Interroger une base SQLite : SELECT et COUNT

Poser des questions à la base ecole.db : filtrer, trier, compter, regrouper et joindre. Tous les exemples de ce cours sont exécutés sur la base fournie — les résultats affichés sont ceux que tu dois obtenir.

5TTR 6TTR Découverte

Une base de données ne sert à rien tant qu'on ne lui pose pas de questions. Ouvre `ecole.db` dans DB Browser, onglet *Exécuter le SQL*, et suis chaque exemple : les résultats affichés ici sont ceux de la base fournie.

🎯 Objectifs

À la fin de ce cours, tu seras capable de :

1. **Sélectionner** des colonnes, les renommer et les combiner.
2. **Filtrer** des lignes avec `WHERE`, y compris sur des valeurs `NULL`.
3. **Trier** et **limiter** un résultat.
4. **Compter** et calculer des agrégats : `COUNT`, `AVG`, `MIN`, `MAX`, `SUM`.
5. **Regrouper** avec `GROUP BY` et filtrer un regroupement avec `HAVING`.
6. **Joindre** plusieurs tables, et choisir entre `JOIN` et `LEFT JOIN`.

Lire une table

Tout voir

```sql
SELECT * FROM classe;
```

```
id | nom  | annee | local
---+------+-------+------
1  | 3TTR | 3     | B12
2  | 4TTR | 4     | B14
3  | 5TTR | 5     | A21
4  | 6TTR | 6     | A23
```

L'étoile signifie « toutes les colonnes ». Pratique pour explorer, à éviter ensuite : si quelqu'un ajoute une colonne à la table, ta requête se met à ramener des données que ton programme n'attend pas.

Choisir ses colonnes, les renommer

```sql
SELECT prenom, nom, date_naissance AS naissance
FROM eleve
WHERE classe_id = 4
ORDER BY nom;
```

```
prenom  | nom      | naissance
--------+----------+-----------
Victor  | Bruyère  | 2007-...
Maya    | Collard  | 2007-...
Gaspard | Dethier  | 2007-08-25
Iris    | Evrard   | 2007-...
Élias   | Falize   | 2007-...
Jade    | Grégoire | 2007-...
```

`AS` donne un **alias** : le nom affiché en en-tête. C'est indispensable dès qu'une colonne est calculée.

Combiner des colonnes

L'opérateur `||` colle deux textes :

```sql
SELECT prenom || ' ' || nom AS eleve
FROM eleve
WHERE classe_id = 4
ORDER BY nom;
```

```
eleve
-----------------
Victor Bruyère
Maya Collard
Gaspard Dethier
...
```

⚠️ **`||`, PAS `+`.** En SQL, `+` est une addition. `'Victor' + 'Bruyère'` ne concatène pas : SQLite tente de convertir les deux textes en nombres, n'y arrive pas, et renvoie `0`. Un résultat faux, sans message d'erreur.

Éliminer les doublons

```sql
SELECT DISTINCT annee FROM classe ORDER BY annee;
```

```
annee
-----
3
4
5
6
```

Filtrer avec `WHERE`

Les opérateurs

Opérateur	Sens	Exemple
`=` `<>`	Égal, différent	`WHERE classe_id = 3`
`<` `>` `<=` `>=`	Comparaisons	`WHERE valeur >= 12`
`AND` `OR` `NOT`	Combinaisons logiques	`WHERE annee = 6 AND local = 'A23'`
`BETWEEN … AND …`	Intervalle, **bornes comprises**	`WHERE valeur BETWEEN 10 AND 12`
`IN (…)`	Fait partie d'une liste	`WHERE classe_id IN (3, 4)`
`LIKE`	Motif texte	`WHERE nom LIKE 'D%'`
`IS NULL` / `IS NOT NULL`	Absence de valeur	`WHERE classe_id IS NULL`

`LIKE` et ses jokers

`%` remplace n'importe quelle suite de caractères, `_` remplace exactement un caractère.

```sql
SELECT nom, prenom, date_naissance
FROM eleve
WHERE nom LIKE 'D%'
ORDER BY nom;
```

```
nom     | prenom  | date_naissance
--------+---------+---------------
Delvaux | Léa     | 2010-04-15
Dethier | Gaspard | 2007-08-25
```

> 💡 En SQLite, `LIKE` **ne distingue pas majuscules et minuscules** pour les caractères ASCII : `'d%'` donne le même résultat. En revanche il les distingue pour les caractères accentués — `'É%'` ne trouvera pas

`élias` . C'est une limite connue du moteur.

