⬅Revenir au menu.

Introduction aux bases de données

I. Introduction

1. Mise en situation

Vous avez fait un super programme en Python qui va faire la moyenne des points obtenus pour le cours de Python. Le résultat est stocké en mémoire. Vous avez encodé les résultats de 4 classes composées de 30 élèves chacune. C'est cool. Vous avez le résultat désiré : la moyenne de la classe en python de chaque classe et la moyenne totale de toutes classes confondues. Bien joué, vous assurez !

Vous éteignez votre ordinateur. Vous allez en terrasse car elles ont réouvert le 8 mai. Et boum, le lendemain, mal de crâne et le trou noir de la journée précédente. Votre directeur vous téléphone et vous demande la moyenne/classe et la moyenne générale. Mais impossible de vous en souvenir… Aie aie aie cette journée commence très mal…

2. Brainstorming : Qu'auriez-vous pu faire pour garder les résultats ?

A partir de l'exemple précédent que peut-on retirer comme conclusion ?

3. Approche intuitive des SGBD

De la faiblesse de notre programme stocké en mémoire. On constate que l'on aurait pu stocker notre résultat dans un fichier. Et le relire par après. Ce qui est déjà une très grande évolution par rapport à un stockage en mémoire vive.

Cependant, notre directeur a dû nous téléphoner pour avoir le résultat. Il est déjà dommage d'avoir eu besoin de transmettre ce fichier stocké sur notre ordinateur.

Dans une infrastructure de type entreprise, ça ne posait pas de problème car notre directeur aurait eu accès à ce fichier via le réseau d'entreprise : Domaine, système de fichiers, partages, etc…
Imaginons maintenant que le conseil de classe de ces 4 classes se déroulent en même temps. Le directeur fournit le fichier aux 4 titulaires de classe.

Chaque titulaire manipule un fichier car il est possible que l'on donne la moyenne aux élèves proches de la moyenne. C'est tout le débat d'un conseil de classe. Et donc de modifier ce fichier.

Chaque titulaire devra envoyer son fichier au directeur. Et le directeur devra remettre le tout dans un fichier reprenant les modifications de chaque titulaire : aie aie aie sur autant d'élèves les risques d'erreurs commencent à augmenter avec autant de manipulations manuelles.

Idéalement il aurait été très intéressant que chaque titulaire puisse encoder ses modifications et que ces modifications soient prises en compte. Avec un programme, il est plus compliqué de faire la gestion concurrentielle d'un même fichier. Mais ça reste faisable. Mais de nos jours nous avons recours à ce qu'on appelle des bases de données. Le programme se connecte sur une base de données où seraient stockés les résultats de tous les élèves de toutes les classes. L'accès concurrent est généré nativement : chaque titulaire peut modifier en même temps la base de données.

II. Base de données

1. Définition

Une base de données est un outil informatique qui permet d'organiser des informations de façon sécurisée, hiérarchisée et sans doublon. Appelée Database en anglais (on voit souvent l'abréviation db), les bases de données sont des logiciels qui permettent surtout de mieux travailler.
C'est donc une collection structurée de données cohérentes, intègres, protégées et accessibles simultanément aux utilisateurs
Concrètement, les informaticiens se sont rapidement retrouvés face à des problèmes difficiles à résoudre en termes de performance et d'intégrité. Comment s'assurer qu'une information saisie dans un système informatique est unique, toujours bien rangée et correctement protégée contre les mauvaises manipulations ?

Un SGBD peut gérer plusieurs bases. En effet, on pourrait très bien avoir les données des cours d'une école comme base et une autre base pour les données d'un garage de voitures Tesla. C'est tout à fait possible et même utile pour sécuriser et séparer des données différentes.

2. Système de gestion de base de données (SGBD)

Le SGBD n'est que l'application concrète de la base de données. Sans SGBD, la BD reste un outil théorique « sur papier ». Le SGBD permet concrètement de mettre en place le travail de modélisation et de se servir de la base de données imaginée.

Implémenter une base de données dans un SGBD impose d'arrêter son choix sur un outil. Pour le choisir, il faut avoir réfléchi aux contraintes et caractéristiques de la base de données (volume d'information, accès depuis un même lieu ou pas, droits et accès simultanés) … Baser son choix de SGBD uniquement sur les outils disponibles (par exemple Ms Access parce qu'il est installé avec la suite Ms Office) est à coup sûr une mauvaise idée.

Petite parenthèse pour MS Access. Attention Ms Access n'est pas non plus la pire idée du monde. Il reste tout de même fort utilisé dans les asbl, les petites PME car il permet de rapidement faire des formulaires pour ajouter/modifier/supprimer des informations dans une base de données Access. Depuis que MS Access gère le transactionnel, il n'est plus le choix aussi risible qu'il était par le passé. Tout va dépendre de la charge et du nombre d'utilisateurs.

Si l'on veut une base de données de type fichier ou monoposte, on se tournera plutôt vers SqlLite que MS Access. Mais à nouveau tout va dépendre du besoin. En informatique, le terme besoin est important car il va nous aider à faire des choix.

Un Système de Gestion de Base de Données (SGBD) est un logiciel qui permet de stocker des informations dans une base de données. Un tel système permet de lire, écrire, modifier, trier, transformer ou même imprimer les données qui sont contenus dans la base de données.
De plus, un SGBD est sécurisé via l'utilisation d'utilisateurs, de mots de passe. On peut définir les objets que l'utilisateur pourra utiliser dans notre SGBD : bases, tables, procédures stockées, etc.

Dans le cadre de ce cours, nous utiliserons MySQL comme SGBD. Car elle c'est un logiciel libre. Elle existe en version libre et propriétaire. En version propriétaire et donc payante vous aurez un support de la part de Sun qui a racheté MySQL en 2010.

3. Base de données relationnelles

La révolution apportée par le modèle relationnel réside dans une indépendance totale par rapport

au modèle physique. Défini par Codd en 1970 sur des bases purement mathématiques, ce modèle s'affranchit résolument de toute contrainte matérielle. Cela explique qu'il ait démarré assez lentement, parce qu'il exigeait les machines puissantes dont nous disposons seulement aujourd'hui.

4. SQL

Les SGBD offrent un langage d'interrogation des données qui s'appelle le SQL. Le SQL a connu plusieurs normes. On pourrait penser que le SQL est un langage universel et identique à tous SGBD. Et bien non ! En gros oui mais avec parfois des adaptations mineures mais posant un problème majeur : mes commandes SQL ne seront pas exactement les mêmes si je veux changer de base de données. Exemple : passer d'Oracle à MySQL. La volonté de ne pas uniformiser vient du fait qu'un fabricant de SGBD n'a pas envie que vous quittiez son SGBD très facilement. De plus, certains fabricants offrent de belles fonctionnalités par rapport à la concurrence. Donc si vous utilisez des fonctionnalités très spécifiques et propriétaires à un SGBD. Il vous sera difficile de migrer vers un autre SGBD ou au prix de beaucoup d'efforts. Mais pour des requêtes classiques la syntaxe de votre commande SQL sera la même partout ou avec de petites modifications.

III. Modèle conceptuel de données (MCD)

Dans la méthodologie Merise destinée à créer des bases de données, il y a des outils dédiés aux traitements et aux données. Le MCD (Modèle Conceptuel des Données) est un des outils majeurs concernant les données. Ce modèle décrit une situation à analyser à l'aide d'entités et de relations. Des entités peuvent être liées à une autre entité par l'intermédiaire de ce qu'on appelle une relation.
Une fois notre modèle fait, on convertira nos entités en tables et nos relations en clés primaires et clés étrangères. Nous verrons plus loin cela.

1. Entité

Une entité a un nom unique afin de la manipuler facilement. Plus tard dans l'analyse, l'entité se

transforme en table et devient concrètement une table lors de la réalisation effective de la base de données.
Cet ensemble d'informations, l'entité, partage les mêmes caractéristiques et peut être manipulé au sein du système d'information mais aussi en discutant entre informaticiens et personnes du métier.
Reprenons notre exemple sur les résultats de nos élèves et les notes des élèves pour le cours de python.
Si on devait essayer de rassembler les informations de base d'un élève : de l'entité élève. Quelles sont les propriétés qui le caractérisent et permettent de l'identifier au mieux ?

2. Clef d'identité

La clef d'identité permet d'identifier de manière sûre et fiable notre élève. Cette clé doit être pensée pour qu'il ne puisse JAMAIS y avoir de doublons. La clef peut être composée d'une ou plusieurs propriétés. Les valeurs de clefs d'identités sont uniques et non nulles.

Dans l'entité, cette clef est soulignée pour marquer justement que c'est une clef.
Il arrive qu'on ajoute le préfixe « id » (pour identifiant) à une clef d'identité. Exemple : id_client, id_etudiant. Cette clef est souvent écrite Id. Rajouter id_eleve dans l'entité Elève est un peu redondant : c'est évident.

Mais ça c'est selon l'endroit et les conventions que vous ou votre équipe utiliserez.

Dans beaucoup de cas, c'est souvent un numéro automatique incrémenté de 1. L'intérêt d'avoir un numéro automatique, c'est que la gestion de ce numéro automatique est laissée au SGBD. Il déduira automatique le nouveau numéro à générer pour le nouvel enregistrement. Si le précédent Eleve avait comme identifiant 43, le nouvel étudiant que l'on encodera aura le numéro 44. Et pour le suivant, ça sera le numéro 45, etc.

Souvent on prend un entier comme clef primaire. Car un entier prend 4 octets. Une clef primaire de type entier prend moins de place et la recherche dans les index est plus rapide.

3. Relation/association

La relation ou association relie plusieurs entités.
Notre entité élève fait partie d'une classe. Une classe est une entité.

Ce qui relie l'entité élève et l'entité classe c'est la relation : « fait partie de » dans le sens élève vers classe.

4. Cardinalités/Multiplicité ou combien ?

Dans la théorie des ensembles, la cardinalité est une propriété des ensembles, y compris infinis, qui généralise la notion de nombre d'éléments aux ensembles finis.
Les cardinalités sont des couples de valeurs que l'on retrouve entre chaque entité et ses relations. La première valeur est la valeur minimale et la seconde est la valeur maximale.
Il existe quatre valeurs : (0,1), (0,N), (1,1) ou (1,N) où N >1
Exemple :

- Entité Élève
- Relation : Fait partie de
- Entité Classe

Les cardinalités traduisent des règles de gestion.

- Un élève fait partie d'une et une seule classe.
  Sens Elève vers Classe : cardinalité 1 (minimum) et 1 (maximum)
- Et une classe a 1 ou N (=plusieurs) élèves.
  Sens Classe vers Elève : cardinalité 1 (minimum) et N (maximum).

5. Passage du MCD au MLD

Le passage du Modèle Conceptuel de Données (MCD) au Modèle Logique de données (MLD).

5.1 Entités, clefs, propriétés

5.1.1 Une entité devient une table

Une entité devient une table. Dans un SGBD, une table est une structure composée de colonnes. Ces colonnes sont typées et peuvent avoir des contraintes : unique, non nulle, etc. Ces colonnes correspondent aux propriétés de l'entité. Une colonne porte le nom de champ dans une table. Par exemple le champ « prénom » de la table Elève.
Par convention, on n'utilise pas de caractère accentué pour le nom des champs.

Dans l'affichage de l'ensemble des données de notre table. Par exemple la table Élève qui contient tous les élèves, chaque ligne de cette table correspond à ce qu'on appelle un enregistrement qui correspond à un élève.
La valeur prise par un champ pour un enregistrement donné se situe à l'intersection entre l'enregistrement et le nom du champ.

5.1.2 La clef d'identité devient la clef primaire

Dans une table, la clef d'identité devient une clef primaire. La clef primaire (Primary key en anglais) permet d'identifier de manière unique un enregistrement d'une table. Si on liste tous les enregistrements de la table élèves = si on liste tous les élèves de la table élèves, nous n'aurons pas deux élèves avec la même clef primaire. Le SGBD ne le permettrait pas et provoquerait une erreur si on essayait de le faire. La création d'une clef primaire donne lieu dans les SGBD la création d'un index qui permet aux SGBD de traiter plus rapidement les recherches, les tris. C'est très intéressant quand on brasse une très grosse quantité de données.

Comme dit, la clef s'appelle clef primaire. Mais on peut aussi l'appeler clef d'identité. Ou encore Primary Key ou PK en anglais. Ou encore Id. Ou encore idEleve. Ou encore id_eleve. Il n'est pas rare de voir des noms de clef primaire avec le préffixe "pk_".

5.1.3 Une propriété devient un attribut

Une propriété d'une entité devient un attribut/champ/une colonne. Ce champ a un type : integer, float, boolean, varchar (chaîne à taille variable), char (chaîne à taille fixe), date (date, datetime, timespamp). Cet attribut peut aussi être qualité de NULL, NOT NULL, UNIQUE par exemple.

5.2 Associations

5.2.1 Une association de type 1:N

C'est à dire qui a les cardinalités maximales positionnées à « 1 » d'un côté de l'association et à « N » de l'autre côté. Elle se traduit par la création d'une clé étrangère dans la relation correspondante à l'entité côté « 1 ». Cette clé étrangère référence la clé primaire de la relation correspondant à l'autre entité.

Exemple 1:_
Elève Fait partie d'une classe : cardinalité 1:1 (Cardinalité maximale = 1)
Et une classe a un ou plusieurs élèves : 1:N (Cardinalité maximale = N)

=> Association de type 1:N

On ajoutera dans la table Elève la clef primaire de la table Classe. En effet, cette clé permettra d'identifier la classe dont fait partie un élève. Quand on ajoute comme champ la clef primaire d'une autre table, cette clef porte le nom de clef étrangère (Foreign Key en anglais). Contrairement à la clef primaire qui doit être unique dans une table, une même valeur de clef étrangère peut y figurer plusieurs fois. Ce qui est logique : plusieurs élèves font partie d'une même classe. Elle ne peut être **NULL** dans ce cas-ci.

Exemple 2:
On pourrait imaginer que notre système d'inscription autorise l'inscription d'étudiants indécis. Ils ne savent pas ce qu'ils veulent faire dans l'école mais savent qu'ils veulent étudier…
Un élève peut faire partie d'une classe mais peut aussi ne pas en faire partie : cardinalité 0,1 (cardinalité maximale = 1)
Une classe a un ou plusieurs élèves : 1:N (Cardinalité maximale = N)

=> Association de type 1 :N

On fera comme précédemment on ajoutera comme clef étrangère, la clef primaire de la table Classe. Mais à la différence qu'ici on acceptera les valeurs **NULL** pour cette clef étrangère pour nos indécis d'étudiants qui ont par conséquent donnés une cardinalité minimum à 0…

5.2.2 Une association de type N :N

C'est-à-dire que les cardinalités maximales des deux entités sont à N.
Dans ce cas précis on doit créer une entité intermédiaire reprenant les deux clefs d'entité. Ces deux clefs d'identité forment alors la clef d'identité de cette nouvelle entité. En effet, à la différence de l'association 1 :N, ici on ne peut avoir qu'une seule valeur mais plusieurs.

Exemple: Cours et Formateur.
Un formateur donne 1 ou plusieurs cours : Cardinalité 1 :N (Cardinalité maximale :N).
Un cours est donné par 1 ou plusieurs cours : Cardinalité 1 : N (Cardinalité maximale :N)

=> Association de type N :N

Par exemple le cours de Python est donné par Philip & Johnny.
Philip donne les cours de Python, PHP, Django, Rattrapage, etc.
Johnny donne les cours de Python, Git, SGBD, etc.

Le cours de Python est donné par Philip et Johnny.

Ici on voit bien qu'on ne sait pas mettre qu'une seule clef étrangère Cours dans l'entité Formateur. Il en faudrait plusieurs.

