2021-2022

Université Clermont Auvergne
IUT Département Informatique
2ème année

Bases de Données Avancées

Raphaël Delage, Anaïs Durand, Franck Glaziou, Pascal Lafourcade

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'accès 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 Confucius :

"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.

La note finale NF est calculée avec la formule suivante :

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

où DS correspond à la note du Devoir Surveillé de 1h30 en amphi sur feuille avec comme seul document autorisé une feuille A4 recto-verso de notes personnelles manuscrites, et CC correspond au Contrôle Continu. La note de CC est calculée avec la formule suivante :

$$CC = 100\%TP$$

où TP correspond à la note obtenue pour les Travaux Pratiques sur machine. Il y a 4 TP à rendre et 2 seront tirés au hasard et seront corrigés a posteriori pour tous les groupes.

▶ **Travaux Pratiques :** Pour accéder aux sujets de TP vous pouvez les consulter sur les deux sites web suivants :

`https://sancy.iut-clermont.uca.fr/~lafourcade/BD2A.html`

et

`https://pretoria.iut-clermont.uca.fr/BD2A/`

Pour rendre les TPs vous avez une semaine après le TP, vous devez utiliser le serveur `pretoria`, mais aussi envoyer par email à vos enseignants les fichiers de TP.

Plan du cours

1. Modélisation de 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 structure 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 1.1 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. Il est important de prendre du temps pour la conception des bases de données car les choix faits lors de cette phase vont significativement impacter la phase d'exploitation des données.

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

Table 1.1 – 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 concepts 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 est notée n.

Exemple 1.2 : Dans la Figure 1.1, 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.

Remarque 1.1. Le numéro de Sécurité Sociale est absolument interdit en tant qu'identifiant d'entité (voir CNIL 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

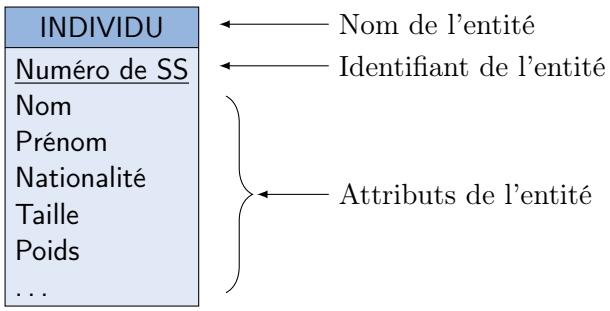


FIGURE 1.1 – Exemple d'entité.

Exemple 1.3 : L'association participer à de la Figure 1.2, 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 $0, n$. À un examen participe au minimum un étudiant et au maximum n, d'où la cardinalité $1, n$.

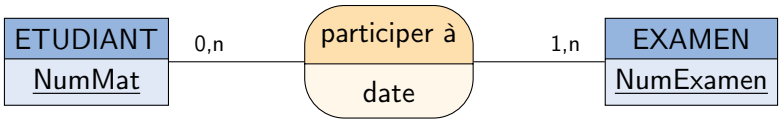


FIGURE 1.2 – Exemple d'association.

1.3. Associations

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

1.3.1. 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 1.3, un client peut acheter 1 à n articles et un article peut être achetée par 0 ou n client.

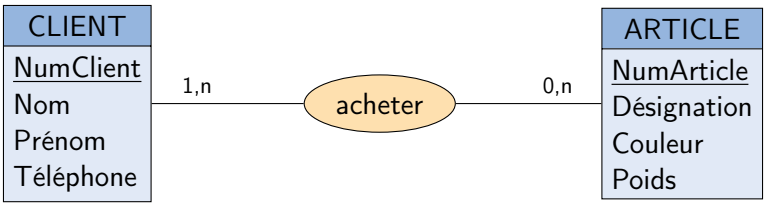


FIGURE 1.3 – Exemple de relation non hiérarchique.

1.3.2. *Association hiérarchique*

Définition 1.3 – Association hiérarchique

Une ***association hiérarchique*** 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 «*fils* », il est possible de retrouver l'occurrence de l'entité «*père*».

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 : Dans la relation de la Figure 1.4, 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.

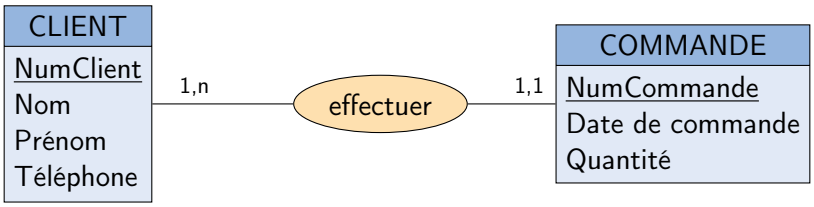


FIGURE 1.4 – Exemple d'association hiérarchique.

Exemple 1.6 : La Figure 1.5 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.

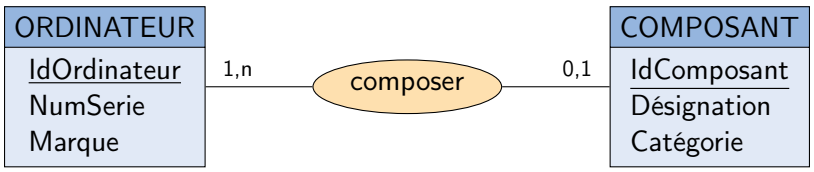


FIGURE 1.5 – Exemple d'association hiérarchique faible.

1.3.3. *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 : Dans l'association réflexive non hiérarchique de la Figure 1.6, une molécule peut être composée de plusieurs molécules.

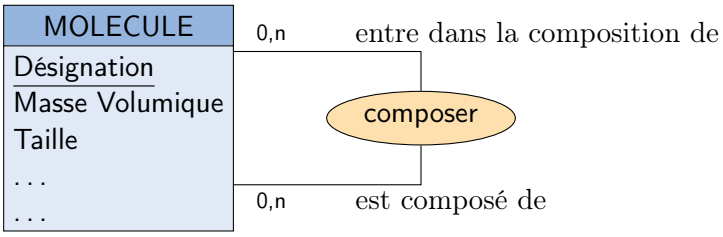


FIGURE 1.6 – Exemple d'association réflexive non hiérarchique.

▶ **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 1.7, un salarié a pour supérieur direct un autre salarié.

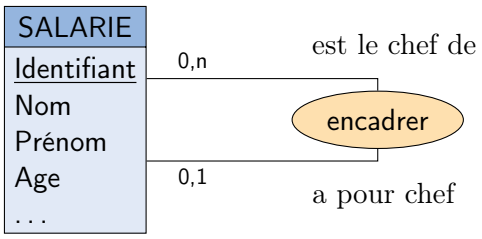


FIGURE 1.7 – Exemple d'association réflexive hiérarchique.

1.3.4. *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 1.8, 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.

Cette relation est équivalente à celle de la la Figure 1.9.

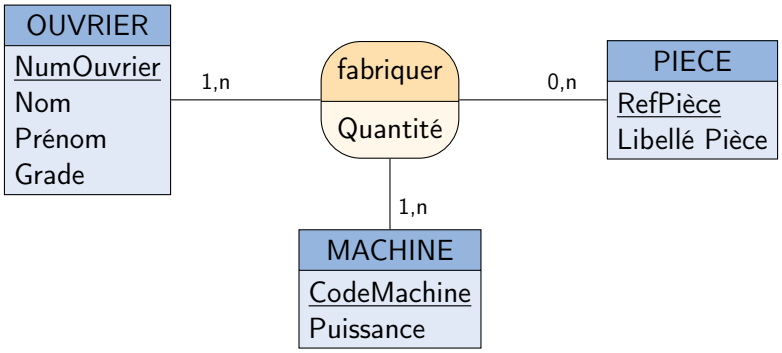


FIGURE 1.8 – Exemple de relation ternaire qui sont équivalentes.

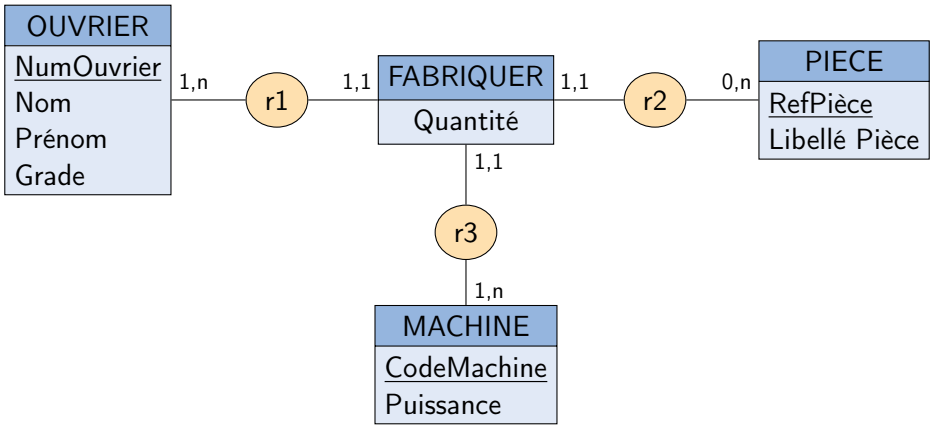
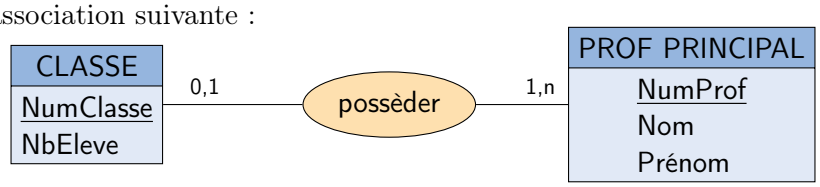


FIGURE 1.9 – Exemple de relation ternaire qui sont équivalentes.

Exercice 1

Caractériser l'association suivante :

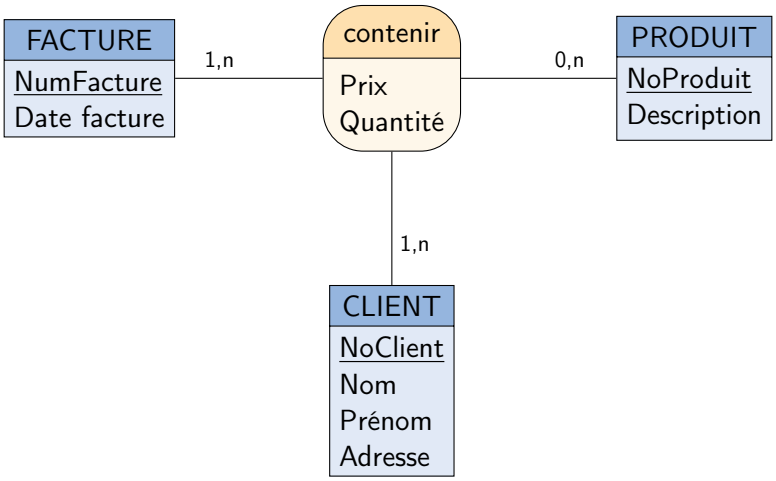


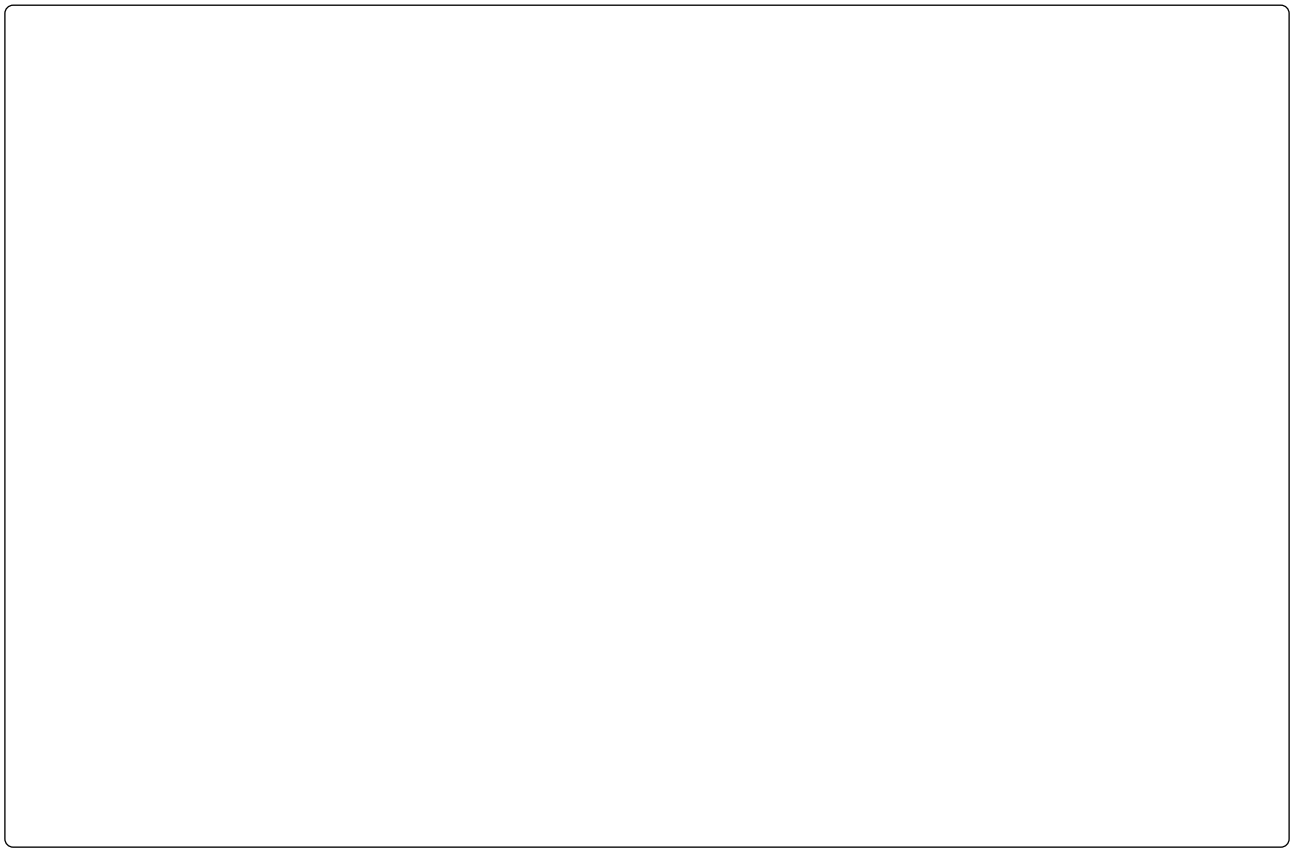
Exercice 2

Donner deux exemples d'association reflexive, une hiérarchique et un non-hiérarchique.