Le cas `NULL`

`NULL` n'est pas zéro, ni une chaîne vide : c'est **l'absence de valeur**. Et une absence ne se compare pas.

```sql
SELECT nom, prenom FROM eleve WHERE classe_id IS NULL;
```

```
nom     | prenom
--------+---------
Nouveau | Camille
```

⚠️ `WHERE CLASSE_ID = NULL` **NE RENVOIE JAMAIS RIEN**, même s'il existe des lignes sans classe. La comparaison d'une valeur inconnue avec quoi que ce soit donne « inconnu », jamais « vrai ». Il faut écrire `IS NULL` ou `IS NOT NULL` . C'est l'erreur la plus fréquente des débutants en SQL — et elle ne provoque aucun message d'erreur.

Trier et limiter

```sql
SELECT intitule, periodes
FROM cours
ORDER BY periodes DESC, intitule
LIMIT 4;
```

```
intitule            | periodes
--------------------+---------
Programmation Python | 6
Bases de données     | 4
Français             | 4
Mathématiques        | 4
```

- `ASC` (croissant) est le défaut, `DESC` inverse.
- On peut trier sur **plusieurs colonnes** : ici, à nombre de périodes égal, l'ordre alphabétique départage.
- `LIMIT n` ne garde que les `n` premières lignes — **après** le tri.

Compter et calculer

`COUNT`, et le piège du `NULL`

```sql
SELECT COUNT(*) AS lignes,
       COUNT(classe_id) AS avec_classe
FROM eleve;
```

```
lignes | avec_classe
-------+------------
33     | 32
```

Deux comptages sur la même table, deux résultats différents :

Écriture	Ce qu'elle compte
`COUNT(*)`	Les **lignes**, sans exception
`COUNT(colonne)`	Les lignes où cette colonne **n'est pas** `NULL`
`COUNT(DISTINCT colonne)`	Les **valeurs différentes**, hors `NULL`

L'écart de 1 est exactement notre élève sans classe. **Retiens ce comportement** : c'est lui qui explique la plupart des comptages faux.

Les autres agrégats

```sql
SELECT MIN(valeur) AS mini,
       MAX(valeur) AS maxi,
       ROUND(AVG(valeur), 2) AS moyenne,
       COUNT(*) AS nb
FROM note;
```

```
mini | maxi | moyenne | nb
-----+------+---------+-----
4    | 20   | 13.16   | 367
```

`ROUND(x, 2)` arrondit à deux décimales : sans lui, `AVG` afficherait `13.161852861035422`. `SUM` existe aussi, mais additionner des points sur 20 n'a aucun sens — un agrégat doit toujours répondre à une vraie question.

Regrouper

`GROUP BY`

Jusqu'ici, les agrégats donnaient **une seule ligne** pour toute la table. `GROUP BY` en donne une **par groupe** :

```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.id
ORDER BY nb_eleves DESC;
```

```
classe | nb_eleves
-------+----------
3TTR   | 10
4TTR   | 9
5TTR   | 7
6TTR   | 6
```

🧠 **LA RÈGLE QUI ÉVITE LES ERREURS.** Toute colonne du `SELECT` doit être soit dans le `GROUP BY`, soit à l'intérieur d'une fonction d'agrégation. SQLite est laxiste et accepte le contraire — il choisit alors une ligne **au hasard** dans le groupe. Le résultat n'est pas faux au sens du moteur, il est simplement dénué de sens. MySQL, lui, refuserait.

`HAVING` : filtrer les groupes

```sql
SELECT e.prenom || ' ' || e.nom AS eleve,
       ROUND(AVG(n.valeur), 1) AS moyenne,
       COUNT(n.id) AS nb_points
FROM eleve e
JOIN note n ON n.eleve_id = e.id
GROUP BY e.id
HAVING moyenne > 14
ORDER BY moyenne DESC;
```

```
eleve              | moyenne | nb_points
-------------------+---------+----------
Nina Xhonneux      | 14.9    | 11
Chloé Fontaine     | 14.6    | 9
Zoé Piret          | 14.6    | 10
Julie Renard       | 14.5    | 11
Elena Vandenbroucke | 14.4    | 14
Camille Jacques    | 14.1    | 8
```

`WHERE` ou `HAVING` ? La distinction est simple une fois posée :

	Agit sur	Peut utiliser un agrégat	Exemple
`WHERE`	Les **lignes**, avant regroupement	Non	`WHERE n.periode = 1`
`HAVING`	Les **groupes**, après regroupement	Oui	`HAVING AVG(n.valeur) > 14`

« Je ne veux que les points du 1er trimestre » → `WHERE`. *« Je ne veux que les élèves dont la moyenne dépasse 14 »* → `HAVING`, parce que la moyenne n'existe qu'une fois le groupe formé.