Et on voit bien qu'on ne sait pas mettre qu'une seule clef étrangère Formateur dans l'entité Cours. Il en faudrait plusieurs.

Il faut donc « tout simplement » créer une nouvelle entité qu'on peut appeler FormateurCours. La clef d'identité sera composée de la clef d'identité de l'entité Cours et de la clef d'identité de l'entité Formateur. Cette entité contient donc une clef composite ainsi qu'éventuellement des propriétés propres.

Ma compagne relisant le cours me dit : oui mais pourquoi ne pas créer autant de champs supplémentaires qu'on en a besoin ? Par exemple dans Formateur on mettrait idCours1, idCours2, idCours3, idCours4, etc.

Alors j'ai été étonné par sa réflexion mais finalement elle a raison : Pourquoi pas ? Car c'est difficilement maintenable. Imaginons qu'un jour le nombre maximum de cours donné par un formateur passe de 5 à 15 (pauvre formateur !). Ça veut dire qu'on doit modifier l'entité et ajouter 10 propriétés qui sont des clefs étrangères. C'est faisable mais peu évident à maintenir. Par contre notre entité FormateurCours est une solution finale/générique. Cette solution tiendra en compte un nombre infini de cours qu'un formateur pourrait donner (pauvre formateur !).

5.2.3 Une association de type 1 :1

Ce type d'association est à proscrire car elle reflète une situation où une entité doit être intégrée dans l'autre entité.

Prenons un Exemple : On doit modéliser les entités entrant en jeu dans une course de voiliers en solitaire.

Nous pourrions avoir deux entités : Marin et Voilier avec comme relation Pilote.
Un marin pilote 1 et 1 seul voilier : cardinalité 1:1 (Cardinalité maximale = 1)
Un voilier est piloté par 1 et 1 seul voilier : cardinalité 1:1 (Cardinalité maximale = 1)

=> Association de type 1 :1

Si fonctionnellement on considère que Marin est plus important : on ramène toutes les propriétés

de Voilier dans Marin.

Si fonctionnellement on considère que Voilier est plus important : on ramène toutes les propriétés de Marin dans Voilier.

Maintenant, si on sait que notre modèle évoluera vers une association de type 1 :N. Ce type d'association pourra gérer par exemple des courses de voiliers. 1 marin pilote 1 et 1 seul voilier. Mais 1 voilier est piloté par 1 ou N marins.

Dans la pratique si les entités ont une distinction fonctionnelle forte. On peut les séparer. En effet, imaginons qu'un voilier ait 100 propriétés qui le caractérisent. Remettre toutes ces propriétés dans l'entité Marin est assez discutable. Personnellement, dans ce cas, je fais deux entités.

IIIb. Les formes normales / Normalisation des bases de données

1. Définition

Les formes normales sont des principes de conception de bases de données relationnelles qui visent à réduire la redondance des données et à augmenter l'intégrité des données. Elles représentent des règles pour la structuration de tables et de relations dans une base de données.

L'application de ces formes normales aide à prévenir les anomalies de base de données, facilite l'entretien des données, et améliore la performance des requêtes.

Il y a 5 formes normales (1FN, 2FN, 3FN, 4FN et 5FN). La plupart des bases de données sont normalisées jusqu'à la 3FN. Nous ne verrons que les 3 premières formes normales. Les 4 et 5FN sont très peu utilisées: elles sont surtout utiles pour les bases de données très complexes.

2. Première forme normale (1FN)

2.1 Définition

Une table est dite en **Première Forme Normale** (`1FN`) si et seulement si tous les champs

contiennent des valeurs atomiques, c'est-à-dire chaque colonne contient une sule valeur.

De plus, chaque ligne doit être unique. C'est-à-dire que chaque ligne doit avoir une valeur unique pour la clef primaire.

2.2 Exemples

2.2.1 Exemple 1 - Table etudiant

Soit la table `etudiant` suivante :

etudiant (0FN)

id	nom	prenom	email	telephones	adresse	cp	loc
1	Dupont	Jean	jean.dupont@dupont.com	0478/45.45.46, 02/555.55.55	Rue de l'insertion, 1	1000	Bru
2	Durand	Marie	Durand.Marie@Durand.com	0478/45.45.47, 02/555.55.55	Rue des sélections, 45	4000	Liè
3	Martin	Jean	jean.Martin@Martin.com	0478/45.05.05, 02/555.55.55	Rue de l'instance, 100	5300	Anc

Dans cette table, le champ `telephones` contient plusieurs valeurs séparées par une virgule. Ce n'est pas une valeur atomique. Pour atteindre la `1FN`, nous devons le diviser :

etudiant (presque 1FN)

id	nom	prenom	email	gsm	telephone	adresse	
1	Dupont	Jean	jean.dupont@dupont.com	0478/ 45.45.46	02/ 555.55.55	Rue de l'insertion, 1	1
2	Durand	Marie	Durand.Marie@Durand.com	0478/ 45.45.47	02/ 555.55.55	Rue des sélections, 45	4
3	Martin	Jean	jean.Martin@Martin.com	0478/ 45.05.05	02/ 555.55.55	Rue de l'instance, 100	5

Cependant, cette table n'est pas encore en `1FN`. En effet, le champ `adresse` contient plusieurs valeurs séparées par une virgule. Pour atteindre la `1FN`, nous devons le diviser :

etudiant (1FN)

id	nom	prenom	email	gsm	telephone	rue	nu
1	Dupont	Jean	jean.dupont@dupont.com	0478/ 45.45.46	02/ 555.55.55	Rue de l'insertion	1
2	Durand	Marie	Durand.Marie@Durand.com	0478/ 45.45.47	02/ 555.55.55	Rue des sélections	45
3	Martin	Jean	jean.Martin@Martin.com	0478/ 45.05.05	02/ 555.55.55	Rue de l'instance	10

2.2.2 Exemple 2 - Table etudiant_cours

Soit la table `etudiant_cours`. Cette table reprend les informations de l'étudiant et les cours qu'il suit.

etudiant_cours (0FN)

id	nom	prenom	date_naissance	email	gsm	telephon
1	Dupont	Jean	01/01/2000	jean.dupont@dupont.com	0478/ 45.45.46	02/ 555.55.5
2	Durand	Marie	10/08/2002	Durand.Marie@Durand.com	0478/ 45.45.47	02/ 555.55.5
3	Martin	Jean	15/03/2003	jean.Martin@Martin.com	0478/ 45.05.05	02/ 555.55.5

On constate que la colonne `cours` contient plusieurs valeurs séparées par une virgule. Ce n'est pas une valeur atomique. Pour atteindre la `1FN`, nous pourrions avoir une solution non générique en ajoutant des colonnes `cours1`, `cours2`, `cours3`, etc. Mais cette solution n'est pas optimale. En effet, si on veut ajouter un cours à un étudiant, il faudra ajouter une colonne à la table etudiant. Si un étudiant suit 50 cours différents, il faudra ajouter 50 colonnes à la table etudiant. Cette solution n'est pas optimale.

etudiant_cours (1FN)

id	nom	prenom	date_naissance	email	gsm	telephon
1	Dupont	Jean	01/01/2000	jean.dupont@dupont.com	0478/ 45.45.46	02/ 555.55.5
2	Durand	Marie	10/08/2002	Durand.Marie@Durand.com	0478/ 45.45.47	02/ 555.55.5
3	Martin	Jean	15/03/2003	jean.Martin@Martin.com	0478/ 45.05.05	02/ 555.55.5

Notre table est en 1FN mais ce n'est pas top top... En effet, on a des redondances de données. Par exemple, on a deux fois les mêmes informations pour Jean Dupont. On a deux fois son nom, son prénom, sa date de naissance, son email, son gsm, son téléphone, son adresse, son numéro, son code postal et sa localité. Ca sera lors de la 2FN que nous allons supprimer ces

redondances de données.

Ou bien on peut imaginer que l'on va mettre qu'une seule donnée atomique pour la colonne cours. Par exemple "Jean Dupont" a suivit les cours de Python et de PHP. On pourrait avoir deux enregistrements dans la table etudiant_cours. Un enregistrement pour le cours de Python et un enregistrement pour le cours de PHP.

Notre table etudiant_cours serait alors comme ceci :

etudiant_cours (1FN)

id	nom	prenom	date_naissance	email	gsm	telephon
1	Dupont	Jean	01/01/2000	jean.dupont@dupont.com	0478/ 45.45.46	02/ 555.55.5
1	Dupont	Jean	01/01/2000	jean.dupont@dupont.com	0478/ 45.45.46	02/ 555.55.5
2	Durand	Marie	10/08/2002	Durand.Marie@Durand.com	0478/ 45.45.47	02/ 555.55.5
3	Martin	Jean	15/03/2003	jean.Martin@Martin.com	0478/ 45.05.05	02/ 555.55.5
3	Martin	Jean	15/03/2003	jean.Martin@Martin.com	0478/ 45.05.05	02/ 555.55.5

Ici, notre table est en 1FN, on sent bien que cette solution n'est pas optimale. En effet, on a des redondances de données. Par exemple, on a deux fois les mêmes informations pour Jean Dupont. On a deux fois son nom, son prénom, sa date de naissance, son email, son gsm, son téléphone, son adresse, son numéro, son code postal et sa localité.

Ca sera lors de la 2FN que nous allons supprimer ces redondances de données.

2.3 En résumé, comment passer de la `0FN` à la `1FN` ?

- Soit diviser les champs qui contiennent plusieurs valeurs en plusieurs champs qui ne contiennent qu'une seule valeur.

- Soit diviser les champs qui contiennent plusieurs valeurs en plusieurs enregistrements qui ne contiennent qu'une seule valeur.

3. Deuxième forme normale (2FN)

3.1 Définition

Une table est dite en **Deuxième Forme Normale** (`2FN`) si et seulement si elle est en `1FN` et si tous les champs non-clés dépendent de la clé primaire.

> Cela signifie que chaque information dans la table doit être associée à cette clé primaire, et non à une partie de celle-ci (dans le cas des clés primaires composées).

En pratique, cela implique souvent de diviser les tables pour éliminer les redondances et les dépendances partielles. Par exemple, si une table contient les informations des étudiants et de leurs cours, où chaque étudiant peut s'inscrire à plusieurs cours, il serait inapproprié de répéter les informations personnelles de l'étudiant pour chaque cours auquel il s'inscrit.

3.2 Exemple - Table etudiant_cours

Reprenons notre table `etudiant_cours` qui était en `1FN` :

etudiant_cours (1FN)

id	nom	prenom	date_naissance	email	gsm	telephon
1	Dupont	Jean	01/01/2000	jean.dupont@dupont.com	0478/ 45.45.46	02/ 555.55.5
1	Dupont	Jean	01/01/2000	jean.dupont@dupont.com	0478/ 45.45.46	02/ 555.55.5
2	Durand	Marie	10/08/2002	Durand.Marie@Durand.com	0478/ 45.45.47	02/ 555.55.5
3	Martin	Jean	15/03/2003	jean.Martin@Martin.com	0478/ 45.05.05	02/ 555.55.5

id	nom	prenom	date_naissance	email	gsm	telephon
3	Martin	Jean	15/03/2003	jean.Martin@Martin.com	0478/ 45.05.05	02/ 555.55.5

On voit que les champs `nom`, `prenom`, `date_naissance`, `email`, `gsm`, `telephone`, `rue`, `numero`, `cp` et `localité` sont redondants. En effet, on a deux fois les mêmes informations pour Jean Dupont. On a deux fois son nom, son prénom, sa date de naissance, son email, son gsm, son téléphone, son adresse, son numéro, son code postal et sa localité.

De plus, on a plusieurs fois les mêmes cours. Par exemple, le cours Python est écrit 2 fois. C'est une redondance de données.

Il est donc difficile de trouver une clé primaire pour cette table. En effet, si on prend le champ `id`, on aura plusieurs fois le même `id` pour un même étudiant. Si on prend le champ `cours`, on aura plusieurs fois le même `cours` pour un même étudiant.

Pour atteindre la `2FN`, nous allons procéder par étapes :

1. Créer une table `etudiant` qui contiendra les champs `id`, `nom`, `prenom`, `date_naissance`, `email`, `gsm`, `telephone`, `rue`, `numero`, `cp` et `localité`. Cette table contiendra les informations des étudiants.
2. Créer une table `cours` qui contiendra les champs `id` et `nom`. Cette table contiendra les informations des cours.
3. Créer une table `etudiant_cours` qui contiendra les champs `etudiant_id` et `cours_id`. Cette table contiendra les relations entre les étudiants et les cours.

etudiant (2FN)

id	nom	prenom	email	gsm	telephone	rue	nu
1	Dupont	Jean	jean.dupont@dupont.com	0478/ 45.45.46	02/ 555.55.55	Rue de l'insertion	1
2	Durand	Marie	Durand.Marie@Durand.com	0478/ 45.45.47	02/ 555.55.55	Rue des sélections	45
3	Martin	Jean	jean.Martin@Martin.com	0478/	02/	Rue de	10

id	nom	prenom	email	gsm	telephone	rue	nu
				45.05.05	555.55.55	l'instance	

cours (2FN)

id	nom
1	Python
2	PHP
3	HTML
4	Bootstrap

etudiant_cours (2FN)

etudiant_id	cours_id
1	1
1	2
2	1
3	3
3	4

On voit donc que la décomposition de la colonne cours en une table à part permet de supprimer les redondances de données. En effet, le cours Python n'est plus écrit 2 fois dans la table etudiant.

3.3 En résumé, comment passer de la `1FN` à la `2FN` ?

- Avant tout, il faut être en `1FN`.
- Décomposer les tables en plusieurs tables pour éliminer les redondances et les

dépendances partielles.

- Déplacer les champs qui ne dépendent pas de la clé primaire dans une autre table.
- Ajouter la clef primairedont dépendent ces champs déplacés dans la nouvelle table.

4. Troisième forme normale (3FN)

- 1FN: ses attributs sont atomiques (pas de champs multiples) et chaque ligne est unique (pas de doublons).
- 2FN: Les attributs non-clés dépendent de la clé primaire.

Une table est en **Troisième Forme Normale** si elle est en `2NF` il faut que Les attributs non-clés ne dépendent pas d'autres attributs non-clés. Cela aide à éliminer les dépendances transitives (lorsqu'un attribut dépend d'un autre attribut qui dépend lui-même de la clé primaire).

4.1 Exemple - Table etudiant

Reprenons notre table `etudiant` qui était en `2FN` :

etudiant (2FN)

id	nom	prenom	email	gsm	telephone	rue	nu
1	Dupont	Jean	jean.dupont@dupont.com	0478/ 45.45.46	02/ 555.55.55	Rue de l'insertion	1
2	Durand	Marie	Durand.Marie@Durand.com	0478/ 45.45.47	02/ 555.55.55	Rue des sélections	45
3	Martin	Jean	jean.Martin@Martin.com	0478/ 45.05.05	02/ 555.55.55	Rue de l'instance	10

Ici, on voit que la localité dépend du code postal. En effet, si on connaît le code postal, on connaît la localité. On peut donc supprimer la colonne `localité` et la remplacer par la colonne `cp` qui contient le code postal.

etudiant (3FN)

id	nom	prenom	email	gsm	telephone	rue	nu
1	Dupont	Jean	jean.dupont@dupont.com	0478/ 45.45.46	02/ 555.55.55	Rue de l'insertion	1
2	Durand	Marie	Durand.Marie@Durand.com	0478/ 45.45.47	02/ 555.55.55	Rue des sélections	45
3	Martin	Jean	jean.Martin@Martin.com	0478/ 45.05.05	02/ 555.55.55	Rue de l'instance	10

cp

cp	localité	Province
1000	Bruxelles	Bruxelles
4000	Liège	Liège
5300	Andenne	Namur

On voit donc que la décomposition de la colonne `localité` en une table à part permet de supprimer les dépendances transitives. En effet, le champ `localité` dépendait du champ `cp` qui dépendait lui-même de la clé primaire `id`. Maintenant, le champ `localité` dépend directement de la clé primaire `cp`. De plus, nous avons ajouté un champ `Province` qui dépend de la clé primaire `cp`.