Exercice 3

Est ce que cette association ternaire est juste ? Si oui expliquez pourquoi, sinon donner l'association qui est correcte :





1.4. Du MCD au modèle relationnel

1.4.1. 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 :

$$< \text{NOM DE LA RELATION} > (\text{attribut}_1, \text{attribut}_2, \ldots, \text{attribut}_n)$$

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 *p*remière 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 : COMMANDE(numérocommande, datecommande, #numéroclient)
   - Clé primaire : numérocommande
   - Clé étrangère : numéroclient référençant CLIENT

1.4.2. *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 1.10 : Dans la Figure 1.10, l'entité COMMANDE avec comme identifiant NumCommande devient la relation COMMANDE avec comme clef primaire NumCommande.

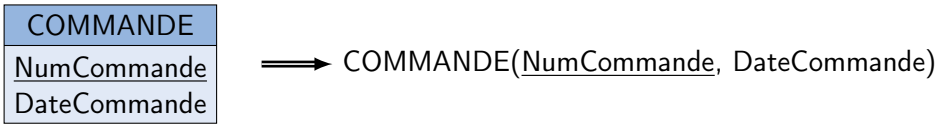


FIGURE 1.10 – Exemple de l'application de la règle 1.

▶ **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 1.11 : Dans la Figure 1.11, l'association hiérarchique CLIENT impose que le champs NumClient soit une clé étrangère dans la relation COMMANDE.

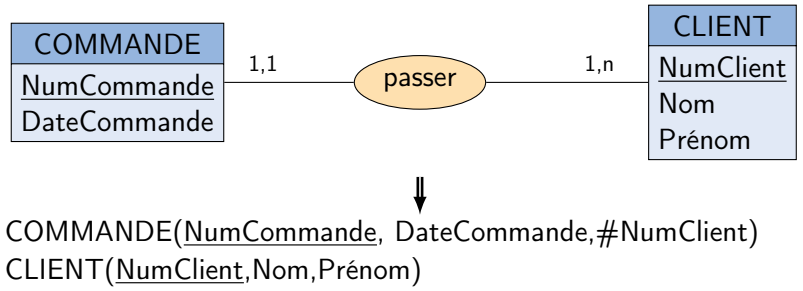


FIGURE 1.11 – Exemple d'application de la règle 2.

▶ **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 1.12 : Dans la Figure 1.12, 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 composée des deux clés primaires des relations COMMANDE et ARTICLE. Par construction ces champs sont aussi des clés étrangères.

Remarque 1.2. 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.

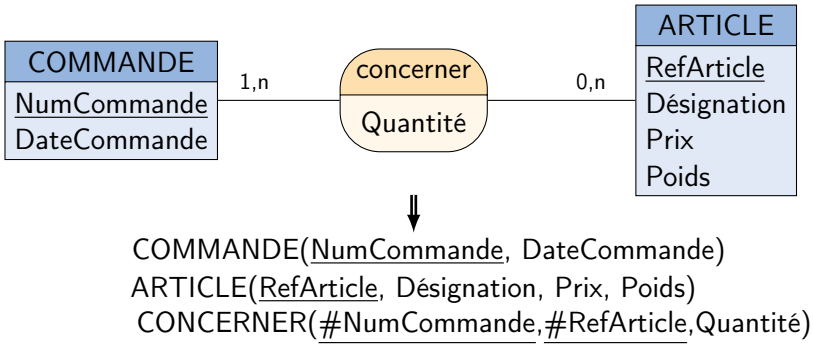


FIGURE 1.12 – Exemple d'application de la règle 3.

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

Exemple 1.13 : Dans la Figure 1.13, 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). Il est donc possible d'appliquer la règle 1 : l'identifiant de l'entité devient clé primaire. Il est aussi

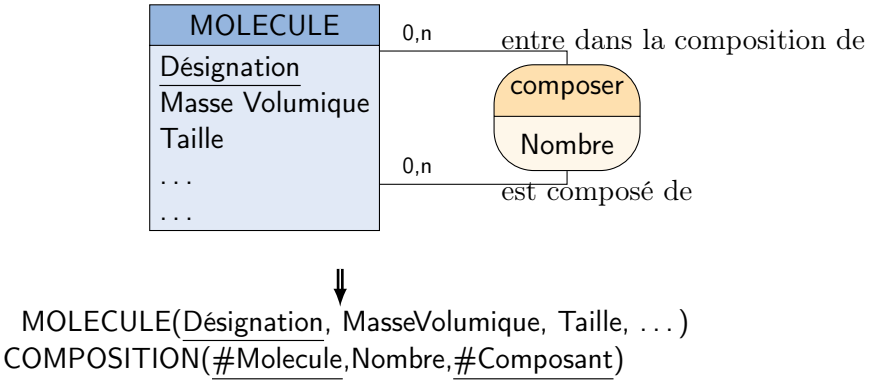


FIGURE 1.13 – Exemple de transformation d'une association réflexive non hiérarchique, où les attributs suivants sont égaux : composant = Désignation.

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.

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 1.14 : Dans la Figure 1.14, 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.

▶ **Passage du MCD au MLD.**

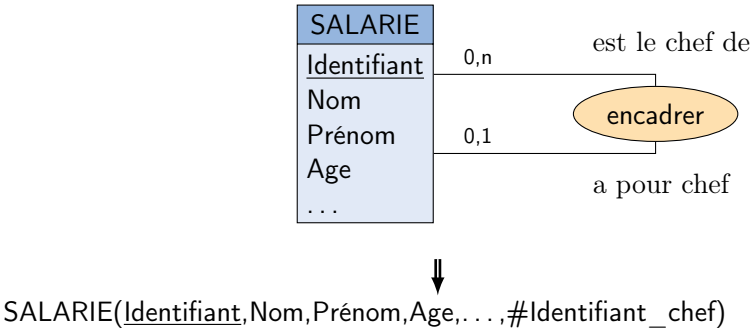
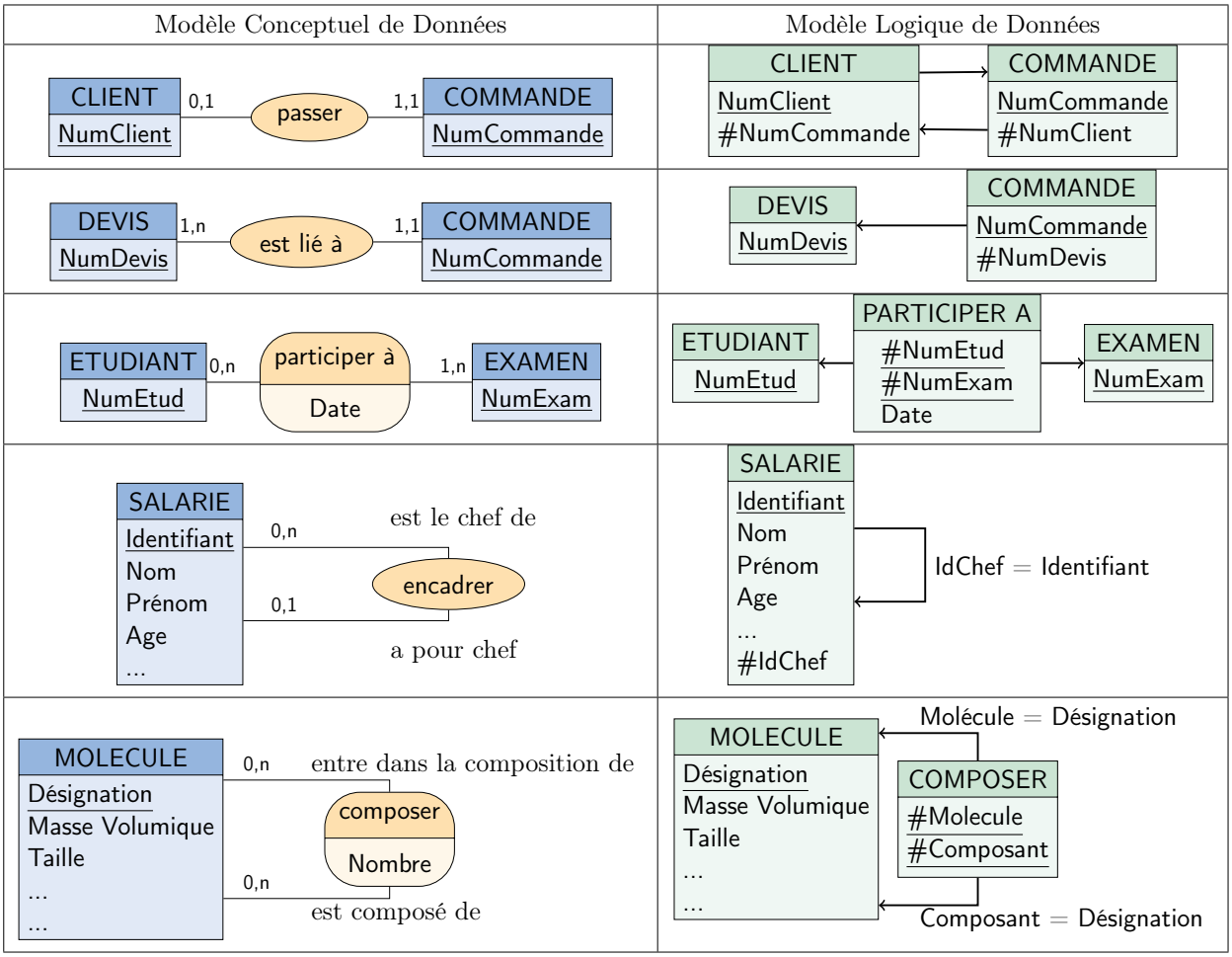


FIGURE 1.14 – Exemple de transformation d'une association réflexive, où les attributs suivants sont égaux identifiant = identifiant_chef.



Remarque 1.3. 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 1.4. Dans la Figure 1.15, le résultat pour le passage du MCD au MLD est donné pour l'association ternaire indiquée dans la Figure 1.8. 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.

