) ;
END IF ;
IF UPDATING THEN
dbms_output.put_line( 'Mise à jour de la table EMP' ) ;
END IF ;
IF DELETING THEN
  dbms_output.put_line( 'Suppression dans la table EMP' ) ;
END IF ;
END ;
/
```

Exemple 5.5.
```
CREATE OR REPLACE TRIGGER Print_salary_changes
  BEFORE DELETE OR INSERT OR UPDATE ON Emp_tab
  FOR EACH ROW
WHEN (new.Empno > 0)
DECLARE
    sal_diff number;
BEGIN
    sal_diff  := :new.sal  - :old.sal;
    dbms_output.put('Old salary: ' || :old.sal);
    dbms_output.put('  New salary: ' || :new.sal);
    dbms_output.put_line('  Difference ' || sal_diff);
END;
/
```

Autres commandes :

```
ALTER TRIGGER nom_déclencheur DISABLE;
ALTER TRIGGER nom_déclencheur ENABLE;
ALTER TABLE nom_table DISABLE ALL TRIGGERS;
ALTER TABLE nom_table ENABLE ALL TRIGGERS;
DROP TRIGGER TRG_commande;
```

Exercice 5.1. Pour continuer l'exercice 3.3, il est nécessaire de créer un trigger qui empêche l'insertion d'une ligne avec n'importe quelle valeur dans la colonne `id_trace`. Écrire les requêtes permettant de tester le trigger et expliquer les résultats.

6 Concurrence d'accès

Il existe 3 catégories de commandes dans une base de données.

1. **DDL** : *Data Definition Language* (en français LDD : Langage de Définition de Données). Cette catégorie regroupe toutes les commandes permettant de gérer la structure des données. Ces commandes sont `CREATE`, `ALTER`, `DROP`, `GRANT`, `REVOKE`, ... (plus simplement, toutes les commandes non DML).
   Exemple : `ALTER TABLE add (nouvelle_colonne NUMBER);`
2. **DML :** *Data Manipulation Language* (en français : Language de Manipulation de Données). Cette catégorie regroupe toutes les commandes permettant de manipuler les données. Ces commandes sont `INSERT, UPDATE, DELETE`. La commande n'affecte pas la structure des données dans la base, mais uniquement le contenu.
   Exemple : `UPDATE commande SET date_commande=sysdate WHERE num_commande=1203;`
3. `SELECT` : Cette commande est à part car elle ne modifie en rien la structure ou les données.

L'objectif est de comprendre comment se passe les accès à la base de données lorsque plusieurs utilisateurs l'utilisent en même temps. Cela demande de regarder comment sont gérés les accès concurrents aux données en écriture et en lecture afin d'assurer la cohérence et l'intégrité de la base de données.

6.1 Transaction

Définition 6.1. Une *transaction* est une unité logique de traitements regroupant un ensemble d'opérations élémentaires. Une transaction effectue un ensemble d'ordre SQL, par exemple `SELECT` puis `UPDATE` puis `DELETE`. Ce n'est qu'à la fin de ces opérations que l'utilisateur décide de VALIDER son travail ou de L'ANNULER.

Les mises à jour ne sont effectives qu'au moment de la validation, sinon il y a annulation des modifications

Début de transaction : Une nouvelle transaction commence dès l'exécution de la première commande agissant sur des données ou prévoyant d'agir sur les données.

Fin de transaction : Il y a deux moyen pour terminer une transaction.

Validation d'une transaction par :

- Exécution explicite de la commande `"commit;"`
- Exécution d'une commande DDL. Dans ce cas toutes les opérations précédentes sont validées, par un `COMMIT` implicite.
- Fin normale d'un programme ou d'une session SQL/Plus, `COMMIT` implicite.

Annulation d'une transaction par :

- Exécution explicite de la commande `"rollback;"`.
- Fin anormale d'un programme ou d'une session SQL/Plus, donc `ROLLBACK` implicite.

6.2 ACID

En 1983, Harder et Reuter ont défini un modèle de transactions pour les actions effectuées sur une base de données. Ce modèle leur permet d'introduire 4 concepts fondamentaux pour garantir al cohérence et l'intégrité des données.

Atomicité : Gestion des modifications de données sous formes de transactions. S'il y a par exemple, un problème technique, l'ensemble de la transaction est annulé. Une transaction se fait donc complètement ou ne se fait pas du tout.

Cohérence : Une transaction fait passer la base d'un état cohérent à un autre état cohérent. Différents mécanismes permettent de garantir le retour dans un état cohérent en cas de panne du SGBDR. Ce principe s'applique aussi au niveau des lectures.

Isolation : Toute transaction doit s'exécuter comme si elle était seule. Il ne peut y avoir aucune dépendance entre les transactions.

Durabilité : Lorsque la base atteint un état cohérent, c'est état est enregistré et pérenne, même en cas de panne matériel.

6.3 Lecture cohérente et undo

Lorsqu'une transaction modifie des données, la nouvelle valeur vient effectivement écraser l'ancienne donnée au niveau du bloc de données physique et ce, dès que la commande de modification de la donnée est exécutée. La transaction n'est pas terminée, donc Oracle ne peut pas savoir si vous allez valider ou invalider les données. Il faut donc que la base de données conserve la valeur avant modification pour pouvoir revenir en arrière en cas de `ROLLBACK`. Au niveau d'Oracle, ce mécanisme s'appelle l'undo.

Undo Il existe un espace de stockage dans lequel Oracle garde les images avant modification. Lorsqu'une donnée est modifiée :

1. L'image de la donnée avant modification est copiée dans un espace dédié `UNDO`.
2. L'adresse permettant d'atteindre cette image est gardée dans une table de correspondance au niveau interne de la base de données.
3. Un indicateur est positionné au niveau du bloc de données pour indiquer que la donnée est en cours de modification par une transaction.

Il y a alors 2 cas possibles.

1. La transaction est validée. Dans ce cas, l'indicateur (3) est enlevé.
2. La transaction est annulée. Dans ce cas, toutes les données qui ont été modifiées par la transaction sont écrasées par leurs valeurs avant modification.

Pourquoi fonctionner ainsi et pas en stockant dans un espace temporaire les nouvelles valeurs ? Oracle part du principe que les transactions annulées sont plus rares que les transactions validées. En mettant à jour les données dans les blocs de table, la validation est presque instantanée par contre, le `ROLLBACK` est plus long.

Lecture cohérente

Lorsqu'une transaction est en cours, les autres sessions ne doivent pas voir les modifications effectuées par la transaction tant que cette dernière n'est pas validée. Soit une session S1 et une session S2.

Temps	Session S1	S2
T1	Modification de la 12045$^{\text{ième}}$ ligne de la table INDIVIDU. L'image avant modification est stockée dans l'undo. La nouvelle valeur est stockée dans les blocs de données de la table.	
T2		`SELECT` sur la table individu
T3		La session arrive à la ligne 12045. La valeur lue n'est pas la valeur présente dans le bloc de données, mais la valeur présente dans l'undo.
T4	Fin de la transaction	

Remarque 6.1. les temps T1 et T2 peuvent être inversé et le fonctionnement reste le même.

Le tableau précédent montre le principe permettant d'effectuer une lecture cohérente. Les données sont cohérentes à l'instant T1, toutes les données retournées à S1 sont celles qui étaient présentes à l'instant T1.

Démonstration de transaction

