Identifier et structurer les données

Dans cette activité, tu vas manipuler un fichier réel contenant des informations de billetterie. Ton objectif est d'observer les données et de comprendre pourquoi une simple liste ne suffit pas toujours.

4TTR niveau

Identifier et structurer les données

On sait maintenant que tout mettre dans un seul tableau pose des problèmes. L'objectif de cette séquence est d'apprendre à **repérer les différents objets** d'un système et à **organiser les informations** qui leur appartiennent.

Objectifs

À la fin de cette séquence, tu seras capable de :

1. Identifier les objets principaux d'un système
2. Regrouper correctement les informations qui appartiennent à chaque objet
3. Comprendre quels objets sont liés entre eux
4. Représenter la structure sous forme d'un schéma simple

Partie 1 — Repérer les objets

Situation

On reprend les données de la billetterie. Voici trois lignes du tableau Excel :

ticket_id	nom_client	prenom_client	email_client	telephone_client	evenement	date_evenement	lieu	ville	prix
T001	Dupont	Marie	marie.dupont@email.be	0471 12 34 56	Soirée Quiz Pop Culture	2026-10-03	Salle des Fêtes	Namur	12
T002	Dupont	Marie	marie.dupont@email.be	0471 12 34 56	Soirée Quiz Pop Culture	2026-10-03	Salle des Fêtes	Namur	12

Tu remarques que dans chaque ligne, les informations parlent de **choses très différentes** : la personne qui achète, l'événement concerné, le ticket lui-même et le lieu.

> Il peut y avoir plusieurs façons d'analyser des données. Dans notre cas présent, un évènement se passe toujours dans le même lieu (salle et ville). On pourrait, par simplicité, considérer que `lieu` et `ville` sont des propriétés (attributs) d'un évènement ou que le lieu est une *chose* différente en elle-même.

Exercice 1

Réponds aux questions suivantes :

1. Quelles colonnes concernent la **personne** qui achète ?
2. Quelles colonnes concernent l'**événement** ?
3. Quelles colonnes concernent le **ticket** lui-même ?
4. Quelles colonnes concernent le **lieu** lui-même ?

> Tu viens d'identifier 4 **OBJETS** du système : un Client, un Événement, un Ticket, un Lieu.

> Chaque objet regroupe des informations qui lui appartiennent logiquement.

Exercice 2

Complète le tableau ci-dessous. Pour chaque objet, note son nom et les informations qui lui appartiennent :

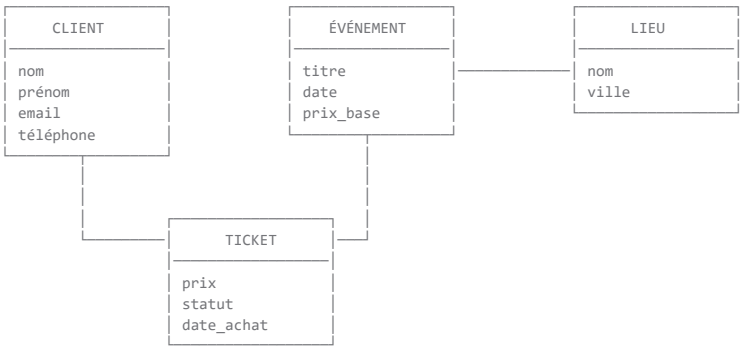
Objet	Nom	Informations
Objet 1	.....	.....
Objet 2	.....	.....
Objet 3	.....	.....
Objet 4	.....	.....

Partie 2 — Dessiner les objets et leurs liens

Maintenant qu'on a identifié les trois objets, on va les **représenter sous forme de boîtes**. Chaque objet est une boîte rectangulaire. On relie les boîtes qui sont **directement liées** entre elles.

> Une boite (entité) peut être reliées à plusieurs autres boites.

Le schéma



Lire le schéma

- Chaque **rectangle** représente un objet (que l'on appelle une **entité**).
- Les **traits** montrent quels objets sont directement liés (c'est une **relation**).
- Le **Ticket** est relié à la fois au Client et à l'Événement : c'est lui qui fait le lien entre les deux.
- Il n'y a **pas de lien direct** entre Client et Événement — ils ne se connaissent qu'à travers le Ticket.

Exercice 3 — Analyser les liens

Réponds aux questions suivantes en observant le schéma :

1. Un client peut-il avoir plusieurs tickets ?
2. Un ticket peut-il appartenir à plusieurs clients ?
3. Un événement peut-il avoir plusieurs tickets ?
4. Un ticket peut-il concerner plusieurs événements ?

> Ces questions serviront de base dans la prochaine séquence pour comprendre comment les tables sont liées.

Partie 3 — Le vocabulaire

Ce que tu viens de faire a un nom précis en bases de données. Voici les trois termes à retenir.

Entité

Une **entité** est un objet du système qu'on veut représenter.

Exemples : `Client`, `Ticket`, `Événement`, `Lieu`

Attribut

Un **attribut** est une information qui appartient à une entité.

Exemples :

- `Client` → nom, prénom, email, téléphone
- `Ticket` → prix, statut, date_achat
- `Événement` → titre, date, lieu, prix_base

Relation

Une **relation** décrit le lien entre deux entités.

Exemples :

- Un client est lié à un ticket
- Un événement est lié à un ticket

> Pour l'instant, il suffit de comprendre ce que ces termes désignent.

> On verra dans la séquence suivante comment les relations s'écrivent techniquement dans une base de données.

À retenir

Pour organiser des données :

1. On identifie les **objets importants** (les entités)
2. On regroupe les **informations** qui appartiennent à chaque entité
3. On sépare ce qui est **différent** — une information n'appartient qu'à une seule entité
4. On **relie** les entités qui sont directement liées par un trait

> Cette étape est indispensable avant de créer une base de données.

Suite

Dans la prochaine séquence, tu apprendras à analyser et mieux spécifier les relations. Mais, d'abord, quelques exercices!

Le chaos des données - Exercices

Dans cet exercice, tu vas analyser des données. Toutes les informations sont à chaque fois stockées dans un seul tableau, ce qui peut poser des problèmes d'organisation et de cohérence. Ton objectif est de comprendre comment les données sont structurées et d'identifier les différents types d'objets présents dans ce système.

4TTR niveau

DONNÉES BRUTES → ANALYSE → IDENTIFICATION DES ENTITÉS

Exercice 1 — Garage (location de voitures)

Contexte

Un garage propose un service de location de voitures. Actuellement, toutes les informations sont stockées dans un seul tableau.

Chaque ligne représente une location effectuée par un client.

Données

id_location	nom_client	prenom_client	voiture	marque	date_location	date_retour	prix_jour	statut	telephone
L001	Dupont	Marie	Clio	Renault	2026-05-01	2026-05-03	45	terminée	0471 12 34 56
L002	Martin	Lucas	208	Peugeot	2026-05-02	2026-05-05	50	en cours	0485 22 11 00
L003	Dupont	Marie	Clio	Renault	2026-05-10	2026-05-12	45	réservée	0471 12 34 56
L004	Lambert	Sophie	Golf	VW	2026-05-03	2026-05-06	60	terminée	0492 44 55 66
L005	Martin	Lucas	208	Peugeot	2026-05-12	2026-05-14	50	réservée	0485 22 11 00
L006	Dubois	Nora	Clio	Renault	2026-05-04	2026-05-06	45	terminée	0478 88 77 66
L007	Peeters	Hugo	Golf	VW	2026-05-07	2026-05-09	60	annulée	0499 10 20 30
L008	Dupont	Marie	208	Peugeot	2026-05-15	2026-05-18	50	en cours	0471 12 34 56
L009	Lambert	Sophie	Clio	Renault	2026-05-20	2026-05-22	45	réservée	0492 44 55 66
L010	Dubois	Nora	Golf	VW	2026-05-21	2026-05-25	60	en cours	0478 88 77 66

Consigne

Analyse les données et détermine quels sont les différents groupes d'objets (**entités**) ainsi que leurs propriétés (**attributs**):

Exercice 2 — Centre de formation

Contexte

Un centre de formation propose des cours donnés par des instructeurs. Les élèves peuvent s'inscrire à différentes formations.

Toutes les informations sont stockées dans un seul tableau.

Données

id_inscription	nom_eleve	prenom_eleve	formation	duree	instructeur	date_debut	prix	salle	statut
I001	Dupont	Marie	Python	5 jours	Durand	2026-06-01	300	A101	confirmée
I002	Martin	Lucas	Web	3 jours	Lefevre	2026-06-02	200	B202	confirmée
I003	Lambert	Sophie	Python	5 jours	Durand	2026-06-01	300	A101	annulée
I004	Dubois	Nora	Réseau	4 jours	Martin	2026-06-03	250	C303	confirmée
I005	Dupont	Marie	Web	3 jours	Lefevre	2026-06-02	200	B202	confirmée
I006	Peeters	Hugo	Réseau	4 jours	Martin	2026-06-03	250	C303	confirmée
I007	Martin	Lucas	Python	5 jours	Durand	2026-06-01	300	A101	confirmée
I008	Lambert	Sophie	Web	3 jours	Lefevre	2026-06-02	200	B202	confirmée
I009	Dubois	Nora	Python	5 jours	Durand	2026-06-01	300	A101	confirmée
I010	Peeters	Hugo	Web	3 jours	Lefevre	2026-06-02	200	B202	annulée

Consigne

Analyse les données et détermine quels sont les différents groupes d'objets (**entités**) ainsi que leurs propriétés (**attributs**):

Exercice 3 — Serveur Discord

Contexte

Un serveur Discord contient plusieurs salons. Les utilisateurs peuvent envoyer des messages dans ces salons.

Toutes les données sont stockées dans un seul tableau.

