⬅Revenir au menu.

Introduction aux bases de données

Johnny Piette

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 souvent qu'on ajoute le préfixe « Id » (pour identifiant) à une clef d'identité. Exemple : IdClient, IdEtudiant. Cette clef est souvent écrite Id. Rajouter IdEleve 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.

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

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

Pour illustrer ces principes, considérons une table simple `Commandes` dans une base de données de commerce électronique :

Commandes

ID_Commande	Produits	Quantité	Prix_Unitaire	ID_Client	Nom_Client
1	T-shirt, Casquette	2, 1	15, 10	101	Alice
2	T-shirt	1	15	102	Bob

2. Première forme normale (1FN)

Une table est dite en **Première Forme Normale** (`1NF`) si et seulement si tous les champs contiennent des valeurs atomiques, c'est-à-dire chaque colonne contient des valeurs indivisibles.

De plus, chaque enregistrement doit être unique.

Cette normalisation élimine les groupes répétitifs, assurant que la table représente une relation correcte. La première forme normale est la forme normale la plus simple.

Elle stipule que toutes les valeurs d'une table doivent être atomiques. C'est-à-dire qu'elles ne doivent pas être décomposables en d'autres valeurs.

En d'autres termes, chaque colonne d'une table doit contenir une seule valeur et cette valeur doit être de même type que les autres valeurs de la colonne.

Dans notre table Commandes, les champs Produits et Quantité contiennent des valeurs non atomiques. Pour atteindre la 1NF, nous devons les diviser :

Commandes (1NF)

ID_Commande	Produit	Quantité	Prix_Unitaire	ID_Client	Nom_Client
1	T-shirt	2	15	101	Alice
1	Casquette	1	10	101	Alice
2	T-shirt	1	15	102	Bob

3. Deuxième forme normale (2FN)

Pour qu'une table soit en **Deuxième Forme Normale** (`2FN`), elle doit d'abord satisfaire toutes les conditions de la 1NF. Ensuite, elle doit s'assurer que tous les attributs non-clés sont pleinement fonctionnels dépendants de la clé primaire. Cela signifie qu'il n'y a pas de dépendance partielle d'un attribut sur une partie seulement de la clé primaire.

Dans notre exemple, Nom_Client dépend de ID_Client et non de ID_Commande. Pour atteindre la 2NF, nous séparons les informations du client dans une table distincte :

Commandes

ID_Commande	Produit	Quantité	Prix_Unitaire	ID_Client
1	T-shirt	2	15	101
1	Casquette	1	10	101
2	T-shirt	1	15	102

Clients

ID_Client	Nom_Client
101	Alice

ID_Client	Nom_Client
102	Bob

4. Troisième forme normale (3FN)

Une table est en Troisième Forme Normale si elle est en 2NF et que tous ses attributs non-clés sont non seulement dépendants de la clé primaire mais aussi mutuellement indépendants. En d'autres termes, aucun attribut non-clé ne doit dépendre d'un autre attribut non-clé. Cela aide à éliminer les dépendances transitives.

Dans notre exemple, Prix_Unitaire pourrait dépendre du Produit et non de la Commande. Nous créons donc une table distincte pour les produits :

Commandes

ID_Commande	Produit	Quantité	ID_Client
1	T-shirt	2	101
1	Casquette	1	101
2	T-shirt	1	102

Clients

ID_Client	Nom_Client
101	Alice
102	Bob

Produits

Produit	Prix_Unitaire
T-shirt	15
Casquette	10

5. Vers une Conception Plus Avancée

À ce stade, notre table `Commandes` est bien structurée selon les principes de la 3NF. Cependant, dans une conception de base de données relationnelle avancée, il est courant d'utiliser des identifiants uniques pour les relations entre les tables. Cela réduit la redondance et améliore l'efficacité.

Nous allons donc faire évoluer notre exemple pour utiliser `ID_Produit` au lieu de Produit. Cela implique l'introduction d'une table `Produits` distincte, où chaque produit est identifié par un `ID_Produit` unique. Voici comment cela se présente :

Produit

ID_Produit	Nom_Produit	Prix_Unitaire
1	T-shirt	15
2	Casquette	10

Nous gardons la même table Clients :

Client

ID_Client	Nom_Client
101	Alice
102	Bob

Et nous modifions la table Commandes pour utiliser les identifiants uniques :

Commande

ID_Commande	ID_Produit	Quantité	ID_Client
1	1	2	101
1	2	1	101
2	1	1	102

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, DateNaissance
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 :

```sql
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 :

```sql
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 :

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

Ou encore nées dans les années 1990 :

```
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.

```
SELECT *
FROM Eleve
WHERE Tel IS NOT NULL ;
```

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

```
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 :

```
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 ;

```
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

```
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 ;

```
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.