```
CREATE TABLE commande (numero number, designation varchar2(50));
```

Temps	Session S1	Session S2
T1	`SELECT count(1) FROM commande;` 0 ligne	
T2		`INSERT INTO commande VALUES (1,'commande 1');` 1 ligne créée.
T3	`SELECT count(1) FROM commande;` 0 ligne **Les modifications effectuées par S2 ne sont pas visibles car la session S2 n'a pas terminé sa transaction.**	
T4		`Commit;`
T5	`SELECT count(1) FROM commande;` 1 ligne **Les données sont visibles !**	

Gestion des accès concurrents

Dans une architecture multi-utilisateurs, la modification simultanée des mêmes données étant impossible, le système devra assurer :

— Un mécanisme de concurrence d'accès aux données.

— Un mécanisme de lecture cohérente.

Oracle assure automatiquement ces mécanismes grâce à l'utilisation de verrous et par un mécanisme de conservation de données. Il n'existe pas de concurrence d'accès aux données en LECTURE (avec un SELECT simple). Un utilisateur qui lit une donnée n'interférera pas avec une transaction.

Verrous : Le but d'un verrou est d'empêcher la modification d'une donnée par plusieurs sessions en même temps. Le verrou est donc posé par la première transaction qui le demande et les autres sessions demandant la modification de la donnée déjà verrouillée, attendent. Une fois que la transaction détenant

le verrou termine sa transaction (par commit ou rollback), la transaction suivante, qui était bloquée par le verrou, pose alors son verrou et peut travailler. Il existe deux niveaux de verrouillage :

— Au niveau de la ligne (DML lock)

— Au niveau de la table (DDL lock)

La possibilité de lever un verrou est attribué en mode FIFO (First In First Out). Soit une session S1 qui, lors d'une transaction pose un verrou. Soit plusieurs sessions bloquées par le verrou ci-dessus, arrivée dans l'ordre A, B, C, D, ... Lorsqu'un verrou est libéré par S1, c'est la première session qui a été bloquée qui pose le nouveau verrou (donc la session A). Lorsque A libérera le verrou, ce sera au tour de B et ainsi de suite.

Verrou au niveau ligne : Avec cette stratégie, chaque ligne peut être verrouillée individuellement. Le verrou empêche donc la modification d'une ou plusieurs lignes mais permet la modification des autres lignes de la table.

Lorsqu'un tel verrou est positionné sur une ligne, la base de données va :

1. Poser un verrou de manipulation de données (DML lock) sur la ligne, pour empêcher les données d'être modifiées.
2. Poser un verrou de définition de données (DDL lock) sur la table pour empêcher les modifications de structure de la table.

Verrou de table : Un verrou de table est un verrou empêchant la modification de la structure (`ALTER`), ainsi que la suppression de la table. Ce verrou n'agit pas au niveau des lignes, ce qui signifie que les commandes `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, ... sont toujours possibles.

Exemple 6.1 (Verrous)**.** Dans le tableau ci-dessous :

— XX : verrou de ligne où XX indique l'identifiant de la ligne concernée par le verrou.

— XX : indique la libération du verrou et la ligne concernée.

— XX : indique que la session est bloquée par un verrou.

— : verrou de table.

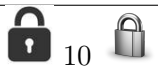
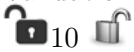
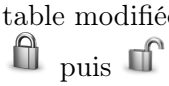
— : libération du verrou de table.

— : bloqué par le verrou de table.

```
CREATE TABLE commande (numero number, designation varchar2(50));
```

Tps	Session S1	Session S2	Session S3

T1	`INSERT INTO commande VALUES (10,'commande 1');`
	1 ligne créée
	`INSERT INTO commande VALUES (11,'commande 2');`
	1 ligne créée
	`INSERT INTO commande VALUES (12,'commande 3');`
	1 ligne créée
	`Commit;`
	Validation effectuée		

T2		`UPDATE commande SET designation='new 1' WHERE numero=10;` 1 ligne créée 	
T3			`UPDATE commande SET designation='new 2' WHERE numero=11 ;` 1 ligne mise à jour 
T4			`UPDATE commande SET designation='new 11' WHERE numero=10;` 
T5	`ALTER TABLE commande ADD (quand date);` 		
T6		`commit;` Validation effectuée 	
T7			1 ligne mise à jour 
T8			`rollback;` annulation effectuée 
T9	table modifiée 		

`Drop table commande;`

Deadlock

Le deadlock, aussi appelé **interblocage** de transactions, arrive quand des sessions s'interbloquent les unes avec les autres. Dans ce cas, Oracle met fin à une des deux sessions pour permettre à l'autre de continuer.

Dans le tableau ci-dessous :

—  : verrou de ligne. XX indique l'identifiant de la ligne concernée par le verrou.

—  : indique la libération du verrou et la ligne concernée.

—  : indique que la session est bloquée par un verrou, où XX indique l'identifiant de la ligne sur laquelle une autre session a posé le verrou.

Les tables sont créées comme suit :