4.2 En résumé, comment passer de la `2FN` à la `3FN` ?

- Avant tout, il faut être en `2FN`.
- Vérifier s'il existe des dépendances transitives, où un champ non-clé dépend d'un autre champ non-clé.
- Éliminer ces dépendances transitives en déplaçant les champs concernés dans des tables séparées et en établissant des relations appropriées.

IV. Le langage SQL

Nous allons maintenant manipuler les données qui se trouvent dans une base de données. Nous utiliserons un langage qui s'appelle le SQL. Les commandes SQL s'écrivent en MAJUSCULES par convention. Ne pas le faire ne provoquera pas une erreur.
Pour faire simple, voici les commandes de base que l'on utilise en SQL :

1. Chercher des informations avec `SELECT`
2. Ajouter des enregistrements avec `INSERT INTO`
3. Modifier des enregistrements avec `UPDATE`
4. Effacer des enregistrements avec `DELETE FROM`

Nous allons présenter chaque type de commande SQL au cours de ce chapitre.

1. SELECT: Chercher des informations

L'instruction la plus célèbre du langage SQL est sans conteste l'instruction SELECT. Cette instruction est utilisée pour faire chercher des résultats d'une table ou plusieurs tables.

Sa forme la plus simple est :

```
SELECT attr1, attr2, attr3, etc…
FROM NomTable ;
```

Ou encore

```
SELECT *
FROM NomTable ;
```

Ici le symbole * prendra tous les attributs de la table en question.|

Exemple:

```
SELECT Nom, Prenom, Sexe, Naissance
FROM eleve ;
```

Cette instruction SQL va nous lister TOUS les élèves de la table Eleve et affichera les attributs Nom, Prenom, Sexe, Naissance. Maintenant, si vous avez 10.000 élèves ça risque d'être le tsunami d'informations C'est pourquoi on couple avec l'instruction WHERE qui permet de mettre une expression booléenne pour restreindre la quantité d'informations reçues.

1.1 DISTINCT

Si l'on veut connaître par exemple toutes les nationalités de la table élève :

```
SELECT Nationalite
FROM eleve ;
```

Le problème ici, c'est que l'on va avoir autant de fois Belge que l'on a des étudiants belges. Donc si on a 12 belges et 1 camerounais, on aura comme résultats : 12 fois 'belges' et 1 fois 'camerounais'. Ce qui n'est pas exactement ce que l'on veut.
Pour y arriver, on va utiliser après notre **SELECT** le mot clef **DISTINCT**

```
SELECT DISTINCT Nationalite
FROM eleve ;
```

Et ici, nous n'aurons plus que 2 résultats : 'Belge' et 'Camerounais'. Dans ce cas, DISTINCT va supprimer les doublons.

2. WHERE : Filtrage

WHERE signifie Où en français. Le où indique qu'on attend une condition pour filtrer notre sélection. Seuls les enregistrements répondants à la condition seront affichés.
Exemple :

```
SELECT Nom, Prenom, Sexe, Naissance
FROM eleve
WHERE Sexe = 'F' ;
```

Nous afficherons ici tous les élèves qui sont du sexe 'F'. Le SGBD va parcourir tous les enregistrements et ne garder que les élèves qui rentrent dans la condition Sexe= 'F'.

a. WHERE : Opérateurs Booléens

Les opérateurs booléens pour Mysql dans un WHERE sont

- Différent: <> ou !=
- Egal: = Plus grand que : >
- Plus grand ou égale : >=
- Plus petit que : <
- Plus petit ou égale : <=
- ET : **AND**
- Ou : **OR**
- Est Null : **IS NULL**
- N'est pas NULL : **IS NOT NULL**
- Par/Comme : **LIKE** par exemple Nom **LIKE** 'Pi%' Cherchera les noms commençait par Pi
- Entre : **BETWEEN** Valeur1 **AND** Valeur2
- Dans : **IN** par Exemple : CP **IN** (6980 , 4000)|

b. WHERE : AND et OR

Les opérateurs sont à ajoutés dans la condition **WHERE**. Ils peuvent être combinés à l'infini pour filtrer les données comme souhaités.
L'opérateur **AND** permet de s'assurer que la condition1 ET la condition2 sont vrai :

```
SELECT nom_colonnes
FROM nom_table
WHERE condition1 AND condition2;
```

L'opérateur **OR** vérifie quant à lui que la condition1 OU la condition2 est vrai :

```
SELECT nom_colonnes FROM nom_table
WHERE condition1 OR condition2;
```

Ces opérateurs peuvent être combinés à l'infini et mélangés. L'exemple ci-dessous filtre les résultats de la table “nom_table” si condition1 ET condition2 OU condition3 est vrai :

```
SELECT nom_colonnes FROM nom_table
WHERE condition1 AND (condition2 OR condition3) ;
```

Attention : il faut penser à utiliser des parenthèses lorsque c'est nécessaire. Cela permet d'éviter les erreurs car ça améliore la lecture d'une requête pour un humain.
Exemple de données. Ici vous devrez importer la petite base de données : Ventes avec la commande source databaseVentes.sql dans le répertoire exercices du cours.
Pour illustrer les prochaines commandes, nous allons considérer la table “produit” suivante :

IdProduit	Nom	Categorie	Stock	Prix
1	Ordinateur	Informatique	5	950
2	Clavier	Informatique	32	35
3	Souris	Informatique	16	30
4	Crayon	Fourniture	147	2

c. Opérateur AND

L'opérateur **AND** permet de joindre plusieurs conditions dans une requête. En gardant la même table que précédemment, pour filtrer uniquement les produits informatiques qui sont presque en rupture de stock (moins de 20 produits disponible) il faut exécuter la requête suivante :

```
SELECT *
FROM produit
WHERE categorie = 'informatique' AND stock < 20;
```

Cette requête retourne les résultats suivants :

IdProduit	Nom	Categorie	Stock	Prix
1	Ordinateur	Informatique	5	950
3	Souris	Informatique	16	30

d. Opérateur OR

Pour filtrer les données pour avoir uniquement les données sur les produits “ordinateur” ou “clavier” il faut effectuer la recherche suivante :

```
SELECT *
FROM produit
WHERE nom = 'ordinateur' OR nom = 'clavier';
```

Cette simple requête retourne les résultats suivants :

IdProduit	Nom	Categorie	Stock	Prix
1	Ordinateur	Informatique	5	950
2	Clavier	Informatique	32	35

e. Combiner AND et OR

Il ne faut pas oublier que les opérateurs peuvent être combinés pour effectuer de puissantes recherches. Il est possible de filtrer les produits “informatique” avec un stock inférieur à 20 OU les produits “fourniture” avec un stock inférieur à 200 avec la recherche suivante :

```
SELECT * FROM produit
WHERE ( categorie = 'informatique' AND stock < 20 )
OR ( categorie = 'fourniture' AND stock > 200 ) ;
```

Cela permet de retourner les 3 résultats suivants :

IdProduit	Nom	Categorie	Stock	Prix
1	Ordinateur	Informatique	5	950
2	Clavier	Informatique	16	35
4	Crayon	Fourniture	147	2

f. BETWEEN .... AND....

On se limite de conserver les lignes dont le champ spécifié est compris dans un intervalle. Les limites sont comprises.

Par exemple on veut afficher les élèves nés dans les années 90. Pour le tri portera sur le sexe, nom, Prenom. Trier de cette manière permettra de trier en premier les filles puis les garçons.

```
SELECT Nom, Prenom, Naissance, Sexe
FROM eleve
WHERE Naissance BETWEEN '1990/01/01' AND '1999/12/31'
ORDER BY Sexe, Nom, Prenom;
```

On pourrait bien entendu utiliser simplement l'opérateur AND :

```
SELECT Nom, Prenom, Naissance, Sexe
FROM eleve
WHERE Naissance >= '1990/01/01' AND Naissance <= '1999/12/31'
ORDER BY Sexe, Nom, Prenom;
```

g. Appartenance (IN)

On ne va garder que les enregistrements dont un champ est compris dans une liste de valeur. Ça nous évite d'écrire une multitude d'opérateurs OR.
Prenons les élèves qui ont comme CP soit 6890, 1348 et 1490 :

```
SELECT Nom, Prenom, Naissance, Sexe
FROM eleve
WHERE CP IN (1490, 6890, 1348);
```

Cette requête aurait pu s'écrire de cette manière :

```
SELECT Nom, Prenom, Naissance, Sexe
FROM eleve
WHERE CP=1490 OR CP = 1348 OR CP = 6890 ;
```

Ce n'est donc pas une obligation d'utiliser le IN ainsi que le BETWEEN mais je trouve qu'ils peuvent simplifier grandement la lecture. Si vous préférer faire vos requêtes autrement sans les utiliser, c'est très bien aussi.

h. Ressemblance LIKE

L'opérateur LIKE permet de faire plusieurs types de recherches :

- Champ qui commence par AB : LIKE 'AB% '
- Champ qui se termine par AB : LIKE '%AB '
- Champ qui contient AB : LIKE '%AB% '

Recherchons les rues qui se terminent par 'SGBD' :

```
SELECT Prenom, Nom, Rue
FROM eleve
WHERE RUE LIKE '%SGBD' ;
```

Recherchons les rues qui commencent par 'Place' ;

```
SELECT Prenom, Nom, Rue
FROM eleve
WHERE RUE LIKE 'Place%' ;
```

Et pour finir les rues qui contiennent ' des ' :

```
SELECT Prenom, Nom, Rue
FROM eleve
WHERE RUE LIKE '% des %' ; #Notez Ici qu'il y a des espaces avant et après 'des'. Pourquoi
```

Ici, je vais vous montrer des exemples de cherches sur des dates à l'aide d'un LIKE.
Normalement, on n'utilise pas un like mais plutôt des fonctions comme MONTH() ou YEAR() qui rendront vos requêtes plus efficaces/rapides. Ici, c'est juste pour vous donner des exemples supplémentaires.
Rechercher les personnes nées au mois de mars :

```sql
SELECT Nom, Prenom, Naissance, Sexe
FROM eleve
WHERE NAISSANCE LIKE '%-03-%'; #On ferait normalement MONTH(Naissance) = 3 ;
```

Ou encore nées dans les années 1990 :

```sql
SELECT Nom, Prenom, Naissance, Sexe
FROM eleve
WHERE NAISSANCE LIKE '199%'; #On ferait normalement YEAR(Naissance) BETWEEN 1990 AND 1999
```

i. NULL

Lorsque l'on veut voir si un champ n'a pas de valeur, c'est-à-dire la valeur **NULL**. On n'utilise pas les opérateurs d'égalité/d'inégalité. On utilise **IS NULL** (pour égale à **NULL**) et **IS NOT** (pour n'est pas égale à **NULL**)

La valeur **NULL** ne pas s'écrire entre guillemets car elle pourrait être confondue par une chaîne de caractère. C'est vraiment l'absence de valeur et s'écrit **NULL**.

Exemple **IS NOT NULL**: Les élèves qui ont un numéro de Téléphone.

```sql
SELECT *
FROM eleve
WHERE Tel IS NOT NULL ;
```

Exemple **IS NULL** : Les élève qui n'ont pas de Téléphone

```sql
SELECT *
FROM eleve
WHERE Tel IS NULL ;
```

3. Champs calculés

Comme je vous ai dit en classe, on ne met pas de champ dans une table qui serait le résultat d'un calcul. Exemple : On a le prix du produit. Il serait inutile de faire un champ PrixTVAC. En effet,

cette colonne va prendre de la place dans notre base de données. Si vous avez un million de produits, vous avez 1 million de valeurs inutiles. En effet, on pourrait procéder par exemple de la manière suivante :

```
SELECT Nom, Categorie, Stock, Prix, Prix + Prix * 0.21 AS PrixTVAC
FROM Produit ;
```

Sinon, généralement on effectue le calcul du prix TVAC dans un langage de programmation en récupérant le prix HTVA.

4. ORDER BY: Les tris

Avoir des données de notre base de données, c'est déjà bien. Mais si en plus le SGBD peut nous les trier selon l'ordre que nous voulons, c'est encore mieux ! C'est là qu'entre en scène ORDER BY.

Par défaut le tri est ascendant : ordre croissant **ASC** c'est pourquoi on ne le met pas mais on peut le mettre : n'oubliez pas l'informaticien est fainéant.

Soit classer nos élèves par Nom de famille :

```
SELECT Nom, Prenom, Naissance, Sexe
FROM eleve
ORDER BY Nom ;
```

Classer nos élèves par Nom et puis par prénom : Ce cas est intéressant si nous avons plusieurs mêmes noms de famille, le SGBD classera alors ensuite sur le prénom.

```
SELECT Nom, Prenom, Naissance, Sexe
FROM eleve
ORDER BY Nom, Prenom ;
```

Pour un tri décroissant, on utlise **DESC**.

Si l'on veut classer nos élèves dans l'ordre décroissant sur le nom puis croissant sur le prénom :

```sql
SELECT Nom, Prenom, Naissance, Sexe
FROM eleve
ORDER BY Nom DESC, Prenom ASC ;
```

5. Fonctions d'agrégation

Les fonctions d'agrégation dans le langage SQL permettent d'effectuer des opérations statistiques sur un ensemble d'enregistrement. Étant donné que ces fonctions s'appliquent à plusieurs lignes en même temps, elles permettent des opérations qui servent à récupérer l'enregistrement le plus petit, le plus grand ou bien encore de déterminer la valeur moyenne sur plusieurs enregistrements.
Les fonctions d'agrégation sont des fonctions idéales pour effectuer quelques statistiques de bases sur des tables.

5.1 La moyenne - AVG

La fonction AVG() permet de calculer une valeur moyenne sur un ensemble d'enregistrement de type numérique et non nul.
Exemple : Trouver l'âge moyen de la table Eleve ;

```sql
SELECT AVG(YEAR(CURDATE())-YEAR(Naissance)) AS AgeMoyen
FROM eleve;
```

Petite explication sur la précédente requête. Pour connaître l'âge approximatif d'un élève on retire de l'année en cours l'année de naissance. C'est sur ce résultat de tous les élèves qu'on fera la moyenne.
La fonction CURDATE() retourne la date du jour et YEAR() l'année d'une date.