Données

id_message	utilisateur	email	salon	date_message	contenu	role	serveur	statut
M001	Marie	marie@mail.be	général	2026-04-01	Bonjour !	membre	DevZone	envoyé
M002	Lucas	lucas@mail.be	gaming	2026-04-01	Qui joue ce soir ?	membre	DevZone	envoyé
M003	Sophie	sophie@mail.be	général	2026-04-02	Salut !	admin	DevZone	envoyé
M004	Nora	nora@mail.be	musique	2026-04-02	Nouvelle playlist	membre	DevZone	envoyé
M005	Hugo	hugo@mail.be	gaming	2026-04-03	Let's go	membre	DevZone	supprimé
M006	Marie	marie@mail.be	musique	2026-04-03	Trop bien	membre	DevZone	envoyé
M007	Lucas	lucas@mail.be	général	2026-04-04	Update serveur	admin	DevZone	envoyé
M008	Sophie	sophie@mail.be	gaming	2026-04-04	GG	admin	DevZone	envoyé
M009	Nora	nora@mail.be	général	2026-04-05	Hello	membre	DevZone	envoyé
M010	Hugo	hugo@mail.be	musique	2026-04-05		membre	DevZone	envoyé

Consigne

Analyse les données et détermine quels sont les différents groupes d'objets (**entités**) ainsi que leurs propriétés (**attributs**):

Le serveur est-il une entité à part entière ?

Objectif pédagogique global

Ces exercices doivent amener les élèves à comprendre que :

un tableau mélange plusieurs “types d’objets” chaque objet doit être séparé c’est la base de la modélisation (Merise)

Les relations

Dans la séquence précédente, tu as appris à identifier les entités d'un système et à les représenter sous forme de boîtes. Tu as aussi tracé des traits pour indiquer quelles entités sont liées. Mais ces traits ne disaient pas grand-chose : ils montraient qu'un lien existe, sans préciser sa nature ni son étendue. C'est exactement ce que cette séquence va creuser. On va donner un nom à ces liens, les représenter correctement, et surtout répondre à une question essentielle : dans ce lien, combien d'éléments peuvent être concernés de chaque côté ?

4TTR niveau

Les relations

Objectifs

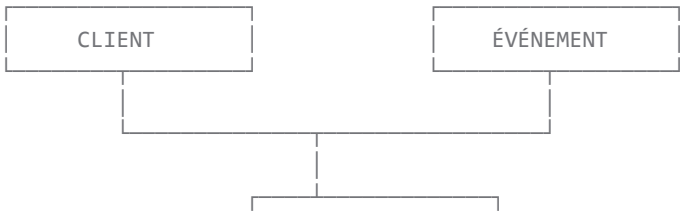
À la fin de cette séquence, tu seras capable de :

1. Définir ce qu'est une relation dans un modèle de données
2. Représenter une relation selon la notation Mérise
3. Lire et écrire des cardinalités
4. Construire un schéma entité-association complet

Partie 1 — Qu'est-ce qu'une relation ?

De l'observation à la formalisation

Dans la séquence 2, on avait ce schéma :



Les traits disaient juste : *ces entités ont un lien*. Mais quel lien, exactement ? Que se passe-t-il concrètement entre un client et un ticket ? Entre un événement et un ticket ?

La réponse, c'est la **relation** — ou plus précisément, **l'association** en Mérise.

Une relation, c'est le verbe qui unit deux entités. Elle décrit ce qui se passe entre elles dans la réalité du système :

- Un client **achète** un ticket.
- Un ticket **concerne** un événement.

Ce n'est pas anodin : nommer la relation oblige à comprendre ce que le système fait vraiment.

Partie 2 — La notation Mérise

Pourquoi Mérise ?

Mérise est une méthode de modélisation de bases de données développée en France dans les années 80. Elle est encore très utilisée dans l'enseignement et dans de nombreuses entreprises. Elle propose une façon standardisée de dessiner des modèles de données, avec des symboles précis.

Le schéma qu'on produit en Mérise s'appelle le **MCD** : Modèle Conceptuel de Données.

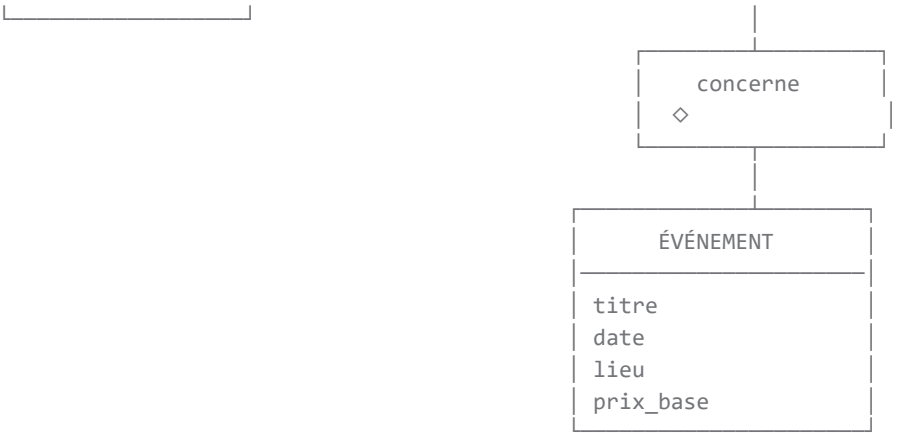
Les symboles

En Mérise, trois éléments ont chacun leur forme :

Élément	Forme	Rôle
Entité	Rectangle	Un objet du système
Relation	Losange (ou hexagone)	Le lien entre deux entités
Attribut	Ellipse ou texte listé	Une information d'une entité

Le schéma avec les relations nommées

En appliquant ces conventions à notre système de billetterie, on obtient :



Le losange `◇` représente la relation. Il porte un verbe qui décrit le lien. C'est ce verbe qui donne du sens au trait.

> Choisir un bon verbe n'est pas un détail : "achète", "réserve", "possède" ne décrivent pas la même chose. Avant de nommer une relation, demande-toi ce qui se passe réellement dans le système.

Partie 3 — Les cardinalités

Le problème du "combien"

On sait maintenant qu'un client *achète* un ticket, et qu'un ticket *concerne* un événement. Mais ces liens ne disent encore pas tout. Dans la réalité du système :

- Un client peut-il acheter plusieurs tickets, ou un seul ?
- Un ticket peut-il être acheté par plusieurs clients, ou un seul ?
- Un événement peut-il avoir plusieurs tickets, ou un seul ?

Les réponses à ces questions s'appellent les **cardinalités**. Une cardinalité indique, pour chaque entité participant à une relation, combien d'occurrences peuvent être impliquées.

La notation

On écrit la cardinalité sous la forme `minimum, maximum` de chaque côté du losange.

Les valeurs utilisées sont :

- `0` : aucune occurrence n'est obligatoire
- `1` : exactement une occurrence
- `n` : plusieurs occurrences possibles (sans limite fixée)

Les paires les plus courantes :

Notation	Signification
`1,1`	Exactement un — ni plus, ni moins
`0,1`	Zéro ou un — facultatif mais pas multiple
`1,n`	Au moins un, potentiellement plusieurs
`0,n`	Zéro ou plusieurs — aucune contrainte

Comment lire une cardinalité

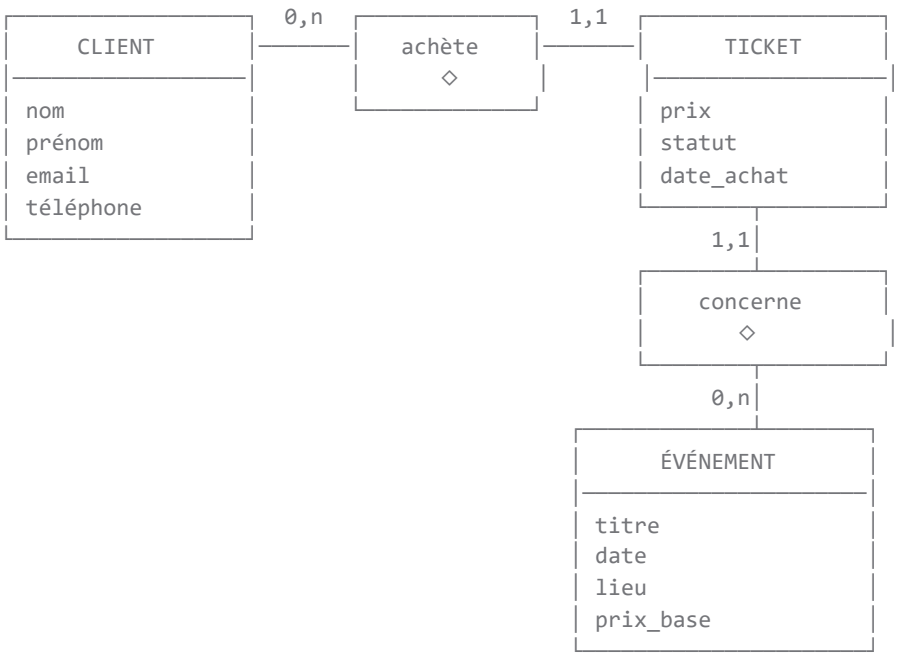
La cardinalité se lit **du côté de l'entité vers la relation**, en se posant la question :

> *Pour une occurrence de cette entité, combien d'occurrences de la relation peut-elle avoir ?*

Exemple avec **CLIENT — achète — TICKET** :

- Du côté de `CLIENT` : un client peut acheter *zéro ou plusieurs* tickets → `0,n`
- Du côté de `TICKET` : un ticket est acheté par *exactement un* client → `1,1`

Le schéma avec les cardinalités



Exercice 1 — Lire les cardinalités

En observant le schéma ci-dessus, réponds aux questions suivantes :

1. Un client peut-il n'avoir acheté aucun ticket ? Justifie.
2. Un ticket peut-il avoir été acheté par deux clients différents ? Justifie.
3. Un événement peut-il n'avoir aucun ticket associé ? Justifie.

4. Un ticket peut-il concerner deux événements différents ? Justifie.

Partie 4 — Construire un MCD complet

Démarche

Pour construire un MCD, on suit toujours le même ordre :

1. **Identifier les entités** — quels sont les objets du système ?
2. **Lister les attributs** de chaque entité
3. **Repérer les relations** — quels verbes lient les entités ?
4. **Fixer les cardinalités** — en se posant la question "combien ?" pour chaque côté

Exercice 2 — Analyser avant de dessiner

Avant de dessiner quoi que ce soit, réponds d'abord à ces questions pour le système de billetterie :

Entités :

- Quels sont les trois objets principaux ? (tu les connais déjà)

Relations :

- Quel verbe décrit le lien entre `CLIENT` et `TICKET` ?
- Quel verbe décrit le lien entre `TICKET` et `ÉVÉNEMENT` ?