```
CREATE TABLE commande (numero number, designation varchar2(50));
INSERT INTO commande VALUES (10,'commande 1');
INSERT INTO commande VALUES (11,'commande 2');
COMMIT;
```

Tps	Session S1	Session S2
T1	`UPDATE commande SET designation='S1 demo 1' WHERE numero=10;` 1 ligne mise à jour  10	
T2		`UPDATE commande SET designation='S2demo 2' WHERE numero=11;` 1 ligne mise à jour  11
T3	`UPDATE commande SET designation='S1 demo 2' WHERE numero=11;`  10  11	 11
T4	 10  11	`UPDATE commandeSET designation='S2 demo 1' WHERE numero=10;`  10  11
T5	ERREUR à la ligne 1 : ORA-00060 : détection d'inter blocage pendant l'attente d'une ressource  10	 10  11

T6	`SELECT * FROM commande;`
	`NUMERO DESIGNATION`
	`------ ------------`
	`10 S1 demo 1`
	`11 commande 2`
	  10
		 10  11

T7	`commit;`  10	 10  11
T8		1 ligne mise à jour  10  11

T9		`SELECT * FROM commande;`
		`NUMERO DESIGNATION`
		`------ ------------`
		`10 S2 demo 1`
		`11 S2 demo 2`
		  10  11

T10		`Commit;`  10  11

```
Drop table commande;
```

Soit le compte de votre professeur (exemple : `PALAFOUR`) comme compte principal et un utilisateur supplémentaire créé pour le TD (exemple : `PALAFOUR2`)

Exercice 6.1 (Mise à jour)**.**

1. A partir du compte principal, mettre à jour toutes les lignes de la table `INDIVIDU` pour mettre `sexe='F'` au lieu de `sexe='M'`.

2. A partir du second compte, supprimer l'individu dont le nom est `'MERCIER'` et le prénom est `'Daniel'`.

3. Que constatez dans le cas où ces commandes sont lancées à un intervalle de temps très court par rapport au temps de traitement de ces requêtes ? Expliquez précisément les mécanismes de verrous qui sont mis en place.

4. A partir du compte principal, supprimer l'individu dont le nom est `'MERCIER'` et le prénom est `'Daniel'`.

5. Que constatez si cette seconde commande est effectuée juste après la seconde commande ? Expliquez précisément, les éléments techniques qui interviennent.

Exercice 6.2 (Deux Sessions)**.**

1. Faire un rollback sur les deux sessions lancées.

2. Effectuer 2 connexions sur le compte principal. Sur la première session, lire le nombre de lignes de la table `INDIVIDU`.

3. Sur la seconde session, faire une suppression de toutes les lignes de la table `INDIVIDU` dont le sexe est `'H'`.

4. Sur la première session, lire le nombre de lignes de la table `INDIVIDU`.

5. Que constatez si la commande de la seconde session prend longtemps par rapport au `SELECT` ?

7 Optimisation Oracle

Dans toute base de données, la rapidité d'accès aux données est un élément à prendre en compte. Les accès disque sont les éléments les plus pénalisant ainsi plus ces accès sont limités, plus le temps de réponse sera court. Pour ce cours, le MLD est supposé être correctement construit.

Est-ce que la requête est bien écrite ? Avant d'optimiser une requête, il est important de comprendre exactement le résultat attendu. Fréquemment, le développeur part d'une requête simple et la fait évoluer. Au fur et à mesure de ces évolutions, le développeur arrive à obtenir le résultat qu'il souhaite mais la requête n'est pas forcément propre et optimale.

Il faut donc vérifier :

— Que les tables (ou vues, ...) renseignées dans la clause `FROM` sont toutes nécessaires.

— Que la clause `WHERE` contient toutes les jointures concernant les tables de la clause `FROM`.

— Que les sous-requêtes sont bien jointes à la requête principale.

— Que les conditions présentes dans la clause `WHERE` sont les plus restrictives possibles.

Jointures Plus les requêtes sont compliquées et plus il est important de se référer au MLD pour construire les jointures et ainsi ne pas oublier de cas.

Exemple : Comment connaître la désignation du type de coordonnées des personnes dont le père s'appelle "DURAND Pierre".

T : TYPE (idtype, designation)
C : COORDONNEES (idcoord, valeur, #idind, #idtype)
P : PARENT (#idparent, #idenfant)
I : INDIVIDU (idind, nom, prenom, datenaissance, lieunaissance)

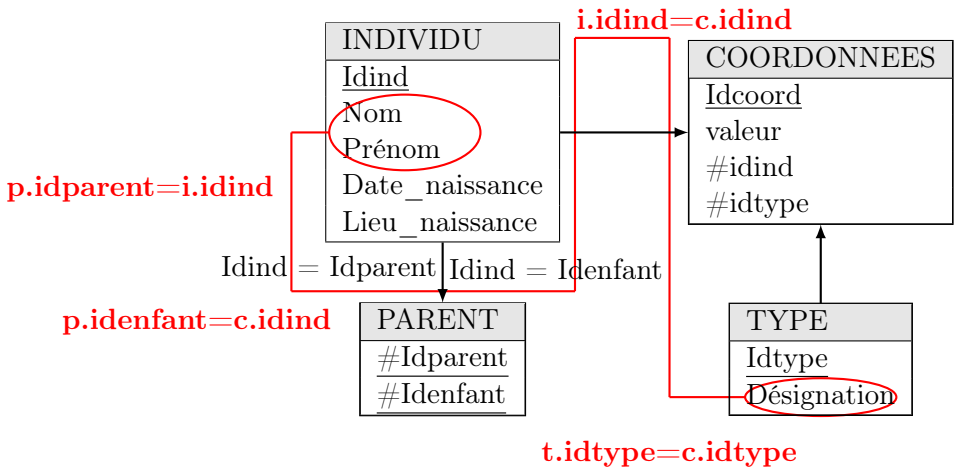


Figure 16 – MLD de gestion d'arbre généalogique.

Dans la requête SQL, il faut joindre :

— `t.idtype` et `c.idtype`

— `c.idind` et `i.idind` (pour l'enfant)

— `i.idind` et `p.idenfant`

— `p.idparent` et `i.idind` (pour le père)

Il est possible d'imbriquer des requêtes de plusieurs manières.

7.1 Sous-requête

Sous-requête au niveau des colonnes

La requête suivante ramène le nom et le prénom du père d'un individu donné.

ATTENTION : dans ce cas de sous-requête, il faut que la sous-requête ramène une seule ligne et une seule valeur.