5.2 Le Minimum - MIN

La fonction MIN() retourne la valeur minimum d'un champ parmi tous les enregistrements non nuls ;
Exemple : le prix minimum d'un produit

```sql
SELECT MIN(prix)
FROM Produit ;
```

5.3 Le Maximum – MAX

La fonction MAX() retourne la valeur Maximum d'un champ parmi tous les enregistrements non nuls ;

```sql
SELECT MAX(prix)
FROM Produit ;
```

5.4 La Somme – SUM

La fonction SUM() retourne la somme de toutes les valeurs non nuls pour un champ donné d'une table.
Exemple : Calculer le nombre total d'articles disponibles de la catégorie informatique.

```sql
SELECT SUM(Stock) AS Total
FROM Produit
WHERE Categorie = 'Informatique' ;
```

5.5 COUNT(*)

Si nous voulons connaître le nombre d'élèves faisant partie de la table eleve : on utilise COUNT

```sql
SELECT COUNT(*)
FROM eleve ;
```

On va voir que le résultat sera une colonne avec comme nom : count(*) avec une cellule ayant 14 comme valeur. Il y a donc 14 élèves.
Cependant le nom de la colonne n'est pas très agréable à lire, on peut lui donner un autre nom simple :

```sql
SELECT COUNT(*) AS NB_Eleves
FROM eleve ;
```

On peut utiliser le mot clef AS qui signifie Comme. On pourrait aussi l'omettre et mettre un espace :

```
SELECT COUNT(*) NB_Eleves
FROM eleve ;
```

Si nous voulions connaître le nombre de garçons parmi nos élèves :

```
SELECT COUNT(*) AS NB_Garcons
FROM eleve
WHERE Sexe = 'M' ;
```

Nous avons comme résultats 11 garçons. Faisons la même chose avec les filles, nous devrions en avoir : 14-13 = 1 fille

```
SELECT COUNT(*) AS NB_Filles
FROM eleve
WHERE Sexe = 'F' ;
```

Dernier exemple, nous voulons connaître le nombre d'élèves nés de 1990 à aujourd'hui :

```
SELECT COUNT(*)
FROM eleve
WHERE Naissance >= '1990/01/01' ;
```

5.6 COUNT(champ)

Ici, cette syntaxe permet de compter le nombre de valeurs non nulles pour un champ donné. Si on veut connaître le nombre de Tel non nul :

```
SELECT COUNT(Tel)
FROM eleve ;
```

qui est équivalent à :

```sql
SELECT COUNT(*) AS NB_Tel
FROM eleve
WHERE Tel IS NOT NULL ;
```

Ajoutons une colonne pour connaître le nombre de boites non nul :

```sql
SELECT COUNT(Tel) AS NB_Tel, COUNT(boite) AS NB_boites
FROM eleve;
```

6. GROUP BY

La clause GROUP BY en SQL permet d'organiser des données identiques en groupes à l'aide de certaines fonctions. C'est-à-dire si une colonne particulière a les mêmes valeurs dans différentes lignes, elle organisera ces lignes dans un groupe.
Prenons un exemple qui a été demandé par un étudiant l'autre jour quand on a vu le DISTINCT pour le sexe. Il a demandé comment savoir combien on a d'hommes et de femmes ? Je peux enfin lui répondre

```sql
SELECT Sexe, Count(*) AS Nombre
FROM eleve
GROUP BY Sexe;
```

Explications:

On va regrouper les élèves par sexe. On va donc avoir un groupe pour les filles et un groupe pour les garçons. Ensuite, on va compter le nombre d'élèves dans chaque groupe. On va donc avoir comme résultats :

Sexe	Nombre
F	1
M	13

Ou bien si l'on veut avoir la moyenne des prix par catégorie :

```
SELECT Categorie, AVG(Prix)
FROM Produit
GROUP BY Categorie;
```

Categorie	AVG(Prix)
Fourniture	2
Informatique	338.33

Ou savoir combien j'ai en stock par catégorie tout produit confondu :

```
SELECT Categorie, SUM(Stock)
FROM Produit
GROUP BY Categorie;
```

Categorie	SUM(Stock)
Fourniture	147
Informatique	53

> La liste des champs qui se trouvent dans le GROUP BY doit se trouver dans le SELECT.
>

6Bis.1 HAVING

Si l'on veut maintenant filtrer le résultat d'un regroupement (GROUP BY) on va utiliser le mot clé HAVING (qui peut se traduire par "ayant"). Donc pour rechercher sur un regroupement on utilise HAVING et non un WHERE. Si l'on utilise HAVING sans regroupement, celui-ci agira comme un WHERE classique.

Soit afficher les CP ayant plusieurs communes:

```
USE Localites;
SELECT CP, COUNT(*) AS nb
FROM Localite
GROUP BY CP
HAVING nb >1;
```

On va regrouper les CP et compter le nombre de fois qu'ils apparaissent. On va ensuite filtrer les CP qui apparaissent plus d'une fois. Si on a un CP qui apparaît plus d'une fois, c'est qu'il y a plusieurs communes avec ce CP.

7. INNER JOIN: Jointure entre tables.

Maintenant que nous savons lire/sélectionner des données depuis une table.
Il est parfois nécessaire de lire les données depuis plusieurs tables en même temps. Et de n'afficher que certaines données de ces tables.

Il faut essayer de trouver un point d'anchrage dans chaque table. Essayer de lier nos tables entre elles. Pour cela, on lie une table à une autre grâce aux clefs primaires/étrangères.

Tiens qu'est-ce encore qu'une clef primaire/étrangère ? 😀

7.1 Jointure sur deux tables

Reprenons les tables eleve et formation.

Table formation:

id (Clef primaire)	Nom	Lieu
1	BlindCode	BXL
2	BlindCodeJava	Mons

Table eleve: Elle a été épurée pour l'exemple. Dans les exercices, elle contient plus de champs.

id (Clef primaire)	Prenom	Nom	formation_id (clef étrangère)
1	Karim	Bouchaïb	1

id (Clef primaire)	Prenom	Nom	formation_id (clef étrangère)
2	Amir	El Gharbi	1
3	Baptiste	Brasseur	1
4	Carmen Ramona	Todorut	1
5	ANTHONY	VELEZ PAEZ	1
6	Filippo	Muratore	1
7	Nabil	Elrhanaoui	1
8	Thomas	Ardui	1
9	Christian	Honore	1
10	Sébastien	Baloge	2
11	David	Dehoust	2
12	Simon	Desseille	2
13	Christian	Vanneste	2
14	Bruno	Defalque	2

Si nous voulons afficher tous les élèves et leur formation, nous faisions ceci en SQL:

```
SELECT *
FROM eleve;
```

Ca nous affiche tous le champs mais malheureusement le champ relatif à la formation (formation_id) est un nombre. Ce nombre est la clef étrangère qui fait référence à la clef primaire de la table Classe.

Si on veut lier/joindre nos tables pour afficher le nom de la classe au lieu d'un identifiant, nous allons utiliser la commande sql suivante: **INNER JOIN**

```sql
SELECT eleve.Nom, Prenom, formation.Nom
FROM eleve
INNER JOIN formation ON eleve.formation_id = formation.id;
```

Décortiquons cette requête:

- Le **SELECT** est particulier car on a mis `eleve.Nom` et `formation.Nom` pour éviter une ambiguité. En effet, MySQL ne saura pas si on veut le nom de l'élève ou le nom de la formation si on ne spécifie pas la table.
- **FROM** `eleve` : On veut prendre des informations de la table `eleve` .
- **INNER JOIN** `formation` **ON** `eleve.formation_id` = `formation.id`
    i. **INNER JOIN** `formation` : on veut joindre la table `formation` à la table `eleve` (du **FROM**).
    ii. **ON** `eleve.formation.id` **=** `formation.id` : On dit comment on va lier nos tables. Ici on va lier l'`formation.id` de la table `eleve` sur (**ON**) l'`id` de la table `formation` . Le SGBD cherchera les enregistrements dans les deux tables où `formation_id` de la table `eleve` = à l'`id` de la table `formation` . On fait donc une égalité sur la clé étrangère de la table `eleve` (`eleve.formation_id`) et sur la clé primaire de la table `formation` (`formation.id`).

Si on a besoin de tous les champs de la table `eleve` et de la table `formation` , on peut changer la requête de cette manière pour éviter de taper tous les champs:

```sql
SELECT eleve.*, formation.*
FROM eleve
INNER JOIN formation ON eleve.formation_id = formation.id;
```

Ou bien ainsi

```sql
SELECT *
FROM eleve
INNER JOIN formation ON eleve.formation_id = formation.id;
```

Il suffit donc d'utiliser le nom de table suivit d'un point et du symbole *****: SELECT nomtable.*****

Avant (les vieux comme moi), on ne faisait pas d'**INNER JOIN** mais on faisait la jointure de table dans un **WHERE**. Notre précédente requête peut s'écrire de cette manière:

```
SELECT eleve.Nom, Prenom, formation.Nom as Formation
FROM eleve, formation
WHERE eleve.formation_id = formation.id ;
```

Mais c'est à déconseiller car c'est plus logique de faire la jointure via un **INNER JOIN**. Le **WHERE** est plutôt là pour faire des recherches spécifiques et non pour les jointures. Je vous le montre car vous verrez parfois des personnes faire des jointures de tables non pas via un **INNER JOIN** mais via un **WHERE**. Au final, on optient le même résultat... Et c'est ce qui compte. 😉

Si par exemple on veut afficher le nom de l'élève, le prénom de l'élève et le nom de la formation des élèves masculins dont le nom de famille commence par un 'b':

```
SELECT eleve.Nom, Prenom, formation.Nom as Formation
FROM eleve
INNER JOIN formation ON eleve.formation_id = formation.id
WHERE Sexe='M' AND eleve.nom LIKE 'b%' ;
```

7.2 Utilisation du AS dans un FROM et INNER JOIN

Lors d'une requête avec un `INNER JOIN`, il peut être intéressant de raccourcir le nom des tables via l'utilisation de `AS`

Reprenons l'exemple suivant:

```
USE Ventes;
SELECT ProduitV2.IdProduit, ProduitV2.Nom, Categorie.Nom
FROM ProduitV2
INNER JOIN Categorie ON ProduitV2.IdCategorie = Categorie.IdCategorie
WHERE ProduitV2.IdProduit <> 4;
```

En utilisant `AS` on peut réduire la requête et lui donner une meilleure visibilité:

```
USE Ventes;
SELECT p.IdProduit, p.Nom, c.Nom
FROM ProduitV2 AS p
INNER JOIN Categorie AS c ON p.IdCategorie = c.IdCategorie
WHERE p.IdProduit <> 4;
```

Autre exemple:

```
USE Ventes;
SELECT ProduitV2.IdProduit, ProduitV2.Stock, ProduitV2.Prix, ProduitV2.Nom AS NomProduit,
FROM ProduitV2
INNER JOIN Categorie ON ProduitV2.IdCategorie = Categorie.IdCategorie
ORDER BY NomProduit;
```

Devient:

```
USE Ventes;
SELECT p.IdProduit, p.Stock, p.Prix, p.Nom AS NomProduit, c.Nom AS NomCategorie
FROM ProduitV2 AS p
INNER JOIN Categorie AS c ON p.IdCategorie = c.IdCategorie
ORDER BY NomProduit;
```

7.3 Jointure sur plus de 2 tables

C'est le même principe que pour deux tables.
Imaginons que nous devons lier 3 tables:

```
SELECT champ1, champ2, champ3, champX
FROM table1
INNER JOIN table2 ON table1.FK_Key = table2.PK_Key
INNER JOIN table3 ON table2.FK_Key = table3.PK_Key;
```

Où les FK_Key seraient les clefs étrangères (Foreign Key) et les PK_Key seraient les clefs primaires.

L'ordre des `INNER JOIN` a son importance. On pourrait pas faire ceci:

```
SELECT champ1, champ2, champ3, champX
FROM table1
INNER JOIN table3 ON table2.FK_Key = table3.PK_Key;
INNER JOIN table2 ON table1.FK_Key = table2.PK_Key
```

En effet, on ne peut pas faire un INNER JOIN sur une table qui n'existe pas encore. Il faut donc que les tables soient dans l'ordre d'utilisation. Ici, on va d'abord lier table1 à table2 puis table2 à table3

Ici, j'ai lié table2 à table1 et table3 à table2. Mais tout dépend de la situation réelle. Vous aurez un exemple concret à l'exercice n°24.

7b. Limitation des résultats - LIMIT

7b.1 Simplement limiter le nombre de résultats

Si on veut limiter le nombre de résultats, on peut utiliser le mot clef **LIMIT**.

Pour afficher les 5 premiers élèves de la table eleve triés par nom:

```
USE BlindCode;
SELECT *
FROM eleve
ORDER BY Nom ASC
LIMIT 5;
```

Donc il faut bien faire attention au fait que l'on limite le résultat à 5 enregistrements. Si on avait mis LIMIT 10, on aurait eu les 10 premiers élèves. Et donc que c'est un résultat tronqué. On n'a pas tous les élèves. On a juste les 5 premiers.

7b.2 Pour avoir les premiers résultats ou les derniers résultats

Il est souvent utilisé avec **ORDER BY** pour avoir les premiers résultats ou les derniers résultats.

vous pouvez créer une requête SQL pour afficher, par exemple, la formation ayant l'âge moyen

des élèves le plus élevé et l'âge moyen le plus bas, en utilisant GROUP BY, ORDER BY et LIMIT.

Pour cela, vous pouvez d'abord calculer l'âge moyen des élèves dans chaque formation, puis trier ces moyennes pour trouver la plus haute et la plus basse. Voici comment cela peut être fait :

Pour afficher la formation ayant l'âge moyen le plus élevé :

```
USE BlindCode;
SELECT formation.Nom, AVG(YEAR(CURDATE())-YEAR(Naissance)) AS AgeMoyen
FROM eleve
INNER JOIN formation ON eleve.formation_id = formation.id
GROUP BY formation.Nom
ORDER BY AgeMoyen DESC
LIMIT 1;
```

On peut voir que l'on a trié par ordre décroissant sur l'âge moyen et on a limité à 1. On aura donc le premier résultat qui sera la formation ayant l'âge moyen le plus élevé.

Trouver la formation avec l'âge moyen des élèves le plus bas :

```
USE BlindCode;
SELECT formation.Nom, AVG(YEAR(CURDATE())-YEAR(Naissance)) AS AgeMoyen
FROM eleve
INNER JOIN formation ON eleve.formation_id = formation.id
GROUP BY formation.Nom
ORDER BY AgeMoyen ASC
LIMIT 1;
```

On peut voir que l'on a trié par ordre croissant sur l'âge moyen et on a limité à 1. On aura donc le premier résultat qui sera la formation ayant l'âge moyen le plus bas.

Bien entendu, on peut demander les 3 élèves les plus âgés de la formation BlindCode en modifiant la requête de cette manière en mettant LIMIT 3 au lieu de LIMIT 1.

8.a Création d'une base de données - CREATE DATABASE

Avant de pouvoir créer nos tables, nos devrons avant tout créer une base de données.
Pour cela on utilise la commande suivante **CREATE DATABASE**

Si je veux créer la base de données Jeux:

```
CREATE DATABASE Jeux;
```

Si j'exécute cette commande, le SGDB va créer notre base de données Jeux. C'est aussi simple que ça. 😀

Cependant, si je réexécute cette instruction mysql me dira que la base de données existe déjà.

C'est pour ça que pour les exercices, je supprime la base de données avant de la créer. De cette manière, vous pouvez modifier comme vous le souhaitez nos bases de données. Après un appel du script de création tout sera supprimé et recréé:

```
DROP DATABASE IF EXISTS Jeux;
CREATE DATABASE Jeux;
```

Attention donc qu'un **DROP DATABASE** supprime toute la base de données ! A utiliser avec les précautions qui s'imposent...

8.b Jeu de caractères - CHARACTER SET

Lors de la création d'une base de données, on peut spécifier le jeu de caractères à utiliser. Par défaut, c'est **latin1**. Mais on peut spécifier **utf8** ou **utf8mb4**.

Pourquoi utiliser **utf8mb4** ? Car il permet de stocker des émojis. En effet, **utf8** ne permet pas de stocker des émojis. Si vous voulez stocker des émojis, il faut utiliser **utf8mb4**. Ensuite utf8 est sur 3 octets et utf8mb4 est sur 4 octets.

Pour spécifier le jeu de caractères, on utilise **CHARACTER SET** suivi du jeu de caractères. Par exemple, pour créer une base de données avec le jeu de caractères **utf8mb4**:

```
CREATE DATABASE Jeux CHARACTER SET utf8mb4;
```

8.c Collation - COLLATE

Non, il n'est pas question ici de manger des collations. 😄 Même si vous pouvez manger des collations pendant que vous créez votre base de données. 😄

La collation permet de spécifier l'ordre de tri des caractères. Pour un jeu de caractères donné, il existe plusieurs collations.

Nous utiliserons la collation **utf8mb4_unicode_ci**. Ci signifie Case Insensitive. C'est-à-dire que les caractères majuscules et minuscules seront considérés comme identiques. Par exemple, si on trie sur le nom de famille, on aura les noms de famille commençant par une majuscule et ceux commençant par une minuscule.

