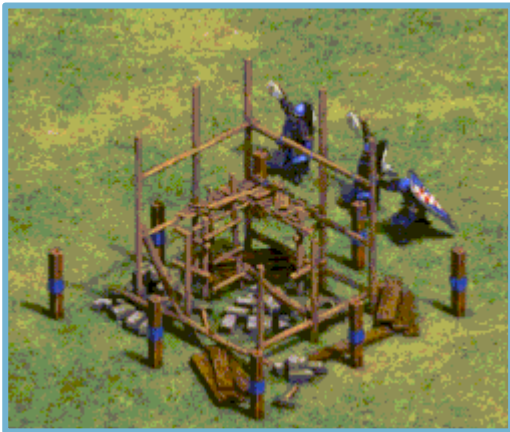


Rencontre 2

DDL et DML

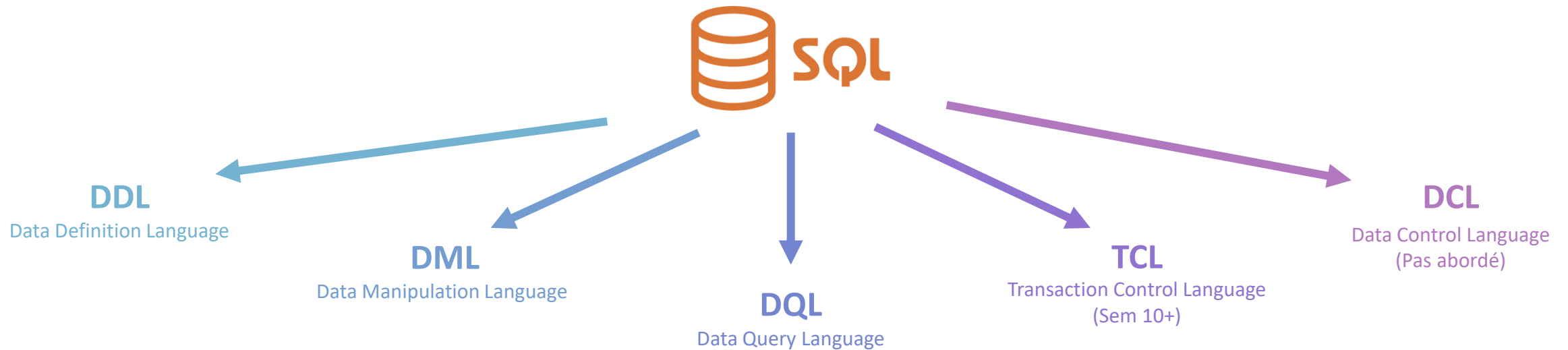
Bases de données et programmation Web



On va finalement *construire* la base de données !



- ❖ Définition de données (DDL)
- ❖ Manipulation de données (DML)












❖ Définition de données

- ◆ Après avoir conçu le **modèle logique**, (qui représente des tables de données), il est temps de créer **concrètement** la base de données et ses **tables** dans un **système de gestion de base de données**, (SGBD) comme Microsoft SQL Server.
- ◆ Pour la définition de données (ainsi que toutes les futures étapes dans le cours), nous utiliserons majoritairement des **scripts SQL**, plutôt que d'utiliser une **interface utilisateur**. (UI)



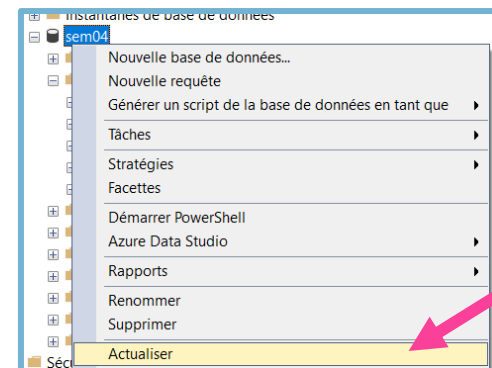
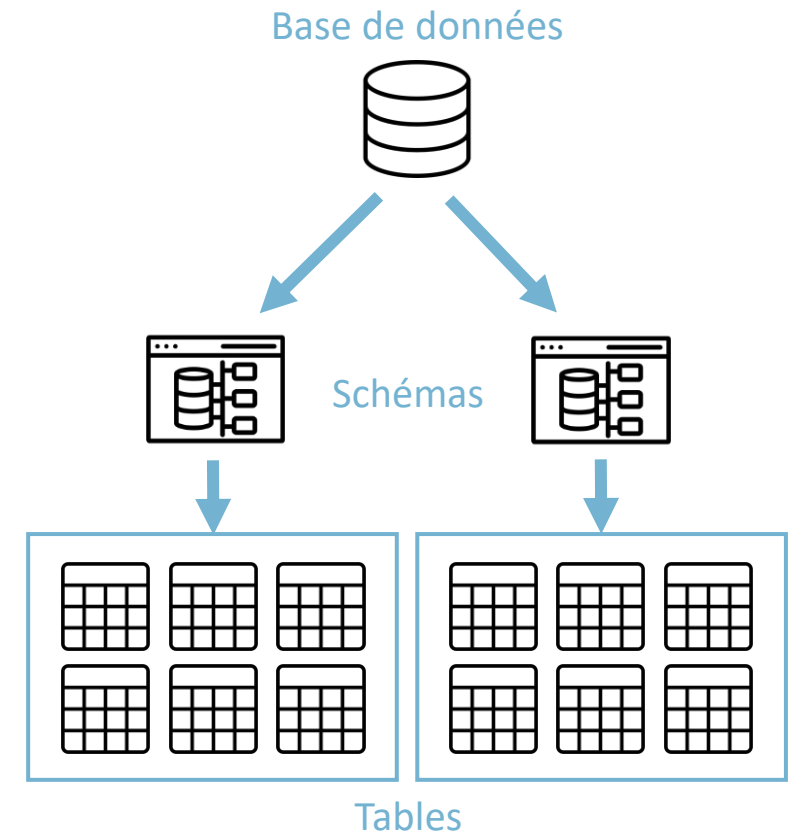
- ❖ Pourquoi des scripts plutôt qu'un UI ?  
- ◆  Les scripts SQL peuvent être **utilisés avec plusieurs SGBD**. (Contrairement à l'UI, qui est unique pour chaque SGBD)
- ◆  Les scripts SQL peuvent être **réutilisés** facilement si plusieurs actions sont similaires.
- ◆  La majorité des opérations peuvent être **automatisées** à long terme grâce à des scripts SQL.
- ◆  **Simplifie** la réalisation de certaines opérations qui sont beaucoup plus complexes avec des UI. (Voir impossibles)
- ◆  Les scripts SQL peuvent parfois être optimisés pour améliorer la **performance**.