Cardinalités :

Complète le tableau :

Relation	Côté gauche	Cardinalité gauche	Cardinalité droite	Côté droit
achète	CLIENT	…	…	TICKET
concerne	TICKET	…	…	ÉVÉNEMENT

Exercice 3 — Dessiner le MCD

À partir de tes réponses à l'exercice 2, dessine le MCD complet sur papier.

Tu dois faire apparaître :

- Les trois entités avec leurs attributs
- Les deux relations avec leur nom dans un losange
- Les cardinalités de chaque côté de chaque relation

Partie 5 — Application

Reprends le système que tu avais choisi en séquence 2 (garage, centre de formation ou serveur Discord) et construis son MCD complet.

Étapes :

1. Reprends tes trois entités et leurs attributs
2. Nomme les relations qui les lient (un verbe pour chaque)
3. Détermine les cardinalités en te posant la question "combien ?" pour chaque côté
4. Dessine le MCD final

> Avant de dessiner, remplis toujours le tableau d'analyse (comme dans l'exercice 2). Un MCD dessiné sans réflexion préalable contient presque toujours des erreurs de cardinalité.

À retenir

Une **relation** est le lien nommé entre deux entités. En Mérise, elle est représentée par un **losange** portant un verbe.

Les **cardinalités** expriment le "combien" de chaque côté d'une relation. On les note sous la forme `minimum,maximum`.

Paire	Sens
`1,1`	Exactement un
`0,1`	Zéro ou un
`1,n`	Au moins un
`0,n`	Zéro ou plusieurs

La règle pour lire une cardinalité : on se place du côté de l'entité, et on se demande combien d'occurrences de la relation cette entité peut avoir.

Suite

Dans la prochaine séquence, tu apprendras à transformer ce MCD en **modèle logique** : les entités deviennent des tables, les relations se traduisent en colonnes spéciales appelées **clés étrangères**, et les cardinalités dictent

comment ces colonnes s'organisent.

© 2025 Mr. Lambrechts
https://v2.ttrinfo.be
7 | 7

Identifier ses données

Clé primaire, identifiant naturel, artificiel, AUTO_INCREMENT, UUID : comprendre pourquoi chaque ligne d'une table doit être unique et comment y parvenir.

4TTR niveau

Les identifiants — donner un nom unique à chaque enregistrement

Dans le cours précédent, on a vu comment les relations permettent de lier des tables entre elles. Mais pour créer un lien entre deux tables, il faut d'abord pouvoir **désigner** un enregistrement précis sans ambiguïté.

C'est le rôle de l'**identifiant** — ou **clé primaire**.

Le problème : deux lignes identiques

Imagine une table `eleves` :

nom	prenom	classe
Dupont	Lucas	4TTR
Dupont	Lucas	4TTR
Martin	Emma	4TTR

Il y a deux Lucas Dupont dans la même classe. Comment savoir de lequel on parle quand on veut lui attribuer une note ? Quand on veut l'inscrire à un cours ? Quand on veut le supprimer ?

On ne peut pas. Et c'est exactement le genre de chaos qu'une base de données bien construite doit éviter.

> **RÈGLE FONDAMENTALE :** dans une table relationnelle, chaque ligne doit être identifiable de façon **unique et permanente**.

Les deux types d'identifiants

Il existe deux grandes familles d'identifiants.

1. L'identifiant naturel

Un **identifiant naturel** est une donnée qui existe *déjà dans la réalité* et qui est naturellement unique.

Exemples :

Contexte	Identifiant naturel
Élève	Numéro de registre national
Voiture	Numéro de châssis (VIN)
Livre	ISBN
Pays	Code ISO (BE, FR, US…)
Produit	Code-barres EAN-13

Avantage : il a une signification dans le monde réel. On peut lire `BE` et comprendre immédiatement que c'est la Belgique.

Inconvénients :

- On dépend d'une source extérieure pour garantir l'unicité (que se passe-t-il si l'ISBN est mal encodé ?)
- Il peut changer (un numéro de téléphone, un matricule interne…)
- Il n'existe pas toujours (comment identifier un film maison sans système externe ?)
- Il peut être trop long ou trop complexe pour servir de clé dans des jointures

> En pratique, les identifiants naturels sont risqués. Un email semble unique — mais que se passe-t-il si quelqu'un change d'adresse ? Toutes les références dans les autres tables deviennent fausses.

---

2. L'identifiant artificiel

Un **identifiant artificiel** est un identifiant qu'on **invente** spécifiquement pour la base de données. Il n'a aucun sens dans la réalité — c'est juste un numéro (ou un code) dont l'unique rôle est de désigner une ligne.

C'est la solution **recommandée** dans la grande majorité des cas.

AUTO_INCREMENT — l'identifiant entier automatique

La forme la plus courante d'identifiant artificiel en MySQL est un entier auto-incrémenté.

```
CREATE TABLE eleves (
    id     INT  NOT NULL AUTO_INCREMENT,
    nom    VARCHAR(100) NOT NULL,
    prenom VARCHAR(100) NOT NULL,
```

```
    PRIMARY KEY (id)
);
```

À chaque `INSERT`, MySQL attribue automatiquement le prochain numéro disponible :

```
INSERT INTO eleves (nom, prenom) VALUES ('Dupont', 'Lucas');
-- id = 1
INSERT INTO eleves (nom, prenom) VALUES ('Dupont', 'Lucas');
-- id = 2  ← même nom, même prénom, mais identifiant différent !
INSERT INTO eleves (nom, prenom) VALUES ('Martin', 'Emma');
-- id = 3
```

Résultat :

id	nom	prenom
1	Dupont	Lucas
2	Dupont	Lucas
3	Martin	Emma

Les deux Lucas sont maintenant **distinguables**. On peut parler de "l'élève n°1" et "l'élève n°2" sans ambiguïté.

Ce qu'il faut savoir sur AUTO_INCREMENT

- Le compteur **ne se réinitialise pas** si on supprime des lignes. Si on supprime l'élève n°3, le prochain élève inséré aura l'id `4`, pas `3`.
- C'est **voulu** : un identifiant ne doit jamais être réutilisé. Une fois qu'une ligne a existé avec l'id `3`, cet id ne doit plus jamais désigner autre chose — même si la ligne a été supprimée.
- La colonne doit être déclarée `NOT NULL` et être la `PRIMARY KEY`.

La clé primaire (`PRIMARY KEY`)

La **clé primaire** est la colonne (ou le groupe de colonnes) qui identifie de façon unique chaque ligne d'une table.

En SQL :

```
PRIMARY KEY (id)
```

MySQL refuse automatiquement d'insérer deux lignes avec la même valeur de clé primaire, et refuse les valeurs `NULL`.

Règles d'une bonne clé primaire

1. **Unique** — deux lignes ne peuvent jamais avoir la même valeur
2. **Non nulle** — elle doit toujours avoir une valeur
3. **Stable** — elle ne doit jamais changer après création
4. **Simple** — idéalement un seul champ (pas une combinaison de colonnes)

UUID — l'identifiant universel

`AUTO_INCREMENT` fonctionne très bien dans une seule base de données. Mais imaginons un scénario différent : on a **deux serveurs** qui génèrent chacun des enregistrements, et on veut les fusionner. Le serveur A a des élèves avec les ids 1, 2, 3 et le serveur B aussi. Impossible de les fusionner sans collision !

C'est là qu'intervient l'**UUID** (*Universally Unique Identifier* — Identifiant Universel Unique).

Un UUID ressemble à ça :

```
550e8400-e29b-41d4-a716-446655440000
```

C'est une chaîne de 32 caractères hexadécimaux (plus des tirets) générée de façon **aléatoire**. La probabilité que deux UUID générés indépendamment soient identiques est astronomiquement faible — on parle d'environ 1 chance sur 5,3 × 10^{36}.

UUID en MySQL

```sql
CREATE TABLE commandes (
    id    CHAR(36)     NOT NULL DEFAULT (UUID()),
    total DECIMAL(8,2) NOT NULL,
    PRIMARY KEY (id)
);
```

Ou en générant l'UUID côté application (PHP, Node.js…) avant l'insertion :

```sql
INSERT INTO commandes (id, total)
VALUES ('550e8400-e29b-41d4-a716-446655440000', 49.99);
```

Avantages et inconvénients

	UUID	AUTO_INCREMENT
Unicité garantie	Partout dans le monde	Dans une seule base
Lisibilité	Illisible (`550e8400…`)	Lisible (`42`)
Performance	Plus lent (index sur 36 chars)	Très rapide (index sur entier)
Cas d'usage	APIs, apps distribuées, microservices	Applications locales, scolaires, simples

> Pour vos projets scolaires et la grande majorité des applications web classiques, `INT AUTO_INCREMENT` est le bon choix. L'UUID devient pertinent quand on expose des ids dans des URLs publiques ou quand on synchronise plusieurs bases.

Clé primaire composite — quand utiliser plusieurs colonnes ?

Il est parfois possible de définir une clé primaire sur **deux colonnes combinées**. On parle de **clé primaire composite** (ou clé naturelle composée).

Exemple : une table `inscriptions` qui enregistre qu'un élève est inscrit à un cours. Un élève ne peut pas être inscrit deux fois au même cours :

```sql
CREATE TABLE inscriptions (
    eleve_id INT NOT NULL,
    cours_id INT NOT NULL,
    date_inscription DATE NOT NULL,
    PRIMARY KEY (eleve_id, cours_id)
);
```

La combinaison `(eleve_id, cours_id)` est unique — même si `eleve_id = 1` apparaît plusieurs fois (l'élève 1 est inscrit à plusieurs cours) et `cours_id = 5` apparaît plusieurs fois (plusieurs élèves dans le cours 5).

Quand l'utiliser ?

Les clés composites sont appropriées pour les **tables de liaison** (aussi appelées tables d'association ou tables pivot), qui matérialisent une relation **plusieurs-à-plusieurs**. Dans ce contexte, elles sont naturelles et recommandées.

Pour les tables "normales" (entités), préférer un `id INT AUTO_INCREMENT`.