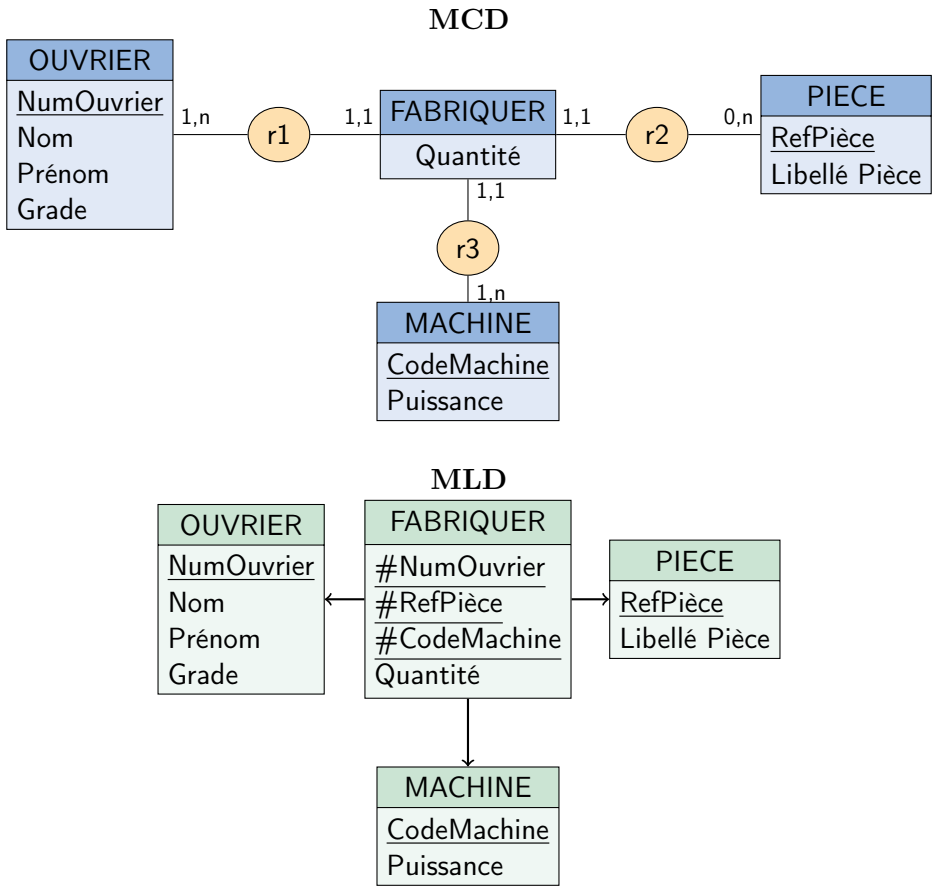


FIGURE 1.15 – Exemple de transformation d'association ternaire.

Exercice 4

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 les coordonnées courantes et passées des membres de la famille.

Q_1). Réaliser le Modèle Conceptuel de Données correspondant.

Q_2). À partir du MCD obtenu, réaliser le Modèle Logique de Données.

2. Gestion des données redondantes

Les notions de clés primaires et clés étrangères jouent des rôles clefs dans la structure des bases de données.

2.1. Clés primaires et Clés étrangères

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° pièce	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	C55	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 « n° pièce » et « quantité » ou introduire des valeurs nulles (ce qui pose d'autres problèmes).

— **Suppression :** la suppression de la pièce n° 102 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. Dépendances fonctionnelles

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 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 possible de lui associer qu'une seule occurrence de Y.

Définition 2.2 – Dépendance fonctionnelle triviale

Une dépendance fonctionnelle $X \to Y$ est dite triviale si $Y \subseteq X$.

Exemple 2.3 :

— Soient A et B des attributs uniques. $AB \to B$ est une dépendance fonctionnelle triviale.

— Soient A, B et C des attributs uniques. $AB \to C$ est une dépendance fonctionnelle non triviale.

Remarque 2.2. À partir d'une table (ou en d'autres termes d'une relation définie en extension), il est possible de déterminer des dépendances fonctionnelles *a priori*. Pour n'importe quelle paire d'ensembles d'attributs X et Y, il est supposé que $X \to Y$. L'exercice est alors de trouver une contradiction au vu du contenu de la table ! En l'absence de contradiction, Y dépend bien fonctionnellement de X. Comment trouver une contradiction pour une DF $X \to Y$? il suffit d'exhiber deux lignes (i.e. enregistrements) telles que l'égalité des valeurs pour l'attribut X n'implique pas l'égalité de valeurs pour l'attribut Y.

Exemple 2.4 : Énumérer toutes les dépendances fonctionnelles non triviales satisfaites par la table (ou relation en extension) suivante :

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

Les dépendances fonctionnelles non triviales sont : A $\rightarrow$ B et C $\rightarrow$ B , et en tant que dépendances fonctionnelles elles impliquent logiquement : AC $\rightarrow$ B. Ensuite, C ne détermine pas fonctionnellement A car les premier et troisième tuples ont le même C mais des valeurs A différentes. Les mêmes tuples montrent également que B ne détermine pas fonctionnellement A . De même, A ne détermine pas fonctionnellement C parce que les deux premiers tuples ont la même valeur A et des valeurs C différentes. Les mêmes tuples montrent également que B ne détermine pas fonctionnellement C.

Exercice 5

Énumérer les dépendances fonctionnelles non triviales (avec un seul attribut but) satisfaites par la relation suivante :

A_1	A_2	A_3	A_4
1	b	vache	rouge
1	c	cochon	rouge
4	b	vache	rouge
3	a	mouton	bleu
5	a	mouton	jaune

Définition 2.3 – Dépendance fonctionnelle élémentaire

Une dépendance fonctionnelle $X \rightarrow A$ est *élémentaire* si elle est non triviale et si :

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

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.

Exemple 2.5 :

— $AB \rightarrow C$ est élémentaire si ni A, ni B pris individuellement ne déterminent C. Par exemple : Nom, DateNaissance, LieuNaissance → Prénom est élémentaire.

— $AB \rightarrow A$ n'est pas élémentaire car A est incluse dans AB.

$AB \rightarrow CB$ n'est pas élémentaire car CB n'est pas un attribut, mais un groupe d'attributs.

Exemple : NumeroSS → Nom, Prénom, n'est pas élémentaire.

Exercice 6

Établir les dépendances fonctionnelles d'une relation IDENTITE ayant pour attributs (Nom, AnneeDeNaissance, CodePostal, Ville, Departement, NumeroDeTelephoneFixe, NumeroDeSecurite-Sociale), dans les deux cas suivants :

1. Seule la résidence principale de l'intéressé est enregistrée.
2. Ses résidences secondaires sont également enregistrées.

▶ **Les règles d'Armstrong.** Les règles suivantes permettent de trouver toutes les dépendances fonctionnelles logiquement induites par un ensemble de dépendances fonctionnelles :

- **Règle de la réflexivité**. Si X est un ensemble d'attributs et $Y \subseteq X$, alors $X \to Y$. Les dépendances de ce type sont des **dépendances fonctionnelles triviales**.
- **Règle d'augmentation**. Si $X \to Y$ et que Z est un ensemble d'attributs, alors $ZX \to ZY$.
- **Règle de transitivité**. Si $X \to Y$ et $Y \to Z$, alors $X \to Z$.
- **Règle de l'union**. Si $X \to Y$ et $X \to Z$, alors $X \to YZ$.
- **Règle de décomposition**. Si $X \to YZ$, alors $X \to Y$ et $X \to Z$.
- **Règle de la pseudotransitivité**. Si $X \to Y$ et $WY \to Z$, alors $WX \to Z$.

Définition 2.4 – 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*.

Définition 2.5 – Clé candidate

Une *clé candidate* est un ensemble d'attributs permettant de rendre chaque tuple unique dans une relation.

Il n'y a pas qu'une clef candidate pour une relation. Une clé candidate fait forcément partie de l'ensemble des clefs primaires.

Exercice 7

Étant donnée une relation R(A, B, C, D) satisfaisant les dépendances fonctionnelles suivantes :

- DF1. $AB \to C$
- DF2. $C \to D$
- DF3 $D \to A$

Q_1). Quelles sont les dépendances fonctionnelles non triviales qui se déduisent de ces dépendances fonctionnelles ? (se restreindre aux dépendances ayant un seul attribut en partie droite)

Q_2). Quelles sont les clés candidates de R ?

2.3. Le problème de la décomposition sans perte

Soit T(X) une table et soient les tables T1(Y) et T2(Z) formant une décomposition de T(X), c'est-à-dire que, en considérant X, Y et Z comme des groupes d'attributs, $X = Y \cup Z$. La décomposition est une *décomposition sans perte* s'il n'y a pas de perte d'information en remplaçant T par les deux tables T1 et T2.

Il y a perte d'information s'il est possible que la table T(X) comprenne des informations non représentables avec les tables T1(Y) et T2(Z).

Définition 2.6 – Décomposition sans perte

Soit la décomposition en $X = Y \cup Z$, des tables $T1(Y)$ et $T2(Z)$ à partir de $T(X)$. La décomposition est *sans perte* si, quelles que soient les informations contenues dans la base de données, la table T contient les mêmes enregistrements (tuples) que le résultat de la requête SQL suivante. Pour simplifier il n'y a que le champs `W = Y` $\cap$ `Z` commun entre `Y` et `Z`, il est bien entendu possible d'avoir plusieurs champs :

```
SELECT *
FROM (SELECT Y FROM T) T1, (SELECT Z FROM T) T2
WHERE T1.W = T2.W;
```

Remarque 2.3. Ceci est énoncé plus succinctement en algèbre relationnelle, de la façon suivante où $\bowtie$ dénote la jointure naturelle entre les deux tables :

$$\prod_{Y}(\mathrm{T}) \bowtie \prod_{Z}(\mathrm{T}) = \mathrm{T}$$

Inversement, une décomposition est avec perte si, lorsque la jointure naturelle des résultats des projections donne un sur-ensemble propre de l'ensemble des enregistrements de la table d'origine. Ceci est énoncé plus succinctement en algèbre relationnelle :

$$\mathrm{T} \subsetneq \prod_{Y}(\mathrm{T}) \bowtie \prod_{Z}(\mathrm{T})$$

▶ **Un exemple de décomposition *avec perte*.** Soit la table `ETUDIANT (ID, prenom, ville, age)` qui se décompose en les deux tables suivantes :

— `ETUDIANT1 (ID, prenom)`

— `ETUDIANT2 (prenom, ville, age)`.

Le défaut de cette décomposition provient de la possibilité que deux étudiants aient le même prénom. Par exemple, si deux étudiant s'appellent Baptiste, étudient à l'université et ont les tuples suivants dans la table d'origine `ETUDIANT` :

(25423, Baptiste, Marseille, 21)
(25424, Baptiste, Toulouse, 22)

Le contenu des tables `ETUDIANT1` et `ETUDIANT2` correspondent alors, respectivement, à la projection de `ETUDIANT` sur les 2 premiers attributs, et à la projection de `ETUDIANT` sur les 3 derniers attributs. La jointure naturelle de ces deux tables (sur l'attribut prenom) donne la table résultante suivante :

(25423, Baptiste, Marseille, 21)
(25423, Baptiste, Toulouse, 22)
(25424, Baptiste, Marseille, 21)
(25424, Baptiste, Toulouse, 22)

Ainsi les deux tuples originaux apparaissent dans le résultat avec deux nouveaux tuples qui mélangent incorrectement les valeurs des données relatives aux deux étudiants nommés Baptiste. De l'information sur les âges et provenance des personnes sont perdues.

Théorème 2.1 – Théorème de Heath

Soit R(X, Y, Z) une relation où X, Y et Z sont des ensembles d'attributs disjoints. Si R vérifie la dépendance fonctionnelle $X \to Y$ alors sa décomposition en R1(X, Y) et R2(X, Z) est *sans perte* d'information.

Exercice 8

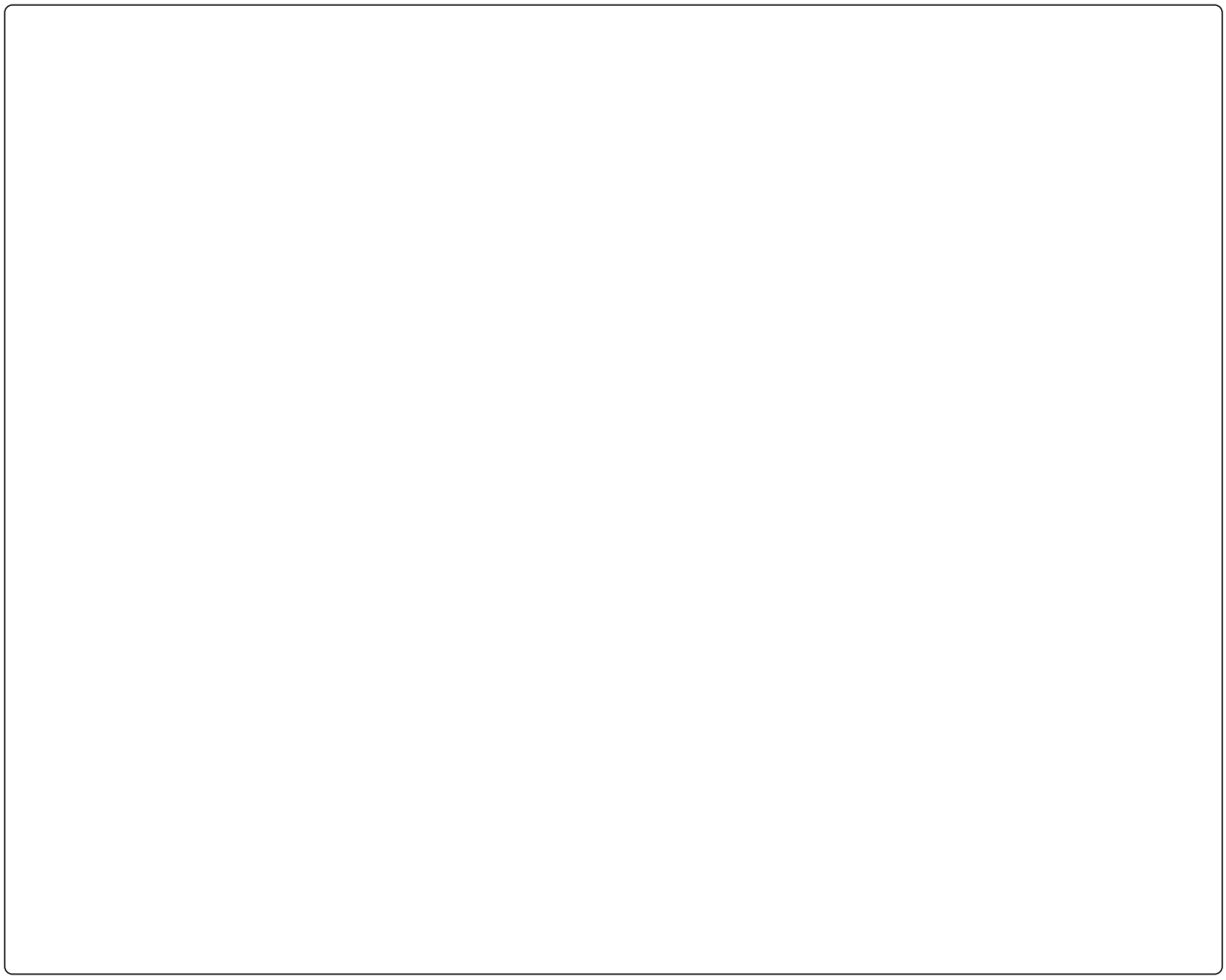
Soit un schéma R(A, B, C, D, E) qui satisfait l'ensemble des dépendances fonctionnelles suivantes :

- $A \to BC$
- $CD \to E$
- $B \to D$
- $E \to A$

Q_1). Montrez que A est une clé candidate pour R.