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

5.5 COUNT(champ) et COUNT(*)

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

```
SELECT COUNT(*)
FROM Eleve ;
```

On va voir que le résultat sera une colonne avec comme nom : count(*) avec une cellule ayant 13 comme valeur. Il y a donc 13 élèves.
Cependant le nom de la colonne n'est pas très agréable à lire, on peut lui donner un autre nom simple :

```
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(Sexe) 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 : 13-11 = 2 filles

```
SELECT COUNT(Sexe) 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' ;
```

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

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

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

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

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;
```

Ou bienn elle pourrait s'écrire ainsi

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

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 Classe.

Table Classe:

IdClasse (Clef primaire)	Nom	Lieu
1	BlindCode	BXL
2	BlindCode4Data	LLN

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

IdEleve (Clef primaire)	Prénom	Nom	IdClasse (clef étrangère)
1	Alain	Dufrasne	2
2	Bruno	Defalque	2
3	Eleonor	Sana	2

IdEleve (Clef primaire)	Prénom	Nom	IdClasse (clef étrangère)
4	Jessie	Bakashika	2
5	Mahsum	Kizmaz	2
6	Maxime	Borsen	2
7	Isaac	Tcheuyassi	2
8	Matthieu	DARFEUILLE	1
9	Simon	DESSEILLE	1
10	Ibrahim	TAMDITI	1
11	Sophie	De BACKER	1
12	Yves	BEYA	1
13	Mounir	BEN AHMED	1

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

```
SELECT *
FROM Eleve
```

Ca nous affiche tous le champs mais malheureusement le champ relatif à la classe 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**

```
SELECT Eleve.Nom, Prenom, Classe.Nom
FROM Eleve
INNER JOIN Classe ON Eleve.IdClasse = Classe.IdClasse ;
```

Décortiquons cette requête:

- Le **SELECT** est particulier car on a mis Eleve.Nom et Classe.Nom pour éviter une ambiguité. En effet, Mysql ne saura pas si on veut le nom de l'élève ou le nom de la

Classe si on ne spécifie pas la table.

- **FROM** *Eleve*: On veut prendre des informations de la table *Eleve*.
- **INNER JOIN** *Classe* **ON** *Eleve.IdClasse* **=** *Classe.IdClasse*
    i. **INNER JOIN** *Classe*: on veut joindre la table *Classe* à la table *Eleve* (du **FROM**).
    ii. **ON** *Eleve.IdClasse* **=** *Classe.IdClasse*: On dit comment on va lier nos tables. Ici on va lier l'*IdClasse* de la table *Eleve* sur (**ON**) l'*IdClasse* de la table *Classe*. Le SGBD cherchera les enregistrements dans les deux tables où l'IdClasse de table Eleve = à l'IdClasse de la table Classe. On fait donc une égalité sur la clef étrangère de la table Eleve (Eleve.IdClasse) et sur la clef primaire de la table Classe (Classe.IdClasse).

Si on a besoin de tous les champs de la table *Eleve*, on peut changer la requête de cette manière pour éviter de taper tous les champs:

```
SELECT Eleve.*, Classe.Nom
FROM Eleve
INNER JOIN Classe ON Eleve.IdClasse = Classe.IdClasse ;
```

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, Classe.Nom
FROM Eleve, Classe
WHERE Eleve.IdClasse = Classe.IdClasse ;
```

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 classe des élèves masculins dont le nom de famille commence par un 'b':

```
SELECT Eleve.Nom, Prenom, Classe.Nom
FROM Eleve
INNER JOIN Classe ON Eleve.IdClasse = Classe.IdClasse
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.

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.

8. 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 fait manière, vous pouvez modifier comme vous le souhaitez nos base 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...

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)
```

Personnellement, je n'utilise que des **VARCHAR** sauf par exemple 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(1) qui est utilisé par MySQL. Donc on pourrait avoir une valeur comprise entre 0 et 9.

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 requête 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 le type d'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.

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... Pour ma part, le SGBD le permet, je l'utilise. ☺

18. 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;

18.1 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.

18.2 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/

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

```
USE Ventes;
DELIMITER $$
DROP FUNCTION IF EXISTS price_tvac;
CREATE FUNCTION price_tvac(price FLOAT(5,2)) RETURNS FLOAT(5,2)
BEGIN
DECLARE price_tvac FLOAT(5,2);
SET price_tvac=price+price*0.21;
RETURN price_tvac;
END;
$$
DELIMITER ;
```

20.2 Supprimer une fonction - DROP FUNCTION

20.3 Modifier une fonction - ALTER FUNCTION

21. Les Procédures stockées

22. Les vues - VIEW

⬅Revenir au menu.