Tableau récapitulatif

Type d'identifiant	Exemple	Avantage	Inconvénient
Naturel simple	ISBN, code pays	Lisible, déjà existant	Peut changer, dépend de l'extérieur
Artificiel entier	`INT AUTO_INCREMENT`	Simple, rapide, automatique	Pas portable entre serveurs
UUID	`550e8400-…`	Unique partout dans le monde	Lourd, peu lisible
Composite	`(eleve_id, cours_id)`	Adapté aux tables de liaison	Complexe à référencer

---

À retenir

- Chaque table doit avoir une **clé primaire** — une colonne dont la valeur est unique pour chaque ligne.
- Dans la plupart des cas, on utilise `id INT NOT NULL AUTO_INCREMENT` avec `PRIMARY KEY (id)`.
- Un identifiant **ne doit jamais changer** après sa création — c'est pour ça qu'on n'utilise pas le nom ou l'email comme clé primaire.
- La clé primaire d'une table devient la **clé étrangère** dans les tables qui lui sont liées (vu dans le cours sur les relations).
- L'UUID existe pour les cas où plusieurs systèmes indépendants doivent générer des ids sans se concerter.

Exercice

Voici une table `produits` mal conçue :

```sql
CREATE TABLE produits (
    nom         VARCHAR(100) NOT NULL,
    prix        DECIMAL(6,2) NOT NULL,
    categorie   VARCHAR(50)  NOT NULL
);
```

Questions :

1. Quel problème pose cette table si on insère deux fois le produit `'Stylo bille'` à `0.99 €` ?
2. Modifie la requête `CREATE TABLE` pour ajouter un identifiant artificiel correct.
3. On veut ajouter une table `commandes_produits` pour enregistrer quels produits sont dans quelle commande. Écris la `CREATE TABLE` avec la clé primaire appropriée.
4. Pourquoi serait-ce une mauvaise idée d'utiliser le champ `nom` comme clé primaire ?

Un SGBDR, c'est quoi ?

Avant d'ouvrir un outil et de créer des tables, on pose le vocabulaire : qu'est-ce qu'une base de données relationnelle, qui s'en occupe, et comment on s'y connecte ?

4TTR niveau

Jusqu'ici, tu as travaillé sur du papier : entités, relations, cardinalités, identifiants. C'est le **modèle**. La prochaine étape est de transformer ce modèle en une vraie base de données qui tourne sur ton ordinateur.

Avant de toucher à un outil, il faut comprendre **ce qui s'exécute** et **avec quoi on lui parle**.

Du schéma aux tables

Dans une **base de données relationnelle**, les informations sont stockées dans des **tables**. Une table, c'est essentiellement un tableau avec :

- des **colonnes** — chacune correspond à un attribut (nom, prix, date…)
- des **lignes** — chacune représente une occurrence (un client précis, un ticket précis…)

La traduction depuis le MCD est directe :

MCD	Base de données
Entité (`CLIENT` , `TICKET` …)	Table
Attribut (`nom` , `prix` …)	Colonne
Une occurrence	Une ligne
Identifiant	Clé primaire (la colonne `id`)

Le **R** de SGBDR — *relationnel* — vient du fait que ces tables sont **reliées entre elles** par des colonnes spéciales appelées **clés étrangères**. Tu verras la mécanique exacte dans le prochain cours.

> Une base de données relationnelle, c'est donc un ensemble de tables qui se référencent mutuellement de façon contrôlée.

Le logiciel qui gère tout ça : le SGBDR

Les tables ne flottent pas dans le vide. Elles sont gérées par un programme : un **SGBDR** — *Système de Gestion de Base de Données Relationnelle* (GB ***RDBMS, Relational Database Management System***).

Son boulot :

- **stocker** les tables sur le disque
- **garantir** que les contraintes sont respectées (pas de doublons sur une clé primaire, pas de clé étrangère qui pointe vers le vide…)
- **exécuter** les requêtes qu'on lui envoie pour lire, ajouter, modifier ou supprimer des données
- **gérer** les accès concurrents (plusieurs personnes ou programmes en même temps)

Les principaux SGBDR

SGBDR	Particularité
MySQL	Le plus répandu sur le web, racheté par Oracle
MariaDB	Fork libre de MySQL, presque 100 % compatible — c'est ce que Laragon installe par défaut
PostgreSQL	Très complet, réputé pour sa rigueur
SQLite	Pas un serveur : un simple fichier `.db`, idéal pour les petites apps
SQL Server	Solution Microsoft, courante en entreprise
Oracle Database	Historique, très présent en grande entreprise

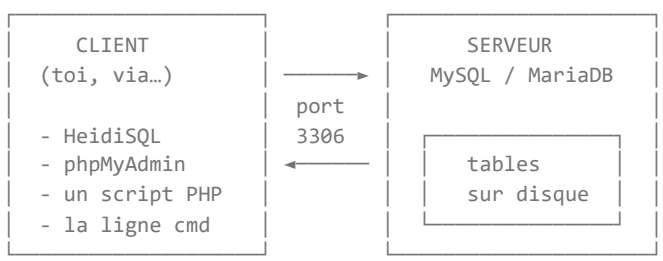
Dans ce cours, on utilise **MySQL / MariaDB** : c'est le SGBDR de loin le plus courant chez les hébergeurs web, et c'est celui que Laragon (ou XAMPP) a installé sur ta machine.

Le modèle client / serveur

Voici le point qui surprend souvent au début.

Quand tu démarres Laragon, un programme se lance en arrière-plan : c'est le **serveur MySQL** (`mysqld`, le *d* pour *daemon*). Il tourne en permanence, écoute sur un **port réseau** (par défaut le `3306`) et attend qu'on lui parle.

Toi, tu ne lui parles **jamais directement**. Tu utilises un **client** — un autre programme qui se connecte au serveur et lui transmet tes ordres.



Conséquences pratiques :

- Si le service MySQL de Laragon n'est pas démarré, **aucun** client ne pourra se connecter — même si HeidiSQL est ouvert.

- Plusieurs clients peuvent être connectés en même temps au même serveur, sur la même base.
- Le client peut être sur la même machine que le serveur (chez toi pour l'apprentissage) ou à l'autre bout du monde (sur un hébergement web).

Les clients qu'on va utiliser

HeidiSQL — le client desktop

Application Windows gratuite, interface graphique riche. On peut créer des bases, des tables, taper des requêtes SQL, voir et éditer les données comme dans un tableur. C'est l'outil de tous les jours sur ta machine.

phpMyAdmin — le client web

Une application web (écrite en PHP) qui tourne sur le serveur web et expose une interface dans le navigateur. Quasiment **tous les hébergements mutualisés** la proposent — c'est souvent le seul moyen d'administrer une base distante sans installer de logiciel.

> **À RETENIR** : HeidiSQL et phpMyAdmin sont **deux clients différents qui parlent au même serveur MySQL**. Tout ce que tu crées avec l'un est visible avec l'autre — c'est la même base de données.

Les autres clients (pour info)

- **La ligne de commande** `mysql` — installée avec Laragon, utile pour les scripts
- **Un programme** — n'importe quel langage (PHP, Python, Node.js, Java…) peut se connecter à MySQL via une bibliothèque dédiée. C'est ce qu'on fera plus tard quand on connectera un site web à sa base.

La langue commune : SQL

Quel que soit le client, on parle au serveur dans le même langage : **SQL** (*Structured Query Language*).

Exemples — pas besoin de comprendre la syntaxe maintenant, c'est juste pour voir à quoi ça ressemble :

```sql
-- Créer une table
CREATE TABLE client (
    id      INT NOT NULL AUTO_INCREMENT,
    nom     VARCHAR(100),
    prenom  VARCHAR(100),
    PRIMARY KEY (id)
);

-- Ajouter une ligne
```

```sql
INSERT INTO client (nom, prenom) VALUES ('Dupont', 'Marie');

-- Lire les données
SELECT * FROM client WHERE nom = 'Dupont';
```

Les interfaces graphiques (HeidiSQL, phpMyAdmin) **génèrent du SQL pour toi** quand tu cliques. Tu apprendras à écrire ce SQL directement dans les cours qui suivront — c'est une compétence indispensable dès qu'on quitte les interfaces toutes faites.

À retenir

- Une **base de données relationnelle** est un ensemble de **tables** reliées par des **clés**.
- Le **SGBDR** est le logiciel qui stocke ces tables et qui exécute les requêtes — ici MySQL ou MariaDB.
- Le SGBDR fonctionne en mode **client / serveur** : un serveur tourne en permanence, des clients s'y connectent.
- **HeidiSQL** (desktop) et **phpMyAdmin** (web) sont deux clients différents pour le même serveur.
- Tous les clients parlent la même langue : **SQL**.

Vérifications

1. Sur ta machine, qu'est-ce qui est *serveur* et qu'est-ce qui est *client* ?
2. Si on ferme HeidiSQL, est-ce que la base de données est perdue ? Pourquoi ?
3. Pourquoi un hébergement web propose-t-il phpMyAdmin plutôt que HeidiSQL ?
4. Cite trois SGBDR et donne un cas d'usage typique pour chacun.

Suite

Maintenant que le décor est planté, on peut faire l'étape qui manque entre ton MCD papier et de vraies tables : traduire le schéma entité-association en **modèle relationnel**, avec ses clés primaires et ses clés étrangères.

Du MCD au modèle relationnel

Comment transformer un schéma entité-association en un ensemble de tables prêtes à être créées : règles de passage, clés primaires, clés étrangères.

4TTR niveau

Tu as un MCD complet : entités, attributs, relations nommées, cardinalités. Tu sais aussi que chaque ligne d'une table doit avoir un identifiant unique. Il manque une étape : **comment relier les tables entre elles** pour matérialiser les relations du MCD.

L'objectif de ce cours est d'apprendre les **règles de traduction** d'un MCD vers un **modèle logique de données** — le **MLD**. À la fin, tu auras une liste de tables avec toutes leurs colonnes, prêtes à être créées dans le SGBDR au prochain cours.