Q_2). Ce schéma est décomposé en R1(A, B, C) et R2(A, D, E). Montrer que cette décomposition est une décomposition sans perte.

Q_3). Ce schéma est maintenant décomposé en R1(A, B, C) et R2(C, D, E). Montrer que cette décomposition n'est pas une décomposition sans perte.
Suggestion : Trouver une table pour R *telle que* $\prod_{A,B,C}(\mathrm{R}) \bowtie \prod_{C,D,E}(\mathrm{R}) \neq R$



2.4. Première forme normale : 1FN

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 2.7 – 1FN

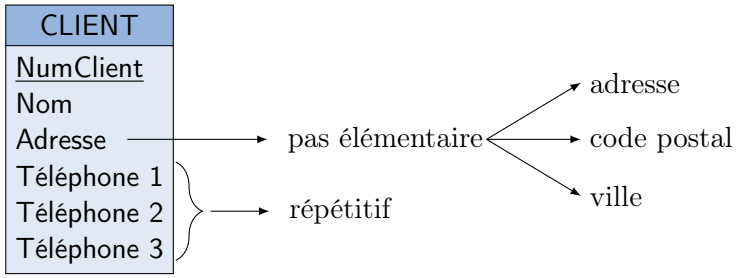
Une relation (une entité) est en 1FN si et seulement si tout attribut contient une valeur atomique.

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 2.6 :

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 9

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	C55	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

Exercice 10

Écrire une relation `COMMANDE V1` avec les données de `COMMANDE V0` qui soit en 1FN.

2.5. Deuxième forme normale : 2FN

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 2.8 – 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 avec **l'ensemble** du groupe définissant la clé.

Par exemple une relation avec comme identifiant `A` et `B`, si `B` → `C` alors la relation n'est pas en 2FN.

Exercice 11

Expliquer pourquoi la relation `COMMANDE V1` n'est pas en 2FN.

2.6. Troisième forme normale : 3FN

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

Définition 2.9 – 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.

Plusieurs exemples :

— Exemple d'une relation avec comme identifiant `A` et `B`, si `C` → `D` alors la relation n'est pas en 3FN.

— Exemple d'une relation avec comme identifiant `A` et `B`, si `C` → `B` alors la relation n'est pas en 3FN.

Exercice 12

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	C55	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

2.7. Quatrième Forme Normale : 4FN

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

Définition 2.10 – 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.

Par exemple une relation avec comme identifinat `A` et `B`, si `A` $\rightarrow$ `B` alors la relation n'est pas en 4FN.

Exercice 13

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

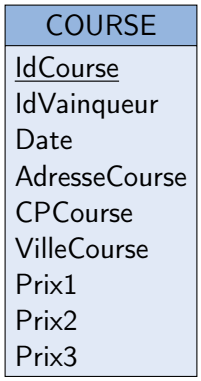
`COMMANDE V4 :`

# Nom four	N° com	N° client	Date com
DURAND	C54	15	12/01/2016
DUPONT	C55	15	10/05/2016
BRUNEAU	C17	18	01/08/2016
DURAND	C574	85	30/12/2015

Exercice 14

— Indiquer quelle FN respecte le MCD ci-dessous.

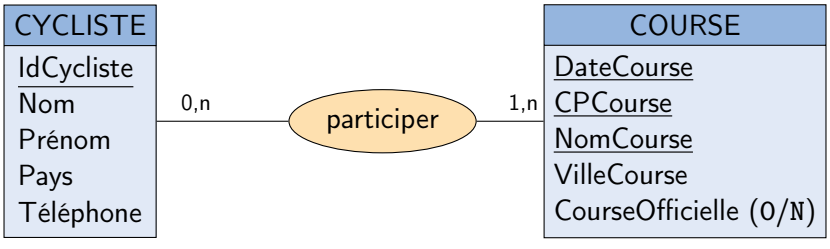
— Indiquer les modifications à effectuer pour le MCD respecte la FN supérieure.



Exercice 15

— Indiquer quelle FN respecte le MCD ci-dessous.

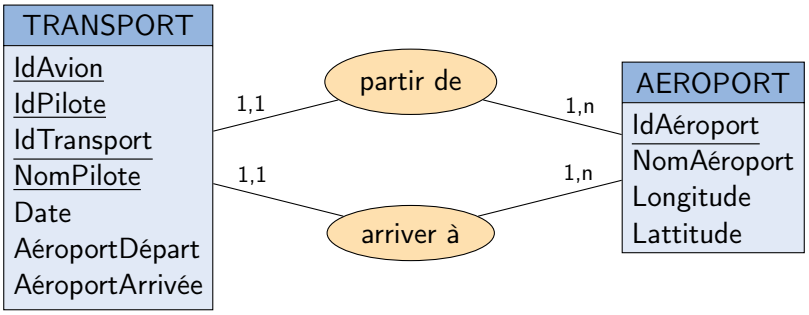
— Indiquer les modifications à effectuer pour le MCD respecte la FN supérieure.



Exercice 16

— Indiquer quelle FN respecte le MCD ci-dessous.

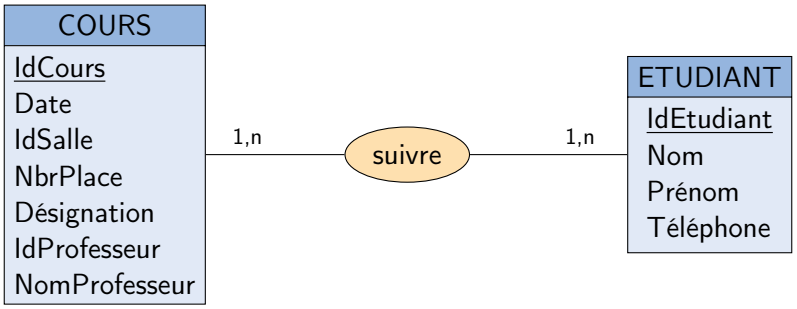
— Indiquer les modifications à effectuer pour le MCD respecte la FN supérieure.



Exercice 17

— Indiquer quelle FN respecte le MCD ci-dessous.

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



Exercice 18

Soit le schéma relationnel suivant :
ELEMENT_FACTURATION(NumCommande, NumArticle, Description, Prix, Quantite)

Q_1). Trouvez les dépendances fonctionnelles et une clé candidate pour la relation ci-dessus.

Q_2). Quelle est la forme normale de cette relation ? Détaillez votre réponse.

Q_3). Quels sont les inconvénients du schéma choisi dans l'énoncé ?

Exercice 19

Le schéma relationnel suivant décrit des événements musicaux en France :
Hypothèses :

- Chaque événement a au moins un chanteur.
- Les chanteurs s'en tiennent à un genre de musique et ne se rendent pas deux fois la même année au même endroit.

Lieu	Annee	Chanteur	Genre
V	2004	The Corrs	Pop
W	2007	Coldplay	Pop
X	2010	Norah Jones	Jazz
Y	2016	Muse	Rock
Z	2017	Fréro Delavega	Pop

Q_1). Cette relation est-elle en 2FN ? 3FN ? Déterminez les dépendances et les clés fonctionnelles et justifiez votre réponse.

Q_2). Le schéma est maintenant modifié pour inclure le nombre de spectateurs :

Lieu	Annee	Chanteur	Spectateurs
V	2004	The Corrs	7 000
W	2007	Coldplay	10 000
X	2010	Norah Jones	6 000
Y	2016	Muse	10 000
Z	2017	Fréro Delavega	7 000

Le nouveau schéma est-il en 2FN ? 3FN ?

Exercice 20

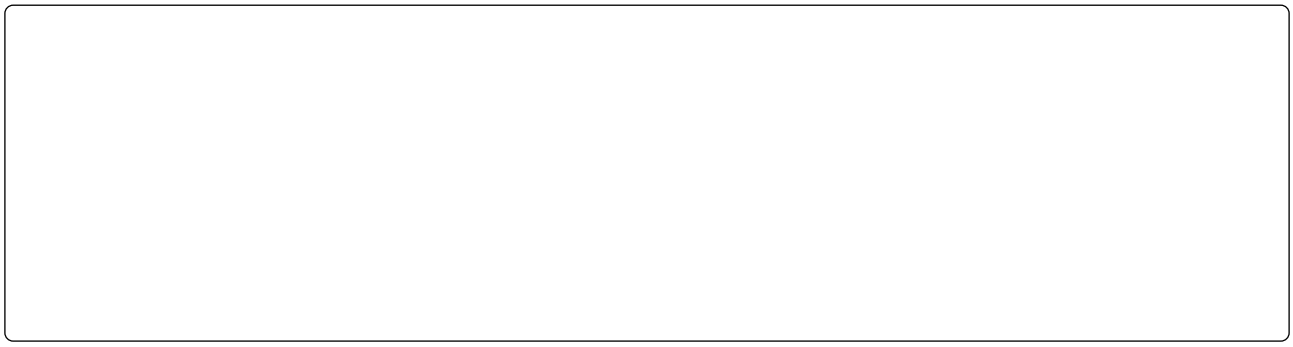
Soit le schéma relationnel qui concerne les employés d'une compagnie implantée sur plusieurs bâtiments :

EMPLOYE(NumEmp, Nom, Salaire, Departement, Batiment)

Q_1). Il est pris pour hypothèses qu'un employé travaille dans un département donné, et qu'aucun département ne possède des locaux dans plusieurs bâtiments.

Déterminez les dépendances fonctionnelles.

Q_2). En quelle forme normale est le schéma ? Le mettre en 3FN le cas échéant.



3. Contraintes et données

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 aux 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 stockées 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`. À 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.	Stockée 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`.

3.1.1. *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.

▶ **`rowid`.** Les données sont stockées dans des tables, mais le fonctionnement interne d'oracle utilise un identifiant unique par tuple, quelque soit la table : le `rowid`.

Comme cela sera abordé dans le Chapitre 8, dans certains plans d'exécution le terme `access by rowid` apparît, celà signifie qu'Oracle a reperé le `rowid` d'un tuple à partir d'un index par exemple, et qu'il accède au tuple entier dans la table par son `rowid`. Car c'est la manière la plus rapide pour accéder aux données, comme dans l'exemple **??**.

Cependant, il n'est pas viable de baser son applicatif sur cet identifiant qui est propre à oracle qui peut être modifié en cas de réorganisation de la base, mais il est possible, dans le cadre d'optimisation, de l'utilisé dans une procédure stockée ou une sous-requête par exemple, la détermination du `rowid` étant faite dans une même transaction. En effet, lors de réorganisation de base par exemple, oracle réattribue ses `rowid` afin de mieux les ordonnancer au sein des tablespaces.