```
SELECT nom, prenom, (SELECT prenom||' '||nom
                     FROM individu i2, parent p
                     WHERE p.idparent=i2.idind
                     AND sexe='M'
                     AND p.idenfant=i1.idind)
FROM individu i1
WHERE idind=36;
```

Sous-requête au niveau de la clause `FROM`

Cette sous-requête peut ramener plusieurs lignes. La jointure est faite dans la clause WHERE de la requête principale.

```
SELECT i1.prenom||' '||i1.nom enfant, sr.prenom||' '||sr.nom as parent
FROM individu i1, (SELECT p.idenfant, i.nom, i.prenom
                   FROM individu i, parent p
                   WHERE p.idparent=i.idind) sr
WHERE i1.idind=sr.idenfant
AND sr.idenfant=36;
```

Sous-requête au niveau de la clause `WHERE`

Cette requête ramène les lignes de la table individu, dont `idind=36`, et pour lesquelles il existe un parent.

```
SELECT idind, nom, prenom
FROM individu i
WHERE exists (SELECT 1 FROM parent WHERE idenfant=i.idind)
AND idind=36;
```

Restriction des données

Il faut si possible limiter les données dès que possible dans les sous-requêtes.

Exercice 7.1. La quelle des deux requêtes est plus rapide et pourquoi ?

```
SELECT idind, nom, prenom
FROM (SELECT * FROM individu) i
WHERE i.idind=36;

SELECT idind, nom, prenom
FROM (SELECT * FROM individu WHERE idind=36);
```

Produit cartésien

Soit une table A comptant 2000 lignes et une table B comptant 6000 lignes. Un produit cartésien est une requête ou un traitement ramenant l'ensemble des lignes de la table B pour chaque ligne de la table A. La plupart du temps, un tel traitement est dû à l'oubli d'une clause de jointure entre les 2 tables.

Exemple 7.1 (Requête engendrant un produit cartésien)**.** La requête suivante renvoie 2000 × 6000 = 12 000 000 lignes.

```
SELECT *
FROM A, B;
```

Pour éviter le produit cartésien, il faut mettre la clause de jointure entre les deux tables :

```
SELECT *
FROM A, B
WHERE A.col=B.col;
```

Parfois, ces produits cartésiens ne sont pas évidents à voir.

```
SELECT i1.prenom||' '||i1.nom enfant, sr.prenom||' '||sr.nom as parent
FROM individu i1, (SELECT p.idenfant, i.nom, i.prenom
                   FROM individu i, parent p
                   WHERE p.idparent=i.idind) sr
WHERE i1.idind=sr.idenfant AND sr.idenfant = 36 OR sr.idenfant=35;
```

La requête ci-dessus génère un produit cartésien. en effet, après la clause `WHERE`, il y a la condition de jointure mais le `OR` génère le traitement suivant : Il renvoie des lignes de sr pour lesquelles `idenfant=36`, et pour chacune de ces lignes, les lignes de i1 pour lesquelles `idind=sr.idenfant`.

renvoi aussi (**à cause du OR**), des lignes de `sr` dont `idenfant=35` et pour chacune de ses lignes, TOUTES les lignes de i1 car la jointure n'est pas présente de ce côté de la condition `OR`.

Solution :