❖ Définition de données

- ◆ Créer une **base de données**
- ◆ Créer un **schéma**
- ◆ Créer une **table** (+ **clés**, + **contraintes**)
- ◆ Créer des vues
- ◆ Créer des procédures stockées
- ◆ Créer des fonctions
- ◆ Créer des déclencheurs

Semaine 4 et 5



N'oubliez pas que dans **SSMS** il faut « réactualiser » la base de données après avoir fait une requête pour remarquer des changements !



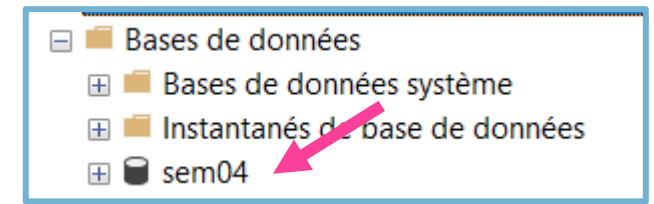
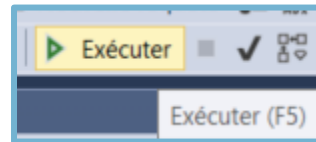
❖ Créer / supprimer une **base de données**

- ◆ Rappel : SQL n'est pas sensible à la casse pour les mots-clés. (CREATE, SELECT, WHERE, etc.)

```
CREATE DATABASE nom;
```



```
CREATE DATABASE sem04;
```



Supprimer une base de données, ses schémas, ses tables, etc.

```
DROP DATABASE nom;
```

Voir la liste des bases de données.


```
SELECT * FROM  
sys.databases;
```

	name	database_id
1	master	1
2	tempdb	2
3	model	3
4	msdb	4
5	sem04	13



❖ Utiliser une **base de données**

- ◆ Une fois la base de données créée, il faut mentionner qu'on veut l'utiliser avec **USE**



```
CREATE DATABASE Sem04;  
GO  
USE Sem04;  
GO
```

Le **GO** entre l'instruction CREATE DATABASE et USE DATABASE permet de s'assurer que la création de la base de donnée est terminée AVANT qu'on essaie de l'utiliser.



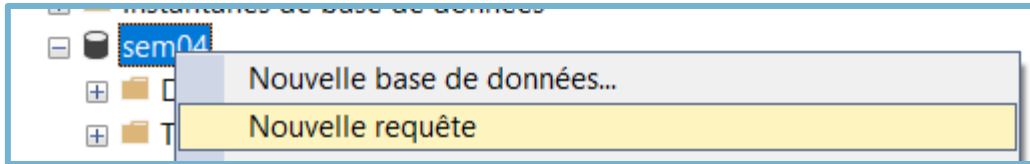
❖ Schémas

- ◆ Les schémas sont optionnels. (Si on n'en crée pas, toutes les tables seront dans le même schéma « dbo ». Cependant, c'est définitivement une BEST PRACTICE d'en définir!!!)
- ◆ Les schémas permettent d'**organiser** / séparer les objets de la BD, (tables, fonctions, procédures, vues, etc.) un peu comme des « namespace ».
 - Simplifie l'organisation des **accès**. (Sécurité)
 - On peut donner des permissions sur un schéma entier.
 - Simplifie la gestion des **backups**.
 - On peut faire des backups sur un schéma spécifique.
 - Si des tables ont des relations (ou sont souvent utilisées ensemble), on les met *généralement* dans le même schéma.
 - Impossible de créer deux tables avec le même nom dans un même schéma.

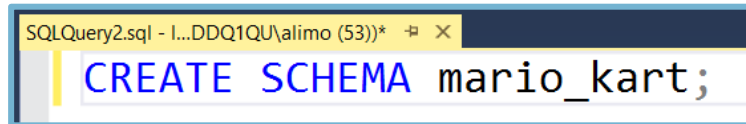


❖ Créer / supprimer un schéma

- ◆ Attention ! Supprimer un schéma **supprime également** toutes les **tables** (et autres objets) qu'il contient.



```
CREATE SCHEMA nom;
```

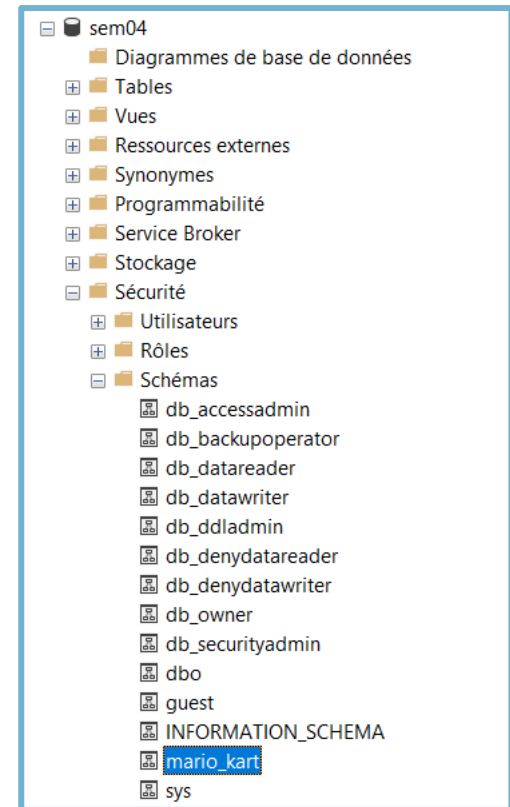


Supprimer un schéma

```
DROP SCHEMA nom;
```

Voir la liste des schémas d'une base de données

```
SELECT * FROM  
database_name.sys.schemas;
```





❖ Créer une table

```
CREATE TABLE schema_name.table_name(  
    col_name type,  
    col_name type,  
    col_name type  
);
```

- **schema_name** peut être omis si on utilise le schéma par défaut. Définitivement une BEST PRACTICE de mettre le nom du schéma par contre.