Objectifs

À la fin de cette séquence, tu seras capable de :

1. Transformer chaque entité en une table avec sa clé primaire
2. Traduire une relation `0,n / 1,1` en clé étrangère
3. Écrire un MLD complet en notation textuelle
4. Repérer où placer la clé étrangère selon les cardinalités

Partie 1 — La clé étrangère, le maillon manquant

Un exemple pour comprendre

Reprenons deux entités du MCD billetterie :

On a vu dans le cours sur les identifiants que chaque table aura un `id` :

```
client(id, nom, prenom)
ticket(id, prix, statut)
```

Mais avec juste ça, **rien ne relie un ticket à son client**. Si je regarde la ligne `ticket` n°42, je ne sais pas qui l'a acheté.

La solution : une colonne qui pointe

L'idée est simple : on ajoute dans la table `ticket` une colonne qui **contient l'id du client** propriétaire :

id	prix	statut	client_id
1	12	payé	**3**
2	12	payé	**3**
3	15	réservé	**7**

Tickets n°1 et n°2 appartiennent au client n°3. Ticket n°3 appartient au client n°7. La relation `achète` du MCD est maintenant matérialisée par cette colonne.

Cette colonne porte un nom précis : **clé étrangère** (GB ***foreign key, FK***). Elle est *étrangère* parce qu'elle référence la clé primaire d'une **autre** table.

> **RÈGLE D'OR** : une clé étrangère contient **toujours** une valeur qui existe dans la clé primaire de la table référencée. Le SGBDR refusera d'insérer un ticket avec `client_id = 999` si aucun client n°999 n'existe.

Partie 2 — Les règles de traduction

Règle 1 — Chaque entité devient une table

Pour chaque entité du MCD :

- créer une **table** du même nom (en minuscules, au singulier par convention)
- chaque attribut devient une **colonne**
- ajouter une colonne `id` comme **clé primaire** (`INT AUTO_INCREMENT`)

```
Entité ÉVÉNEMENT (titre, date, prix_base)
       │
       ▼
Table  evenement (id, titre, date, prix_base)
```

Règle 2 — Une relation `0,n / 1,1` devient une clé étrangère

C'est le cas le plus fréquent et celui qui te servira partout.

Principe : la clé étrangère est placée du côté `1,1`.

Pourquoi ? Parce que du côté `1,1`, chaque occurrence pointe vers **exactement un** élément de l'autre côté — une seule colonne suffit pour stocker cette référence.

```
CLIENT  ———0,n—— achète ——1,1——►  TICKET
                                  (↑ la FK va ici : client_id)
```

À l'inverse, du côté `0,n`, un client peut avoir plusieurs tickets — on ne pourrait pas stocker plusieurs ids dans une seule colonne.

Résultat sur le MCD billetterie :

```
client    (id, nom, prenom, email, telephone)
evenement (id, titre, date, prix_base)
ticket    (id, prix, statut, date_achat, #client_id, #evenement_id)
```

Le `#` signale une clé étrangère. La table `ticket` reçoit **deux** clés étrangères parce qu'elle participe à **deux** relations `0,n / 1,1` (avec `client` et avec `evenement`).

Règle 3 — Une relation `0,1 / 0,n` se traite pareil

Même logique : la FK va du côté du **maximum à 1**. La différence avec `1,1` : la colonne peut être vide (`NULL` autorisé).

```
LIEU   ———0,n—— accueille ——0,1——►  ÉVÉNEMENT
                                    (FK : lieu_id, NULL autorisé)
```

Un événement peut ne pas avoir de lieu encore défini → `lieu_id` peut être `NULL`.

Partie 3 — Notation textuelle du MLD

Sur papier (ou dans un cahier), on écrit un MLD avec les conventions suivantes :

- **Nom de la table** en minuscules, suivi des colonnes entre parenthèses
- La **clé primaire** est **soulignée**
- Les **clés étrangères** sont préfixées d'un `#`
- Une flèche `→` (ou une note) précise vers quelle table pointe la FK

Exemple complet — la billetterie

```
client    (id, nom, prenom, email, telephone)
           —
```

```
evenement (id, titre, date, prix_base)
            _

ticket    (id, prix, statut, date_achat, #client_id, #evenement_id)
            _                             └──→ client.id
                                                    └──→ evenement.id
```

Cette écriture est **suffisante** pour créer ensuite les tables dans le SGBDR. Toutes les informations nécessaires sont là.

Partie 4 — Quel type pour chaque colonne ?

Au moment de créer la table, le SGBDR exigera un **type** pour chaque colonne. Voici les plus utiles à ce stade :

Type	Pour stocker	Exemple
`INT`	un entier	un id, un prix en centimes, un nombre
`VARCHAR(n)`	du texte court (max `n` caractères)	un nom, un email, un titre
`TEXT`	du texte long sans limite pratique	une description, un commentaire
`DATE`	une date `AAAA-MM-JJ`	une date d'événement
`DATETIME`	une date + heure	une date d'achat précise
`DECIMAL(p,d)`	un nombre à virgule fixe	un prix : `DECIMAL(6,2)` = jusqu'à 9999,99
`BOOLEAN`	vrai / faux	un statut binaire

> **ÉVITE `FLOAT` POUR DE L'ARGENT.** `DECIMAL` garantit la précision exacte ; `FLOAT` arrondit subtilement et provoque des erreurs de centimes.

MLD enrichi avec les types

```
client (
    id          INT  PK  AUTO_INCREMENT
    nom         VARCHAR(100)  NOT NULL
    prenom      VARCHAR(100)  NOT NULL
    email       VARCHAR(150)
    telephone   VARCHAR(20)
)

evenement (
    id          INT  PK  AUTO_INCREMENT
    titre       VARCHAR(200)  NOT NULL
    date        DATETIME      NOT NULL
    prix_base   DECIMAL(6,2)  NOT NULL
)
```

```
ticket (
    id           INT  PK  AUTO_INCREMENT
    prix         DECIMAL(6,2)  NOT NULL
    statut       VARCHAR(20)   NOT NULL
    date_achat   DATETIME      NOT NULL
    client_id    INT  FK → client.id        NOT NULL
    evenement_id INT  FK → evenement.id     NOT NULL
)
```

C'est ce que tu vas littéralement encoder dans HeidiSQL au prochain cours.

Partie 5 — Cas particulier : la relation `0,1 / 0,1`

Plus rare, mais ça arrive : deux entités liées au maximum une seule fois de chaque côté.

Exemple : un `employe` est rattaché à au plus un `casier`, et un `casier` est attribué à au plus un employé.

```
EMPLOYE  ———0,1—— possède ——0,1——►  CASIER
```

Règle : on peut placer la FK **d'un côté ou de l'autre** — peu importe techniquement. On choisit :

- selon ce qui est le plus naturel (l'employé existe d'abord, le casier est attribué après → FK côté casier)
- en ajoutant une contrainte `UNIQUE` sur la FK pour interdire qu'elle se répète (sinon on retomberait sur du `0,n / 0,1`)

```
employe (id, nom, prenom)
casier  (id, numero, #employe_id UNIQUE)
```

> Ce cas est **RARE EN PRATIQUE**. On préfère souvent regrouper les deux entités dans une seule table quand la relation est aussi exclusive. Mais c'est utile à connaître.

Partie 6 — La méthode, étape par étape

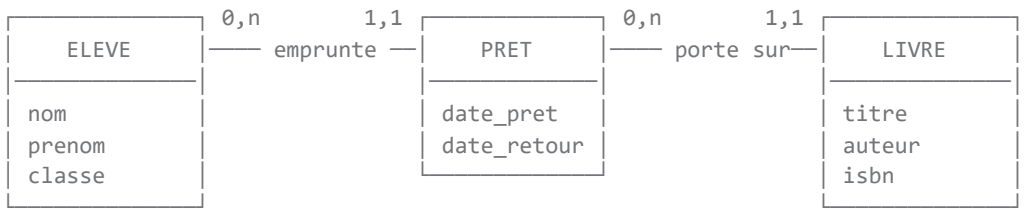
Quand tu reçois un MCD, voici la procédure à suivre :

1. **Une table par entité**, avec un `id` comme clé primaire
2. **Recopier les attributs** en choisissant un type pour chacun
3. **Pour chaque relation** `0,n / 1,1` : ajouter une FK dans la table du côté `1,1`
4. **Pour chaque relation** `0,n / 0,1` : ajouter une FK dans la table du côté `0,1`, autorisée à `NULL`
5. **Vérifier** : chaque relation du MCD doit se retrouver sous forme de FK quelque part

Si tu termines avec autant de relations que de FK ajoutées, ton MLD est cohérent.

Exercice 1 — Bibliothèque scolaire

Voici un MCD simplifié :



Questions :

1. Combien de tables ce MLD aura-t-il ?
2. Pour chacune, donne sa clé primaire.
3. Où va la clé étrangère `eleve_id` ? La clé étrangère `livre_id` ?
4. Écris le MLD complet en notation textuelle (avec les types).

Exercice 2 — Refaire la billetterie sans regarder

Sans consulter la Partie 3, écris le MLD complet du système de billetterie en notation textuelle. Compare ensuite avec la correction.

Exercice 3 — Ton projet

Reprends le MCD que tu as construit en séquence 4 (garage, formation, Discord, ou autre). Applique la procédure de la Partie 6 et écris le MLD complet — types compris.

> Vérification finale : chaque relation du MCD doit avoir donné lieu à **EXACTEMENT UNE** clé étrangère dans ton MLD. Ni plus, ni moins.

À retenir

- Une **entité** devient une **table** avec un `id` comme clé primaire.
- Une **relation** se matérialise par une **clé étrangère** : une colonne qui contient l'id d'une ligne d'une autre table.
- Pour une relation `0,n / 1,1` (ou `0,n / 0,1`) : la **FK va du côté** `1`.
- Le SGBDR garantit que chaque FK pointe vers une ligne existante — c'est l'**intégrité référentielle**.
- Un MLD bien écrit (avec PK, FK et types) contient tout ce qu'il faut pour créer les tables.

Suite

Tu as maintenant un MLD prêt à l'emploi. Dans le prochain cours, on ouvre HeidiSQL, on lance MySQL via Laragon, et on crée concrètement la première base et ses premières tables. Les clés étrangères, elles, auront leur cours dédié juste après.

Créer sa base et ses tables avec HeidiSQL

Premier contact avec un vrai SGBDR : on lance MySQL via Laragon, on se connecte avec HeidiSQL, on crée la base de données 'billetterie' et ses deux premières tables. Les clés étrangères suivent au cours suivant.

