Interroger deux tables

Lire ensemble deux tables reliées par une clé étrangère. On commence par la requête naïve, qui donne 36 lignes pour 12 élèves, puis on écrit la bonne : la jointure.

4TTR 5TTR 6TTR Découverte

Dans une base bien découpée, la classe d'un élève n'est pas écrite en toutes lettres : la table `eleve` contient seulement un numéro. Pour afficher « Lucas Adam, 4TTR », il faut lire deux tables en même temps. Cet article commence par la requête la plus simple, qui donne un résultat absurde, puis construit la bonne.

> 💡 **BASE DE TRAVAIL**
>
> Télécharge `mon-ecole-classes.db` dans la carte « Téléchargements » de la sidebar et ouvre-la avec DB Browser for SQLite. Dans l'onglet *Éditer les pragmas*, vérifie que la case **Foreign Keys** est cochée.

🎯 Objectifs

À la fin de cet article, tu seras capable de :

1. **Reconnaître** un produit cartésien et **expliquer** d'où il vient.
2. **Écrire** une jointure avec `JOIN … ON`.
3. **Utiliser** des alias de table pour rendre tes requêtes lisibles.
4. **Choisir** entre `JOIN` et `LEFT JOIN` selon la question posée.
5. **Compter** par groupe à travers une jointure.
6. **Exécuter** une jointure depuis Python.

Deux tables reliées

La base contient une table `classe` et une table `eleve`. Voici leur description, au format du modèle logique de données (MLD) : le nom de la table, puis ses colonnes, avec la clé primaire soulignée.

- classe (id, nom, annee, local)
- eleve (id, nom, prenom, date_naissance, commune, email, absences, #classe_id → classe.id)

La colonne `classe_id` est une **clé étrangère** : elle contient l'`id` de la classe de l'élève. L'écriture `#classe_id → classe.id` rappelle vers quelle colonne elle mène. Regarde les trois premiers élèves :

```sql
SELECT id, nom, prenom, classe_id FROM eleve WHERE id <= 3;
```

```
id | nom      | prenom | classe_id
---+----------+--------+----------
1  | Adam     | Lucas  | 1
2  | Bastin   | Emma   | 1
3  | Charlier | Noah   | 1
```

Personne ne veut lire un bulletin où il est écrit « classe 1 ». Pour afficher `4TTR`, il faut aller chercher le nom de la classe dans l'autre table.

La requête naïve : deux tables dans le `FROM`

Essayons le plus simple : écrire les deux tables dans le `FROM`, séparées par une virgule. Comme les deux tables ont une colonne `nom`, on précise à chaque fois de quelle table elle vient : `eleve.nom` et `classe.nom`.

```sql
SELECT eleve.nom, classe.nom
FROM eleve, classe;
```

Le résultat compte **36 lignes**. Voici les neuf premières :

```
nom      | nom
---------+-----
Adam     | 4TTR
Adam     | 5TTR
Adam     | 6TTR
Bastin   | 4TTR
Bastin   | 5TTR
Bastin   | 6TTR
Charlier | 4TTR
Charlier | 5TTR
Charlier | 6TTR
```

36 lignes pour 12 élèves, et Lucas Adam apparaît dans les trois classes à la fois.

D'où viennent ces 36 lignes

12 élèves × 3 classes = 36. SQLite a fait exactement ce qu'on lui a demandé : il a associé chaque élève à chaque classe, sans rien trier. Sur ces 36 combinaisons, seules 11 sont vraies : celles où le `classe_id` de l'élève est égal à l'`id` de la classe.

> 📖 **NOUVELLE NOTION : LE PRODUIT CARTÉSIEN**
>
> Le **produit cartésien** de deux tables est l'ensemble de toutes les combinaisons possibles entre une ligne de la première table et une ligne de la seconde. Son nombre de lignes est le produit des deux nombres de lignes : 12 élèves × 3 classes = 36 lignes.
>
> En anglais : GB ***cartesian product***.