Supprimer une table (et ses données)

```
DROP TABLE schema_name.table_name;
```

```
CREATE TABLE Mario_kart.Kart(  
    KartID int,  
    Nom nvarchar(15),  
    Vitesse numeric(3,2),  
    Acceleration numeric(3,2),  
    Poids numeric(3,2),  
    TenueDeRoute numeric(3,2),  
    Traction numeric(3,2),  
    MiniTurbo numeric(3,2)  
);
```

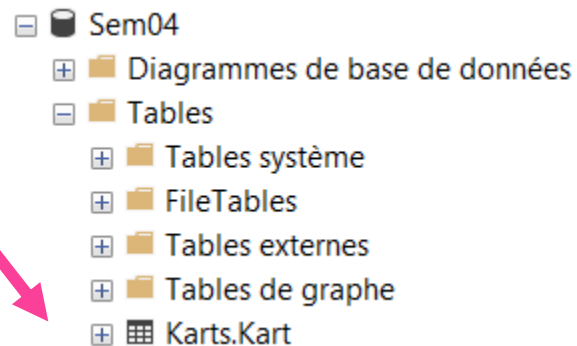
Convention : Tous les noms de tables sont au singulier.



❖ Transférer une table d'un schéma à un autre

- ◆ Souvent, on se rend compte qu'on a créé une table dans le mauvais schéma.

```
GO
CREATE SCHEMA Karts;
GO
ALTER SCHEMA Karts
TRANSFER Mario_kart.Kart;
GO
```



On se rends compte qu'on n'a pas pensé assez à notre affaire quand on a créé un schéma général pour toutes les tables de la BD.

Clairement, pour s'occuper des karts il nous faudra une expertise spéciale et on ne voudra pas que tout le monde puisse manipuler les données de nos karts. C'est certain que plus tard, on voudra mettre de la sécurité sur la table Kart. Donc le mieux est de créer un schéma pour nos Karts.



❖ Modifier une table

Opération	Requête
Ajouter une colonne	<code>ALTER TABLE schema_name.table_name ADD nomCol type;</code>
Supprimer une colonne	<code>ALTER TABLE schema_name.table_name DROP nomCol;</code>
Renommer une colonne	<code>ALTER TABLE schema_name.table_name RENAME COLUMN old_name TO new_name;</code>
Changer le type d'une colonne	<code>ALTER TABLE schema_name.table_name ALTER COLUMN col_name type;</code>



❖ Choix des types

◆ Chaînes de caractères

- **char(N)** : Chaîne de caractères de N caractères maximum. (1 à 8000) Taille fixe occupée en mémoire, peu importe la taille de la chaîne. À utiliser si toutes les chaînes ont une taille TRÈS similaire.
- **varchar(N)** : Chaîne de caractères de N caractères maximum. (1 à 8000) Taille variable occupée en mémoire lorsque la chaîne est plus petite que N. À utiliser si les tailles varient.
- **nchar(N) / nvarchar(N)** : Identique, sauf que les caractères Unicode sont utilisés. (Beaucoup plus d'alphabets et de symboles), mais 2 bytes de stockage par caractère au lieu de 1. *BEAUCOUP PLUS UTILISÉ DE NOS JOURS.*
- **text / ntext** : Jusqu'à 2 Go. Pour les très longues chaînes de caractères.

◆ Images

- **varbinary(max)** : Jusqu'à 2 Go. Nous l'utiliserons tard dans la session.
- **image** : Similaire à varbinary(max), mais sera retiré par Microsoft bientôt.



❖ Choix des types

◆ Numérique

- **bit** : Vaut 0, 1 ou NULL. Parfait pour les booléens. On met 1 ou 0 au lieu de true ou false.
- **tinyint** : 0 à 255. 1 byte.
- **smallint** : $0 \pm 32\,767$. 2 bytes.
- **int** : $0 \pm 2\,147\,483\,647$. 4 bytes.
- **bigint** : $0 \pm 9\,223\,372\,036\,854\,775\,807$. 8 bytes.
- **numeric(p,s)** / **decimal(p,s)** : Les deux sont identiques à 90%. Nombres décimaux avec un maximum de p chiffres au total et un maximum de s chiffres après la virgule. Ex : avec **numeric(5,2)**, je peux stocker 123.45, mais pas 1234.56 ou 12.345. 5 à 17 bytes.
- **float(n)** : n = 24 pour stocker sur 4 bytes ou n = 53 pour stocker sur 8 bytes. Moins précis que **numeric(p,s)**.
- **real** : équivalent à **float(24)**.



❖ Choix des types

◆ Dates

- **datetime** : 1^{er} janvier 1753 au 31 décembre 9999. Précision de 3.33 milli-s. 8 bytes.
 - Format : 'YYYY-MM-DD HH:MI:SS' (ex : '2020-12-20 17:59:59')
- **datetime2** : 1^{er} janvier 0001 au 31 décembre 9999. Précision de 100 nano-s. 8 bytes.
 - Format : 'YYYY-MM-DD HH:MI:SS.nnnnnnn' (ex : '2020-12-20 17:59:59.1234567')
- **date** : Stocke seulement la date, sans le moment de la journée. Année 0001 à 9999. 3 bytes.
 - Format : 'YYYY-MM-DD'
- **time** : Stocke seulement le moment de la journée. Précision de 100 nano-s. 5 bytes.
 - Format : 'HH:MI:SS'



❖ Contraintes

- ◆ Pour assurer l'intégrité de certaines données, SQL nous permet d'ajouter des contraintes parmi les suivantes :

Contrainte	Description	Exemple
NOT NULL	La colonne ne peut pas être vide. (avoir des valeurs Null)	<code>ColName datatype NOT NULL</code>
UNIQUE	Aucune donnée de la colonne ne peut se répéter.	Une colonne avec des valeurs uniques : <code>ColName varchar(20) UNIQUE</code> Un <u>ensemble</u> de valeurs uniques sur plusieurs colonnes : <code>CONSTRAINT nom UNIQUE (Col1, Col2)</code>
CHECK	S'assure que les valeurs respectent une condition.	<code>CONSTRAINT nom CHECK (ColName >=10)</code>
DEFAULT	Définit une valeur par défaut si la colonne n'est pas remplie.	<code>CONSTRAINT nom DEFAULT 'voiture' FOR NomCol</code>

Les contraintes peuvent être indiquées après le type d'une colonne, mais pour certaines il est préférable de créer et **nommer** la contrainte séparément. (À suivre dans 2 diapos)