Pour spécifier la collation, on utilise **COLLATE** suivi de la collation. Par exemple, pour créer une base de données avec le jeu de caractères **utf8mb4** et la collation **utf8mb4_unicode_ci**:

```
CREATE DATABASE Jeux CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Ne vous prenez pas la tête avec les collations. Utilisez **utf8mb4_unicode_ci** et ça ira très bien. 😄

9. CREATE TABLE

9.1 La commande

Lors de vos échanges en groupe, vous avez réussi à déterminer les champs primordiaux des entités Eleve et Classe.

Ensuite, dans le script de création de la base de données, nous avons vu que la creation d'une table avait des points communs avec notre entité: champs, clefs, type de donnée, **NULL**, **NOT NULL**, etc.

La commande **CREATE TABLE** permet donc de créer une table dans une base de données.

9.2 La syntaxe

De manière très résumée, elle se traduit ainsi:

```
USE UneBaseDeDonnees;
CREATE TABLE nomtable
(
    champ1 type_donnees,
    champ2 type_donnees,
    champ3 type_donnees,
    champ4 type_donnees
);
```

Ici, on utilise la base de données ayant pour nom: `UneBaseDeDonnees` et nous avons créé une table avec pour nom: nomtable.

Nous avons 4 champs. Chaque champ a une nom (champ1 à champ2) et chaque champ a un type de données (**INT**, **DATE**, **VARCHAR**, **CHAR**, etc.). Entre chaque champ, il y a une virgule.

Notez l'utilisation du mot clef `USE` qui permet de changer de base données. En effet, soyez bien sûrs d'être dans la base de données avant de faire des créations de tables ou toute autre manipulation.

9.3 Types de données / types des champs

Un champ a un type de données. Voici les principaux:

9.3.1 VARCHAR - Chaîne de caractères à longueur variable

Ce type de donnée est utilisé pour les chaînes de caractères. On lui donne une longueur maximum. Lors de l'affectation de notre enregistrement pour ce champ si la longueur, n'est pas atteinte, il ne complètera pas par des espaces.

```
nom VARCHAR(50)
```

9.3.2 CHAR - Chaîne de caractère à longueur fixe

Ici aussi c'est une chaîne de caractères. A la différence que si lors de l'affectation, la chaîne est

plus petite que la longueur définie, MySQL remplira d'espaces le reste des caractères non affectés. Donc notre chaîne aura toujours une taille fixe même si on affecte une chaîne avec moins de caractères que la taille fixe.

```
prenom CHAR(50)
```

Ou bien pour un champ Sexe.

```
Sexe CHAR(1)
```

9.3.3 INT / TINYINT / SMALLINT / MEDIUMINT / BIGINT

Alors très très souvent vous allez mettre des INT pour des champs entiers.
Cependant il faut bien réfléchir avant de faire son choix sur le type entier à utiliser.

En effet, la limite négative et positive varie en fonction du type. Et choisir un type à un impacte sur la taille de stockage du champ.

Il faut savoir qu'il existe des types signés et non signés. Signé c'est qu'on tient compte des valeurs négatives.
Non signé, on ne tient pas compte de valeur négative. Par défaut c'est **SIGNED** mais on peut qualifier le type de **UNSIGNED**.

- **TINYINT**: 1 octet, valeurs [-128, 127] ou bien 255 valeurs positives.
- **SMALLINT**: 2 octets, valeurs [-32768, 32767] ou bien 65535 valeurs positives.
- **MEDIUMINT**: 3 octets, valeurs [-8388608, 8388607] ou bien 16777215 valeurs positives.
- **INT**: 4 octets, valeurs [-2147483648, 2147483647] ou bien 4294967295 valeurs positives.
- **BIGINT**: 8 octets, valeurs [-2^63, 2^63-1] ou bien 264-1 valeurs positives. (^ signifie ici exposant)

Par exemple avec le type **BIGINT**, si dans une table un champ a le type **BIGINT** et que vous avez 128 enregistrements, ça prendra 1 Ko de stockage uniquement pour ces champs **BIGINT**.

Ca peut paraître fort peu mais il faut parfois imaginer des bases de données énormes pour se rendre compte que notre base de données aurait pu prendre nettement moins de place en

utilisant par exemples:

- un **INT**: 2 fois moins de place en stockage.
- un **SMALLINT**: 4 fois moins de place en stockage.

```
IdUser INT UNSIGNED,
Annee SMALLINT,
NBEnfants TINYINT UNSIGNED //Normalement TINYINT devrait suffire... 😁
```

Dans le cas IdUser, j'ai pris **INT**. Pourquoi ? Imaginons que votre application dépasse les 50 millions d'utilisateurs, vous ne pouvez donc pas prendre le type **MEDIUMINT**. Mais plutôt le type **INT** qui pourra en gérer 4 milliards... Mais si vous être Facebook (2,85 milliards d'utilisateurs), Microsoft, Google, Apple, ce type de donnée ne sera pas suffisant et il faudra sans doute passer par un type **BIGINT**...

Maintenant, je n'ai aucune idée du design des DB des GAFAM (Google Amazon Facebook Microsoft) mais c'est juste pour vous montrer que le nombre d'utilisateurs est fonction de vos besoins: Le nombre d'utilisateurs Facebook vs nombre de membres d'un club de foot...

Le choix du type est donc TRES important. Mais tout va dépendre de la taille de la base de données. Pour les types entiers, vous verrez quasi toujours des **INT** même quand ça n'est pas justifié...

9.3.4 FLOAT / DOUBLE / DECIMAL

Pour les nombres à virgule flottante vous devrez choisir où vous aurez besoin de la plus grande précision.

- FLOAT: simple précision (4 octets)
- DOUBLE: double précision (8 octets)

```
pourcentage FLOAT(5,2)
```

5 représente le nombre total de chiffres et 2, le nombre de décimales. 'pourcentage' stockera les valeurs entre -999.99 et 999.99.

Le problème réside dans leurs approximations:

- Une valeur telle que 1 / 3.0 = 0.3333333... sera stockée sous la forme 0.33 (2 décimales)
- Une valeur telle que 33.009 sera stockée sous la forme 33.01 (arrondie à 2 décimales)

Le type DECIMAL(M,D), utilisez-le lorsque vous vous souciez de la précision exacte, comme de l'argent.

```
salaire DECIMAL(8,2)
```

8 est le nombre total de chiffres, 2 le nombre de décimales. 'salaire' sera dans la plage de -999999.99 à 999999.99

La précision se définit par DECIMAL(M,D) où M représente le nombre total de chiffres allant de 1 à 65 et où D représente le nombre de chiffres après la virgule allant de 1 à 30. D doit être inférieur à M.

Donc si on écrit DECIMAL(65,30), on aura à gauche de la virgule un nombre a 35 (65-30) chiffres et on aura 30 chiffres après la virgule. Cela définit ici un champ qui accepte un nombre entre: -99999999999999999999999999999999.999999999999999999999999999999 et 99999999999999999999999999999999.999999999999999999999999999999

9.3.5 DATE / DATETIME

Pour les Dates on peut choisir entre: DATE et DATETIME.

- Le type DATE ne prend qu'une DATE.
- Le type DATETIME prend une date et aussi une heure. Si on n'a pas besoin de l'heure autant utiliser le type DATE.

Un champ nommé DateNaissance peut avoir besoin d'un simple DATE pour la table USER.

```
DateNaissance DATE NOT NULL
```

Tandis que dans un hopital, ce champ DateNaissance sera du type DATETIME car il est important de connaître la date et l'heure exacte de la naissance d'un nouveau né par exemple.

```
DateNaissance DATETIME NOT NULL
```

9.3.6 BOOLEAN

Ce type permet de définir une valeur booléenne.

```
EstActif boolean NOT NULL
```

Cependant, il faut faire attention car en fait c'est le type TINYINT qui est utilisé par MySQL. Donc on pourrait avoir une valeur comprise entre 0 et 255.

Lors de l'insert on utilisera soit 0 ou 1 pour rester cohérent. On peut aussi utiliser true ou false qui seront remplacé par MySQL par 0 ou 1.

```
INSERT INTO User(Nom, Prenom, EstActif) VALUES('Piette','Johnny', true);
INSERT INTO User(Nom, Prenom, EstActif) VALUES('Dupont','Philip', 0);
```

Lors d'un SELECT il sera affiché pour le champ EstActif soit 0 ou 1.

9.4 NULL / NOT NULL

A droite du type de donnée on peut ajouter **NULL** ou **NOT NULL**.

- **NULL** signifie que la donnée peut être nulle. C'est par défaut. Donc si vous ne mettez pas **NULL**, c'est comme si vous le mettiez.
- **NOT NULL** signifie que la donnée ne peut être nulle.

```
Nom VARCHAR(20) NOT NULL,
Lieu VARCHAR(20) NOT NULL,
Nickname VARCHAR(20) NULL
```

9.5 DEFAULT

DEFAULT permet de définir une valeur par défaut.

```
Actif BOOLEAN DEFAULT TRUE
```

On définit un champ booléen nommé Actif ayant vrai (true) comme valeur par défaut.

9.6 PRIMARY KEY

Dans MySQL, une clef primaire se définit soit

- sur le champ en le qualifiant de **PRIMARY KEY**.
- après la définition de tous les champs de la table avec **PRIMARY KEY**. (identifiant_de_la_clef_primaire)

```
CREATE TABLE Toto(
  ID int NOT NULL PRIMARY KEY,
  Nom VARCHAR(30) NOT NULL
)
```

ou bien

```
CREATE TABLE Toto(
  ID int NOT NULL,
  Nom VARCHAR(30) NOT NULL,
  PRIMARY KEY (ID)
)
```

Dans une clef primaire, il n'est pas nécessaire de mettre **NOT NULL**. En effet, il est implicite que le champ soit **NOT NULL** vu que c'est une clef primaire. Lors de l'ajout d'un enregistremet si on omet la valeur de la clef primaire la base de données va générer une erreur.

Mais ce n'est pas une erreur de mettre **NOT NULL**. C'est juste inutile. Pour les exemples, j'ai mis **NOT NULL** pour que lors votre apprentissage vous compreniez bien qu'un champ de clef primaire doit avoir une valeur.

Cependant, j'ai lu que beaucoup de personnes ajoutent **NOT NULL** à une clef primaire par style et une relecture plus aisée. C'est aussi pour éviter le cas où MySQL aurait un bug qui permettrait un **NULL** où ne mettrait pas **NOT NULL** par défaut (très rare).

Update: Après une énième lecture, j'ai lu que dans la norme SQL-92, on devait explicitement ajouter **NOT NULL** à tout champ primaire:

> If **PRIMARY KEY** or **UNIQUE** is specified, then the **<column defi-nition>** for each column whose **<column name>** is in the **<unique column list>** shall specify **NOT NULL**.

Mais la norme SQL-99, ne l'oblige pas car c'est implicite que le champ est **NOT NULL**:

> If the **<unique specification>** specifies **PRIMARY KEY**, then for each **<column name>** in the explicit or implicit **<unique column list>** for which **NOT NULL** is not specified, **NOT NULL** is implicit in the **<column definition>**.

Voilà pourquoi aussi on peut trouver sur un champ clef primaire/unique la contrainte **NOT NULL** ou non.

9.7 FOREIGN KEY

Dans MySQL, une clef étrangère se définit après la définition de tous les champs de la table.

```
CREATE TABLE Personne (
    PersonID int UNSIGNED AUTO_INCREMENT,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int UNSIGNED,
    PRIMARY KEY (PersonID)
);

CREATE TABLE Commande (
    OrderID int UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    OrderNumber int UNSIGNED NOT NULL,
    PersonID int UNSIGNED NOT NULL,
    FOREIGN KEY (PersonID) REFERENCES Personne(PersonID)
);
```

Dans le précédent exemple on a ajouté une clef étrangère à la table Commande après la définition des champs:

```
FOREIGN KEY(PersonID) REFERENCES Personne(PersonID)
```

Cela signifie que l'on définit la clef étrangère sur le champ PersonID de la table Commande qui référence la clef primaire de la table Personne.

On peut aussi définir la clef étrangère sur le champ directement comme pour la clef primaire:

```
CREATE TABLE Commande (
    OrderID int UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    OrderNumber int UNSIGNED NOT NULL,
    PersonID int UNSIGNED NOT NULL FOREIGN KEY REFERENCES Personne(PersonID)
);
```

A la différence de la clef primaire, une clef étrangère peut avoir la valeur **NULL**. Donc, si vous mettez **NOT NULL** sur le champ de la clef étrangère ça veut dire que l'on veut absolument qu'il y ait un lien vers une autre table.

Si on met **NULL**, ça signifie qu'il peut y avoir des enregistrements qui n'ont pas de lien vers une autre table.

9.8 UNIQUE

Comme son nom l'indique, le champ doit être unique. Par exemple un numéro national.

Si votre clef primaire est composée d'un seul champ, il n'est pas nécessaire d'indiquer UNIQUE. Car par définition, une clef primaire est unique.

Pour les clefs primaires composites, ce n'est pas le cas car c'est la compostion des champs qui forment la clef primaire qui est unique.

9.9 AUTO_INCREMENT

L'auto-incrémentation d'un champ est très très très utilisée dans les bases de données. Surtout pour générer un identifiant unique pour une clef primaire.

Un champ qualifié d'AUTO_INCREMENT se verra incrémenté de 1 pour le prochain enregistrement.

```
IdClasse INT UNSIGNED NOT NULL AUTO_INCREMENT
```

9.10 Exemples

Création de la table Classe

```
CREATE TABLE Classe (
    IdClasse INT UNSIGNED NOT NULL AUTO_INCREMENT,
    Nom VARCHAR(20) NOT NULL,
    Lieu VARCHAR(20) NOT NULL,
    Nickname VARCHAR(20) NULL,
    PRIMARY KEY(IdClasse)
);
```

La table Eleve DOIT être créée après la table Classe car nous avons une clef étrangère dans la table Eleve qui référence la clef primaire de la table Classe.

```
CREATE TABLE Eleve (
    IdEleve INT UNSIGNED NOT NULL AUTO_INCREMENT,
    Prenom VARCHAR(20) NOT NULL,
    Nom VARCHAR(20) NOT NULL,
    Naissance DATE NOT NULL,
    RN VARCHAR(20) UNIQUE NOT NULL,
    Actif boolean NOT NULL DEFAULT 1,
    Nationalite VARCHAR(20) NOT NULL,
    Rue VARCHAR(50) NOT NULL,
    Numero VARCHAR(5) NULL,
    Boite VARCHAR(3) NULL,
    CP SMALLINT UNSIGNED NOT NULL,
    Localite VARCHAR(30) NOT NULL,
    Sexe char(1) NOT NULL CHECK(Sexe IN ('M','F')),
    Email VARCHAR(40),
    Tel VARCHAR(20),
    GSM VARCHAR(20),
    IdClasse INT UNSIGNED NOT NULL,
    PRIMARY KEY (IdEleve),
    FOREIGN KEY (IdClasse) REFERENCES Classe(IdClasse)
);
```

10. INSERT INTO

10.1 La commande

La commande **INSERT INTO** est utilisée pour ajouter des enregistrements dans une table.
Il faut donc bien évidemment que notre table ait été créée avec un **CREATE TABLE**.

10.2 La Syntaxe

Elle est assez simple.