4TTR niveau

Tu as un MLD complet sur papier : trois tables, leurs colonnes, leurs types. Il est temps de le transposer dans une vraie base MySQL. Dans ce cours, on couvre la **base de données** et **deux tables sans relation entre elles** (`client` et `evenement`). Les clés étrangères (`ticket`) ont leur cours dédié juste après — c'est trop important pour le bâcler ici.

Objectifs

À la fin de cette séquence, tu seras capable de :

1. Démarrer le service MySQL depuis Laragon
2. Te connecter à ton serveur MySQL avec HeidiSQL
3. Créer une base de données
4. Créer une table avec colonnes, types et clé primaire
5. Insérer des lignes via l'interface
6. Lire le SQL généré par HeidiSQL

Partie 1 — Démarrer MySQL

Vérifier que Laragon/Xampp tourne

Lance **Laragon**. Tu vois une petite fenêtre avec des boutons : **Démarrer tout**, **Arrêter tout**, **Menu**…

Clique sur **Démarrer tout** (ou **Start All**). Deux services s'allument :

- **Apache** (le serveur web — utile plus tard pour phpMyAdmin)
- **MySQL** (le SGBDR — c'est lui qu'on attaque ici)

> Si tu vois deux pastilles vertes à côté des noms de services, c'est bon : le SERVEUR MYSQL

tourne et écoute sur le port `3306`. Rappelle-toi du cours 06 : sans ça, aucun client ne peut se connecter.

Vérification rapide

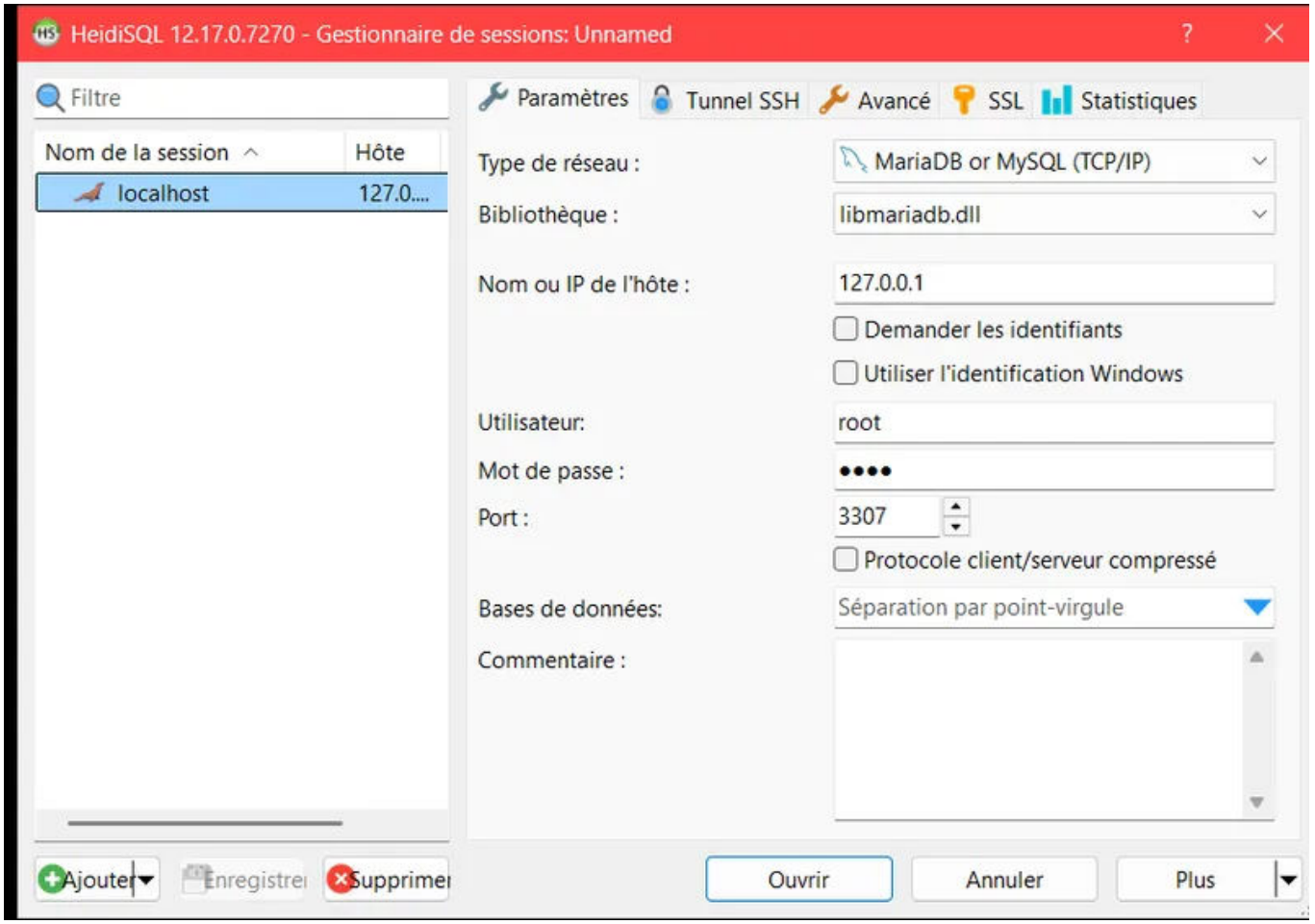
Dans le menu de Laragon, tu peux aller dans **MySQL → Version** : tu verras la version exacte de MariaDB ou MySQL installée. Pas critique, mais c'est un bon réflexe pour savoir avec quoi on travaille.

Partie 2 — Installer et lancer HeidiSQL

HeidiSQL est généralement **déjà installé avec Laragon**. Sinon, télécharge-le depuis heidisql.com (version *Installer*, gratuite).

Au premier lancement, tu arrives sur l'écran **Gestionnaire de sessions** : c'est là qu'on enregistre les connexions vers les serveurs MySQL.

Créer une session



Clique sur **Nouveau** en bas à gauche. Remplis :

Champ	Valeur	Pourquoi
Réseau	**MySQL (TCP/IP)**	On parle à MySQL en réseau, même si c'est local
Hôte / IP	`127.0.0.1` ou `localhost`	Le serveur tourne sur ta propre machine
Utilisateur	`root`	L'admin par défaut de MySQL
Mot de passe	*(vide)*	Laragon ne met pas de mot de passe par défaut
Port	`3306`	Le port standard de MySQL

Renomme la session en haut à gauche (par exemple `Laragon local`) puis clique **Ouvrir**.

> Le compte `root` sans mot de passe est acceptable **en local** pour apprendre. Sur un hébergement réel, c'est interdit — on aura un utilisateur dédié avec un mot de passe sérieux.

Tu es connecté(e)

À gauche : un arbre avec ton serveur et la liste des **bases existantes** (`information_schema`, `mysql`, `performance_schema`…). Ces bases sont **internes au SGBDR** — n'y touche pas. La tienne, on va la créer.

Partie 3 — Créer la base `billetterie`

Dans l'arbre de gauche, **clic droit sur le nom de la session** (la racine, pas une base existante) → **Créer nouveau** → **Base de données…**

Remplis :

Champ	Valeur
Nom	`billetterie`
Interclassement	`utf8mb4_unicode_ci`

Clique **OK**.

© 2025 Mr. Lambrechts https://v2.ttrinfo.be 4 | 10

L'INTERCLASSEMENT (*collation*) définit comment MySQL stocke et compare les caractères. `utf8mb4_unicode_ci` accepte tous les caractères Unicode (accents, emojis, alphabets non latins) et ignore la casse pour les comparaisons. C'est le bon choix par défaut en 2026 — c'est généralement l'option proposée d'office par HeidiSQL.

SI `UTF8MB4_UNICODE_CI` N'APPARAÎT PAS dans la liste (cela dépend de la version de MariaDB/MySQL), prends dans l'ordre :

- `utf8mb4_unicode_520_ci` — équivalent, basé sur Unicode 5.20
- `utf8mb4_uca1400_ai_ci` (ou `uca1400_ai_ci`) — sur les MariaDB récentes, basé sur Unicode 14
- `utf8mb4_general_ci` — fallback de dernier recours, plus rapide mais moins précis (suffisant à ce stade)

L'essentiel est que le nom commence par `utf8mb4_` — pas `utf8_` tout court, qui ne stocke ni les emojis ni certains caractères asiatiques.

La base `billetterie` apparaît dans l'arbre — vide. On va la peupler.

Partie 4 — Créer la table `client`

Rappel du MLD :

```
client (
    id          INT  PK  AUTO_INCREMENT
    nom         VARCHAR(100)  NOT NULL
    prenom      VARCHAR(100)  NOT NULL
    email       VARCHAR(150)
    telephone   VARCHAR(20)
)
```

Lancer le formulaire

Clic droit sur la base `billetterie` → **Créer nouveau** → **Table**.

Une fenêtre de conception s'ouvre. En haut, le champ **Nom** : tape `client`.

Ajouter les colonnes

Dans le tableau central, chaque ligne représente une **colonne**. Clique **Ajouter** pour insérer une nouvelle colonne, puis remplis les champs :

Colonne 1 — `id` *(la plus importante)*

Champ	Valeur
Nom	`id`
Type de données	`INT`
Autoriser NULL	**décoché**
Défaut	*(rien)*
Cocher la case	`AUTO_INCREMENT`

Puis, à droite du tableau, ajoute un **index** de type **PRIMARY** sur la colonne `id` (onglet *Index* ou bouton dédié — l'emplacement varie selon la version).

POURQUOI CES DEUX RÉGLAGES ?

- `AUTO_INCREMENT` dit à MySQL : *à chaque nouvelle ligne, attribue automatiquement le prochain entier disponible*. Tu n'as plus à inventer ou compter — la base s'occupe de générer un `id` unique pour chaque insertion. Sans ça, il faudrait fournir un `id` manuellement à chaque `INSERT`, avec le risque de doublons.
- **L'index** `PRIMARY` déclare que cette colonne est la **clé primaire** de la table. Trois conséquences techniques : MySQL refuse les doublons sur cette colonne, refuse les valeurs `NULL`, et crée un **index**

qui permet de retrouver n'importe quelle ligne par son `id` en une fraction de seconde, même avec des millions d'enregistrements.