Exemple 3.1 : Soit une table `T (C1, C2, C3)` ne contenant pas d'index unique et de clé primaire, avant de créer une clé primaire sur les deux premières colonnes, il faut s'assurer qu'aucun doublon n'existe sur le couple `(C1, C2)`. Pour cela, il est possible d'utiliser une requête s'appuyant sur `rowid` comme suit :

```
SELECT *
FROM T a
WHERE (C1, C2) IN (SELECT C1, C2
                   FROM T b
                   WHERE b.rowid <> a.rowid);
```

▶ **`rownum`.** Le `rownum` est quant à lui propre au résultat de chaque requête, il s'agit en quelque sorte du numéro de la ligne du résultat dans l'ordre de retour. Ainsi, lorsque l'information ne dépend pas du nombre de lignes ramenées, il est utilisé afin qu'Oracle ne satisfasse pas toute la requête mais s'arrête au premier enregistrement trouvé (un ordre `hint first_row` est alors utilisé).

Remarque 3.1. Sans `ORDER BY`, l'ordre du résultat n'est pas prédictible.

Exemple 3.2 : Les pseudo-colonnes `rowid` et `rownum` s'utilisent comme ceci :

- `SELECT rowid, PRENOMNOMJ FROM joueur WHERE POSTE='F' AND TAILLE>6.9;`
  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).
- `DELETE stats WHERE rownum=986;`
  efface la ligne 986 de la table stats ;

Une alternative est possible pour créer un index dans un nouveau champs en utilisant la commande suivante :

```
ALTER TABLE T ADD idcom NUMBER GENERATED ALWAYS AS
                IDENTITY(START WITH 1 INCREMENT BY 1);
```

3.2. Manipulation des tables

La table est l'objet principal permettant de stocker des données. Il est possible de créer une table avec la commande **CREATE TABLE**, de supprimer une table avec la commande **DROP TABLE** et de modifier la structure d'une table avec la commande **ALTER TABLE**.

Remarque 3.2. Il ne faut jamais mettre les noms de table entre guillemets. Cela est autorisé mais complexifie énormément les commandes SQL.

Exemple 3.3 :

```
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 CONSTRAINTS**" 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;


CREATE TABLE ma_table(
    <nom_colonne 1> <type de donnee> [contrainte],
    <nom_colonne 2> <type de donnee> [contrainte],
    ...,
    [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** : force 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 3.4 :

```
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> MODIFY (age NUMBER NOT NULL) ;
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);
```

Exemple 3.5 :

```
CREATE TABLE new (
    nu NUMBER CHECK (nu IN (1,2,3,4,5))
);
```

À 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 **EXCEPTIONS**, il est possible de stocker dans une table (à définir), l'ensemble des lignes (**rowid**) posant problème.

3.4. Contraintes référentielles

▶ **Syntaxe :** 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 3.6 :

```
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 21

Soit le MLD de la Figure 3.1 et le modèle relationnel correspondant :
INDIDIVU(IdInd, Nom, Prénom, DateNaissance, LieuNaissance, Sexe)
COORDONNEES (IdCoord, Valeur, #IdInd, #IdType)
TYPE(IdType, Désignation)
PARENT (#IdParent, #IdEnfant)

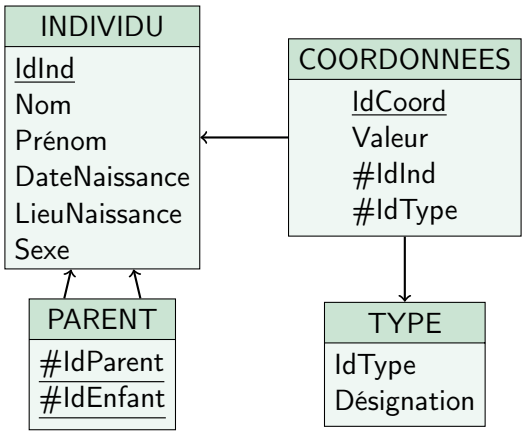
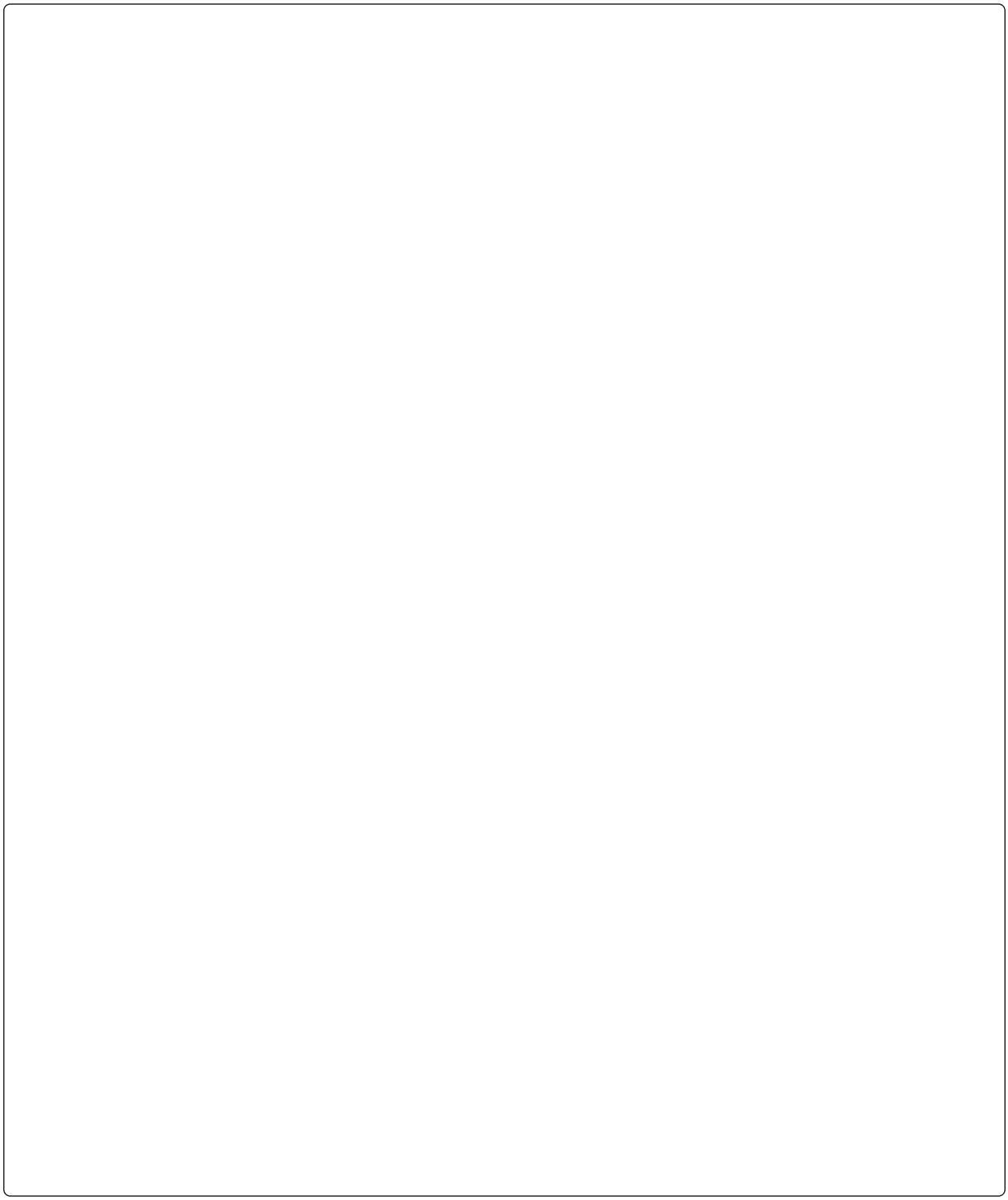


FIGURE 3.1 – MLD de gestion d'arbre généalogique.

À partir du MLD ci-dessus, lister les contraintes référentielles puis écrire les commandes de création des tables et des contraintes référentielles uniquement pour la table **PARENT** et **INDIVIDU**, dans un premier temps, sans commande **ALTER** et dans un second temps avec les commandes **ALTER**.



3.5. 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 3.7 :

```
CREATE SEQUENCE ma_sequence START WITH 100 INCREMENT BY 1 NOCYCLE;
SELECT ma_sequence.NEXTVAL FROM dual;  -- valeur 101 suivante de la sequence.
SELECT ma_sequence.CURRVAL FROM dual;  -- valeur 101 courante de la sequence.
DROP SEQUENCE ma_sequence;
```

Pour définir une colonne en clé primaire :

```
CREATE SEQUENCE ma_sequence START WITH 100 MAXVALUE 1000 INCREMENT BY 1
   NOCYCLE;
INSERT INTO ma_table VALUES (ma_sequence.NEXTVAL,'champ1');
SELECT ma_sequence.NEXTVAL FROM dual; -- valeur suivante de la sequence.
SELECT ma_sequence.CURRVAL FROM dual; -- valeur courante de la sequence.
DROP SEQUENCE ma_sequence;
```

Pour définir une colonne en clé primaire :

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

Exercice 22

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.
Que fait la commande suivante avec cette séquence :

```
SELECT 'Nombre A='||seq_trace.nextval||' B='||seq_trace.nextval
                  ||'C=' ||seq_trace.currval
FROM dual;
```

3.6. Fonctions mono-lignes

Les fonctions mono-lignes agissent sur chaque ligne indépendamment. Ces 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` décimales	`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 nième 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 mième 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(chaine,' ') FROM ma_table;`	DUR Supprime les blancs en fin de chaine
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 le format d'une date est 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.** Il est possible de convertir les différents types entre eux, en utilisant les fonctions suivantes :

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

TO_DATE(chaîne, format)

TO_NUMBER(chaîne)
```

Exemple 3.8 : Cette commande permet d'afficher le jour de la semaine en français.

```
SELECT TO_CHAR(sysdate,'DAY','NLS_DATE_LANGUAGE =FRENCH') FROM dual;
```

Exemple 3.9 : Un exemple d'applications pour chaque catégories de fonction de conversion.

```
SELECT SUBSTR(TO_CHAR(SYSDATE,'dd/mm/yyyy'),1,10) FROM dual; -- 05/09/2021
SELECT TO_DATE('15/12/05','dd/mm/yy') FROM dual;              --   15/12/21
SELECT TO_NUMBER('50') FROM dual;                             --          50
```

Exercice 23

Écrire les requêtes suivantes.

Q_1). Afficher la date sous cette forme : `'nous sommes le mardi 5 septembre 2016. Il est 10:59'`.

Q_2). Afficher la date de demain, mais avec heure, minutes et secondes à 0.

Q_3). Pour un jour donné, il faut écrire la date, le jour et une autre colonne : Si c'est samedi : "c'est le début du we" Si c'est dimanche : "c'est repos" Sinon : "il faut bosser"

3.7. Fonctions de regroupement (Agrégation)

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 <table>
WHERE <conditions>
GROUP BY <colonnes indiquant le regroupement>
HAVING <condition suite au regroupement>;
```

Exemple 3.10 : 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 3.3. 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 3.11 : 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 3.4. La clause "**GROUP BY**" doit référencer toutes les colonnes indiquées dans la clause **SELECT**, hormis les fonctions de regroupement, donc dans notre cas : `id_etudiant`.

Exemple 3.12 : 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 3.5. La clause "`GROUP BY`" fait référence à `id_etudiant` et `id_matiere`.

3.8. Fonctions courantes

▶ **Traitement des valeurs nulles.** `NVL2(x, y, z)` : si `x` n'est pas nul, renvoie `y` sinon renvoie `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
   WHEN decouvert = 0 THEN 'pas de decouvert'
   WHEN decouvert = 2000 THEN 'Eleve'
   WHEN decouvert = 5000 THEN 'Privilegie'
   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`".

4. Consultation

4.1. 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 OR REPLACE VIEW total_salaire AS SELECT SUM(salaire) FROM salaire;
```
— Interrogation d'une vue : `SELECT * FROM total_salaire;`

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

4.1.1. *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 24

Créer la vue "MAMANS" permettant de voir la liste des mamans. Les données doivent s'afficher en lançant la requête suivante :

```
SELECT * FROM mamans;
```

4.2. 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;
```

Exercice 25

Créer un synonyme pour utiliser la table `palafour.nba`.

4.3. SQL dynamique

Le SQL dynamique permet d'écrire le résultat de commande SQL dans un fichier. Cela offre la possibilité de générer des commandes SQL construites à partir de requêtes. Pour cela il faut utiliser la commande `spool`. Cette commande a plusieurs options possibles :

- `SPOOL OUTPUTFILE CREATE;` l'option `CREATE` crée un nouveau fichier `OUTPUTFILE` et tous les résultats des commandes SQL seront écrits dans ce fichier. Si le fichier existe déjà un message d'erreur est affiché.
- `SPOOL OUTPUTFILE APPEND` cette option ajoute les résultats des commandes SQL à la suite du contenu du fichier `OUTPUTFILE`.
- `SPOOL OUTPUTFILE REPLACE` avec cette option tous les résultats des commandes SQL seront écrits dans le fichier `OUTPUTFILE`, le contenu de ce fichier sera écrasé par les nouvelles données.
- `SPOOL OFF` cette commande arrête l'écriture des résultats des commandes SQL dans le fichier choisi.
- `SPOOL OUT` cette commande arrête l'écriture et envoie sur la sortie standard le contenu du fichier créé dynamiquement avec la commande `SPOOL`.