```
INSERT INTO nomTable (champ1,champ2, champ3)
VALUES (valeur1, valeur2, valeur3);
```

- nomTable: nom de la table où l'on veut insérer
- champ1, champ2, champ3: c'est l'énumération des champs où l'on veut ajouter une valeur.
- valeur1, valeur2, valeur3: ce sont les valeurs que nous affecterons. valeur1 mettre sa valeur dans le champ1, valeur2 mettra sa valeur dans le champ2, etc.

Partons d'un exemple:

```
INSERT INTO Classe(Nom, Lieu, Nickname)
VALUES ('BlindCode','BXL','BlindBXL');

INSERT INTO Classe(Nom, Lieu, Nickname)
VALUES ('BlindCode4Data','LLN','BlindLLN');
```

11. UPDATE

11.1 La commande

La commande **UPDATE** permet de mettre à jour un ou plusieurs enregistrements. Ces enregistrements peuvent correspondre à un motif de recherche.

11.2 La syntaxe

Par exemple, on veut changer le prénom 'jooooohnny' en 'johnny' et le nom 'Pietttttuuuus' en 'Piette' pour l'utilisateur ayant l'iduser = 47

```
UPDATE User
SET Prenom='Johnny', nom='Piette'
WHERE IdUser=47;
```

Il faut faire SUPER attention quand on met à jour des données. Si vous oubliez par exemple un WHERE. Boummmm ça met à jours tous les enregistrements de la table. Donc par une mauvaise manipulation on pourrait avoir tous nos utilisateurs s'appeller 'Johnny Piette'...

==> D'où l'intérêt des backups et/ou d'une base de données de tests.

Si on veut désactiver l'envoie des emails pour tout le monde:

```
UPDATE User
SET AcceptEmail=false;
```

12. DELETE

12.1 La commande

La commande **DELETE** dans le langage SQL permet de supprimer des enregistrements dans une table. Cela signfie qu'il faut la manipuler avec prudence !

12.2 La syntaxe

```
DELETE FROM NomTable
WHERE Condition;
```

Exemple:

```
DELETE FROM Produit
WHERE IdProduit = 123;
```

12.2 Suppression ou champ Deleted ?

Parfois, il vaut mieux créer un champ ayant pour nom Deleted et mettre sa valeur à 1 pour l'enregistrement que l'on veut supprimer. En effet, parfois il faut toujours garder une trace de cet enregistrement. Il ne sera pas effacé de notre base de données mais nous ne l'utiliserons plus.

```
UPDATE Produit
SET Deleted = 1
WHERE IdProduit = 123;
```

Affichons tous les produits à vendre:

```
SELECT *
FROM Produit
WHERE Deleted = 0;
```

Affichons les produits qui ne sont plus à vendre:

```
SELECT *
FROM Produit
WHERE Deleted = 1;
```

12.3 Pourquoi mon DELETE provoque une erreur ?

Il peut arriver que l'id de l'enregistrement que vous voulez supprimer soit utilisé ailleurs. Par exemple vous voulez supprimer l'article 'Oculus Quest 2 - 256 GB' ayant pour IdProduit '123':

```
DELETE FROM Produit
WHERE IdProduit = 123;
```

Si vous avez déjà eu des commandes pour ce Produit, MySQL devrait provoquer une erreur car certains enregistrements de nos commandes concernent ce produit. Et donc MySQL ne sait pas

le supprimer. Heureusement aussi que MySQL ne l'ait pas fait car alors il aurait dû supprimer toutes nos commandes comportants ce produit. Ce qui pourrait être catastrophique... On pourrait y arriver en utilisant le **ON DELETE CASCADE** mais je n'en parlerai pas car c'est trop risqué. ☺ Et je ne veux pas vous embrouiller.

12.4 Droit à l'oubli ?

Depuis le GDPR/RGPD, Il est possible qu'un utilisateur faisant partie d'une de vos bases de données viennent vous dire qu'il ne veut plus en faire partie. Il faudra en tenir compte.

Cependant, il faut bien se dire que ça ne sera pas toujours possible dans certains cas comptables: commandes, achats, livraisons, etc. Ou dans certaines institutions publiques qui doivent garder des informations importantes sur les personnes.

Mais ça sera par exemple possible sur un forum: on supprimera l'utilisateur ainsi que ses commentaires, ses posts.
Dans l'ordre on devra procéder de la sorte:

- Les commentaires
- Les posts
- L'utilisateur

En effet, Un commentaire est lié à un post et a utilisateur. Et un post est lié à un utilisateur.
Car ces tables contiennent des références de l'id de l'utilisateur.

```
DELETE FROM commentaire
WHERE IdUtilisateur=45;

DELETE FROM post
WHERE IdUtilisateur=45;

DELETE FROM Utilisateur
WHERE IdUtilisateur=45;
```

Idéalement il faudrait effectuer ses trois opérations consécutives dans une transaction...

Ceci ne rentre pas dans le cadre de ce cours d'introduction aux SGBDR.

13. Limiter le résultat - LIMIT

Sur un grand nombre de résultats, il est parfois utile de n'en prendre qu'un certain nombre.
Par exemple:

```
Use Pays;
SELECT *
FROM Pays
ORDER BY Name
LIMIT 50;
```

Nous aurons les 50 premiers pays classés par ordre alphabétique.
Cette requète pourrait s'écrire de la manière suivante en faisant: LIMIT 0,50:

- Où 0 est la borne inférieure non comprise.
- 50 est le nombre d'enregistrements à prendre.

Utiliser LIMIT avec un interval peut-être très utile quand on veut paginer les résultats d'une recherche. Il arrive souvent que l'on doive cliquer sur une flèche pour avoir les résultats suivant. Par exemple 10 résultats par page.

Soit on affiche les 10 premiers résultats puis on a une bouton qui permet d'afficher les 10 suivants.
Pour les 10 premiers:

```
Use Pays;
SELECT *
FROM Pays
ORDER BY Name
LIMIT 0, 10;
```

Et les 10 suivants:

```
Use Pays;
SELECT *
FROM Pays
ORDER BY Name
LIMIT 10, 10;
```

Etc...

14. DROP TABLE

DROP TABLE supprime complètement une table.

```
DROP TABLE Joueur;
```

Le SGBD supprimera la table Joueur.
Cependant, si nous essayons de supprimer la table Equipe

```
DROP TABLE Equipe;
```

Le SGBD nous donnera l'erreur suivante:

> ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint fails

Le SGBD nous indique qu'il ne peut pas supprimer la table Equipe car il y a une contrainte de clef étrangère de la table Joueur qui pointe sur la table Equipe.

15. TRUNCATE

TRUNCATE supprime toutes les données d'une table. A la différence de **DROP TABLE** qui supprime toute la table.

```
TRUNCATE Joueur;
```

Comme pour **DROP TABLE**, s'il y a une clef étrangère qui référence la table sur laquelle on veut appliquer un **TRUNCATE**, le SGBD ne nous permettra pas de le faire.

16. ALTER TABLE

16.1 Ajouter une colonne - ADD

Soit la table suivante:

```
CREATE TABLE Eleve (
    IdEleve int NOT NULL AUTO_INCREMENT,
    Prenom varchar(20) NOT NULL,
    Nom varchar(20) NOT NULL,
    Sexe char(1) NOT NULL CHECK(Sexe IN ('M','F'))
);
```

Si l'on veut ajouter la colonne Tel à la table Eleve, on fera de la sorte:

```
ALTER TABLE Eleve
ADD Tel VARCHAR(30) NULL,
ADD Tel2 VARCHAR(30) NULL;
```

16.2 Changer le type d'une colonne - MODIFY COLUMN

Si l'on veut mettre 30 caractères pour le champ Prenom au lieu des 20 définis.

```
ALTER TABLE Eleve
MODIFY COLUMN Prenom VARCHAR(30);
```

Dans d'autres bases de données que MySQL ça peut être ALTER COLUMN.

16.3 Supprimer une colonne - DROP COLUMN

On supprime une colonne d'une table.

```
ALTER TABLE Eleve
DROP COLUMN Tel2;
```

Attention que si vous avez des enregistrements, vous risquez bien entendu la perte de données.

16.4 Renommer une colonne - CHANGE COLUMN

Si l'on veut renommer le champ Pranom en Prenom:

```
ALTER TABLE Eleve
CHANGE COLUMN Pranom Prenom VARCHAR(30);
```

Ici on a fait le cas simple où le champ n'est pas une clef primaire utilisée dans une clef étrangère. Si c'est le cas, il faudra faire une requête plus complexe.

16.5 Renommer une clef primaire utilisée dans une clef étrangère

Pour renommer la colonne `PersonID` en `id` dans la table `Personne` et mettre à jour la table Commande pour qu'elle fasse référence au nouveau nom de colonne, tu devras suivre plusieurs étapes pour assurer l'intégrité référentielle de ta base de données.
Je ne vais pas insister sur cette partie mais je vais vous donner un exemple de comment faire. Comme ça, si vous avez un jour à le faire, vous saurez comment procéder.

Voici comment procéder étape par étape :

Étape 1: Supprimer la Contrainte de Clé Étrangère

D'abord, supprime la contrainte de clé étrangère de la table Commande qui fait référence à PersonID dans la table Personne.

```
ALTER TABLE Commande DROP FOREIGN KEY Commande_ibfk_1;
```

Note : Remplace Commande_ibfk_1 par le nom réel de ta contrainte de clé étrangère. Tu peux trouver ce nom en utilisant la commande `SHOW CREATE TABLE Commande;`.

```
SHOW CREATE TABLE Commande;
```

Résultat:

```
+----------+-------------------------------------------------------------------------------
| Table    | Create Table
+----------+-------------------------------------------------------------------------------
| Commande | CREATE TABLE `Commande` (
  `OrderID` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `OrderNumber` int(10) unsigned NOT NULL,
  `PersonID` int(10) unsigned NOT NULL,
  PRIMARY KEY (`OrderID`),
  KEY `PersonID` (`PersonID`),
  CONSTRAINT `Commande_ibfk_1` FOREIGN KEY (`PersonID`) REFERENCES `Personne` (`PersonID`
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci |
+----------+-------------------------------------------------------------------------------
1 row in set (0,000 sec)
```

Étape 2: Renommer la Colonne dans la Table Personne

Ensuite, renomme la colonne PersonID en id dans la table Personne.

```
ALTER TABLE Personne CHANGE PersonID id int UNSIGNED AUTO_INCREMENT;
```

Étape 3: Mettre à Jour la Table Commande

Maintenant, tu dois mettre à jour la table Commande pour qu'elle fasse référence au nouveau nom de la colonne (id) dans la table Personne.

1. Modifier la Colonne pour Correspondre au Nouveau Nom :

Tu dois d'abord modifier la colonne PersonID dans la table Commande pour qu'elle corresponde au nouveau nom de colonne référencée.

```
ALTER TABLE Commande CHANGE PersonID person_id int UNSIGNED NOT NULL;
```

Cette étape peut sembler redondante, mais elle est nécessaire si tu souhaites ajuster des attributs de colonne ou si le système de gestion de base de données exige une modification explicite avant de recréer les contraintes.

2. Recréer la Contrainte de Clé Étrangère :

Ensuite, ajoute la contrainte de clé étrangère pour référencer le nouveau nom de colonne.

```
ALTER TABLE Commande ADD CONSTRAINT fk_personne_id FOREIGN KEY (PersonID) REFERENCES Perso
```

Étape 4: Vérification

Après avoir effectué ces changements, il est bon de vérifier que tout fonctionne comme prévu. Tu peux utiliser SHOW CREATE TABLE pour confirmer que les modifications ont été appliquées correctement.

```
SHOW CREATE TABLE Personne;
SHOW CREATE TABLE Commande;
```

Ces commandes te permettront de voir les définitions actuelles des tables, y compris les noms de colonnes et les contraintes de clé étrangère, pour confirmer que tout est configuré correctement.

17. CHECK - Validation

Lors de la définition d'un champ, il peut être utile de directement vérifier la validité d'un champ. Par exemple le sexe d'une personne doit être soit F ou M et le code postal doit être compris entre 1000 et 9992.

```
CREATE TABLE Eleve (
    IdEleve int NOT NULL AUTO_INCREMENT,
    Prenom varchar(20) NOT NULL,
    Nom varchar(20) NOT NULL,
    Sexe char(1) NOT NULL CHECK(Sexe IN ('M','F')),
    Rue varchar(50) NOT NULL,
    Numero varchar(5) NULL,
    Boite varchar(3) NULL,
    CP int NOT NULL CHECK(CP BETWEEN 1000 AND 9992),
    Localite varchar(30) NOT NULL
);
```

Evidemment cette validation devrait être faite en plus depuis le programme qui utilise la base de

données mais si la validation n'est pas implémentée, MySQL veillera aux grains.

Comme je vous l'ai déjà dit, certains développeurs vous diront que les règles de gestion n'ont pas sa place dans la définition d'une table...

Si la règle de gestion change, il faudra changer la table. C'est pour cela que certains développeurs ne veulent pas de règles de gestion dans la table: ce qui est logique.

Reprenons le champ Sexe. Si la règle de gestion change et que l'on doit ajouter un troisième choix, il faudra changer la table. Si la règle de gestion est dans le programme, il suffira de changer le programme.

18. Les vues - VIEW

Une vue est une table virtuelle. Elle ne contient pas de données. Elle est créée à partir d'une ou plusieurs tables. Elle est le résultat d'une requête.

Elle permet de simplifier les requêtes: au lieu d'avoir une requête complexe, on peut créer une vue qui contient cette requête complexe.

Elle permet aussi de ne pas donner accès à certaines données. Par exemple, si vous avez une table avec des données sensibles, vous pouvez créer une vue qui ne contient pas ces données sensibles. Vous pourrez de cette manière donner accès à cette vue sans que les utilisateurs/ développeurs aient accès aux données sensibles.

Syntaxe:

```
CREATE VIEW nomVue AS
SELECT ... (et le reste de votre requête terminée par un point-virgule)
```

Soit la création une vue qui fournit une liste des employés actuels, avec leurs noms, leur département, mais sans inclure les informations sensibles comme la date de naissance ou le genre.