❖ Contraintes

- ◆ Pour assurer l'intégrité de certaines données, SQL nous permet d'ajouter des contraintes parmi les suivantes :

Contrainte	Description	Exemple
PRIMARY KEY	Combinaison de NOT NULL et UNIQUE. Définit une clé primaire.	Clé primaire atomique : <code>CONSTRAINT nom PRIMARY KEY (Col1)</code> Clé primaire composée : <code>CONSTRAINT nom PRIMARY KEY (Col1, Col2)</code>
FOREIGN KEY	La valeur doit être une référence vers une PRIMARY KEY qui existe.	<code>ALTER TABLE schema.table ADD CONSTRAINT FK_nom FOREIGN KEY (ColName1) REFERENCES schema.table(ColName2)</code>
IDENTITY(x,y)	Auto-incrémentation des valeurs. x = première valeur, y = valeur d'incrément+++. Pas besoin de fournir de valeur lors d'un INSERT.	<code>ColName int IDENTITY(1,1)</code>



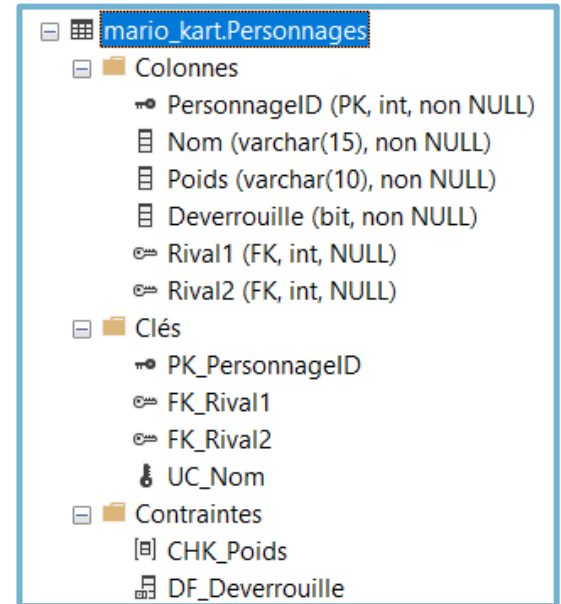
❖ Contraintes

◆ Option 1 : Les ajouter lors de la création de la table

- À privilégier pour les contraintes de **clé primaire** et **NOT NULL**.
- Incontournable pour **IDENTITY**.
 - (On ne peut pas utiliser l'option 2 pour IDENTITY)

```
CREATE TABLE Courses.Personnage (  
    PersonnageID int IDENTITY (1,1) ,  
    Nom varchar(15) NOT NULL,  
    Poids varchar(10) NOT NULL,  
    CONSTRAINT PK_Personnage_PersonnageID PRIMARY KEY (PersonnageID)  
);
```

Toutes les contraintes (**PRIMARY KEY**, **FOREIGN KEY**, **SET DEFAULT**, etc.) pourraient être spécifiées directement après le type de chaque colonne, mais on préfère les spécifier séparément car ça nous permet de les **NOMMER**. Nommer une contrainte simplifie leur suppression et permet de respecter certains standards de nommage.





❖ Contraintes

◆ Convention pour les noms des contraintes

○ Type_**Table**_Colonne

- **Type** : PK (Primary Key), FK (Foreign Key), CK (Check), UC (Unique), DF (Default)
- **Table** : Nom de la table
- **Colonne** : Nom de la colonne concernée. Si c'est un groupe de colonnes, un autre nom cohérent peut être utilisé.

```
ALTER TABLE Joueurs.Joueur ADD CONSTRAINT UC_Joueur_Pseudo UNIQUE (Pseudo)
GO
```

```
ALTER TABLE Courses.Kart ADD CONSTRAINT CK_Kart_Stats CHECK
( (Vitesse BETWEEN 0 AND 6) AND
  (Acceleration BETWEEN 0 AND 6) AND
  (Poids BETWEEN 0 AND 6) AND
  (TenueDeRoute BETWEEN 0 AND 6)
)
```



❖ Contraintes

◆ Convention pour les noms des contraintes

○ Type_**Table**_Colonne

- **Type** : PK (Primary Key), FK (Foreign Key), CK (Check), UC (Unique), DF (Default)
- **Table** : Nom de la table
- **Colonne** : Nom de la colonne concernée. Si c'est un groupe de colonnes, un autre nom cohérent peut être utilisé.

```
ALTER TABLE Joueurs.Joueur ADD CONSTRAINT UC_Joueur_Pseudo UNIQUE (Pseudo)  
GO
```

Cette convention pour les noms des contraintes est très importante.

On a souvent des colonnes qui ont le même nom de tables en tables.

La 'date' par exemple. On a la date d'embauche de l'employé, de l'entraîneur de l'équipe, etc...

En ajoutant le nom de la table AVANT la colonne, on s'assure que le nom de la contrainte est vraiment unique, car on ne peut pas avoir le même nom de colonne dans la même table.



❖ Contraintes

◆ Option 2 : Modifier une table existante

- Méthode à utiliser pour **toutes** les autres contraintes que **PK**, **NOT NULL** et **IDENTITY**

Pour qu'une colonne devienne NOT NULL , on écrase l'ancien type.	<pre>ALTER TABLE Personnages.Personnage ALTER COLUMN Nom nvarchar(15) NOT NULL;</pre>
Pour ajouter une contrainte nommée de type DEFAULT	<pre>ALTER TABLE Personnages.Personnage ADD CONSTRAINT DF_Personnage_Deverouille DEFAULT 0 FOR Deverouille;</pre>
Pour ajouter une contrainte nommée de type CHECK	<pre>ALTER TABLE Personnages.Personnage ADD CONSTRAINT CK_Personnage_Poids CHECK (Poids in ('leger', 'moyen', 'lourd'))</pre>
Pour supprimer une contrainte	<pre>ALTER TABLE Personnages.Personnage DROP CONSTRAINT DF_Personnage_Deverouille</pre>



❖ Contraintes

◆ « CHECK » une contrainte

- Lorsqu'une contrainte existe déjà (ou vient d'être créée), on peut utiliser l'instruction suivante :

```
ALTER TABLE schema.table CHECK CONSTRAINT nom_contrainte
```

- S'il y avait déjà des données dans la table qui **violent** la **contrainte**, (si la contrainte a été ajoutée après **l'insertion de données**) une **erreur** sera lancée, nous permettant ainsi d'identifier les données qui ne respectent pas la nouvelle **contrainte**.
 - Cela ne supprime / modifie aucune donnée automatiquement.
 - On doit ensuite utiliser une stratégie de notre choix (ex : un **UPDATE**) pour modifier ces données.