> ⚠️ **AUCUN MESSAGE D'ERREUR**

Le produit cartésien est dangereux parce que la requête s'exécute sans erreur. Avec 12 élèves et 3 classes, l'absurdité saute aux yeux. Avec 800 élèves et 40 classes, on obtient 32 000 lignes qui ont l'air plausibles, et quelqu'un finira par en tirer une statistique fausse.

Le symptôme à reconnaître : un résultat beaucoup plus gros que prévu, où chaque ligne se répète. Réflexe : vérifie la condition qui relie les deux tables.

La jointure

Il manque une phrase à notre requête : « garde seulement les combinaisons où l'élève appartient vraiment à la classe ». Autrement dit, les lignes où `classe.id` est égal à `eleve.classe_id`.

```sql
SELECT eleve.prenom, eleve.nom, classe.nom AS classe
FROM eleve
JOIN classe ON classe.id = eleve.classe_id
ORDER BY classe.nom, eleve.nom;
```

```
prenom  | nom       | classe
--------+-----------+-------
Lucas   | Adam      | 4TTR
Emma    | Bastin    | 4TTR
Noah    | Charlier  | 4TTR
Léa     | Delvaux   | 4TTR
Hugo    | Englebert | 4TTR
Chloé   | Fontaine  | 5TTR
Nathan  | Gérard    | 5TTR
Manon   | Hubert    | 5TTR
Louis   | Istas     | 5TTR
Camille | Jacques   | 6TTR
Arthur  | Kevers    | 6TTR
```

Onze lignes, et chacune est vraie. Pour rappel, `AS` veut dire *comme* : la colonne `classe.nom` s'affiche sous le nom `classe`.

📖 NOUVELLE NOTION : LA JOINTURE

Une **jointure** assemble les lignes de deux tables qui correspondent l'une à l'autre. `JOIN` veut dire *joindre* et `ON` veut dire *sur* : on joint la table `classe` *sur* la condition `classe.id = eleve.classe_id`. Seules les combinaisons qui respectent cette condition sont gardées.

En anglais : GB ***join***.

Morceau	Rôle
`FROM eleve`	la première table
`JOIN classe`	la table à joindre
`ON classe.id = eleve.classe_id`	la condition de liaison : la clé primaire d'un côté, la clé étrangère de l'autre

> 🧠 **LE `ON` S'ÉCRIT TOUJOURS PAREIL**
>
> Clé étrangère d'un côté, clé primaire de l'autre. Le MLD te donne donc la condition sans rien inventer : `#classe_id → classe.id` devient `ON classe.id = eleve.classe_id` .

Les alias de table

Écrire `eleve.` et `classe.` devant chaque colonne devient vite long. On peut donner un surnom court à chaque table :

```sql
SELECT e.prenom, e.nom, c.nom AS classe
FROM eleve e
JOIN classe c ON c.id = e.classe_id
ORDER BY c.nom, e.nom;
```

Le résultat est exactement le même que plus haut. `FROM eleve e` donne à la table `eleve` le surnom `e` ; on écrit ensuite `e.nom` au lieu de `eleve.nom` .

> 📖 **NOUVELLE NOTION : L'ALIAS DE TABLE**
>
> Un **alias de table** est un surnom donné à une table dans une requête, écrit juste après son nom : `FROM eleve e` . Il sert à préfixer les colonnes (`e.nom` , `c.nom`) pour dire de quelle table chacune vient. Il n'existe que le temps de la requête.
>
> En anglais : GB ***table alias***.

Préfixer les colonnes n'est pas qu'une question de confort. Les deux tables ont une colonne `nom` , et sans préfixe, SQLite ne sait pas de laquelle tu parles :

```sql
SELECT nom FROM eleve JOIN classe ON classe.id = eleve.classe_id;
```

```
ambiguous column name: nom
```

Le message veut dire « nom de colonne ambigu : nom ». La requête est refusée.

> 💡 **PRÉFIXE TOUTES LES COLONNES**
>
> Dès qu'une requête contient une jointure, préfixe toutes les colonnes, même celles qui ne sont pas ambiguës. On voit alors d'un coup d'œil d'où vient chaque information.