```sql
CREATE VIEW VueEmployesDepartements AS
SELECT
    e.emp_no,
    e.first_name,
    e.last_name,
    d.dept_name
FROM
    employees e
JOIN
    dept_emp de ON e.emp_no = de.emp_no
JOIN
    departments d ON de.dept_no = d.dept_no
WHERE
    de.to_date = '9999-01-01';  -- Cela assure que nous sélectionnons seulement les emplo
```

Notez que **de.to_date = '9999-01-01'** nous assure que nous sélectionnons seulement les employés actuellement dans un département.

Nous pouvons maintenant utiliser cette vue comme une table normale. Par exemple, nous pouvons sélectionner tous les employés du département de la vente:

```sql
SELECT *
FROM VueEmployesDepartements
WHERE dept_name = 'Sales'
LIMIT 10;
```

Résultat limité à 10 enregistrements:

emp_no	first_name	last_name	dept_name
10002	Bezalel	Simmel	Sales
10016	Kazuhito	Cappelletti	Sales
10041	Uri	Lenart	Sales
10050	Yinghua	Dredge	Sales
10053	Sanjiv	Zschoche	Sales

emp_no	first_name	last_name	dept_name
10061	Tse	Herber	Sales
10068	Charlene	Brattka	Sales
10089	Sudharsan	Flasterstein	Sales
10093	Sailaja	Desikan	Sales
10095	Hilari	Morton	Sales

L'intérêt ce qu'il ne faut plus jouer avec des jointures complexes pour obtenir ce résultat. On a juste à faire un SELECT sur la vue.

19. Les indexes

Comme vous le savez MySQL optimise les requêtes avec des PRIMARY KEY, FOREIGN KEY et des champs UNIQUE. Tout simplement car MySQL crée ce que l'on appelle un index. Un index permet de vite retrouver une donnée en fonction d'un critère de recherche.

Maintenant, il peut arriver que de nombreuses requêtes récurentes sur un même champ posent un problème de rapidité/performance. De là, vient alors la question de mettre ou non un index sur ce champ.

Ces problèmes de performance n'ont lieu bien entendu que pour des bases de données +/- importantes.

Par exemple dans une base de données récupérée sur GitHub, j'ai une table employees qui contient 300 mille employés. Si l'on fait une recherche sur last_name='Simmel' celle-ci va nous prendre 0,117ms

Si je fais une analyse de la requête avec:

```
EXPLAIN SELECT * FROM employees WHERE last_name='Simmel';
+------+-------------+-----------+------+---------------+------+---------+------+--------
| id   | select_type | table     | type | possible_keys | key  | key_len | ref  | rows
+------+-------------+-----------+------+---------------+------+---------+------+--------
|    1 | SIMPLE      | employees | ALL  | NULL          | NULL | NULL    | NULL | 299423
+------+-------------+-----------+------+---------------+------+---------+------+--------
```

On voit qu'il va TOUT (ALL) analyser de la table employees, qu'il n'y a pas de clef, qu'il y a 299423 lignes et qu'il utilisera un WHERE;

19.1 Création d'un INDEX - CREATE INDEX

Je vais créer maintenant un index sur la colonne last_name

```
CREATE INDEX employees_last_name
ON employees(last_name);
```

La création de l'index réclame un nom d'index, ici: ***employees_last_name*** on doit indiquer sur (ON) quelle table (employees) on crée l'index et on fournit entre parenthèses le nom du champ: last_name.

Je relance la requête sur last_name='Simmel' et la requête ne met que 0,001 ms. Quelle différence !
Voyons le plan d'éxécution:

```
EXPLAIN SELECT * FROM employees WHERE last_name='Simmel';
+------+-------------+-----------+------+---------------------+---------------------+-----
| id   | select_type | table     | type | possible_keys       | key                 | key_
+------+-------------+-----------+------+---------------------+---------------------+-----
|    1 | SIMPLE      | employees | ref  | employees_last_name | employees_last_name | 66
+------+-------------+-----------+------+---------------------+---------------------+-----
```

On voit qu'il va analyser par référence (REF), qu'il y a une clef/index (employees_last_name) Que la référence est de type constante et qu'il va utiliser un index avec condition (WHERE). Il ne va analyser que 167 enregistrement ce qui correspond au nombre total où last_name='Simmel'. Ce qui est nettement mieux que sur 299423 enregistrements quand nous n'avions pas d'index

sur le champ last_name.

19.2 Suppression d'un INDEX - DROP INDEX

La suppresion d'un index est assez simple. On supprimer l'index par son nom et sur quelle table il porte.

```
DROP INDEX employees_last_name ON employees;
```

19. Naming convention - Covention de nommage

Au début de ce cours, je n'ai donné aucune discipline de nommage des tables, champs, des clefs, des contraintes, etc.

Avant d'appliquer telle ou telle convention de nommage. Le plus important est de rester cohérent partout. Si vous avez votre habitude et que vous restez fidèle à vous-même gardez cette manière de nommer qui vous correspond.

Cependant, il est intéressant de se forcer d'utiliser une autre méthode de nommage car si vous travaillez un jour dans le monde du dev, il y aura des conventions de nommage.

19.1 Règles communes

Une règle qui va s'appliquer partout.

- **Aucun accent !**
- Pas de chiffres.
- Ne pas utiliser de mot réservé. Par exemple ne pas nommer un champ date car il existe le type DATE.
- Eviter les majuscules. Utilisez les miniscules.
- Si vous devez écrire plusieurs mots par exemple DateNaissance. Utilisez les underscores: date_naissance

19.1 Nom d'une base de données

On peut utiliser le pluriel dans le nom de la base de données.

Exemples:

- inscriptions
- voiries
- jeux
- cryptos

Tout dépend du sujet:

- bibliotheque
- comptabilite

19.2 Nom d'une table

- Le nom d'une table doit être écrit au singulier mais ça peut être sujet à de lourds débats. Par exemple, WordPress et Joomla ont pris le parti de mettre au pluriel. Cependant, lorsque vous utiliserez un ORM il sera plus simple de manipuler vos données.
  Un ORM (Object Relational Mapping) peut faire correspondre une classe à une table. Dès lors, lorsque nous voudrons créer une nouvelle voiture: Car car = new Car("Peugeot 207",2017); Il sera aisé d'ajouté une voiture dans la base de données.
- Dans certains cas, préfixer le nom des tables: Si vous avez d'autres tables dans votre base de données crées par un autre soft. joomla et WordPress préfixent leurs tables. Exemples: jos_users, wp_users. De cette manière si une application vient se greffer à leur base de données, l'application pourra aussi préfixer ses tables pour éviter une ambiquité par exemple avec la table users: app_users;

19.3 Nom d'une clef primaire

id est très souvent utilisé pour une clef primaire. Si vous ajoutez par exemple id_eleve. On se doute que c'est l'id de la table eleve. Ca n'apporte rien à la clef. Autant faire au plus simple.

19.4 Nom d'une clef étrangère

Pour le nom de la clef étrangère, elle s'écrit **nomtable_clefprimaire**, par exemple **equipe_id** où **equipe** est le nom de la table et **id** la clef primaire de la table equipe.

19.5 Nom des champs

- Les noms doivent être faciles à comprendre. Et ne pas être trop vagues.
  Mettre par exemple un champ date sur une table livre est trop imprécis. On utilisera par exemple date_acquision, date_edition, etc.
- Pour les champs booléens (TRUE ou FALSE) on indique l'état: au lieu d'actif on mettra is_actif, au lieu de deleted on mettra is_deleted.

19.6 Nom de la contrainte de Clef étrangère

Lorsque nous avons vu les clefs étrangères, je n'ai pas été expliqué qu'une clef étrangère est une contrainte (CONSTRAINT). Et que par défaut quand on met une clef étrangère, MySQL sait que c'est une contrainte. Cette contrainte peut avoir un nom et MySQL en définira une pour nous par défaut.

Par exemple

```
CREATE TABLE livre(
  id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
  titre VARCHAR(30) NOT NULL,
  auteur_id INT NOT NULL,
  CONSTRAINT fk_livre_auteur_id FOREIGN KEY (auteur_id) REFERENCES auteur(id)
);
```

Donner un nom à une contrainte permet:

- d'avoir des messages d'erreur beaucoup plus compréhensibles si ceux-ci utilisent le nom de la contraite de clef étrangère.
- de modifier une contrainte via un ALTER TABLE. Par exemple: ALTER TABLE eleve DROP CONSTRAINT fk_eleve_classe_id

20. Les fonctions

Comme pour les langages de programmation, nous pouvons utiliser et créer des fonctions.

Vous avez déjà utilisé des fonctions: MIN, MAX, AVG, etc.

MariaDB fournit nativement pas mal de fonctions intéressantes:

- sur les chaînes de caractères: https://mariadb.com/kb/en/string-functions/
- sur les dates: https://mariadb.com/kb/en/date-time-functions/
- sur les fonctions d'agrégats: https://mariadb.com/kb/en/aggregate-functions/
- sur les fonctions mathématiques: https://mariadb.com/kb/en/numeric-functions/
- etc...

Pour avoir une liste complète: https://mariadb.com/kb/en/built-in-functions/

Mais vous pouvez aussi créer vos propres fonctions ! C'est ce que nous allons voir maintenant.

Cependant, il faut faire attention à ne pas mettre de la logique métier dans une base de données. C'est le rôle de l'application. Mais parfois, il peut être utile de créer une fonction pour simplifier une requête complexe ou pour une question de performance.

20.1 FONCTION sans paramètre

On va prendre le cas où d'une fonction sans paramètre.
Soit la fonction `whois_the_best()` 😀

```
USE BlindCode;
CREATE FUNCTION whois_the_best() RETURNS VARCHAR(50)
RETURN 'Johnny Piette';
```

On peut tester le résultat avec un simple SELECT:

```
SELECT whois_the_best();

+------------------+
| whois_the_best() |
+------------------+
| Johnny Piette    |
+------------------+
1 row in set (0,001 sec)
```

20.2 FONCTION avec paramètres

Une fonction avec paramètres est plus intéressante. On va prendre l'exemple d'une fonction qui calcule le prix TVAC d'un produit. On va donc avoir un paramètre qui est le prix HTVA.

Cependant, ici c'est assez particulier. On définit le délimiteur par défaut qui est normalement le point-virgule. Mais comme nous allons utiliser des points-virgules dans notre fonction, nous devons changer le délimiteur par défaut. Nous utilisons donc la commande DELIMITER $$ pour changer le délimiteur par $$.

```
USE Ventes;
DELIMITER $$
DROP FUNCTION IF EXISTS price_tvac;
CREATE FUNCTION price_tvac(price FLOAT(10,2)) RETURNS FLOAT(10,2)
BEGIN
DECLARE price_tvac FLOAT(10,2);
SET price_tvac=price+price*0.21;
RETURN price_tvac;
END;
$$
DELIMITER ;
```

Expliquons un peu ce code:

1. **`DELIMITER $$`**
    - Cette commande change le délimiteur par défaut de MySQL de `;` à `$$`.
    - Cela est nécessaire car, dans le corps des fonctions ou des procédures stockées, vous pouvez avoir plusieurs instructions SQL, chacune se terminant par `;`. Pour indiquer à MySQL où se termine l'ensemble de la fonction ou de la procédure, un délimiteur différent est utilisé.
2. **`DROP FUNCTION IF EXISTS price_tvac;`**
    - Cette commande vérifie si une fonction nommée `price_tvac` existe déjà dans la base de données. Si c'est le cas, elle la supprime.
    - Cela garantit qu'il n'y aura pas d'erreur de "fonction déjà existante" lorsque vous créerez la nouvelle fonction.
3. **`CREATE FUNCTION price_tvac(price FLOAT(10,2)) RETURNS FLOAT(10,2)`**

- `CREATE FUNCTION` est la commande pour créer une nouvelle fonction.
- `price_tvac` est le nom de la fonction.
- `(price FLOAT(10,2))` déclare un paramètre nommé `price` avec un type de données `FLOAT(10,2)`. Cela signifie que le paramètre `price` peut être un nombre à virgule flottante avec jusqu'à 10 chiffres au total, dont 2 après la virgule.
- `RETURNS FLOAT(10,2)` indique que le type de retour de la fonction est également un `FLOAT(10,2)`.

4. **Corps de la fonction (entre `BEGIN` et `END;`)**
    - `BEGIN` et `END;` délimitent le corps de la fonction.
    - `DECLARE price_tvac FLOAT(10,2);` déclare une variable locale nommée `price_tvac` (qui est différente de la fonction `price_tvac`) avec un type de données `FLOAT(10,2)`.
    - `SET price_tvac = price + (price * 0.21);` est une instruction qui calcule le prix avec TVA en ajoutant 21% au prix original et stocke le résultat dans la variable `price_tvac`.
    - `RETURN price_tvac;` renvoie le résultat stocké dans la variable `price_tvac`.
5. **`$$`**
    - Ce `$$` est utilisé pour indiquer la fin de la fonction, conformément au délimiteur personnalisé défini au début.
6. **`DELIMITER ;`**
    - Cette commande rétablit le délimiteur par défaut à `;` pour les opérations normales après la création de la fonction.

En résumé, cette série de commandes crée une fonction SQL `price_tvac` qui prend un prix (sans TVA), ajoute 21% de TVA à ce prix, et renvoie le prix total avec TVA. La modification du délimiteur garantit que la fonction est correctement interprétée et créée dans MySQL.

Testons notre fonction:

```
SELECT price_tvac(100);
+----------------+
| price_tvac(100)|
+----------------+
|          121.00|
+----------------+
1 row in set (0,001 sec)
```

20.2 Supprimer une fonction - DROP FUNCTION

Si vous voulez supprimer une fonction, il suffit de faire:

```
DROP FUNCTION price_tvac;
```

20.3 Modifier une fonction - ALTER FUNCTION

Pour modifier une fonction:

- Vous la supprimez:

```
DROP FUNCTION price_tvac;
```

- Vous la recréez comme vu précédemment mais auparavant, je vous conseille d'utiliser commande `SHOW CREATE FUNCTION price_tvac` pour voir la définition de la fonction. Cela vous permettra de ne pas perdre la définition de la fonction.

21. Création d'un utilisateur

Pour créer un utilisateur, il faut utiliser la commande `CREATE USER`.
On crée des utilisateurs pour que ceux-ci puissent se connecter à la base de données.

Ces utilisateurs peuvent être des utilisateurs locaux ou des utilisateurs distants.

Il est conseillé de créer des utilisateurs avec des droits limités. Par exemple, si un utilisateur n'a besoin que de lire des données, il ne faut pas lui donner des droits d'écriture.

Un utilisateur par application est une bonne pratique: de cette manière si un utilisateur est compromis, seul l'application est compromise et pas toute la base de données.

21.1 Création d'un utilisateur de la db pour php

Il est recommandé de créer un utilisateur pour chaque application qui se connecte à la base de données. Cela permet de limiter les droits de chaque application. Nous allons créer un utilisateur pour PHP pour simplifier les choses. Mais le principe est le même pour chaque application.

```
CREATE USER 'php'@'localhost' IDENTIFIED BY 'php';
GRANT ALL PRIVILEGES ON *.* TO 'php'@'localhost' WITH GRANT OPTION;
FLUSH PRIVILEGES;
```

Détaillons ces commandes:

1. Création d'un utilisateur

```
CREATE USER 'php'@'localhost' IDENTIFIED BY 'php';
```

- `CREATE USER` : Cette commande est utilisée pour créer un nouvel utilisateur dans le système de gestion de base de données.
- `'php'@'localhost'` : Ici, `'php'` est le nom de l'utilisateur que vous créez. L'adresse `'localhost'` spécifie que cet utilisateur aura uniquement le droit de se connecter à la base de données depuis la machine locale. En d'autres termes, une connexion distante utilisant cet utilisateur sera refusée. Le format est `'nom_utilisateur'@'hôte'`.
- `IDENTIFIED BY 'php'` : Cette partie de la commande définit le mot de passe de l'utilisateur. Dans ce cas, le mot de passe est également `'php'`. Il est crucial de choisir un mot de passe fort dans un environnement de production pour sécuriser l'accès à la base de données.

2. Attribution de privilèges

```
GRANT ALL PRIVILEGES ON *.* TO 'php'@'localhost' WITH GRANT OPTION;
```

- `GRANT` : Cette commande est utilisée pour attribuer des privilèges à un utilisateur.
- `ALL PRIVILEGES` : Cela signifie que vous donnez à l'utilisateur tous les droits

possibles sur les bases de données et les tables. C'est une configuration large qui devrait être utilisée avec prudence, surtout dans un environnement de production.
- `ON *.*` : Ici, le premier `*` fait référence à toutes les bases de données, et le deuxième `*` à toutes les tables au sein de ces bases de données. Ensemble, `*.*` signifie que les privilèges sont accordés sur toutes les bases de données et toutes leurs tables.

> Donc, dans une utilisation en production, il faudra remplacer `*.*` par le nom de la base de données à laquelle l'utilisateur aura accès. Car il est en effet très risqué de donner tous les privilèges à un utilisateur surtout s'il utilisé par une application. Par exemple, on donne accès à la base de données Pays: **ON `Pays.*`**

- `TO 'php'@'localhost'` : Cette partie spécifie à quel utilisateur les privilèges sont accordés, en reprenant le format `'nom_utilisateur'@'hôte'`.
- `WITH GRANT OPTION` : Cette option permet à l'utilisateur non seulement d'avoir tous les privilèges mais aussi de les accorder à d'autres utilisateurs. C'est un niveau élevé de privilège qui doit être accordé avec une grande prudence.

> Il est très rare que vous ayez besoin d'accorder cette option à un utilisateur. C'est une bonne pratique de limiter les privilèges à ce qui est strictement nécessaire pour l'utilisateur.

3. Actualisation des privilèges

```
FLUSH PRIVILEGES;
```

- `FLUSH PRIVILEGES` : Cette commande force le système de gestion de base de données à recharger les tables de privilèges. Cela signifie que tous les changements de privilèges que vous avez faits seront pris en compte immédiatement. Sans cette commande, il se pourrait que certains changements ne soient pas appliqués immédiatement, en particulier si vous avez modifié directement les tables de privilèges sans passer par les commandes `GRANT` ou `CREATE USER`.

Ces commandes ensemble permettent de créer un utilisateur avec un accès complet à toutes les bases de données depuis la machine locale, et lui donnent également la capacité d'accorder ces privilèges à d'autres utilisateurs. Il est important de comprendre et de contrôler l'attribution des privilèges pour sécuriser votre base de données.

21.2 Création d'un utilisateur de la db pour une utilisation externe

```
CREATE USER 'php'@'%' IDENTIFIED BY 'php';
GRANT ALL PRIVILEGES ON *.* TO 'php'@'%' WITH GRANT OPTION;
FLUSH PRIVILEGES;
```

21.3 Accès à une base de données spécifique

Si vous voulez que l'utilisateur php ait accès à une seule base de données, vous pouvez faire:

```
GRANT ALL PRIVILEGES ON `Pays`.* TO 'php'@'localhost' WITH GRANT OPTION;
FLUSH PRIVILEGES;
```

21.4 Accès à une table spécifique en lecture seule

Si vous voulez que l'utilisateur php ait accès à une seule table ou une seule vue, vous pouvez faire:

```
GRANT SELECT ON `Employees`.`employees_info` TO 'php'@'localhost';
FLUSH PRIVILEGES;
```

L'exemple précédent donne accès à la vue employees_info de la base de données Employees que vous avez créée dans un exercice précédent. Et il ne pourra que lire les données de cette vue (`GRANT SELECT`).

21.5 Accès en lecture et écriture à une table spécifique

Si vous voulez que l'utilisateur php ait accès à une seule table ou une seule vue, vous pouvez faire:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON `Employees`.`employees_info` TO 'php'@'localhost'
FLUSH PRIVILEGES;
```

L'exemple précédent donne accès à la vue employees_info de la base de données Employees

que vous avez créée dans un exercice précédent. Et il pourra lire, insérer, mettre à jour et supprimer les données de cette vue (`GRANT SELECT, INSERT, UPDATE, DELETE`).

