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.

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.

```sql
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 :

```sql
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 :

```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 ?