Sans option la commande `SPOOL fichier.sql` se comporte comme avec l'option `REPLACE` et écrit l'ensemble des résultats des requêtes exécutées par la suite dans le fichier `fichier.sql`.

Il est possible d'exécuter les commandes générées avec les requêtes SQL en lançant la commande `@toto.sql;` ainsi créée. Pour éviter d'avoir les commandes entrées par l'utilisateur dans la sortie du `SPOOL`, il faut utiliser la commande `SET ECHO OFF;` et écrire un script SQL de la forme suivante dans par exemple le fichier `script.sql` et le lancer avec la commande `START script.sql` :

```
SPOOL monfichier.sql
  <commandes sql>
SPOOL OFF
```

Remarque 4.1. Les commandes suivantes gérant l'affichage peuvent être utiles :

```
SET ECHO OFF
SET VERIFY OFF
SET FEEDBACK OFF
SET HEADING OFF
```

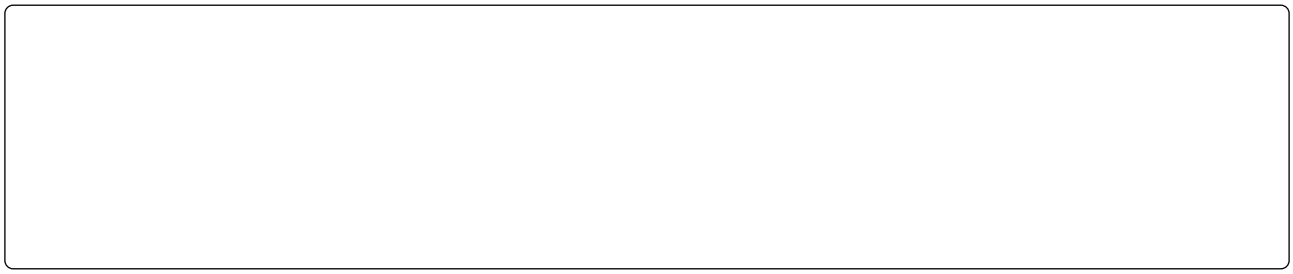
Pour éviter d'afficher les commandes SQL, il faut lancer un fichier .sql.

Exercice 26

Écrire les commandes SQL qui permettent d'écrire dans un fichier texte les noms et les notes des étudiants qui ont plus de 18 ans à partir de la table `ELEVES(NOM, PRENOM, AGE, NOTES)`.

Exercice 27

Écrire les commandes SQL qui permettent d'afficher la description de toutes les tables d'un utilisateur. Sachant que la commande `SELECT TABLE_NAME FROM USER_TABLES` affiche tous les noms des tables de l'utilisateur.

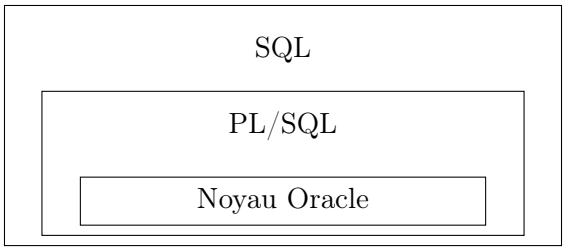


5. Fonctionnalités avancées en PL/SQL

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.



Il permet de créer des fonctions personnalisées, de gérer les exceptions mais aussi de créer des déclencheurs qui réagissent à des actions faites sur la base de données.

5.1. Fonctions personnalisées

Une fonction est un code (PL/SQL) qui peut être appelé 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;
```

Après le **IN** il faut indiquer uniquement type sans spécifier la taille pour les **VARCHAR2** par exemple.

5.2. Description du PL/SQL

Le language PL/SQL est structuré comme suit :

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;
/
```

5.2.1. *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;`
CHAR	Chaîne de caractère de longueur fixe	`Ref CHAR(3);`
VARCHAR2	Chaîne de caractère de longueur variable	`Nom VARCHAR2(30);`

▶ **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. 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'; -- pour avoir une côte dans une chaîne, il faut la
   doubler
    F := E;
END;
/
```

▶ **Exemple d'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;
/
```

5.2.2. *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 :

```
SET ECHO OFF;
SET VERIFY OFF;
SET FEED OFF;

DROP TABLE LOGS;
CREATE TABLE LOGS(
    Champs VARCHAR2(300)
);

VARIABLE vnclient CHAR(3);

PROMPT Saisir un numero de client :
ACCEPT vnclient

DECLARE
    dnumclient VARCHAR2(3);
BEGIN
    dnumclient := 'RDE';
    INSERT INTO LOGS VALUES(dnumclient);
    dnumclient := '&vnclient';
    INSERT INTO LOGS VALUES(dnumclient);
END;
/

SELECT * FROM LOGS;

SET ECHO ON;
SET VERIFY ON;
SET FEED ON;
```

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';
```

▶ **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;`

Dans la suite les syntaxes des principales commande de PL/SQL : instructions conditionnelles, itératives, exceptions ...

5.2.3. *Instructions conditionnelles*

```
IF ... THEN ... END IF;

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

Exemple 5.2 :

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

5.2.4. *Boucle WHILE*

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

Exemple 5.3 :

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

5.2.5. Boucle LOOP

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

Exemple 5.4 :

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

5.2.6. 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;
```

Exemple 5.5 :

```
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`.

5.3. 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.

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

5.3.1. *Exceptions prédéfinies*

▶ **Syntaxe générale :** 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 il faut traiter un maximum d'erreurs possibles, et donner aux 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 avec le message d'erreur sympathique du style :

`"Erreur non prévue no : 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 récupére 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. Il faut 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 5.6 : 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_DIVIDE THEN INSERT INTO result VALUES ('division par zero !!!');
    WHEN OTHERS THEN
        mess1 := SUBSTR(SQLERRM,1,100);
        code1 := sqlcode;
END;
/
```

5.3.2. *Exceptions utilisateur*

La mise en œuvre de ce type d'exceptions est fonction de l'analyse faite. 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.

```
PROMPT 'introduire un age'
ACCEPT age NUMBER

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;
/
```

5.4. 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 packages spécifiques.

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

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

5.4.1. *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;
/
```

5.4.2. *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('Resultats');
    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`, il n'est pas nécessaire de déclarer le curseur, ouvrir le curseur, tester la fin du curseur, faire des `FETCH`, et fermer le curseur.

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

Exemple 5.7 :

```
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;
/
```

Avec un curseur :

```
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;
/
```

5.5. 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** ou **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 à procedure / bloc PL/SQL> : Code à exécuter.

Exercice 28

Q_1). Que fait ce déclencheur ?

```
DROP TABLE ACHAT;
CREATE TABLE ACHAT(
     nocmd NUMBER
);
INSERT INTO ACHAT VALUES(20);

DROP TABLE TRACES;
CREATE TABLE TRACES(
    dd          DATE,
    commentaire VARCHAR2(300)
);

CREATE OR REPLACE TRIGGER trace_delete_commande
    BEFORE DELETE ON commande FOR EACH ROW
BEGIN
   INSERT INTO traces VALUES(SYSDATE,'suppression:'||:OLD.nocmd );
END;
/

DELETE ACHAT WHERE nocmd=20;
SELECT * FROM traces;
```

Q_2). Que se passe-t-il si **BEFORE** est remplacé par **AFTER**

Exercice 29

Que fait ce déclencheur ?

```
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;
/
```

Exercice 30

Que fait ce déclencheur ?

```
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 31

Soit la table et la séquence suivante :

```
CREATE TABLE trace(
    Id_trace NUMBER,
    Trace VARCHAR2(50)
);
CREATE SEQUENCE seq_trace START WITH 1 INCREMENT BY 1;
INSERT INTO TRACE VALUES (seq_trace.NEXTVAL,'nouvelle trace');
```

Q_1). Créer un trigger qui empêche l'insertion d'une ligne avec n'importe quelle valeur dans la colonne `id_trace`.

Q_2). Écrire les requêtes permettant de tester le trigger et expliquer les résultats.

6. Gestion des utilisateurs

6.1. Création des utilisateurs

Les utilisateurs (`users`) permettent d'établir une connexion au SGBD. 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 **TABLESPACE**.
- **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 6.1 :

```
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.

6.2. Gestions des droits

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 :

▶ **Privilèges objet.** Les privilèges **objet** permettent d'attribuer des droits sur des **objets**.

- — `ALTER` : droit de modifier un objet.
- — `DELETE` : droit de supprimer des lignes.
- — `INDEX` : droit de créer un index.
- — `INSERT` : droit d'insérer des lignes.
- — `REFERENCES` : droit d'utiliser la table dans une contrainte de clé externe (`FOREGIN KEY`).
- — `SELECT` : droit de lire la table.
- — `UPDATE` : droit de mettre à jour la table.
- — `EXECUTE` : droit d'exécuter un code (procédure, fonction, ...).
- — `READ` : droit de lire un fichier présent dans un objet de type `"directory"`.
- — `WRITE` : droit d'écrire dans un fichier présent dans un objet de type `"directory"`.
- — `FLASHBACK` : droit d'utiliser la corbeille pour un objet, si la corbeille est activée.

▶ **Privilèges système.** 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 complet de la base.
- — `IMP_FULL_DATABASE` : permet de faire un import complet 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.**

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

▶ **Attribuer et révoquer un privilège système.**

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

Exercice 32

À 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`). Donner les droits permettant d'autoriser le second utilisateur de travailler sur les données du premier utilisateur.

7. 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.
   Exemple : `SELECT age FROM ETUDIANTS WHERE note=10;`

7.1. Transaction

Définition 7.1 – Transaction

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 moyens 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.

7.2. 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.

7.3. 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.

7.3.1. *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 7.1. 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.

7.3.2. *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 !**	

7.3.3. *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'interfére 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 attribuée 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, arrivées 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 7.1 : 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.

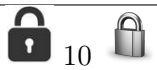
```
CREATE TABLE commande(
    numero      NUMBER,
    designation VARCHAR2(50)
);
```

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  10	

T3		 10	`UPDATE commande;`
			`SET designation='new 2'`
			`WHERE numero=11;`
			1 ligne mise à jour
			  11

T4		 10	`UPDATE commande`
			`SET designation='new 11'`
			`WHERE numero=10;`
			  11  10

T5	`ALTER TABLE commande ADD (quand DATE);` 		 10
T6		`COMMIT;` Validation effectuée  10 	 10
T7			1 ligne mise à jour  11  10 
T8			`ROLLBACK;` annulation effectuée  11  10 
T9	table modifiée  puis 		

`DROP TABLE commande;`

7.3.4. *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.

Exemple 7.2 : 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='S2 demo 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 commande`
		`SET 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

Tps	Session S1	Session S2

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;
```

Exercice 33

Q_1). À partir du compte principal, mettre à jour toutes les lignes de la table `INDIVIDU` pour mettre `sexe='F'` au lieu de `sexe='M'`.

Q_2). À partir du second compte, supprimer l'individu dont le nom est `'MERCIER'` et le prénom est `'Daniel'`.

Q_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.

Q_4). À partir du compte principal, supprimer l'individu dont le nom est `'MERCIER'` et le prénom est `'Daniel'`.

Q_5). Que constatez si cette seconde commande est effectuée juste après la seconde commande ? Expliquez précisément, les éléments techniques qui interviennent.

Exercice 34

Q_1). Faire un rollback sur les deux sessions lancées.

Q_2). Effectuer 2 connexions sur le compte principal. Sur la première session, lire le nombre de lignes de la table `INDIVIDU`.

Q_3). Sur la seconde session, faire une suppression de toutes les lignes de la table `INDIVIDU` dont le sexe est `'H'`.

Q_4). Sur la première session, lire le nombre de lignes de la table `INDIVIDU`.

Q_5). Que constatez si la commande de la seconde session prend longtemps par rapport au `SELECT` ?

8. 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.

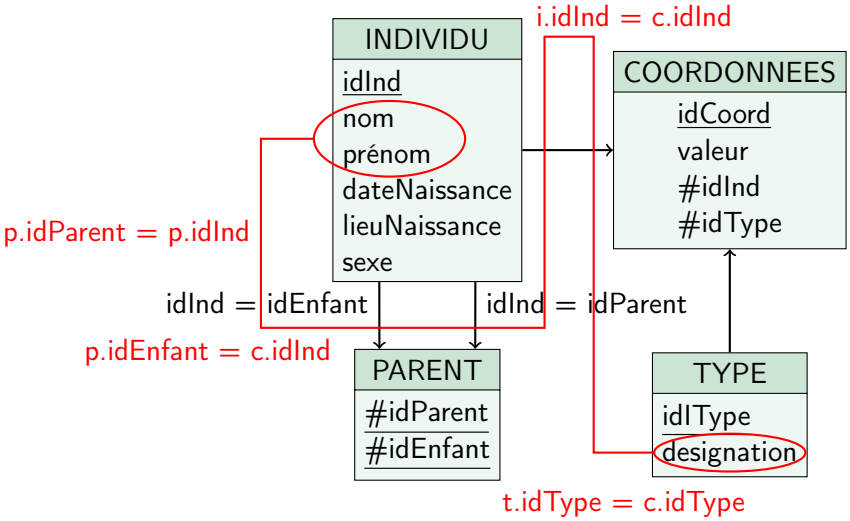
Il faut donc vérifier :

- 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 8.1 : 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, sexe)



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.

8.1. 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 35

Laquelle des deux requêtes est la 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 8.2 : 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, les 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`.

La requête renvoie 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 );
```

8.2. 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 8.3 : Le schéma de la Figure 8.1 montre concrètement comment il est organisé un B-tree.