21.5 Accès à plusieurs tables spécifiques

Si vous voulez que l'utilisateur php ait accès à employees_info et à salaries, vous pouvez faire:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON `Employees`.`employees_info` TO 'php'@'localhost'
GRANT SELECT, INSERT, UPDATE, DELETE ON `Employees`.`salaries` TO 'php'@'localhost';
FLUSH PRIVILEGES;
```

Il n'y a pas de problème à répéter la commande GRANT plusieurs fois pour le même utilisateur. Vous pouvez aussi donner des droits différents pour chaque table. Mais il ne vous sera pas possible de tout faire en une seule commande GRANT.

22. Les sous-requêtes

Une sous-requête est une requête imbriquée dans une autre requête. Elle est utilisée pour récupérer des données à partir d'une ou plusieurs tables.

22.1 1. Sous-requêtes corrélées

- **Définition**: Une sous-requête corrélée est exécutée une fois pour chaque ligne traitée par la requête principale. Elle peut référencer des colonnes de la requête externe.
- **Exemple**: Supposons que nous voulons trouver les noms des employés qui gagnent plus que le salaire moyen de tous les employés. Voici comment nous pourrions structurer cette requête avec une sous-requête :

```
SELECT e.first_name, e.last_name, s.salary
FROM employees e
JOIN salaries s ON e.employee_id = s.employee_id
WHERE s.salary > (
    SELECT AVG(salary) FROM salaries
);
```

Dans cet exemple, la sous-requête `(SELECT AVG(salary) FROM salaries)` est utilisée pour obtenir le salaire moyen de tous les employés. Ensuite, nous comparons chaque salaire individuel avec le salaire moyen pour filtrer les résultats.

22.2 Sous-requête de liste

- **Définition**: Une sous-requête de liste est une forme de sous-requête SQL qui retourne un ensemble de zéro ou plusieurs valeurs. Elle est généralement utilisée avec des opérateurs conditionnels tels que IN, NOT IN, ANY, SOME, ou ALL dans la clause WHERE (ou parfois HAVING) d'une requête principale pour tester si les valeurs d'une colonne correspondent à n'importe quelle valeur dans l'ensemble retourné par la sous-requête.
- **Exemple**: Voici un autre exemple intéressant qui illustre l'utilisation d'une sous-requête dans une clause SELECT, mais cette fois pour identifier les employés qui travaillent dans un département ayant plus de 60000 employés. Cette requête peut être utile pour identifier les membres d'équipes importantes ou pour analyser la distribution des employés par département.

```
SELECT e.first_name, e.last_name, d.dept_name
FROM employees e
JOIN dept_emp de ON e.emp_no = de.emp_no
JOIN departments d ON de.dept_no = d.dept_no
WHERE d.dept_no IN (
    SELECT dept_no
    FROM dept_emp
    GROUP BY dept_no
    HAVING COUNT(emp_no) > 60000
);
```

Dans cette requête hypothétique :

- La table employees est supposée contenir les informations sur les employés, avec emp_no comme identifiant de l'employé.
- La table dept_emp lie les employés aux départements via emp_no pour l'identifiant de l'employé et dept_no pour l'identifiant du département.
- La table departments contient les noms des départements (dept_name) et leurs identifiants (dept_no).

Cette requête sélectionne les noms des employés et le nom de leur département, pour ceux qui sont dans des départements ayant plus de 10 employés. La sous-requête dans la clause WHERE détermine quels départements ont plus de 60000 employés en comptant le nombre d'emp_no dans dept_emp pour chaque dept_no, en utilisant GROUP BY et HAVING.

22.3 Sous-requêtes scalaires/de colonnes

- **Définition**: Ces sous-requêtes retournent une seule valeur (c'est-à-dire un seul élément scalaire) pour chaque ligne traitée dans la requête principale. Elles sont souvent utilisées pour effectuer des calculs ou obtenir des informations spécifiques qui sont ensuite incluses comme colonnes supplémentaires dans les résultats de la requête principale.
- **Exemple**: Supposons que nous voulions obtenir le nom et le nom complet de chaque pays, le continent ainsi que le nombre total de pays dans le même continent. Voici comment nous pourrions structurer cette requête avec une sous-requête scalaire :

```
SELECT p.name, p.full_name, p.continent,
       (SELECT COUNT(*)
        FROM Pays p2
        WHERE p2.continent = p.continent) AS total_countries_in_continent
FROM Pays p;
```

Dans cet exemple, nous avons une requête qui sélectionne le nom et le nom complet de chaque pays, ainsi que le nombre total de pays dans le même continent. La sous-requête est utilisée pour compter le nombre de pays dans le même continent que le pays actuel.

22. CTE (Common Table Expression)

Une CTE est une sous-requête nommée qui peut être utilisée dans une requête SELECT, INSERT, UPDATE ou DELETE. Elle est utile pour simplifier les requêtes complexes et pour améliorer la lisibilité du code.

Attention que le point-virgule ne se met pas à la fin de la CTE mais à la fin de la requête qui utilise la CTE.

22.1 Utilisation d'une CTE

```sql
WITH cte AS (
    SELECT
        e.emp_no,
        e.first_name,
        e.last_name,
        d.dept_name
    FROM
        employees e
    JOIN
        dept_emp de ON e.emp_no = de.emp_no
    JOIN
        departments d ON de.dept_no = d.dept_no
    WHERE
        de.to_date = '9999-01-01'
)
SELECT *
FROM cte
WHERE dept_name = 'Sales';
```

Ce code SQL est utilisé pour récupérer les informations (numéro, prénom, nom de famille) des employés actuellement actifs (c'est-à-dire, sans date de fin ou avec une date de fin fixée à '9999-01-01') dans le département des ventes (Sales). La CTE simplifie la gestion de cette requête complexe en segmentant la logique de sélection et de jointure, avant d'appliquer le filtre final sur le département.

22.2 Exemple récursif

Supposons que nous voulions afficher chaque année depuis l'embauche de l'employé ayant l'ID 10001 jusqu'à l'année actuelle.

```sql
WITH RECURSIVE years_of_service (year, emp_no) AS (
  SELECT YEAR(hire_date) AS year, emp_no
  FROM employees
  WHERE emp_no = 10001
  UNION ALL
  SELECT year + 1, emp_no
  FROM years_of_service
  WHERE year < YEAR(CURDATE())
)
SELECT year
FROM years_of_service;
```

22.2 Différences entre une CTE et une vue

22.2.1 CTE (Common Table Expression)

- Temporaire : Une CTE est une construction temporaire qui existe uniquement durant l'exécution de la requête dans laquelle elle est définie. Elle n'est pas stockée dans la base de données comme un objet permanent.
- Portée : Elle est accessible seulement dans la requête qui la définit, ce qui en fait un bon choix pour structurer des requêtes complexes et pour améliorer la lisibilité sans affecter la base de données avec des objets supplémentaires.
- Usage : Très utile pour des requêtes récursives, pour décomposer des requêtes complexes en parties plus simples, ou pour effectuer des opérations qui seraient autrement plus complexes ou moins performantes.

22.2.2 Vue

Une vue est un objet de base de données qui permet de sauvegarder une requête SQL spécifique pour la réutiliser. Contrairement à une CTE, une vue est stockée dans la base de données comme un objet permanent et peut être utilisée comme une table dans les requêtes SQL. Voici quelques caractéristiques principales des vues :

- Permanence : Les vues sont des objets stockés dans la base de données. Une fois créées, elles restent disponibles jusqu'à ce qu'elles soient explicitement supprimées, permettant leur réutilisation dans plusieurs requêtes ou applications.
- Réutilisabilité : Étant donné qu'elles sont stockées comme des objets de la base de

données, les vues peuvent être réutilisées dans différentes requêtes par différents utilisateurs, sans nécessiter de redéfinir la logique de la requête originale.

- Sécurité des Données : Les vues peuvent être utilisées pour fournir un niveau d'abstraction et de sécurité des données, en exposant seulement certaines colonnes ou en appliquant des filtres pour restreindre les données accessibles aux utilisateurs.
- Simplification des Requêtes : Les vues permettent de masquer la complexité des requêtes sous-jacentes, offrant une interface simplifiée pour accéder aux données. Cela est particulièrement utile dans les environnements où les utilisateurs finaux ne sont pas familiarisés avec le SQL ou la structure de la base de données.
- Performance : Bien que les vues puissent simplifier l'accès aux données, elles ne stockent pas les données elles-mêmes. Les requêtes qui utilisent des vues exécutent la requête SQL sous-jacente chaque fois qu'elles sont appelées, ce qui peut affecter la performance pour des vues complexes sur de grands volumes de données. Certains SGBD offrent des vues matérialisées, qui stockent le résultat de la requête de la vue sur le disque pour améliorer la performance.

21. Les transactions

Une transaction est un ensemble d'opérations qui doivent être exécutées ensemble. Si une seule opération échoue, toutes les opérations de la transaction doivent être annulées.

Par exemple, si vous avez une transaction qui transfère de l'argent d'un compte à un autre, vous ne voulez pas que l'argent soit retiré d'un compte sans être ajouté à l'autre.

Pour commencer une transaction, vous utilisez la commande `START TRANSACTION`. Vous exécutez ensuite toutes les opérations de la transaction. Si tout se passe bien, vous utilisez la commande `COMMIT` pour valider la transaction. Si quelque chose ne va pas, vous utilisez la commande `ROLLBACK` pour annuler la transaction.

```
START TRANSACTION;
UPDATE compte SET solde=solde-100 WHERE id_compte=1;
UPDATE compte SET solde=solde+100 WHERE id_compte=2;
COMMIT;
```

Si une erreur se produit entre START TRANSACTION et COMMIT, le SGBD garantira l'atomicité de la transaction soit par un ROLLBACK automatique en cas d'erreur système, soit

vous permettant d'utiliser ROLLBACK manuellement pour annuler la transaction et les modifications apportées par les opérations de la transaction, en fonction du contexte de l'erreur.

Ici, on a considéré qu'il y a assez d'argent sur le compte 1 pour retirer 100 et qu'il n'y a pas de problème pour ajouter 100 sur le compte 2. Si l'une des deux opérations échoue, on annule tout.

Maintenant, on va voir un exemple plus complet avec une vérification de solde avant de faire le débit.

Je vais ici faire du pseudo code pour que vous compreniez bien le principe. En effet, il n'est pas possible de faire un IF dans une transaction. Mais je vais vous montrer comment on pourrait faire. Vous devrez le faire côté application.

```
-- Démarrage de la transaction
START TRANSACTION;

-- Tentative de débit de 100 unités du compte 1
UPDATE compte SET solde = solde - 100 WHERE id_compte = 1;

-- Vérification que le solde du compte 1 ne devient pas négatif après le débit
IF (SELECT solde FROM compte WHERE id_compte = 1) < 0 THEN
    -- Si le solde est négatif, annulation de la transaction
    ROLLBACK;
    -- Sortie de la procédure ou du script avec un message d'erreur ou une action spécifi
ELSE
    -- Si le solde est suffisant, continuation avec le crédit sur le compte 2
    UPDATE compte SET solde = solde + 100 WHERE id_compte = 2;
    -- Validation de la transaction
    COMMIT;
END IF;
```

L'exemple est plus complexe car il vérifie que le solde du compte 1 ne devient pas négatif après le débit. Si c'est le cas, la transaction est annulée. Sinon, le crédit est effectué sur le compte 2 et la transaction est validée.

Mais si le solde a changé au niveau du IF ??? On pourra alors utiliser un FOR UPDATE pour verrouiller (lock) le compte 1 et éviter que le solde change entre le IF et le UPDATE.

```sql
-- Démarrage de la transaction
START TRANSACTION;

-- Verrouillage pessimiste du solde du compte pour le débit
SELECT solde FROM comptes WHERE id_compte = 1 FOR UPDATE;

-- Effectuer le débit
UPDATE comptes SET solde = solde - 100 WHERE id_compte = 1;

-- Vérification et opérations suivantes
IF (SELECT solde FROM comptes WHERE id_compte = 1) < 0 THEN
    ROLLBACK;
ELSE
    -- Effectuer le crédit avec le verrouillage pessimiste aussi si nécessaire
    UPDATE comptes SET solde = solde + 100 WHERE id_compte = 2;
    COMMIT;
END IF;
```

Souvent, on fait cela côté application comme en PHP mais je vous le montre en SQL pour que vous sachiez que c'est possible. En effet, il n'est pas logique de retirer 100 d'un compte si le solde ne le permet pas. Donc on ne devrait pas faire l'update si le solde est négatif.

```sql
-- Démarrage de la transaction
START TRANSACTION;

-- Verrouillage pessimiste du solde du compte pour le débit
SELECT solde FROM comptes WHERE id_compte = 1 FOR UPDATE;

-- Vérification et opérations suivantes
IF (SELECT solde FROM comptes WHERE id_compte = 1) < 100 THEN
    ROLLBACK;
ELSE
    -- Effectuer le débit
    UPDATE comptes SET solde = solde - 100 WHERE id_compte = 1;
    -- Effectuer le crédit avec le verrouillage pessimiste aussi si nécessaire
    UPDATE comptes SET solde = solde + 100 WHERE id_compte = 2;
    COMMIT;
END IF;
```

Donc, en résumé, une transaction est un ensemble d'opérations qui doivent être exécutées ensemble. Si une seule opération échoue, toutes les opérations de la transaction doivent être annulées. Cela permet de garantir l'intégrité des données.

Le IF n'est pas possible dans une transaction. Vous devrez le faire côté application. Mais je vous ai montré comment on pourrait faire en SQL. Le IF est utilisable dans une procédure stockée ou une fonction.

30.

⬅Revenir au menu.