Joindre des tables

`JOIN` : les lignes qui correspondent des deux côtés

```sql
SELECT co.intitule, p.nom AS professeur, co.periodes
FROM cours co
JOIN professeur p ON p.id = co.professeur_id
ORDER BY p.nom
LIMIT 6;
```

```
intitule             | professeur | periodes
---------------------+------------+---------
Éducation physique   | Colin      | 2
Gestion de projet    | Depré      | 2
Mathématiques        | Dubois     | 4
Bases de données     | Lambrechts | 4
Programmation Python | Lambrechts | 6
Français             | Leroy      | 4
```

Les **alias de table** (`cours co`, `professeur p`) rendent la requête lisible et deviennent indispensables dès qu'on joint trois tables.

`LEFT JOIN` : garder tout le côté gauche

C'est ici que la base d'exemple prend son sens. Comparons deux requêtes sur les inscriptions par cours :

```sql
-- Avec JOIN : le cours de Néerlandais DISPARAÎT du résultat
SELECT co.intitule, COUNT(i.eleve_id) AS nb_inscrits
FROM cours co
JOIN inscription i ON i.cours_id = co.id
GROUP BY co.id
ORDER BY nb_inscrits;
```

```sql
-- Avec LEFT JOIN : il apparaît, avec 0
SELECT co.intitule, COUNT(i.eleve_id) AS nb_inscrits
FROM cours co
LEFT JOIN inscription i ON i.cours_id = co.id
GROUP BY co.id
ORDER BY nb_inscrits;
```

```
intitule               | nb_inscrits
-----------------------+------------
Néerlandais            | 0
Systèmes d'exploitation | 6
Gestion de projet      | 6
Électronique           | 9
```

```
Bases de données        | 13
Réseaux                 | 13
Programmation Python    | 32
Mathématiques           | 32
Français                | 32
Anglais technique       | 32
Éducation physique      | 32
```

🧠 **LA QUESTION QU'IL FAUT SE POSER.** *« Est-ce que je veux voir les lignes qui n'ont pas de correspondance ? »*

- « Les cours **et leur** nombre d'inscrits » → `LEFT JOIN`. Un cours à zéro inscrit est une information précieuse : c'est peut-être une erreur d'encodage.
- « Les cours **qui ont** au moins un inscrit » → `JOIN`.

Choisir `JOIN` par habitude, c'est faire disparaître silencieusement les cas qui méritaient justement ton attention.

Note aussi le `COUNT(i.eleve_id)` : avec `COUNT(*)`, le Néerlandais afficherait **1** au lieu de 0, parce que le `LEFT JOIN` produit bien une ligne — remplie de `NULL`. Le piège du `COUNT` vu plus haut, en situation.

Joindre trois tables

```sql
SELECT c.nom AS classe,
       ROUND(AVG(n.valeur), 2) AS moyenne,
       COUNT(n.id) AS nb_points
FROM classe c
JOIN eleve e ON e.classe_id = c.id
JOIN note  n ON n.eleve_id  = e.id
GROUP BY c.id
ORDER BY moyenne DESC;
```

```
classe | moyenne | nb_points
-------+---------+----------
5TTR   | 13.35   | 88
4TTR   | 13.22   | 97
3TTR   | 13.14   | 84
6TTR   | 12.95   | 98
```

On chaîne les jointures de proche en proche : `classe` → `eleve` → `note`. Chaque `ON` relie la clé étrangère à la clé primaire correspondante.

L'ordre dans lequel SQLite travaille

Une requête ne s'exécute **pas** dans l'ordre où on l'écrit. Comprendre cet ordre explique presque toutes les erreurs :

```
1. FROM / JOIN    → constituer l'ensemble des lignes
2. WHERE          → éliminer des lignes
3. GROUP BY       → former les groupes
4. HAVING         → éliminer des groupes
5. SELECT         → calculer les colonnes affichées et les alias
6. ORDER BY       → trier
7. LIMIT          → couper
```

Deux conséquences très concrètes :

- `WHERE` ne peut pas utiliser un alias défini dans le `SELECT` : à l'étape 2, cet alias n'existe pas encore.
- `ORDER BY` **peut** l'utiliser, puisqu'il passe après. C'est pourquoi `ORDER BY moyenne DESC` fonctionne dans nos exemples.