Filtrer une jointure

`WHERE` (*où*) fonctionne comme d'habitude, et il peut porter sur les colonnes des deux tables :

```sql
SELECT e.prenom, e.nom, c.local
FROM eleve e
JOIN classe c ON c.id = e.classe_id
WHERE c.nom = '5TTR'
ORDER BY e.nom;
```

```
prenom | nom      | local
-------+----------+------
Chloé  | Fontaine | A21
Nathan | Gérard   | A21
Manon  | Hubert   | A21
Louis  | Istas    | A21
```

Cette requête traverse deux tables : le critère porte sur la classe, et le résultat parle des élèves. C'est tout l'intérêt d'avoir séparé les données puis de les avoir reliées.

`JOIN` ou `LEFT JOIN` ?

Regarde encore le résultat de la première jointure : **onze lignes pour douze élèves**. Sarah Lambert a disparu.

Elle n'a pas encore de classe : son `classe_id` est vide (`NULL`). Aucune ligne de `classe` ne correspond, donc `JOIN` l'écarte. Selon la question posée, c'est le bon comportement… ou une perte silencieuse.

Pour garder tous les élèves, on écrit `LEFT JOIN` :

```sql
SELECT e.prenom, e.nom, c.nom AS classe
FROM eleve e
LEFT JOIN classe c ON c.id = e.classe_id
ORDER BY e.nom;
```

```
prenom  | nom       | classe
--------+-----------+-------
Lucas   | Adam      | 4TTR
Emma    | Bastin    | 4TTR
Noah    | Charlier  | 4TTR
Léa     | Delvaux   | 4TTR
Hugo    | Englebert | 4TTR
Chloé   | Fontaine  | 5TTR
Nathan  | Gérard    | 5TTR
Manon   | Hubert    | 5TTR
Louis   | Istas     | 5TTR
Camille | Jacques   | 6TTR
Arthur  | Kevers    | 6TTR
Sarah   | Lambert   | NULL
```

Sarah Lambert est là, avec une classe vide.

> 📖 **NOUVELLE NOTION : `LEFT JOIN`**
>
> `LEFT` veut dire *gauche*. `LEFT JOIN` garde **toutes** les lignes de la table de gauche, celle écrite dans le `FROM`, même celles qui n'ont aucune correspondance dans l'autre table. Pour ces lignes-là, les colonnes de la table de droite valent `NULL`.
>
> En anglais : GB ***left join***.

Tu demandes…	Tu écris
« Tous les élèves, avec leur classe quand elle est connue »	`LEFT JOIN`
« Les élèves qui ont une classe, avec cette classe »	`JOIN`

🧠 LA QUESTION À SE POSER

Est-ce que je veux voir les lignes qui n'ont **pas** de correspondance ?

Choisir `JOIN` par habitude fait disparaître, sans bruit, exactement les cas qui méritaient ton attention. Un élève sans classe en octobre, c'est probablement un oubli d'encodage : c'est précisément la ligne qu'il fallait voir.

Retrouver les lignes sans correspondance

Pour rappel, `IS NULL` veut dire *est vide*. On l'écrit toujours ainsi, jamais `= NULL`. Combiné à `LEFT JOIN`, il isole les élèves qui n'ont pas de classe :

```
SELECT e.prenom, e.nom
FROM eleve e
LEFT JOIN classe c ON c.id = e.classe_id
WHERE c.id IS NULL;
```

```
prenom | nom
-------+--------
Sarah  | Lambert
```

💡 UNE REQUÊTE DE CONTRÔLE QUALITÉ

`LEFT JOIN … WHERE … IS NULL` liste exactement les cas incomplets. Garde cette requête sous la main et relance-la régulièrement : si elle renvoie des lignes, un encodage est à compléter.