❖ Cascade sur clés étrangères

```
ALTER TABLE Ventes.Vente ADD CONSTRAINT FK_Vente_ClientID  
FOREIGN KEY (ClientID)  
REFERENCES Clients.Client (ClientID)  
ON DELETE CASCADE
```



Pour l'exemple ci-dessus, on a une **clé étrangère** qui fait référence à une clé primaire nommée **ClientID** dans la table **Client**. Disons que la **clé primaire** vaut **7**. La **clé étrangère** vaut donc **7** aussi. Comme on a choisi **CASCADE** pour **DELETE**. Si la **clé primaire** avec la valeur **7** est supprimée, la rangée de données avec la **clé étrangère** ayant la valeur **7** sera supprimée aussi.

Notez que pour des raisons fiscales, très souvent, on ne peut pas mettre cette contrainte. Certaines infos ne peuvent pas être supprimées: Les employés, l'historique d'emploi des employés, les ventes, le détail des ventes, etc. On va alors faire un 'soft delete' en utilisant les déclencheurs qu'on verra plus tard.



❖ Cascade sur clés étrangères

- ◆ Lorsqu'une **clé primaire** est **supprimée**, les **clés étrangères** qui lui faisaient référence peuvent provoquer des **erreurs**. (Car elles sont obligées de faire référence à une clé qui existe)
- ◆ Dans une **contrainte** de **clé étrangère**, on peut spécifier la **réaction** à la modification / suppression de la **clé primaire** associée.

```
CONSTRAINT Nom FOREIGN KEY (Colonne) REFERENCES Schema.Table (Colonne) ON X Y
```

- ◆ **X** peut être remplacé par **DELETE** ou **UPDATE**. (Pour quelle action on souhaite avoir une réaction ?)
- ◆ **Y** peut-être remplacé par **CASCADE**, **SET NULL**, **SET DEFAULT** ou **NO ACTION**.
 - **CASCADE** supprime / met à jour la rangée de données si sa clé étrangère est impactée.
 - **SET NULL** donne la valeur **NULL** à la clé étrangère impactée. (Assurez-vous que la FK ait le droit d'être NULL...)
 - **SET DEFAULT** donne la valeur par défaut prédéterminée pour la colonne qui contient la clé étrangère. (Assurez-vous que la colonne de la FK a une contrainte **DEFAULT** définie...)
 - **NO ACTION** ne spécifie aucune réaction.

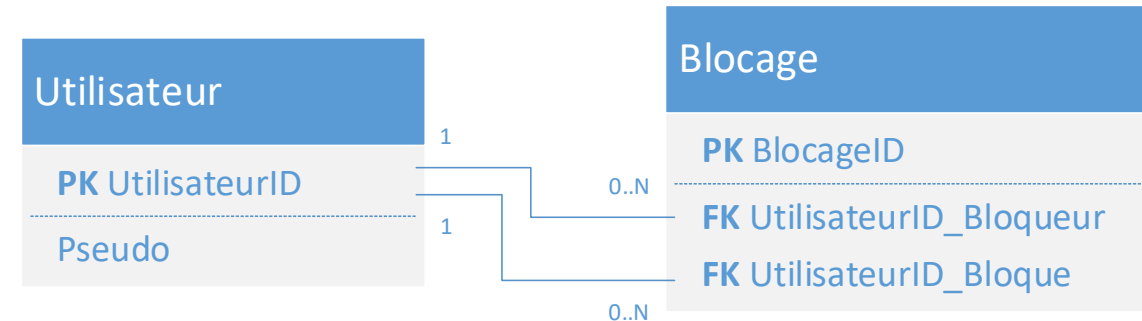


❖ Cascade sur clés étrangères

◆ Cycles et / ou cascades multiples

- Parfois, malheureusement, certaines contraintes en cascade ne sont pas compatibles.
- Dans l'exemple ci-dessous, on a deux **clés étrangères** qui font référence à la même **clé primaire**. Impossible de spécifier **ON CASCADE DELETE** ou **ON CASCADE UPDATE** pour les deux clés étrangères. (Alors que techniquement, on en a besoin)
- Dans ce genre de situation, **on laisse tout simplement tomber les cascades**. (Tout en conservant les contraintes de clé étrangère) Nous pourrions implémenter ce comportement avec les **déclencheurs**. (Semaine 5+)
 - Pour l'instant, ce serait impossible de supprimer un utilisateur associé à un blocage.

```
ALTER TABLE GestionUtilisateurs.Blocage ADD CONSTRAINT FK_Blocage_UtilisateursID_Bloqueur  
FOREIGN KEY (UtilisateurID_Bloqueur) REFERENCES GestionUtilisateurs.Utilisateur(UtilisateurID)  
ON DELETE CASCADE } À retirer  
ON UPDATE CASCADE }  
ALTER TABLE GestionUtilisateurs.Blocage ADD CONSTRAINT FK_Blocage_UtilisateursID_Bloque  
FOREIGN KEY (UtilisateurID_Bloque) REFERENCES GestionUtilisateurs.Utilisateur(UtilisateurID)  
ON DELETE CASCADE } À retirer  
ON UPDATE CASCADE }
```