🧪 Exercices

Toutes les questions portent sur `ecole.db`. Écris la requête, exécute-la, et note le résultat.

Exercice 1 — Prise en main ★☆☆☆☆

1. La liste des professeurs, par ordre alphabétique de nom.
2. Les cours de plus de 2 périodes.
3. Les élèves nés en 2008. *(Indice : `LIKE '2008%'`.)*
4. Le nombre total de cours.
5. Les trois plus mauvais points de la table `note`.

Exercice 2 — Filtrer finement ★★☆☆☆

1. Les élèves de 5TTR **ou** 6TTR, en une seule requête, sans écrire deux fois `classe_id`.
2. Les points strictement compris entre 8 et 10, bornes exclues.
3. Les professeurs dont le nom contient la lettre `e`, quelle que soit la casse.
4. Les élèves dont on ne connaît pas la date de naissance. *(Combien y en a-t-il ?)*
5. Les cours dont l'intitulé fait exactement 8 caractères. *(Indice : `LENGTH()`.)*

Exercice 3 — Compter et regrouper ★★☆☆☆

1. Le nombre de cours donnés par chaque professeur, du plus chargé au moins chargé.
2. Le nombre total de périodes par professeur.
3. Le nombre de points encodés par période (1 et 2).
4. La moyenne par cours, arrondie à une décimale, triée de la meilleure à la moins bonne.
5. Les professeurs qui donnent **plus d'un** cours.

Exercice 4 — Jointures ★★★☆☆

1. La liste `élève — classe` pour tous les élèves, **y compris celui qui n'a pas de classe**.

2. Pour chaque cours : son intitulé, le nom de son professeur, et son nombre d'inscrits.
3. Les points de `Iris Evrard`, avec l'intitulé du cours, triés du meilleur au moins bon.
4. Les élèves qui n'ont **aucun** point encodé. *(Indice : `LEFT JOIN` + `IS NULL`.)*
5. Pour chaque classe, la meilleure moyenne d'élève de cette classe.

Exercice 5 — Diagnostic ★★★☆☆

Ces quatre requêtes ont un problème. Pour chacune : dis ce qui cloche, ce que SQLite renvoie réellement, et écris la version correcte.

```sql
-- a)
SELECT nom, prenom FROM eleve WHERE classe_id = NULL;

-- b)
SELECT nom || ' a ' || COUNT(*) FROM eleve;

-- c)
SELECT c.nom, COUNT(*) AS nb
FROM classe c LEFT JOIN eleve e ON e.classe_id = c.id
GROUP BY c.id;

-- d)
SELECT prenom, AVG(valeur) AS moyenne
FROM eleve e JOIN note n ON n.eleve_id = e.id
WHERE moyenne > 14
GROUP BY e.id;
```

Exercice 6 — Le bulletin ★★★★☆

Écris **une seule requête** qui produit, pour la classe 6TTR, un tableau à quatre colonnes : nom complet de l'élève, nombre de cours suivis, nombre de points encodés, moyenne générale arrondie à une décimale — trié par moyenne décroissante.

Vérifie la cohérence : le nombre de cours suivis doit correspondre à ce que donne la table `inscription`.

✍️ À retenir

- `SELECT colonnes FROM table WHERE condition GROUP BY … HAVING … ORDER BY … LIMIT …`
- `AS` renomme ; `||` concatène (**jamais** `+`).
- `NULL` ne se compare pas : `IS NULL`, jamais `= NULL`.
- `COUNT(*)` compte les lignes ; `COUNT(colonne)` ignore les `NULL`. L'écart révèle les trous.
- `WHERE` filtre les **lignes** avant regroupement, `HAVING` filtre les **groupes** après.
- `JOIN` ne garde que les correspondances ; `LEFT JOIN` garde tout le côté gauche, complété par des `NULL`.
- Dans un `LEFT JOIN`, compte une **colonne** de la table de droite, pas `*`.

- L'exécution suit `FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT` : d'où les alias utilisables dans `ORDER BY` mais pas dans `WHERE`.

Suite

Tu sais interroger la base à la main. Passe au cours Accéder à une base SQLite avec Python : les mêmes requêtes, mais pilotées par un programme — et la question de sécurité qui va avec.