Le piège du `WHERE` sur la table de droite

Nouvelle question : « tous les élèves, avec leur classe, sauf ceux de 4TTR ». Sarah Lambert n'est pas en 4TTR : elle doit donc apparaître. `<>` veut dire *différent de*.

```
SELECT e.prenom, e.nom, c.nom AS classe
FROM eleve e
LEFT JOIN classe c ON c.id = e.classe_id
WHERE c.nom <> '4TTR'
ORDER BY e.nom;
```

```
prenom  | nom      | classe
--------+----------+-------
Chloé   | Fontaine | 5TTR
Nathan  | Gérard   | 5TTR
Manon   | Hubert   | 5TTR
Louis   | Istas    | 5TTR
```

```
Camille | Jacques  | 6TTR
Arthur  | Kevers   | 6TTR
```

Sarah Lambert a disparu, malgré le `LEFT JOIN`. Pour sa ligne, `c.nom` vaut `NULL`. Or une comparaison avec `NULL` n'est jamais vraie : SQLite ne sait pas si « rien » est différent de `4TTR`, et le `WHERE` écarte la ligne.

Pour la garder, il faut le demander explicitement : `OR` veut dire *ou*.

```sql
SELECT e.prenom, e.nom, c.nom AS classe
FROM eleve e
LEFT JOIN classe c ON c.id = e.classe_id
WHERE c.nom <> '4TTR' OR c.id IS NULL
ORDER BY e.nom;
```

```
prenom  | nom      | classe
--------+----------+-------
Chloé   | Fontaine | 5TTR
Nathan  | Gérard   | 5TTR
Manon   | Hubert   | 5TTR
Louis   | Istas    | 5TTR
Camille | Jacques  | 6TTR
Arthur  | Kevers   | 6TTR
Sarah   | Lambert  | NULL
```

> ⚠️ **UN `WHERE` SUR LA TABLE DE DROITE ANNULE LE `LEFT`**
>
> Une condition du `WHERE` sur une colonne de la table de droite élimine toutes les lignes où cette colonne vaut `NULL`, c'est-à-dire justement celles que le `LEFT JOIN` devait garder. Le résultat devient celui d'un simple `JOIN`, sans aucun message d'erreur.

Compter par classe : jointure et `GROUP BY`

Pour rappel, `COUNT` veut dire *compter* et `GROUP BY` veut dire *regrouper par* : `GROUP BY c.nom` range les lignes par classe, et `COUNT` compte les lignes de chaque groupe. Combien d'élèves compte chaque classe ?

```sql
SELECT c.nom AS classe, COUNT(*) AS nb_eleves
FROM eleve e
JOIN classe c ON c.id = e.classe_id
GROUP BY c.nom
ORDER BY c.nom;
```

```
classe | nb_eleves
-------+----------
4TTR   | 5
5TTR   | 4
6TTR   | 2
```

L'école ouvre maintenant une classe de 3TTR, dans le local B10. Aucun élève n'y est encore inscrit. Ajoute-la dans ta base :

```sql
INSERT INTO classe (nom, annee, local) VALUES ('3TTR', 3, 'B10');
```

Relance la requête précédente : le résultat ne change pas, la 3TTR n'apparaît pas. Aucun élève ne la désigne, donc `JOIN` n'a rien à assembler pour elle. Pour voir **toutes les classes**, il faut partir de la table `classe` et écrire un `LEFT JOIN` vers `eleve` :

```sql
SELECT c.nom AS classe, COUNT(e.id) AS nb_eleves
FROM classe c
LEFT JOIN eleve e ON e.classe_id = c.id
GROUP BY c.nom
ORDER BY c.nom;
```

```
classe | nb_eleves
-------+----------
3TTR   | 0
4TTR   | 5
5TTR   | 4
6TTR   | 2
```

La classe vide apparaît, avec 0 élève. Le sens de la jointure compte : la table dont on veut voir **toutes** les lignes s'écrit dans le `FROM`, à gauche du `LEFT JOIN`.

> ⚠️ **`COUNT(*)` OU `COUNT(e.id)` ?**
>
> `COUNT(*)` compte les lignes, même celles remplies de `NULL`. `COUNT(e.id)` ne compte que les lignes où `e.id` n'est pas vide. Avec `COUNT(*)`, la 3TTR afficherait 1 élève au lieu de 0 :
>
> ```
> classe | nb_eleves
> -------+----------
> 3TTR   | 1
> 4TTR   | 5
> 5TTR   | 4
> 6TTR   | 2
> ```
>
> Avec un `LEFT JOIN`, compte toujours une colonne de la table de droite.