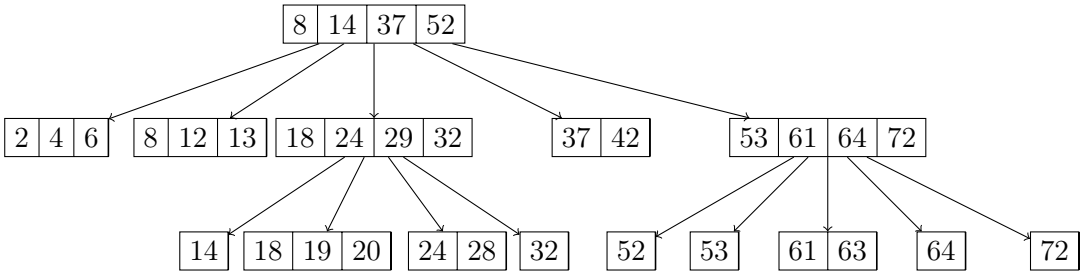


FIGURE 8.1 – Exemple de B-Tree.

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, où les nœuds sont numérotés de gauche à droite et de haut en bas.

▶ **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 8.4 : La Figure 8.2 montre l'organisation d'un index Bitmap.

rowid / couleur	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 8.2 – Exemple d'index Bitmap.

La valeur 0 ou 1 est stockée sous 1 bit et les recherches dans cet index se font sous forme d'opérations logiques. De plus ces index prennent peu de place en mémoire 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 `piece` contenant les résultats de tests de conformité de pièces automobiles. Pour contrôler 4000 pièces par jour, dans cette table, la colonne `"reference"` 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 ma_table(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);`

8.2.1. Quelles colonnes indexer ?

▶ **Clés primaires.** Oracle crée automatiquement des index sur les colonnes concernées par les contraintes `UNIQUE` ou `PRIMARY KEY`. C'est un 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)` ne lit pas de bloc de données et n'utilise que l'index.

Exercice 36

Q_1). Donner la requête permettant d'obtenir le plus grand âge 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`.

Q_2). Indiquer le ou les index pertinents pour optimiser cette requête.

8.3. Plan d'exécution

Lorsqu'une requête SQL est soumise, Oracle définit la méthode à 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`. Cette commande 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 8.5 : 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 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 index 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 8.6 :

```
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 modifiée, 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 8.7 : Soit une table contenant 1 million de lignes.

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

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

▶ **Statistiques.** Oracle utilise 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, mettre à jour les statistiques peut aider à obtenir de meilleurs plans d'exécution.

— `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 utilisateur.

Exemple 8.8 : Soit l'exécution des commandes suivantes.

```
SET autotrace traceonly

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

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name   | Rows | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |        | 14   |   224 |  5      (0)| 00:00:01|
|*  1 |  TABLE ACCESS FULL| MODELE | 14   |   224 |  5      (0)| 00:00:01|
-------------------------------------------------------------------------------
```

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.
-------------------------------------------------------------------------------------------
| Id  | Operation                              | Name         |Rows| Bytes| Cost(%CPU)| Time |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                       |              | 14 |  224 |   1   (0)|00:00:01|
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| MODELE       | 14 |  224 |   1   (0)|00:00:01|
|*  2 |   INDEX RANGE SCAN                     | MODELE_IDX_1| 14 |      |   1   (0)|00:00:01|
-------------------------------------------------------------------------------------------
```

Exemple 8.9 :

```
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 8.10 :

```
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 8.11 : 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 37

Q_1). Compter le nombre de lignes dans la table **TYPE**.

Q_2). Afficher la ligne pour laquelle la désignation est '`Email`'

Q_3). Quel est le plan d'exécution.

Q_4). Indiquer quel index pertinent pourrait améliorer le temps de traitement de cette requête et vérifier la pertinence.

Exercice 38

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.

Commandes utiles pour les TP

▶ **Première séance :** Connexion : `sqlplus <user>/<password>@KIROV`

`source /etc/profile`

Lancer `sqlplus` avec cette commande `rlwrap sqlplus` vous permet d'avoir l'historique.

Lors de la première connexion modifier le mot de passe avec la commande SQL : `PASSWORD;`

Ce qui est équivalent à : `ALTER USER dupond IDENTIFIED BY password;`

Si le mot de passe est égaré quelque part, il faut se connecter en `ssh` sur `pretoria` et faire `oracle_passwd`

Pour lancer un fichier `.sql` en sqlplus, il suffit de taper : `@toto.sql;`

Pour quitter sqlplus, il suffit de taper : `quit;`

▶ **Mise en forme :** Sous SQL/PLUS :

```
SET LINESIZE 150 -- positionne la taille d'une ligne
SET PAGESIZE 300 --  positionne le nombre de lignes avant de réafficher les
    entêtes
SET PAGES 0 --  n'affiche pas les entêtes
COL <nom_colonne> FOR A10 --  défini que la colonne nom_colonne va être affich
    é sur 10 caractères
alphanumériques, 999.99 pour les valeurs numériques.
```

▶ **Corbeille :** Vider les tables `BIN$$xxxx` : avec la commande `PURGE RECYCLEBIN;`

▶ **Débug :** Afficher la strucutre de la table `nba` : `DESCRIBE nba;`

Connaître l'utilisateur connecté : `SHOW user;`

Liste des tables d'un `user` : `SELECT table_name FROM user_tables;`

Lister les tables accessibles par l'utilisateur : `SELECT table_name FROM all_tables;`

Lister toutes les tables possédées par un utilisateur :

```
SELECT * FROM all_tables WHERE owner='PALAFOUR';
```

Liste des vues d'un `user` : `SELECT view_name FROM user_views;`

Liste des contraintes : `SELECT * FROM user_constraints WHERE table_name='<table>';`

`SELECT * FROM user_cons_columns WHERE table_name='<table>';`

`SHOW ERRORS` affiche les erreurs des fonctions, triggers ...

▶ **Contraintes :** Description d'une table : `DESCRIBE <table>` ou `DESC <table>` ;

Liste des colonnes concernées par les contraintes : `SELECT * FROM user_cons_columns;`

Lire une table d'un autre schéma :

```
SELECT * FROM <schema>.<table_name>; -- ou schema = login de connexion de l'utilisateur

SELECT TABLE_NAME FROM USER_TABLES; -- affiche les tables d'un utilisateur
```

▶ **Affichage :**

```
SET HEADING OFF
SET HEADSEP OFF
SET FEEDBACK OFF
SET TERMOUT OFF
SET ECHO OFF
SET VERIFY OFF
SET DEFINE OFF
```

▶ **Accès fichiers.**

`https://homeweb.iut-clermont.uca.fr/login`

▶ **Aide en ligne.**

`https://s2i.iut.uca.fr/page/documentation/sgbd/`

▶ **En cas de blocage :** Dans un terminal executer la commande suivante pour voir les `pid` des processus zombies : `ps -aux | grep sqlplus`

Ensuite tuer ces zombies grâce à la commande `kill -9 numerodepid`

TP1 : NBA Statistiques

L'objectif des TPs est de modéliser une base de données adaptée pour les statistiques des équipes et joueurs de basket de la NBA [1] (National Association Basketball).

En NBA, il y a 30 équipes réparties en deux conférences : Est et Ouest, comme indiqué dans la Figure 8.3. Chaque équipe a un nom et elle est située dans une ville. Un équipe ne peut faire qu'un match par jour. Lors d'un match il y a une équipe qui joue à domicile et un équipe qui joue à l'extérieur. Chaque joueur a un nom, prénom, une date de naissance, une taille, un poids, un poste de jeu, une date de début et une date de fin de carrière en NBA. Chaque joueur d'une équipe ne peut faire qu'un seul match par jour et a un temps de jeu variable par match. Pour commencer un match de basket, 5 joueurs de chaque équipe sont présents sur le terrain et les autres sont sur le banc. Durant son temps de jeu, un joueur peut marquer des points (lancers francs, tir à 2 pts, tirs à 3 pts) avec plus ou moins de réussite, commettre des fautes, prendre des rebonds offensifs ou défensifs, perdre des balles, faire des interceptions, faire des passes décisives. Les équipes s'affrontent dans un championnat qui se déroule en deux phases :

— la saison régulière : où chaque équipe rencontre au moins deux fois chaque équipe ;

— les playoffs : phases finales où les 8 meilleurs équipes de chaque conférence s'affrontent en un tournoi à élimination directe en 4 manches gagnantes pour déterminer le champion de la saison en cours.

Conférence Ouest

ID	Ville	Nom
DAL	Dallas	Mavericks
DEN	Denver	Nuggets
GS	Golden State	Warriors
HOU	Houston	Rockets
LAC	Los Angeles	Clippers
LAL	Los Angeles	Lakers
MEM	Memphis	Grizzlies
MIN	Minnesota	Timberwolves
NO	New Orleans	Pelicans
OKC	Oklahoma City	Thunder
PHO	Phoenix	Suns
POR	Portland	Trail Blazers
SA	San Antonio	Spurs
SAC	Sacramento	Kings
UTA	Utah	Jazz

Conférence Est

ID	Ville	Nom
ATL	Atlanta	Hawks
BKN	Brooklyn	Nets
BOS	Boston	Celtics
CHA	Charlotte	Hornets
CHI	Chicago	Bulls
CLE	Cleveland	Cavaliers
DET	Detroit	Pistons
IND	Indiana	Pacers
MIA	Miami	Heat
MIL	Milwaukee	Bucks
NY	New York	Knicks
ORL	Orlando	Magic
PHI	Philadelphia	76ers
TOR	Toronto	Raptors
WAS	Washington	Wizards

Figure 8.3 – Répartition des équipes NBA par conférences.

1. https://fr.wikipedia.org/wiki/National_Basketball_Association

Les données brutes pour la saison 2017-2018 sont disponibles sur le compte de `palafour` en tapant la commande suivante `SELECT * FROM palafour.nba;` Attention il y a beaucoup de tuples.

1. Combien de tuples contient cette table ?
2. Suite à l'étude des données de la table `palafour.nba`, notez les possibles anomalies que vous pouvez trouver (insertion, suppression, modification)..
3. Indiquez dans quelle forme normale se trouve la table et justifiez votre réponse.
4. À partir des éléments précédents, réalisez un MCD permettant de gérer les statistiques de la NBA
5. À partir de ce MCD en déduire un MLD.
6. Construire le fichier de configuration correspondant avec les contraintes correspondantes et aussi remplir votre base à partir des données fournies dans la table `palafour.nba` le chapitre 3 du polycopié et les commandes utiles-ci-dessous vous aideront sûrement !

▶ **Commandes utiles :**

— Lister les tables accessibles par l'utilisateur : `SELECT table_name FROM all_tables;`

— Lister les contraintes :

```
SELECT * FROM user_constraints WHERE table_name='<table_en_majuscules>';
SELECT * FROM user_cons_columns WHERE table_name='<table_en_majuscules>';
```

— Création de table à partir d'autre table :

```
CREATE TABLE client AS
SELECT ....
```

▶ **Première séance :** Connexion : `sqlplus <user>/<password>@KIROV`

Lors de votre première connexion modifier votre mot de passe avec la commande : `PASSWORD;`

Ce qui est équivalent à : `ALTER USER dupond IDENTIFIED BY password;`

Pour lancer un fichier `.sql` en `sqlplus`, il suffit de taper : `@toto.sql;` ou bien `START toto;`

Pour quitter `sqlplus`, il suffit de taper : `quit;`

Lancer `sqlplus`avec cette commande `rlwrap sqlplus`vous permet d'avoir l'historique.