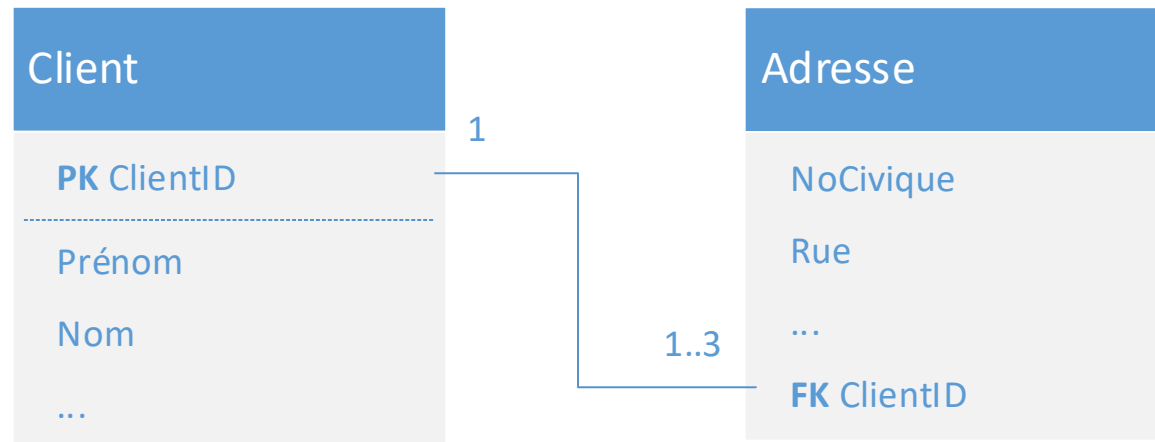


```
Msg 1785, Niveau 16, État 0, Ligne 29  
Introducing FOREIGN KEY constraint 'FK_Blocage_UtilisateursID_Bloque' on table 'Blocage' may cause cycles or multiple cascade paths.  
Msg 1750, Niveau 16, État 1, Ligne 29
```



❖ Contrainte de cardinalité

- ◆ Certaines relations ont des contraintes de cardinalités.
 - Ex : Un client doit posséder 1 à 3 adresses. **Client** et **Adresse** sont deux tables séparées et **Adresse** contient **FK ClientID** pour savoir à quel client appartient une adresse.
- ◆ Il y a moyen d'appliquer une contrainte sur la cardinalité, mais pour cela il faut utiliser des **déclencheurs** (triggers).
 - Nous aborderons les déclencheurs à la semaine 5.
 - Pour le moment, on ne gère pas les contraintes de cardinalité !





❖ Script de définition de données

- ◆ En combinant plusieurs requêtes SQL consécutives, on construit la base de données, ses schémas, leurs tables et leurs contraintes.
- ◆ Quand on crée le script, on exécute habituellement chacune des requêtes une à la fois à mesure qu'on les écrit. Pour voir plus facilement où sont nos erreurs.
- ◆ Par la suite, habituellement, on exécute le script tout d'un seul coup.

```
USE master
GO
CREATE DATABASE BD_Mario_Kart;
GO
USE BD_Mario_Kart
GO

CREATE SCHEMA Joueurs;
GO
CREATE SCHEMA Courses;
GO
CREATE SCHEMA Personnages;

GO

CREATE TABLE Joueurs.Joueur(
    JoueurID int IDENTITY (1,1),
    Pseudo varchar(20) NOT NULL,
    Cote int NOT NULL,
    CONSTRAINT PK_Joueur_JoueurID PRIMARY KEY (JoueurID)
);

CREATE TABLE Courses.Course(
    CourseID int IDENTITY (1,1),
    Nom varchar(20) NOT NULL,
    CONSTRAINT PK_Course_CourseID PRIMARY KEY (CourseID)
);

CREATE TABLE Personnages.Personnage(
    PersonnageID int IDENTITY (1,1),
    Nom varchar(15) NOT NULL,
    Poids varchar(10) NOT NULL,
    EstDeverrouille bit NOT NULL,
    CONSTRAINT PK_Personnage_PersonnageID PRIMARY KEY (PersonnageID)
);

CREATE TABLE Courses.Kart(
    KartID int IDENTITY (1,1),
    Nom varchar(15) NOT NULL,
    Vitesse numeric(3,2) NOT NULL,
    Accelération numeric(3,2) NOT NULL,
    Poids numeric(3,2) NOT NULL,
    TenueDeRoute numeric (3,2) NOT NULL,
    CONSTRAINT PK_KartID PRIMARY KEY (KartID)
);

CREATE TABLE Courses.Participation(
    ParticipationID int IDENTITY (1,1),
    Position int NOT NULL,
    Chrono int NOT NULL,
    DatePartie datetime NOT NULL,
    JoueurID int NOT NULL,
    CourseID int NOT NULL,
    KartID int NOT NULL,
    PersonnageID int NOT NULL,
    CONSTRAINT PK_ParticipationID PRIMARY KEY (ParticipationID)
);

GO

-- Création des contraintes de clés étrangères
```

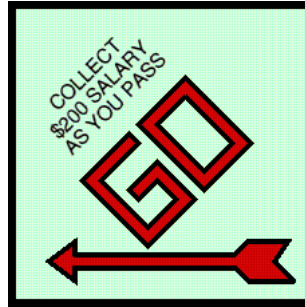


❖ Séparateur « GO »

- ◆ Permet de segmenter un script en **lots**. (En « **batch** »)

```
CREATE SCHEMA Joueurs;  
GO  
CREATE SCHEMA Courses;  
GO  
  
CREATE TABLE Joueurs.Joueur (  
    JoueurID int IDENTITY (1,1) ,  
    Pseudo varchar(25) NOT NULL,  
    CONSTRAINT PK_Joueur_JoueurID PRIMARY KEY (JoueurID)  
);  
GO
```

Trois « batch »



◆ Pourquoi faire ?

- Pour le moment, disons simplement que si une **batch** échoue, les autres ne sont pas impactées et elles sont exécutées malgré tout. (Une **erreur** interrompt seulement le reste de la **batch** où elle se situe)
- Permet de seulement réexécuter certaines parties d'un script lorsqu'il y a des erreurs.