Remets ta base dans son état de départ en supprimant la 3TTR :

```sql
DELETE FROM classe WHERE nom = '3TTR';
```

Cette fois, la suppression est acceptée, même avec le pragma activé : aucun élève ne désigne cette classe, donc aucune ligne ne devient orpheline.

Depuis Python

Pour rappel, le module `sqlite3` permet à un programme Python d'ouvrir une base SQLite : `connect` ouvre la base, `execute` envoie une requête et `close` ferme la connexion. Place ce programme dans le même dossier que `mon-ecole-classes.db` :

```python
import sqlite3

conn = sqlite3.connect("mon-ecole-classes.db")
conn.execute("PRAGMA foreign_keys = ON")
conn.row_factory = sqlite3.Row

requete = """
    SELECT e.prenom, e.nom, c.nom AS classe
    FROM eleve e
    JOIN classe c ON c.id = e.classe_id
    ORDER BY c.nom, e.nom
"""

for ligne in conn.execute(requete):
    print(f"{ligne['prenom']:8} {ligne['nom']:10} {ligne['classe']}")

conn.close()
```

```
Lucas    Adam       4TTR
Emma     Bastin     4TTR
Noah     Charlier   4TTR
Léa      Delvaux    4TTR
Hugo     Englebert  4TTR
Chloé    Fontaine   5TTR
Nathan   Gérard     5TTR
Manon    Hubert     5TTR
Louis    Istas      5TTR
Camille  Jacques    6TTR
Arthur   Kevers     6TTR
```

Quatre points méritent ton attention.

- `PRAGMA foreign_keys = ON` vient juste après l'ouverture. Un programme Python part toujours d'un pragma désactivé. Avec le pragma activé, une écriture incohérente, comme `UPDATE eleve SET classe_id = 99 WHERE id = 12`, arrête le programme avec l'erreur `sqlite3.IntegrityError: FOREIGN KEY constraint failed`.
- `conn.row_factory = sqlite3.Row` permet de lire chaque colonne par son nom : `ligne['prenom']` plutôt que `ligne[0]`.
- **Les guillemets triples** `"""` permettent d'écrire la requête sur plusieurs lignes, avec la mise en page qui la rend lisible. Utilise-les dès que la requête dépasse une ligne.
- `AS classe` **est indispensable.** Sans lui, les deux colonnes s'appelleraient `nom`.

Voici ce qui se passe sans `AS`, avec la requête `SELECT e.nom, c.nom FROM eleve e JOIN classe c ON c.id = e.classe_id` : `ligne.keys()` renvoie deux noms identiques, et `ligne["nom"]` donne seulement le nom de l'élève.

```
['nom', 'nom']
Adam
```

Le nom de la classe est dans le résultat, mais impossible de l'atteindre par son nom. Aucune erreur ne te prévient.

Exercices

Tous les exercices portent sur `mon-ecole-classes.db`, avec le pragma activé.

Exercice 1 — Prédire le produit cartésien ★★☆☆☆

`COUNT(*)` compte les lignes d'un résultat : `SELECT COUNT(*) FROM eleve;` renvoie le nombre d'élèves.

1. Compte les lignes de `eleve`, puis celles de `classe`.
2. Sans l'exécuter, prédis le nombre de lignes de `SELECT * FROM eleve, classe;`. Vérifie avec `SELECT COUNT(*) FROM eleve, classe;`.
3. La base compte aussi 6 professeurs. Prédis le nombre de lignes de `SELECT * FROM eleve, classe, professeur;`, puis vérifie.
4. Une école de 800 élèves répartis en 40 classes écrit la requête naïve. Combien de lignes obtient-elle ?

Exercice 2 — Écrire des jointures ★★☆☆☆

Utilise des alias et préfixe toutes les colonnes.