En pratique, **les deux vont toujours ensemble** sur la colonne `id` d'une table : `AUTO_INCREMENT` garantit la génération automatique, `PRIMARY` garantit l'unicité et la rapidité d'accès. C'est ce qu'on a posé conceptuellement au cours 05 sur les identifiants — on le matérialise enfin ici.

Colonnes 2 à 5

Nom	Type	NULL
`nom`	`VARCHAR(100)`	non
`prenom`	`VARCHAR(100)`	non
`email`	`VARCHAR(150)`	oui
`telephone`	`VARCHAR(20)`	oui

Clique **Enregistrer**. La table `client` apparaît dans l'arbre.

Le SQL généré

En bas, HeidiSQL affiche **la requête SQL** qu'il vient d'envoyer au serveur :

```sql
CREATE TABLE `client` (
    `id`        INT          NOT NULL AUTO_INCREMENT,
    `nom`       VARCHAR(100) NOT NULL,
    `prenom`    VARCHAR(100) NOT NULL,
    `email`     VARCHAR(150) NULL,
    `telephone` VARCHAR(20)  NULL,
    PRIMARY KEY (`id`)
);
```

REGARDE-LA ATTENTIVEMENT. C'est exactement ce qu'on aurait tapé à la main dans un terminal `mysql`. HeidiSQL est juste un générateur de SQL — comprendre le SQL généré te rendra autonome.

Partie 5 — Créer la table `evenement`

Refais la même procédure pour la table `evenement` :

Nom	Type	NULL	Note
`id`	`INT`	non	PK, AUTO_INCREMENT
`titre`	`VARCHAR(200)`	non	
`date`	`DATETIME`	non	
`prix_base`	`DECIMAL(6,2)`	non	

Vérifie le SQL généré — il doit ressembler à :

```
CREATE TABLE `evenement` (
    `id`         INT           NOT NULL AUTO_INCREMENT,
    `titre`      VARCHAR(200)  NOT NULL,
    `date`       DATETIME      NOT NULL,
    `prix_base` DECIMAL(6,2)  NOT NULL,
    PRIMARY KEY (`id`)
);
```

Partie 6 — Insérer des données via l'interface

Sélectionne la table `client` dans l'arbre, puis va sur l'onglet **Données**. Tu vois un tableau vide.

Clique **Insérer une ligne** (en bas, ou bouton `+`). Remplis :

nom	prenom	email	telephone
Dupont	Marie	marie.dupont@email.be	0471 12 34 56

Ne remplis **pas** la colonne `id` — laisse `AUTO_INCREMENT` faire son travail. Valide (Enter ou bouton **Publier**).

Ajoute deux ou trois autres clients. Tu verras que les `id` se suivent : 1, 2, 3…

Fais pareil pour `evenement` :

titre	date	prix_base
Soirée Quiz Pop Culture	2026-10-03 20:00:00	12.00
Blind Test Années 90	2026-11-15 21:00:00	10.00
Quiz Cinéma	2026-12-05 20:30:00	15.00

> Le format `DATETIME` est `AAAA-MM-JJ HH:MM:SS`. HeidiSQL accepte aussi un petit calendrier graphique si tu cliques dans la cellule.

Partie 7 — Vérifier avec une requête SQL

Onglet **Requête** (en haut). Tape :

```sql
SELECT * FROM client;
```

Appuie sur **F9** ou clique **Exécuter**. Le résultat s'affiche en dessous : toutes les lignes de la table `client`.

```sql
SELECT id, titre, date FROM evenement WHERE prix_base < 15;
```

Tu apprendras `SELECT` en détail dans un cours dédié — c'est juste pour valider que tes données sont bien là.

Exercices

Exercice 1 — Créer une 3e table simple

Crée une table `salle` avec ces colonnes :

nom	type	NULL
`id`	INT	non (PK + AUTO_INCREMENT)
`nom`	VARCHAR(150)	non
`ville`	VARCHAR(100)	non
`capacite`	INT	oui

Insère 3 salles.

Exercice 2 — Lire le SQL

Va dans la table `client`, fais clic droit → **Exporter en SQL** → **Vers le presse-papiers** (ou équivalent). Colle le résultat dans un fichier texte. Tu obtiens un **dump SQL** : c'est ce qui te permettra de transférer ta base ailleurs (et c'est exactement ce que phpMyAdmin produira aussi).

Exercice 3 — Ton projet

Sur ta machine, crée la base de données de **ton projet personnel** (celui de la séquence 4). Crée toutes ses tables **sauf** celles qui ont des clés étrangères. Insère quelques lignes dans chacune.

> Si toutes tes tables ont des FK, garde la table principale (celle dont les autres dépendent) pour ce cours, et fais les autres au cours suivant.

Petits réflexes utiles dans HeidiSQL

- **F5** : rafraîchir l'arbre des bases (utile si tu as créé quelque chose sans que ça apparaisse)
- **F9** : exécuter la requête SQL en cours
- **Ctrl + clic** sur une cellule : édition rapide
- **Onglet *Vue d'ensemble*** d'une table : un résumé des colonnes, indexes et taille
- **Bouton *Annuler*** : tant que tu n'as pas **Publié**, tes modifications de données ne sont pas envoyées au serveur

---

À retenir

- Avant tout : **MySQL doit tourner** dans Laragon (pastille verte).
- HeidiSQL se connecte avec `localhost`, user `root`, pas de mot de passe (en local seulement).
- Une **base de données** est créée à part — elle contient les tables.
- Pour chaque table : nom, colonnes avec leur type, et **clé primaire** `id INT AUTO_INCREMENT`.
- L'interface graphique est pratique mais **regarde toujours le SQL généré** — c'est lui la vérité.
- L'insertion de données ne nécessite **jamais** de remplir l'`id` à la main.

Suite

Tes deux tables `client` et `evenement` existent, mais **rien ne les relie**. Au prochain cours, on crée la table `ticket` avec ses **clés étrangères** — c'est elle qui matérialisera les relations du MCD dans la base.

Les clés étrangères en pratique

Créer la table 'ticket' qui relie clients et événements via des clés étrangères. C'est le moment où le MCD prend vie dans la base : le SGBDR refusera désormais toute incohérence.

4TTR niveau

Au cours précédent, tu as créé `client` et `evenement`. Ce sont deux tables isolées : aucune ne sait qu'elle est liée à l'autre. C'est maintenant qu'on matérialise la relation `achète` / `concerne` du MCD avec une table `ticket` munie de **deux clés étrangères**.

Objectifs

À la fin de cette séquence, tu seras capable de :

1. Créer une table qui contient des clés étrangères
2. Déclarer une contrainte `FOREIGN KEY` dans HeidiSQL
3. Choisir un comportement `ON DELETE` / `ON UPDATE` adapté
4. Constater concrètement l'intégrité référentielle (le SGBDR refuse les incohérences)
5. Lire le `CREATE TABLE` complet généré avec ses contraintes

Partie 1 — Le rappel du MLD

```
client    (id, nom, prenom, email, telephone)
evenement (id, titre, date, prix_base)
ticket    (id, prix, statut, date_achat, #client_id, #evenement_id)
                                    ↳ client.id    ↳ evenement.id
```

La table `ticket` a deux clés étrangères, parce qu'elle participe à **deux relations** dans le MCD :

- `CLIENT 0,n — achète — 1,1 TICKET` → FK `client_id` côté ticket
- `TICKET 1,1 — concerne — 0,n ÉVÉNEMENT` → FK `evenement_id` côté ticket

C'est cohérent avec la règle du cours 07 : la FK va toujours côté `1,1`.

Partie 2 — Créer la table `ticket` (colonnes seulement)

Comme au cours précédent : clic droit sur la base `billetterie` → **Créer nouveau** → **Table**. Nom : `ticket`.

Les colonnes

Nom	Type	NULL	Note
`id`	`INT`	non	PK, AUTO_INCREMENT
`prix`	`DECIMAL(6,2)`	non	
`statut`	`VARCHAR(20)`	non	
`date_achat`	`DATETIME`	non	
`client_id`	`INT`	non	future FK
`evenement_id`	`INT`	non	future FK

Enregistre la table. Elle existe maintenant, mais `client_id` et `evenement_id` sont pour l'instant **de simples colonnes entières** : rien n'empêche d'y mettre `999` même si aucun client n°999 n'existe.

> Avant la prochaine étape, vérifie que les colonnes FK ont **EXACTEMENT LE MÊME TYPE** que la PK référencée. Toutes les deux doivent être `INT NOT NULL`. Si les types diffèrent, MySQL refusera de créer la contrainte.

Partie 3 — Ajouter les contraintes FOREIGN KEY

Ouvrir l'onglet des clés étrangères

Dans la fenêtre d'édition de la table `ticket`, va sur l'onglet **Clés étrangères** (ou **Foreign keys** selon ta langue). Tableau vide. Clique **Ajouter**.

Configurer la première FK : `client_id → client.id`

Champ	Valeur
Nom de la contrainte	`fk_ticket_client`
Colonnes	`client_id`
Table référencée	`billetterie.client`
Colonnes référencées	`id`
ON UPDATE	`CASCADE`
ON DELETE	`RESTRICT`

Configurer la seconde FK : `evenement_id → evenement.id`

Clique **Ajouter** à nouveau :

Champ	Valeur
Nom de la contrainte	`fk_ticket_evenement`
Colonnes	`evenement_id`
Table référencée	`billetterie.evenement`
Colonnes référencées	`id`
ON UPDATE	`CASCADE`
ON DELETE	`RESTRICT`

Enregistre. Si tout est bon, les deux contraintes apparaissent dans la liste.

Partie 4 — Que veulent dire `ON DELETE` et `ON UPDATE` ?