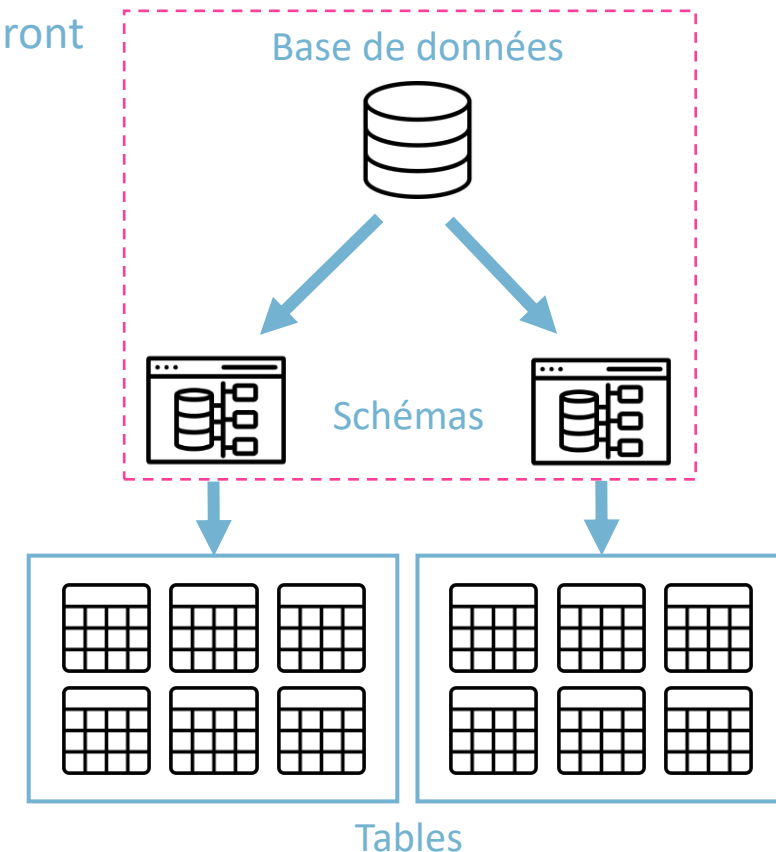


❖ Exemple de structure d'un script de définition de données

◆ Partie 1 : Créer la BD et ses schémas

- **USE MASTER** permet de s'assurer que la BD est créée à partir de la racine du SGBD.
- **USE Nouvelle_BD** permet de s'assurer que les schémas et tables seront créés dans la bonne base de données.

```
1  USE master
2  GO
3  CREATE DATABASE BD_Mario_Kart;
4  GO
5  USE BD_Mario_Kart
6  GO
7
8  CREATE SCHEMA Joueurs;
9  GO
10 CREATE SCHEMA Courses;
11 GO
```





❖ Exemple de structure d'un script de définition de données

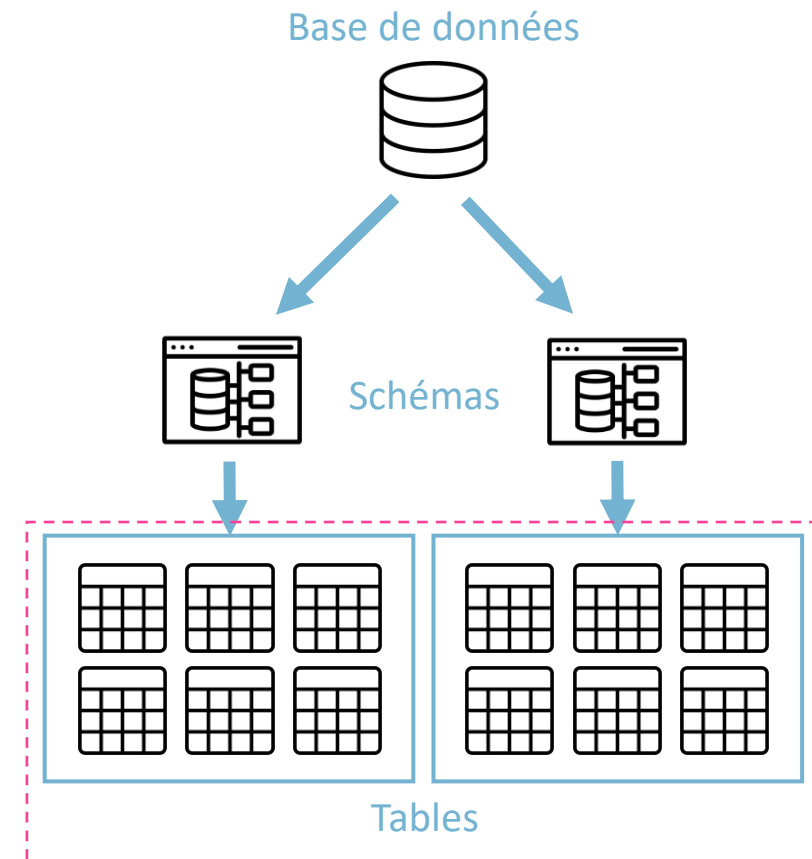
◆ Partie 2 : Créer les tables et les contraintes de **clé primaire**

- Nous allons spécifier les autres contraintes d'intégrité **plus tard** !
- Celles qui sont absolument prioritaires :
 - **Clé primaire**
 - **NOT NULL** (Nous sommes obligés de le spécifier avec le type)
 - **IDENTITY** (Nous sommes obligés de le spécifier avec le type)

```
-- Création des tables

CREATE TABLE Joueurs.Joueur (
  JoueurID int IDENTITY (1,1) NOT NULL,
  Pseudo varchar(25) NOT NULL,
  CONSTRAINT PK_Joueur_JoueurID PRIMARY KEY (JoueurID)
);

CREATE TABLE Courses.Course (
  CourseID int IDENTITY (1,1) NOT NULL,
  Nom varchar(25) NOT NULL,
  CONSTRAINT PK_Course_CourseID PRIMARY KEY (CourseID)
);
```





❖ Exemple de structure d'un script de définition de données

◆ Partie 3 : Créer les contraintes de clé étrangère

```
-- Création des contraintes de clés étrangères

ALTER TABLE Courses.Participation ADD CONSTRAINT FK_Participation_JoueurID
FOREIGN KEY (JoueurID) REFERENCES Joueurs.Joueur (JoueurID)
ON DELETE CASCADE
GO

ALTER TABLE Courses.Participation ADD CONSTRAINT FK_Participation_KartID
GO

ALTER TABLE Courses.Participation ADD CONSTRAINT FK_Participation_CourseID
FOREIGN KEY (CourseID) REFERENCES Courses.Course (CourseID)
ON DELETE CASCADE
```

Vous devez vous demander « Qu'est-ce que je veux qui se produise si le côté un de la relation est supprimé? »

Si vous avez une composition, cela signifie que vous voulez que les données du côté plusieurs de la relation soient supprimées. Donc « **ON DELETE CASCADE** »



❖ Exemple de structure d'un script de définition de données

◆ Partie 4 : Créer les autres contraintes (CHECK, UNIQUE et DEFAULT)

```
80 -- Création des contraintes autres que celles de clés primaires et étrangères
81
82 ALTER TABLE Joueurs.Joueur ADD CONSTRAINT UC_Joueur_Pseudo UNIQUE (Pseudo)
83 GO
84 ALTER TABLE Courses.Course ADD CONSTRAINT UC_Course_Nom UNIQUE (Nom)
85 GO
86 ALTER TABLE Courses.Personnage ADD CONSTRAINT CHK_Personnage_Poids CHECK (Poids in ('léger', 'moyen', 'lourd'))
87 GO
```

◆ Pourquoi l'ordre PK -> FK -> Autres contraintes ?