```
SELECT i1.prenom||' '||i1.nom enfant, sr.prenom||' '||sr.nom as parent
FROM individu i1, (SELECT p.idenfant, i.nom, i.prenom
                   FROM individu i, parent p
                   WHERE p.idparent=i.idind) sr
WHERE i1.idind=sr.idenfant AND ( sr.idenfant = 36 OR sr.idenfant=35 ) ;
```

7.2 Index

Un *index* est une structure de données permettant d'accéder plus rapidement aux données recherchées. Un index contient l'ensemble des valeurs contenues dans la colonne indexée. Ces données sont ordonnées pour permettre une recherche rapide. Pour chaque valeur, l'index contient aussi le `rowid` de la ligne. Cette information permet donc, une fois que la donnée est trouvée dans l'index, de faire le lien rapidement vers la ligne de la table. Si un index concerne plusieurs colonnes, l'ordre des colonnes lors de la création de l'index a de l'importance.

Index B-tree

Ce type d'index est le plus courant. Un index de ce type est organisé sous forme d'arbre. Les terminaisons s'appellent des feuilles. Les éléments intermédiaires, reliés aux feuilles par les branches sont les nœuds.

Exemple 7.2 (B-tree)**.** Le schéma de la Figure 17 montre concrètement comment il est organisé un B-tree.

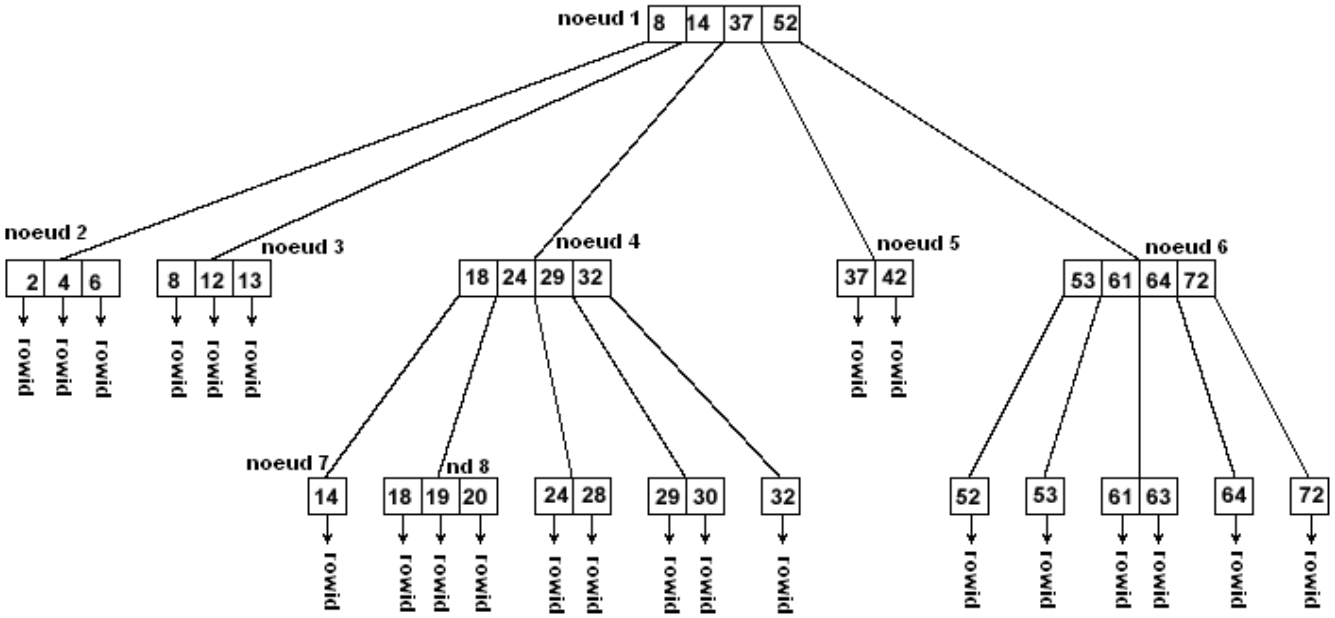


Figure 17 – Exemple de B-Tree.

Un nœud contient des valeurs seuils et des pointeurs. Pour le premier nœud, il faut lire : Jusqu'à 8, aller dans le nœud 2. De 8 à 14, aller dans le nœud 3...

Pour trouver le `rowid` pour la valeur 19, il faut parcourir : nœud 1 nœud 4 nœud 8.

Index Bitmap

Ce type d'index est rarement utilisé alors qu'il s'avère très performant dans le cas où le nombre de valeurs possibles pour la colonne indexée est très faible (exemple : couleur des yeux dans la table individu). Cet index se présente sous forme d'un tableau dans lequel il y a les différentes valeurs possibles de la colonne et les `rowid`. Les valeurs possibles sont alors 0 ou 1.

Exemple 7.3. La figure 18 montre l'organisation d'un index Bitmap.

La valeur 0 ou 1 est stockée sous 1 bit et les recherche dans cet index se font sous forme d'opérations logiques. De plus ces index prennent peu de place physique.

Remarques sur les index

— un index peut être descendant. c'est-à-dire qu'au lieu de commencer par la valeur la plus faible, il commence par la valeur la plus forte.

— `Reverse Index`. Cet index permet de résoudre certains problèmes de performance. Prenons le cas d'une table "pièce" contenant les résultats de tests de conformité de pièces automobiles. Pour

couleur \ rowid	1	2	3	4	5	6	7	8	9	10	...
Bleu	**1**	0	0	0	0	0	0	0	**1**	0	...
Vert	0	**1**	0	**1**	0	**1**	0	0	0	0	...
Noisette	0	0	**1**	0	**1**	0	**1**	0	**1**	**1**	...
Gris	0	0	0	0	0	0	0	0	0	0	...

Figure 18 – Exemple d'index Bitmap.

contrôler 4000 pièces par jour, dans cette table, la colonne "référence" est au format suivant : YYYY-MM-C avec :

- `YYYY` : année sur 4 chiffres
- `MM` : mois sur 2 chiffres
- `C` : code de la pièce contrôlée

Pour obtenir les données concernant la pièce "P" pour les 5 années passées, il faut descendre beaucoup de nœuds dans un index B-tree classique afin de trouver l'ensemble des données recherchée. L'index inversé repère la donnée en la lisant depuis la fin, c'est-à-dire qu'il stocke sous la forme "C-MM-YYYY". L'index b-tree ainsi organisé permet de trouver très rapidement les valeurs commençant par "P".

Commandes des index

Suppression d'un index : `DROP INDEX nom_i1;`
Création d'un index : `CREATE INDEX nom_i1 ON ma_table (colonne_1);`
Création d'un index sur valeur calculée : `CREATE INDEX nom_i1 ON (prix* taux_tva/100);`
Création d'un index basé sur une fonction : `CREATE INDEX nom_i1 ON ma_table(trunc(date_achat)));`
Création d'un index bitmap : `CREATE BITMAP INDEX nom_i1 ON ma_table(yes_color);`
Création d'un index unique : `CREATE UNIQUE INDEX nom_i1 ON ma_table(identifiant);`

Quelles colonnes indexer ?

Clés primaires

Oracle crée automatiquement des index sur les colonnes concernées par les contraintes `UNIQUE` ou `PRIMARY KEY`. C'est le seul moyen rapide de savoir si une valeur est unique.

Clés étrangères

Rajouter une clé étrangère permet de faire des jointures. Ainsi un index est créé sur les colonnes concernées par une contrainte FK.

Clause `WHERE`

Des index peuvent être pertinents sur les colonnes concernées par la clause `WHERE`.