Ces options répondent à une question simple : **quand la ligne référencée bouge ou disparaît, qu'est-ce qui doit arriver aux lignes qui la référencent ?**

Les options

Option	Effet quand le client référencé est supprimé / modifié
`RESTRICT`	**Refuser** l'opération tant que des tickets référencent ce client
`CASCADE`	**Propager** : supprimer aussi les tickets / mettre à jour la valeur
`SET NULL`	Mettre `client_id` à `NULL` (seulement si la colonne autorise NULL)
`NO ACTION`	Équivalent à `RESTRICT` en MySQL

Le choix par défaut raisonnable

- `ON DELETE RESTRICT` — on **refuse** de supprimer un client qui a des tickets. Sinon, on perdrait silencieusement de l'historique de vente. Pour supprimer le client, il faudra d'abord traiter ses tickets explicitement.
- `ON UPDATE CASCADE` — si pour une raison étrange l'id d'un client changeait, on veut que les tickets suivent. En pratique, les `id AUTO_INCREMENT` ne changent jamais, mais cette option ne coûte rien.

> `CASCADE` sur `ON DELETE` peut sembler pratique ("je supprime le client, tout son historique part avec") mais c'est **dangereux** : une fausse manip efface des données qu'on ne voulait pas perdre. On l'utilise consciemment, jamais par défaut.

Partie 5 — Le SQL complet

Voici ce que HeidiSQL a envoyé à MySQL :

```sql
CREATE TABLE `ticket` (
    `id`           INT          NOT NULL AUTO_INCREMENT,
    `prix`         DECIMAL(6,2) NOT NULL,
    `statut`       VARCHAR(20)  NOT NULL,
    `date_achat`   DATETIME     NOT NULL,
    `client_id`    INT          NOT NULL,
    `evenement_id` INT          NOT NULL,
    PRIMARY KEY (`id`),
    CONSTRAINT `fk_ticket_client`
        FOREIGN KEY (`client_id`) REFERENCES `client` (`id`)
        ON UPDATE CASCADE ON DELETE RESTRICT,
    CONSTRAINT `fk_ticket_evenement`
        FOREIGN KEY (`evenement_id`) REFERENCES `evenement` (`id`)
        ON UPDATE CASCADE ON DELETE RESTRICT
);
```

Garde ce modèle sous les yeux : c'est exactement ce que tu écriras un jour à la main quand tu n'auras plus HeidiSQL sous la main.

Partie 6 — Tester l'intégrité référentielle

Le moment où le SGBDR montre **enfin** son utilité. Onglet **Requête**, exécute (F9) ces tests un par un.

Test 1 — Insertion valide

Vérifie d'abord les ids existants dans tes tables `client` et `evenement` (par exemple `SELECT id FROM client;`). Puis :

```
INSERT INTO ticket (prix, statut, date_achat, client_id, evenement_id)
VALUES (12.00, 'payé', '2026-10-01 14:30:00', 1, 1);
```

→ Succès. Le ticket est créé.

Test 2 — Client inexistant

```
INSERT INTO ticket (prix, statut, date_achat, client_id, evenement_id)
VALUES (12.00, 'payé', '2026-10-01 14:30:00', 999, 1);
```

→ **Erreur** :

```
Cannot add or update a child row: a foreign key constraint fails
```

Traduction : « Je refuse parce que ta clé étrangère pointe dans le vide. » Sans cette contrainte, tu te serais retrouvé avec un ticket orphelin — exactement le genre de bug silencieux qui pourrit une application en production.

Test 3 — Événement inexistant

```
INSERT INTO ticket (prix, statut, date_achat, client_id, evenement_id)
VALUES (15.00, 'réservé', '2026-10-01 15:00:00', 1, 999);
```

→ Même erreur, sur l'autre FK.

Test 4 — Supprimer un client référencé

```
DELETE FROM client WHERE id = 1;
```

→ **Erreur** :

```
Cannot delete or update a parent row: a foreign key constraint fails
```

C'est `ON DELETE RESTRICT` qui bloque : le client 1 a des tickets, on ne peut pas l'effacer sans gérer ses tickets d'abord. C'est exactement le comportement qu'on voulait.

> Pour vraiment supprimer le client, il faudrait d'abord supprimer ses tickets (`DELETE FROM ticket WHERE client_id = 1;`), puis le client. C'est plus de travail — c'est aussi le but : t'obliger à réfléchir avant de perdre des données.

Partie 7 — Visualiser les relations

HeidiSQL ne propose pas un vrai éditeur visuel de schéma, mais tu peux voir les relations en :

- Clic droit sur la table `ticket` → **Afficher CREATE TABLE** : tu vois toutes les contraintes en SQL
- Onglet **Clés étrangères** d'une table : la liste des FK sortantes

> Pour un schéma visuel propre, des outils comme **MYSQL WORKBENCH** ou **DBeaver** offrent une vue *Entity-Relationship*. Optionnel à ce stade — le SQL et le MLD papier suffisent.

Exercices

Exercice 1 — Insérer une vraie cohorte

Insère **5 tickets** valides, répartis entre tes clients et tes événements existants. Vérifie le résultat avec :

```sql
SELECT t.id, c.nom, e.titre, t.prix, t.statut
FROM ticket t
JOIN client c    ON c.id = t.client_id
JOIN evenement e ON e.id = t.evenement_id;
```

Tu n'as pas vu `JOIN` formellement — l'idée : la requête montre côte à côte les infos de trois tables grâce aux FK. C'est tout l'intérêt du modèle relationnel.

Exercice 2 — Casser intentionnellement

Tente d'insérer un ticket avec un `client_id` qui n'existe pas. Relève le message d'erreur exact. À quoi sert ce blocage dans une vraie application (billetterie en ligne par exemple) ?

Exercice 3 — Comprendre ON DELETE

Modifie la contrainte `fk_ticket_client` pour mettre `ON DELETE CASCADE`. Réessaie de supprimer le client n°1. Que se passe-t-il pour ses tickets ? Pourquoi est-ce dangereux dans la vraie vie ? Remets ensuite la valeur sur

`RESTRICT`.

Exercice 4 — Ton projet

Ajoute toutes les **clés étrangères** manquantes de ton projet personnel. Pour chaque FK, justifie en une phrase ton choix de `ON DELETE` (`RESTRICT` ou `CASCADE`).

À retenir

- Une clé étrangère est une colonne déclarée comme `FOREIGN KEY (...) REFERENCES autre_table(id)`.
- La colonne FK doit avoir **exactement le même type** que la PK qu'elle référence.
- Le SGBDR **refuse** toute insertion ou mise à jour qui pointerait vers une ligne inexistante : c'est l'**intégrité référentielle**.
- `ON DELETE RESTRICT` est le défaut prudent ; `CASCADE` ne s'utilise que consciemment.
- Une table peut avoir **plusieurs** clés étrangères (autant que de relations auxquelles elle participe).
- HeidiSQL est un outil pratique ; le **SQL généré** est la source de vérité — apprends à le lire.

Suite

Tu as une base complète, propre, avec ses contraintes. Au prochain cours, on fait exactement le même travail dans **phpMyAdmin** — l'outil que tu utiliseras sur un hébergement web. Tu verras qu'au-delà de l'interface, ce sont les mêmes concepts et le même SQL en dessous.

La chaos de données peu structurées

Dans cette activité, tu vas manipuler un fichier réel contenant des informations de billetterie. Ton objectif est d'observer les données et de comprendre pourquoi une simple liste ne suffit pas toujours.

4TTR niveau

Dans cette activité, tu vas manipuler un fichier réel contenant des informations de billetterie. Ton objectif est d'observer les données et de comprendre pourquoi une simple liste ne suffit pas toujours.

Objectifs

À la fin de cette activité, tu seras capable de :

- repérer des problèmes dans un tableau de données
- identifier des répétitions et des incohérences
- comprendre qu'un tableau peut contenir plusieurs types d'informations mélangées
- proposer une organisation plus claire des données

Fichier à utiliser

Ouvre le fichier CSV fourni par ton professeur dans un tableur (Excel, LibreOffice, Google Sheets…).

Partie 1 — Observation des données

Observe attentivement le contenu du fichier.

Réponds aux questions suivantes :

- Combien de fois le client **Marie Dupont** apparaît-il dans le fichier ?
- Les informations de ce client sont-elles toujours identiques (nom, téléphone…) ?
- Combien de fois l'événement **Soirée Quiz Pop Culture** apparaît-il ?

- Les noms des événements sont-ils toujours écrits de la même manière ?
- Les dates sont-elles toutes écrites avec le même format ?
- Les prix sont-ils toujours stockés de la même façon ?
- Trouves-tu des informations manquantes ? Donne un exemple.
- Ce fichier te semble-t-il fiable et facile à utiliser ? Explique pourquoi.

Partie 2 — Manipulation

Effectue les actions suivantes directement dans le fichier ou en expliquant comment tu ferais :

- Retrouver tous les tickets du client **Lucas Martin**
- Retrouver tous les participants du **Quiz Cinéma**
- Retrouver tous les tickets pour des évènements qui se passent au Forum de Liège
- Modifier le numéro de téléphone de **Marie Dupont**
- Changer le lieu du **Blind Test Années 90**
- Supprimer un ticket annulé
- Compter le nombre de tickets vendus pour chaque événement

Partie 3 — Analyse

Réponds aux questions suivantes :

- Pourquoi certaines informations apparaissent-elles plusieurs fois ?
- Que se passe-t-il si tu modifies une information à un seul endroit ?
- Quels problèmes peuvent apparaître si plusieurs personnes modifient ce fichier ?
- Est-ce que toutes les informations présentes dans une ligne concernent la même chose ?

Partie 4 — Comprendre les données

Dans le tableau, on retrouve plusieurs types d'informations.

Classe les colonnes dans les trois catégories suivantes :

Informations sur le client

→ .....

Informations sur le ticket

→ .....

Informations sur l’événement

→ .....

Informations sur la salle / le lieux

→ .....

Partie 5 — Proposition d’amélioration

Propose une meilleure organisation des données.

Tu dois :

- regrouper les informations qui vont ensemble
- donner un nom à chaque groupe
- expliquer pourquoi cette organisation est plus claire

Tu peux répondre sous forme de schéma simple ou de liste.

À retenir

Un tableau unique peut devenir difficile à gérer lorsque :

- les mêmes informations sont répétées plusieurs fois
- les données ne sont pas toujours cohérentes
- plusieurs types d’informations sont mélangés

Dans ce cas, il est nécessaire de réfléchir à une meilleure structure des données.

© 2025 Mr. Lambrechts https://v2.ttrinfo.be 4|4