- PK -> FK : Pas vraiment le choix, des PK doivent exister pour créer des FK.
- Autres contraintes en dernier : Pas strict, mais les clés sont généralement plus importantes car elles permettent d'établir les relations entre les tables et c'est ce qu'on veut faire fonctionner d'abord.

◆ Attention, on ne crée pas de contrainte UNIQUE sur la clé primaire car une telle contrainte est automatiquement faite par SQL Server.



Une fois nos tables bien structurées et l'intégrité des données bien protégée, on peut peupler nos tables !

❖ Manipulation des données

- ◆ Insertion de données
- ◆ Modification de données
- ◆ Suppression de données



❖ Insertion de données

```
INSERT INTO schema_name.table_name (Col1, Col2, Col3, ...)
VALUES (valeur, valeur, valeur, ...);
```

```
-- Insertion de données
INSERT INTO Personnages.Personnage (Nom,Poids, EstDeverrouille, Rival1, Rival2)
VALUES ('Luigi', 'moyen', 1, null, null),
       ('Peach', 'moyen', 1, null, null),
       ('Bowser', 'lourd', 1, 1, 2)
```

On peut faire plusieurs ajouts d'un coup.

PersonnageID	Nom	Poids	EstDeverrouille	Rival1	Rival2
1	Luigi	moyen	1	NULL	NULL
2	Peach	moyen	1	NULL	NULL
3	Bowser	lourd	1	1	2



❖ Insertion de données

On insère des données de type date en mettant la valeur **entre apostrophes** et en utilisant le format international des dates:

Le format international pour les dates est **'YYYYMMDD'** sans séparateur entre les chiffres de l'année, du mois et du jour...

```
INSERT INTO RendezVous (Maintenant, AnneeMoisJour, AvecDateEtHeures)  
VALUES (getDate(), '20230211', '20230211 13:30:00')
```



❖ Insertion de données provenant d'une autre table

```
INSERT INTO schema_name.table_name (Col1, Col2, Col3, ...)
SELECT Champ1, Champ1, Champ3, ...
FROM schema_name.AutreTable;
```

```
INSERT INTO Personnages.PersonnageHistory ( PersonnageID, Nom, Poids, EstDeverrouille, Rival1, Rival2, Date_MAJ)
SELECT PersonnageID, Nom, Poids, EstDeverrouille, Rival1, Rival2, GetDate()
FROM Personnages.Personnage
WHERE EstDisparu = 1;
```

On utilise souvent ce type de commande INSERT pour archiver des données dans une table Historique, afin d'enlever des tables utilisées fréquemment les données qui sont maintenant inactives: anciennes adresses, clients qui n'ont pas fait affaire avec nous depuis plus de 5 ans, courriels non utilisés depuis plus de 5 ans, etc.

On s'en sert aussi quand une partie de nos données proviennent d'une autre base de données existante.

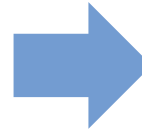


❖ Modification de données

```
UPDATE schema_name.table_name  
SET Col1 = valeur, Col2 = valeur, Col2 = valeur  
WHERE condition;
```

```
UPDATE Personnages.Personnage  
SET Nom = 'Wario', Poids = 'lourd', Rival1 = 2  
WHERE Nom = 'Mario'
```

PersonnageID	Nom	Poids	Deverrouille	Rival1	Rival2
1	Mario	moyen	1	NULL	NULL
2	Luigi	moyen	1	NULL	NULL
3	Peach	moyen	1	NULL	NULL
4	Bowser	lourd	1	1	2



PersonnageID	Nom	Poids	Deverrouille	Rival1	Rival2
1	Wario	lourd	1	2	NULL
2	Luigi	moyen	1	NULL	NULL
3	Peach	moyen	1	NULL	NULL
4	Bowser	lourd	1	1	2

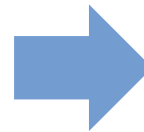


❖ Suppression de données

```
DELETE FROM schema_name.table_name  
WHERE condition;
```

```
DELETE FROM Personnages.Personnage  
WHERE Nom = 'Wario'
```

PersonnageID	Nom	Poids	Deverrouille	Rival1	Rival2
1	Wario	lourd	1	2	NULL
2	Luigi	moyen	1	NULL	NULL
3	Peach	moyen	1	NULL	NULL
4	Bowser	lourd	1	NULL	2



PersonnageID	Nom	Poids	Deverrouille	Rival1	Rival2
2	Luigi	moyen	1	NULL	NULL
3	Peach	moyen	1	NULL	NULL
4	Bowser	lourd	1	NULL	2

Lorsqu'on supprime une clé artificielle auto-incrémentée, il est normal que cela fasse un « trou » qui ne sera jamais rempli à nouveau. C'est le fonctionnement normal d'IDENTITY et cela évite certains problèmes lors d'insertions concurrentes. C'est aussi utile pour les AUDIT du département du revenu des différents gouvernements.

Si on ne met pas de condition, toutes les rangées seront supprimées ! (La structure de la table sera conservée) Toutefois, **TRUNCATE TABLE nom** est plus performant si l'objectif est bel et bien de supprimer toutes les données en conservant la table.



❖ ATTENTION pour UPDATE et DELETE

```
UPDATE schema_name.table_name  
SET Col1 = valeur, Col2 = valeur, Col2 = valeur  
WHERE condition;
```

```
DELETE FROM schema_name.table_name  
WHERE condition;
```

Si on ne met pas de condition, toutes les rangées seront mises à jour OU supprimées !

PENSEZ À UTILISER LA CLAUSE WHERE avec UPDATE et DELETE



❖ TRUNCATE

```
TRUNCATE schema_name.table_name
```

Quand on veut supprimer toutes les données d'une table mais qu'on veut garder sa structure, on utilise **TRUNCATE**.

C'est plus efficace qu'une clause DELETE sans WHERE....