```
SELECT *
FROM individu
WHERE telephone = '09.09.09.09.09'
```

Clause SELECT

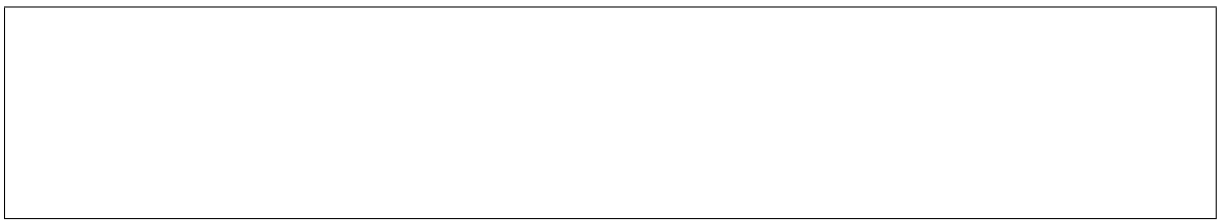
Certaines requêtes peuvent être optimisées en posant un index sur une colonne de la clause `SELECT` car l'index peut éviter d'aller chercher les blocs de données et n'utiliser alors que les blocs d'index.

```
SELECT id_etudiant
FROM individu
WHERE matricule='2143231';
```

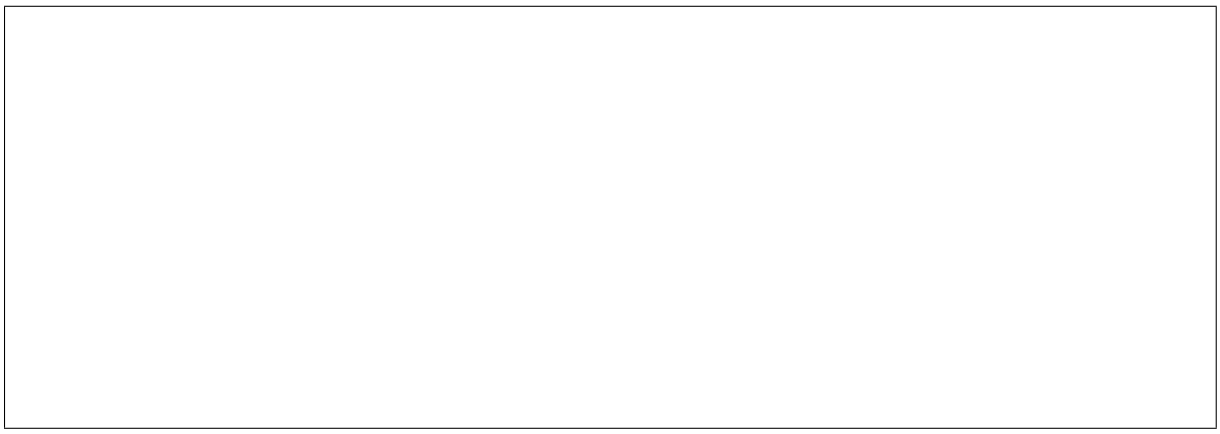
Un index sur (matricule, id_etudiant) n'ira pas lire de bloc de données et n'utilisera que l'index.

Exercice 7.2 (Index)**.**

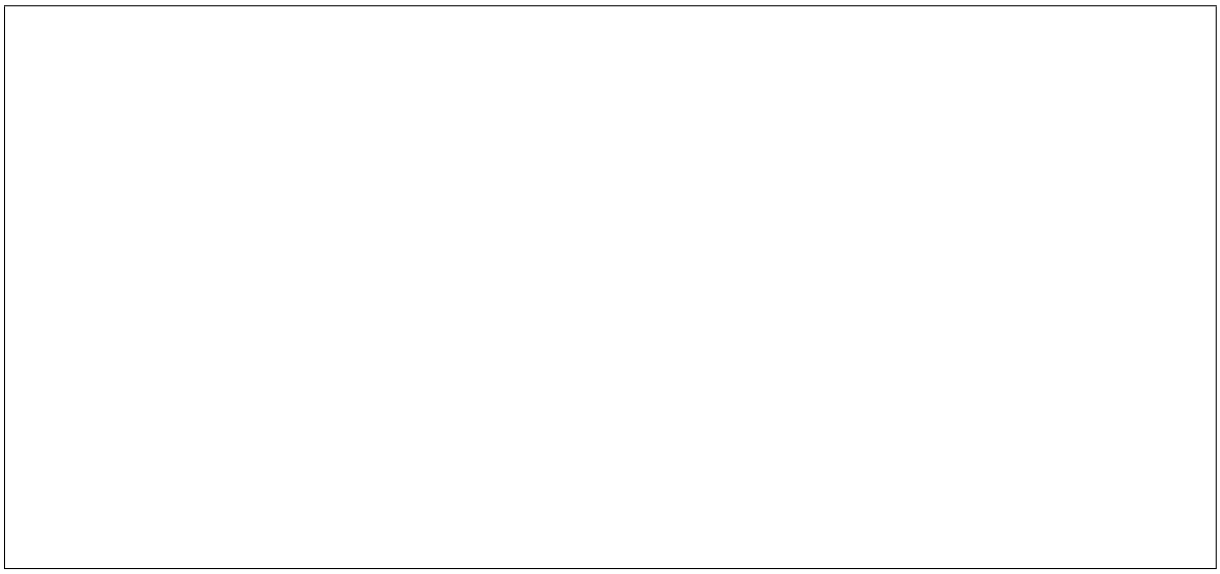
1. Donner la requête permettant d'obtenir l'âge moyen des personnes en fonction du sexe, sachant que la table individu contient la date de naissance d'une personne dans le champ `ddn` et son sexe dans le champs `sexe`.



2. Indiquer quel est le plan d'exécution sachant qu'il existe un index sur `individu(id_ind)`.



3. Indiquer le ou les index pertinents pour optimiser cette requête.



7.3 Plan d'exécution

Lorsqu'une requête SQL est soumise, Oracle définit la méthode qu'il faut utiliser pour obtenir les données. Il se base sur différentes informations (nombre de lignes, taille de lignes, ...) et définit un *plan d'exécution*. Plus un plan d'exécution a un coût faible, plus l'exécution de la requête SQL est rapide.

Sous SQL/PLUS, il est possible de voir le plan d'exécution des requêtes en lançant la commande : `SET autotrace traceonly` ne montre que le plan d'exécution mais ne lance pas la requête. la commande `SET autotrace on` lance la requête et montre le plan d'exécution.

Dans le plan d'exécution, il y a les informations suivantes (liste non exclusive) :

Ligne du plan	Explication
`TABLE ACCESS FULL`	Lecture de toutes les lignes de la table
`TABLE ACCESS BY INDEX ROWID`	L'étape précédente donne une liste de `rowid`.
`INDEX UNIQUE SCAN`	Un index permet d'obtenir un unique rowid permettant d'accéder à la ligne de la table.
`INDEX RANGE SCAN`	Un index ramène un ensemble de rowid permettant d'accéder directement aux différentes lignes de la table.
`HASH JOIN`	Pour les deux tables concernées par la jointure, des valeurs de hash sont calculées à partir des clés, et les valeurs hash sont comparées.
`BUFFER SORT`	Tri des lignes
`MERGE JOIN CARTESIAN`	Indication d'un produit cartésien
`NESTED LOOP`	Pour chaque clé de la table A, vérifier qu'il existe la même valeur dans la table B.

Exemple 7.4 (Plan d'exécution)**.** Soit la requête suivante :