1. La liste de tous les élèves qui ont une classe, avec le nom de leur classe.
2. Les élèves de 4TTR uniquement.
3. Les élèves dont la classe se trouve dans le local `A21`.
4. Le nom et l'année de la classe de Manon Hubert.
5. Les élèves des classes de 5e et de 6e année, triés par année puis par nom.

Exercice 3 — `JOIN` ou `LEFT JOIN` ? ★★☆☆☆

Pour chaque demande, dis quelle jointure tu utilises et **pourquoi**, puis écris la requête.

Lettre	Demande
a	La liste complète des élèves pour l'appel, avec leur classe quand elle est connue
b	Les élèves à convoquer pour la photo de classe
c	Les élèves qui ne sont rattachés à aucune classe
d	Le nombre d'élèves de chaque classe, classes vides comprises
e	Les classes qui n'ont encore aucun élève (attention au sens de la jointure)

Exercice 4 — Diagnostiquer ★★★☆☆

Ces quatre requêtes ont chacune un problème. Exécute-les, note ce qui se passe, explique la cause et corrige.

```sql
-- a)
SELECT nom, prenom FROM eleve JOIN classe ON classe.id = eleve.classe_id;

-- b)
SELECT e.nom, c.nom FROM eleve e, classe c WHERE c.nom = '4TTR';

-- c)
SELECT e.nom, c.nom AS classe FROM eleve e JOIN classe c ON c.id = e.id;

-- d)
SELECT e.nom, c.nom AS classe FROM eleve e LEFT JOIN classe c ON c.id = e.classe_id WHERE c.nom = '4TT
```

- La requête (b) renvoie douze lignes. Tous les élèves sont-ils vraiment en 4TTR ?
- La requête (c) est la plus sournoise : elle s'exécute, elle renvoie des lignes, et tout est faux. Compare son résultat à celui de la jointure correcte et explique d'où viennent ces lignes.

- La requête (d) mélange `LEFT JOIN` et un `WHERE` sur la table de droite. Que devient Sarah Lambert ? Le `LEFT JOIN` sert-il encore à quelque chose ?

Exercice 5 — Rapport de contrôle qualité en Python ★★★★☆

Écris un programme Python qui affiche ce rapport de cohérence de la base :

```
Rapport de cohérence
  Élèves ................ 12
  Élèves sans classe .... 1   → Sarah Lambert
  Classes ............... 3
  Classes sans élève .... 0
```

Contraintes :

- chaque nombre vient d'une requête, jamais d'une valeur écrite à la main ;
- le programme active `PRAGMA foreign_keys = ON` et utilise `row_factory` ;
- les requêtes de plus d'une ligne sont écrites entre guillemets triples ;
- pour les deux lignes « sans », réfléchis au sens de la jointure.

Vérifie ensuite ton programme : ajoute une classe vide dans une copie de la base, relance le rapport et contrôle que la dernière ligne passe à 1.

✍️ À retenir

- Deux tables dans le `FROM` sans condition donnent le **produit cartésien** : toutes les combinaisons, sans aucun message d'erreur. Symptôme : un résultat beaucoup trop gros, où chaque ligne se répète.
- `JOIN autre_table ON cle_primaire = cle_etrangere` ne garde que les combinaisons réelles ; la condition se lit directement dans le MLD.
- Les **alias de table** (`FROM eleve e`) raccourcissent les requêtes ; préfixer les colonnes évite l'erreur `ambiguous column name`.
- `JOIN` écarte les lignes sans correspondance ; `LEFT JOIN` les garde, avec des `NULL`. La question à se poser : est-ce que je veux voir ceux qui n'ont pas de correspondance ?
- `LEFT JOIN … WHERE … IS NULL` isole les cas incomplets ; un `WHERE` sur la table de droite annule le `LEFT`.
- Pour compter avec un `LEFT JOIN`, écris `COUNT(colonne_de_droite)`, pas `COUNT(*)`.
- En Python : pragma activé à chaque connexion, guillemets triples pour le SQL, `AS` dès que deux colonnes portent le même nom.

Suite

Tu sais maintenant découper des données en deux entités, les relier par une clé étrangère et les interroger ensemble. Pour réviser tout le niveau, passe par la fiche à étudier.