TP2 : NBA

Les quatre fichiers `nba-game-2017-18.sql`, `nba-player_data.sql`, `nba-player-2017-18.sql`, et `nba-tp2019.sql` sont à télécharger sur le site suivant

`http://sancy.univ-bpclermont.fr/~lafourcade/BDNBA/`

Pour créér la base de données correspondante au MLD de la Figure 8.4 il faut faire `START nba-tp2019.sql;`

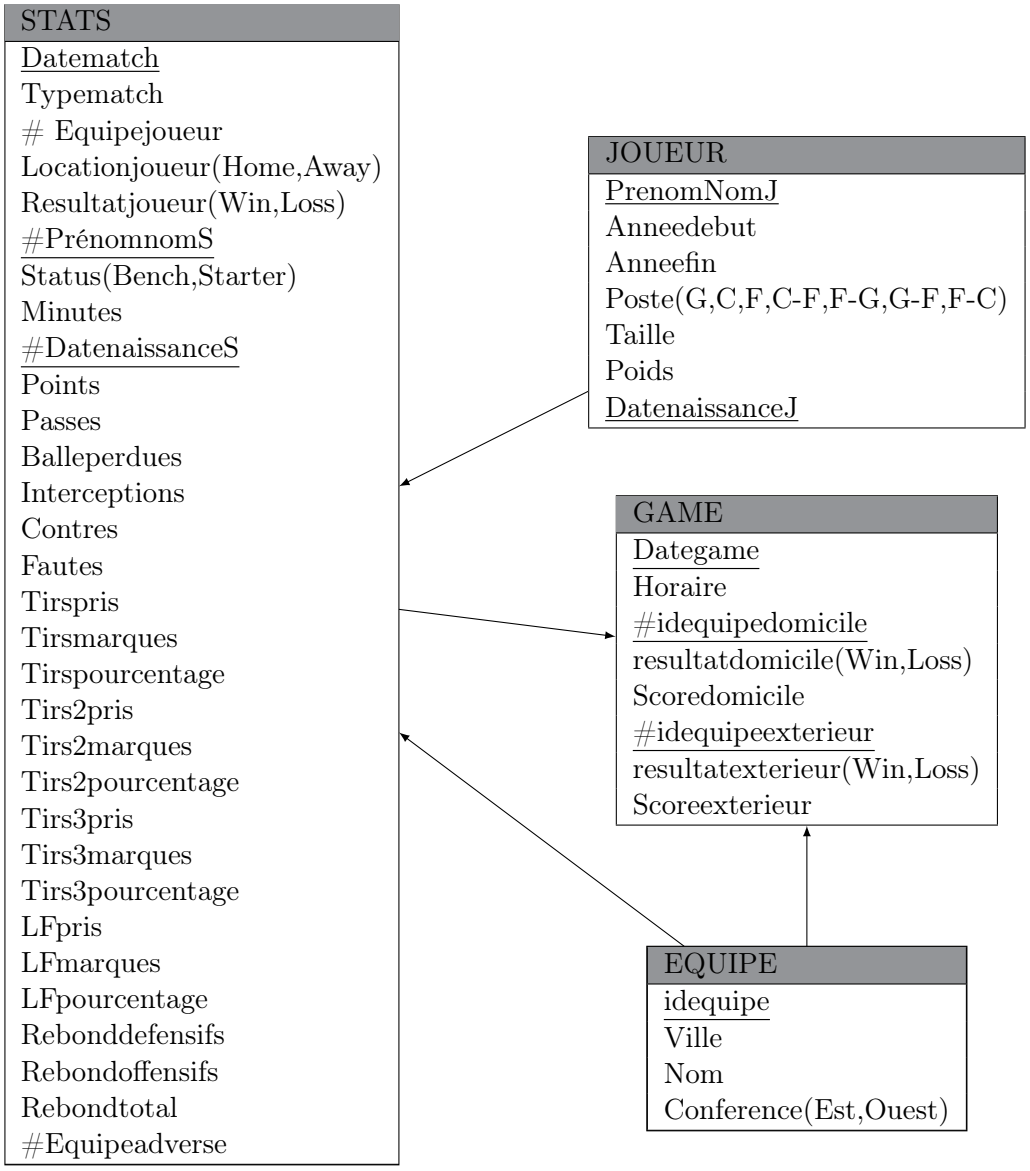


FIGURE 8.4 – MLD utilisé

Exercice 1 : Construction d'une fonction (20 points)

1. (1 points) Trouver le nombre de matchs joués par Kevin Durant.
2. (2 points) Lister le(s) joueur(s) ayant marqué le plus de points dans un match dans la saison en indiquant son nom, la date du match, l'`idequipe` de léquipe adverse et le nombre de points marqués.
3. (2 points) Lister le(s) nom(s) joueur(s) et le nombre de match joué pour ceux ayant joué le plus de match pendant la saison.
4. (5 points) Expliquer pourquoi cette requête renvoie 136

```
SELECT COUNT(*) FROM GAME G, STATS S
WHERE S.Equipejoueur='GS' and S.datematch = G.dategame
AND prenomnomS ='Kevin Durant' AND (idequipedomicile=S.Equipejoueur
                            OR idequipeexterieur=S.Equipejoueur);
```

5. (5 points) Déterminer pour `Kevin Durant` pour le match du `30-oct-17` quelle est l'`idequipe` de l'équipe qui joue à domicile.
6. (5 points) Écrire une fonction `pointsmarques` qui détermine pour une date de match et un joueur donnés quelle est le nombre de points marqués par ce joueur.

Exercice 2 : Construction d'une vue (20 points)

1. (15 points) Créér une vue qui vous permettra à partir de ces tables de reconstruire une table identique à celle de `palafour.nba`
2. (5 points) Vérifier par 2 requêtes que le nombre de tuples de votre vue et de `palafour.nba` sont les mêmes.

Exercice 3 : Clef primaire (10 points)

1. (5 points) Modifier la table `JOUEUR` pour ajouter un champ `idjoueur`. Remplir ce champs avec un identifiant unique. Faire en sorte que cela devienne la clef primaire. Pour répondre à cette question vous pouvez utiliser `ALTER`, `UPDATE` et `rownum`.
2. (5 points) Faire les modifications nécessaires sur la table `STATS`. Vérifier que les contraintes de clefs étrangères sont bien en place en "violant des enfants".

TP3 : Triggers

Déclencheur (15 points)

1. (1 point) Créer la table `TRACE ('quand' date, 'qui' sur 20 caractères, 'quoi' sur 800 caractères)`
2. (1 point) Lancer la commande suivante qui appelle la fonction `SYS_CONTEXT` et commenter le résultat obtenu.

```
SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER') FROM DUAL;

-- PALAFOUR
```

3. (10 points) Réaliser un déclencheur qui permet de tracer la suppression d'un joueur afin de connaître l'identifiant de session, la date et les informations supprimées.
4. (3 points) Tester le déclencheur créé.

Meilleur Marqueur (25 points)

1. (5 points) Créér une table `BEST` qui contient la liste des maximums des points marqués par match et la date du match.

```
MAXPOINTS  DATEMATCH
---------- ---------
32 26-DEC-17
30 11-JAN-18
50 10-JAN-18
```

2. (10 points) Mettre en place un déclencheur qui met à jour cette table pour toutes nouvelles insertions de résultats de match.
3. (5 points) Tester les déclencheurs créés.
4. (3 points) Donner les droits en lecture, écriture et effacement sur cette table à tout le monde.
5. (2 points) Tester si cela fonctionne en ajoutant une entrée dans la table d'un de vos camarades et en effaçant vos traces une fois ce test réalisé. Utilisez un de vos camarades pour faire les tests ou bien `palafour2` sera votre ami.

TP4

Chaîne de caractères (5 points)

1. (3 points) Afficher la date sous la forme `"Il est 18 heure 45 minutes et 26 secondes, nous sommes LUNDI"`
   Cette commande peut vous aider :
   `select to_char(sysdate,'DAY','NLS_DATE_LANGUAGE =FRENCH') from dual;`
2. (2 points) Afficher le jour de la semaine en toute lettre suivi d'autant de `$` qu'il faut pour que la chaine fasse 20 caractères sans espace.

SQL Dynamique (5 points)

Le SQL dynamique permet de générer des commandes SQL. La commande `spool toto.sql` écrit l'ensemble des résultats des requêtes exécutées par la suite dans le fichier `toto.sql`. Pour arrêter cette fonctionalité il faut faire `spool off;`. Il est donc possible d'exécuter les commandes générées avec les requêtes SQL en lançant la commande `@toto.sql;` ainsi créée.

À partir de la liste de vos tables, indiquer la liste des commandes permettant de supprimer les tables. Les seules tables a supprimer doivent contenir la lettre 'A' au moins une fois et avoir plus de 6 lettres. Si la table ne répond pas à ce critère, afficher "la table XXX ne doit pas être supprimée". Créer des tables fictives pour tester votre code.

Concurrence d'accès (10 points)

Suivez les entêtes de colonne à la lettre. Les tables doivent exister dans les deux comptes étudiants. Les chaînes entre <> sont à remplacer par la valeur. <User1> le compte d'un des étudiants <User2> le compte de l'AUTRE etudiant.

Pour les questions suivantes, vous indiquerez :

— Quels sont les résultats des commandes exécutées ?

— Est-ce que des verrous sont positionnés ?

— Que constatez-vous ? Expliquez vos réponses.

Lancer la commande : `CREATE TABLE tp4 AS select * FROM STATS;`

Puis les commandes indiquées par utilisateurs dans l'ordre indiqué.

1. (1 point) Trouver l'`idjoueur` de Kévin Durant. Cette valeur n'est pas forcément la même pour tous les étudiants. Dans la suite, si cette valeur est différente de `920`, remplacez la par la valeur trouvée.

2. (1 point) Uniquement pour cette question, d'un point de vue théorique quel est le comportement obtenu pour chaque utilisateur. Pour les autres questions exécutez les commandes sur machines dans plusieurs fenêtres.

Tps	User1 (session S1, transaction T1)	User2 (session S2, transaction T2)
t1	`DELETE tp4 WHERE datenaissances= TO_DATE('04/01/1997','DD/MM/YYYY');`	
t2		`DELETE tp4 WHERE datenaissances= TO_DATE('04/01/1997','DD/MM/YYYY');`
t3	`Rollback;`	
t4		`Rollback;`

3. (2 points)

Tps	User1 (session S1, transaction T1)	**User1** (session S2, transaction T2)
t1	`DELETE tp4 WHERE idjoueur='920';`	
t2		`DELETE tp4 WHERE idjoueur='920';`
t3	`Rollback;`	
t4		`Rollback;`

4. (2 points)

Tps	User1 (session S1, transaction T1)	User1 (session S2, transaction T2)
t1	`DELETE tp4 WHERE idjoueur='920';`	
t2		`DELETE tp4 WHERE prenomnomj='Kevin Durant';`
t3	`Rollback;`	
t4		`Rollback;`

5. (2 points)

Tps	User1 (session S1, transaction T1)	User1 (session S2, transaction T2)
t1	`UPDATE tp4 SET prenomnomj='Kevin Durant' WHERE idjoueur='920';`	
t2		`ALTER TABLE tp4 MODIFY (STATUS VARCHAR2(7));`
t3	`Rollback;`	
t4		`ALTER TABLE tp4 MODIFY (STATUS VARCHAR2(7));`

6. (2 points)

Tps	User1 (session S1, transaction T1)	User1 (session S2, transaction T2)
t1	`DELETE tp4;`	
t2	`SELECT count(1) FROM tp4;`	
t3		`SELECT count(1) FROM tp4;`
t4	`commit;`	
t5		`SELECT count(1) FROM tp4;`

TP5 : Ecriture de requêtes et optimisation

1. (0.5 point) Écrire la requête qui donne le joueur ayant évolué le plus longtemps en NBA.
2. (0.5 point) Donner ensuite la liste d'équipes dans lesquelles il a évolué en 2017-2018.
3. (3 points) Écrire la requête qui donne l'équipe qui a le plus de victoires pendant la saison 2017-2018.
4. (3 points) Écrire la requête qui donne le(s) joueur(s) le(s) plus adroit(s) à trois points pour le match du '25-DEC-17' contre les Lakers.
5. (3 points) Écrire la requête permettant d'obtenir la liste des équipes battues avec les dates des rencontres pour l'équipe qui a le moins de victoire sur la saison. Que remarquez-vous sur les données de la base ?
6. (3 points) Écrire la requête permettant d'afficher la liste des joueurs qui ont fait un triple double sur la saison en donnant leurs statistiques dans les 5 catégories. Un triple double c'est avoir au moins dans 3 catégories des statistiques à 2 chiffres (points, rebonds, passes, contres, interceptions)
7. (2 points) Écrire la requête permettant d'obtenir la liste des joueurs de plus de 2 mètres pour lesquels il existe des matchs lors desquels ils ont fait des interceptions.
8. (1 points) Écrire la requête qui, pour un nom d'équipe fixée (Golden State par exemple), trouve la somme des rebonds pris sur l'ensemble de la saison.
9. (2 points) Écrire la requête qui va trouver le maximum des points marqués dans un match. Indiquer le plan d'exécution et le commenter.
10. (1 point) Créer un index bitmap permettant d'améliorer les performances. Justifier votre la pertinence de votre action.
11. (1 point) Créer un index b-tree permettant d'améliorer les performances. Justifier votre la pertinence de votre action.