```
SELECT *
FROM parent p, individu i, coordonnees c, type t
WHERE p.id_parent=i.id_ind
AND  c.id_ind=i.id_ind
AND  t.id_type=c.id_type
AND  t.designation='email';
```

Le plan d'exécution (pour des tables vides) est le suivant :

```
Plan d'exécution
----------------------------------------------------------
Plan hash value: 2358650952

-------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                  |     1 |   547 |     9  (12)| 00:00:01 |
|   1 |  NESTED LOOPS                 |                  |     1 |   547 |     9  (12)| 00:00:01 |
|   2 |   NESTED LOOPS                |                  |     1 |   521 |     8  (13)| 00:00:01 |
|*  3 |    HASH JOIN                  |                  |     1 |   340 |     7  (15)| 00:00:01 |
|*  4 |     TABLE ACCESS FULL         | TYPE             |     1 |    40 |     3   (0)| 00:00:01 |
|   5 |     TABLE ACCESS FULL         | COORDONNEES      |   215 | 64500 |     3   (0)| 00:00:01 |
|   6 |    TABLE ACCESS BY INDEX ROWID| INDIVIDU         |     1 |   181 |     1   (0)| 00:00:01 |
|*  7 |     INDEX UNIQUE SCAN         | SYS_C0015341     |     1 |       |     0   (0)| 00:00:01 |
|*  8 |   INDEX RANGE SCAN            | EST_PARENT_DE_PK |     1 |    26 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("T"."ID_TYPE"="C"."ID_TYPE")
   4 - filter("T"."DESIGNATION"='email')
   7 - access("C"."ID_IND"="I"."ID_IND")
   8 - access("P"."ID_PARENT"="I"."ID_IND")

Note
-----
   - dynamic sampling used for this statement (level=2)
```

Gains et pertes

Les indexes sont des structures organisées contenant des données. Lors d'une modification sur les données de la table concernées par cet index, l'index est modifié afin de prendre en compte cette modification.

Exemple : `UPDATE modele SET id_modele=id_modele+100 WHERE id_modele BETWEEN 10 and 20;`

Si un index est posé sur la colonne `id_modele` de la table `modele`, pour chaque ligne modifié, une feuille de l'index est supprimée puis une autre feuille est ajoutée. Ces accès peuvent pénaliser les performances de la base.

Avantages Accélération des performances en lecture et accélération des performances en `DELETE` et `UPDATE` (si clause `WHERE`) pour aller chercher les lignes à modifier ou supprimer.

Inconvénients Pertes de performances dans certains cas sur les commandes DML.

Exemple 7.5. Soit une table contenant 1 million de lignes.

— Si suppression d'une ligne bien identifiée par l'index : **gain** !

— Si suppression de 70% des lignes : **perte** !

Statistiques

Oracle utilise sur des statistiques pour savoir le coût d'utilisation des index et autres objets concernés par les requêtes SQL. Il peut arriver que le plan d'exécution ne soit pas optimal, il faut mettre à jour les statistiques.

`ANALYZE TABLE modele COMPUTE STATISTICS;`
récolte de statistiques rigoureuses
`ANALYZE TABLE modele ESTIMATE STATISTICS;`
récolte de statistiques à partir d'un panel de lignes.
`Execute dbms_stats.gather_schema_stats(ownname=>'IUT2');`
calcul des statistiques sur tous les objets d'un user.

Exemple 7.6. Soit l'exécution des commandes suivantes.

```
Set autotrace traceonly

SELECT id_modele
FROM modele
WHERE conso=3.7;

=> Table access full
```

Création d'un index sur la colonne conso

```
CREATE INDEX modele_idx_1 ON modele (conso);

=> Utilisation de l'index plus lecture partielle de la table.
```

Exemple 7.7.
```
SELECT id_modele
FROM modele
WHERE id_moteur IN (SELECT id_moteur
FROM moteur
WHERE puissance>400) ;

=> Accès full aux 2 tables
```
Création d'un index sur `puissance`

```
CREATE INDEX moteur_idx_1 ON moteur(puissance);
=> Utilisation de l'index sur moteur et lecture complète de la table MODELE
```

Création d'un index sur `id_moteur` de la table `MODELE`.

```
CREATE INDEX modele_idx_3 ON modele(id_moteur);
```

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |              |     7 |   224 |     7   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                 |              |       |       |            |          |
|   2 |   NESTED LOOPS                |              |     7 |   224 |     7   (0)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID | MOTEUR       |     2 |    52 |     2   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN          | MOTEUR_IDX_1 |     2 |       |     1   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN           | MODELE_IDX_3 |     4 |       |     1   (0)| 00:00:01 |
|   6 |   TABLE ACCESS BY INDEX ROWID | MODELE       |     4 |    24 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("PUISSANCE">400)
   5 - access("ID_MOTEUR"="ID_MOTEUR")
```

Exemple 7.8. `Set autotrace traceonly`

```
SELECT id_modele
FROM modele
WHERE conso=3.7;
```

```
=> Utilisation de l'index + lecture partielle de la table.
```

Suppression de l'index `modele_idx_1`.

```
DROP INDEX modele_idx_1;
```

```
=> Table access full
```

Création d'un index sur `conso` ET `id_modele`.

```
CREATE INDEX modele_idx_2 ON modele (conso, id_modele);
```

```
=> Utilisation uniquement de l'index sans lire la table.
```

Exemple 7.9 (Comparaison des temps d'exécution)**.** Lancer les commandes suivantes pour créer une table ayant beaucoup de lignes.

```
CREATE TABLE obj AS SELECT * FROM all_objects;
INSERT INTO obj SELECT * FROM obj;
```

Lancer 10 fois ou plus la commande insert pour avoir un grand nombre de lignes et effectuer la requête suivante :

```
SELECT distinct owner
FROM (
SELECT a.*
FROM obj a, obj b
WHERE a.owner=b.owner)
WHERE object_name='MODELE';
```

Index uniquement sur `owner` :

```
DROP INDEX obj_idx_1;
CREATE INDEX obj_idx_1 ON obj (owner);
=>  Une lecture full et une lecture index
```

Index sur `owner` et `object_name` :

```
DROP INDEX obj_idx_1;
CREATE INDEX obj_idx_1 ON obj (owner, object_name);
```

Exercice 7.3 (Plan)**.**

1. Compter le nombre de lignes dans la table `TYPE`.

2. Afficher la ligne pour laquelle la désignation est 'Email'

3. Quel est le plan d'exécution.

4. Indiquer quel index pertinent pourrait améliorer le temps de traitement de cette requête et vérifier la pertinence.

Exercice 7.4. Vérifier que la requête suivante qui donne tous les enfants dont la mère s'appelle 'Annie' ou le père est né avant le 1er janvier 1950 est bien écrite.

```
SELECT i.id_ind, i.prenom||' '||i.nom||' est l''enfant de
                '||ip.prenom||' '||ip.nom||' et
                '||im.prenom||' '||im.nom as "parents"
FROM individu i, parent p, individu ip, parent m, individu im
WHERE p.id_enfant=i.id_ind
AND m.id_enfant=i.id_ind
AND ip.id_ind=p.id_parent
AND im.id_ind=m.id_parent
AND ip.sexe='M'
AND im.sexe='F'
AND  im.prenom='Annie'
OR ip.ddn < to_date('01/01/1950','DD/MM/YYYY')
ORDER BY i.nom, i.prenom;
```

Définissez les index pertinents et donner le plan d'exécution.

Index

ABS, 35
ACCOUNT
    LOCK, 32
    UNLOCK, 32
ACID, 53
ADD_MONTHS, 36
ALTER, 25
Arbre, 66
Association, 4
    hiérarchique, 5
    faible, 5
    forte, 5
    non hiérarchique, 5
    réflexive, 6
      hiérarchique, 7
      non hiérarchique, 6
    ternaire, 7
Atomicité, 54
Attribut, 4
AVG, 38

B-tree, 66
BFILE, 24
BINARY_INTEGER, 42
BLOB, 24
BOOLEAN, 42

Cardinalité, 4
CASACADE CONSTRAINT, 25
CASE, 40
CEIL, 35
CHAR, 24
CHECK, 25
Clé, 8
    étrangère, 8
CLOB, 24
Cohérence, 54
CONCAT, 35
Contrainte d'Intégrité Fonctionnelle, 5
COUNT, 38
CREATE, 25
Curseur
    explicite, 48
    implicite, 48
CYCLE, 31

Déclencheur, 50
Dépendance fonctionnelle, 5
Dépendance fonctionnelle, 9
    élémentaire, 9
    directe, 10
DATE, 24
DECODE, 40
DEFAULT, 25
DEFAULT TABLESPACE, 32
Domaine d'un attribut, 4
Durabilité, 54

Entité, 4
EXCEPTION, 42, 45

FALSE, 42
Feuille, 66
FLOOR, 35
Fonction
    personnalisée, 39
FOREIGN KEY, 26
Forme Normale, 9

GRANT, 33

Identifiant, 4
INCREMENT BY, 31
Index, 65
    Bitmap, 66
INITCAP, 35
Isolation, 54

LAST_DAY, 36
LENGTH, 35
LONG, 24
LOWER, 35
LPAD, 36
LTRIM, 36

MAX, 38
MAXVALUE, 31
MCD, 3
Merise, 3
MIN, 38
MINVALUE, 31
MLD, 3
MOD, 35
MONTHS_BETWEEN, 36

Nœud, 66
NATURAL, 42
NEXT_DAY, 36
NOCYCLE, 31
Norbert Wiener, 3

NOT NULL, 25
NULL, 25, 42
NUMBER, 24
NVL2, 39

ON DELETTE CASCADE, 26

PASSWORD EXPIRE, 32
PL/SQL, 41
Plan d'exécution, 69
POWER, 35
PRIMARY KEY, 25
Privilège, 33
Produit
    cartésien, 65
PROFILE, 32
pseudo-colonnes, 24

QUOTA, 32

Référence, 8
REFERENCES, 26
Relation, 4
REPLACE, 36
REVOKE, 33
ROLLBACK, 54
ROUND, 35, 36
rowid, 24
rownum, 24
RPAD, 36
RTRIM, 36

Séquence, 31
SIGN, 35
SQLCODE, 46
SQLERRM, 46
START WITH, 31
SUBSTR, 35
SUM, 38
Synonyme, 31
SYSDATE, 36

TEMPORARY TABLESPACE, 32
TO_CHAR, 37
TO_DATE, 37
TO_NUMBER, 37
Transitivité, 10
TRIGGER, 50
TRUE, 42
TRUNC, 35, 36
Tuple, 8

Undo, 54
UNIQUE, 25
UPPER, 35
Utilisateur, 32

VARCHAR2, 24
Verrou, 55
Vue, 29
    matérialisée, 30

Commandes de TP
TODO