Compter et regrouper

Combien d'élèves compte l'école ? Quelle est la moyenne des absences ? Combien d'élèves par commune ? Les fonctions d'agrégation et GROUP BY répondent par un calcul plutôt que par une liste.

4TTR 5TTR 6TTR Découverte

Combien d'élèves compte la base ? Combien d'absences au total ? Combien d'élèves habitent chaque commune ? Ces questions n'attendent pas une liste d'élèves. Elles attendent un nombre, calculé sur plusieurs lignes à la fois.

> 💡 Télécharge `mon-ecole-eleves.db` dans la sidebar, ouvre-la dans DB Browser for SQLite, puis va dans l'onglet **Exécuter le SQL**. Tape chaque requête dans la zone du haut et exécute-la avec ▶ ou Ctrl+Entrée.

🎯 Objectifs

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

1. compter des lignes avec `COUNT(*)` et expliquer la différence avec `COUNT(colonne)` ;
2. calculer un minimum, un maximum, une somme et une moyenne ;
3. arrondir un résultat avec `ROUND` ;
4. combiner une fonction d'agrégation avec `WHERE` ;
5. obtenir un résultat par groupe avec `GROUP BY`, et trier ces groupes ;
6. écrire les clauses dans l'ordre imposé, `GROUP BY` compris.

Compter les lignes avec COUNT(*)

La base contient une table `eleve`, avec une ligne par élève. Pour la lire, on écrit une **requête** : `SELECT` (*sélectionne*) indique ce qu'on veut afficher, `FROM` (*depuis*) indique la table où chercher.

Au lieu d'afficher des colonnes, on peut demander à SQLite de compter les lignes :

```sql
SELECT COUNT(*)
FROM eleve;
```

```
COUNT(*)
--------
12
```

`COUNT` veut dire *compter*. L'étoile entre parenthèses signifie « les lignes entières » : `COUNT(*)` compte toutes les lignes, quel que soit leur contenu.

Le résultat ne contient qu'**une seule ligne**. Les douze élèves ont été résumés en un seul nombre.

Le nom de la colonne, `COUNT(*)`, n'est pas très parlant. `AS`, qui veut dire *en tant que*, donne un autre nom à la colonne dans le résultat :

```sql
SELECT COUNT(*) AS nb_eleves
FROM eleve;
```

```
nb_eleves
---------
12
```

> 📖 **NOUVELLE NOTION : LA FONCTION D'AGRÉGATION**
>
> Une **fonction d'agrégation** calcule **une seule valeur à partir d'un ensemble de lignes** : un nombre de lignes, une somme, une moyenne, un minimum, un maximum. Au lieu d'afficher les lignes, elle les résume. `COUNT`, `SUM`, `AVG`, `MIN` et `MAX` sont des fonctions d'agrégation.
>
> En anglais : GB ***aggregate function***.

COUNT(*) ou COUNT(colonne)

Combien d'élèves ont une adresse email ? Place le nom de la colonne entre les parenthèses, et compare avec `COUNT(*)` :

```sql
SELECT COUNT(*) AS nb_lignes,
       COUNT(email) AS nb_emails
FROM eleve;
```

```
nb_lignes | nb_emails
----------+----------
12        | 9
```

Douze lignes, mais seulement neuf adresses. Trois élèves n'ont pas d'adresse email : leur case contient `NULL`, la marque d'une valeur absente ou inconnue.

`COUNT(*)` compte les **lignes**. `COUNT(email)` compte les **valeurs présentes** dans la colonne `email` : les cases `NULL` ne sont pas comptées.

⚠️ Pour compter des élèves, utilise `COUNT(*)` . Si tu écris `COUNT(email)` par habitude, tu obtiens 9 au lieu de 12, sans aucun message d'erreur. Choisis `COUNT(colonne)` seulement quand la question est « combien de valeurs connues ? ».

Compter les valeurs différentes

Combien de communes différentes y a-t-il ? Ajoute `DISTINCT` (*différent*) dans les parenthèses : SQLite ne compte alors chaque valeur qu'une fois.

```sql
SELECT COUNT(DISTINCT commune) AS nb_communes
FROM eleve;
```

```
nb_communes
-----------
6
```

Six, alors que les élèves habitent cinq communes : Jodoigne, Wavre, Perwez, Hannut et Ramillies. La sixième vient d'une faute de frappe : la commune de Sarah Lambert a été encodée `jodoigne`, en minuscules. Pour SQLite, `Jodoigne` et `jodoigne` sont deux textes différents.

Minimum, maximum, somme et moyenne

Quatre autres fonctions d'agrégation travaillent sur les valeurs d'une colonne.

Fonction	Mot d'origine	Calcule
`MIN(colonne)`	minimum	la plus petite valeur
`MAX(colonne)`	maximum	la plus grande valeur
`SUM(colonne)`	*sum*, somme	le total des valeurs
`AVG(colonne)`	*average*, moyenne	la moyenne des valeurs

```sql
SELECT MIN(absences) AS minimum,
       MAX(absences) AS maximum
FROM eleve;
```

```
minimum | maximum
--------+--------
```

```
0       | 12
```

`MIN` et `MAX` fonctionnent aussi sur les dates. Comme elles sont écrites au format `AAAA-MM-JJ`, la plus petite date est la plus ancienne :

```sql
SELECT MIN(date_naissance) AS plus_ancienne,
       MAX(date_naissance) AS plus_recente
FROM eleve;
```

```
plus_ancienne | plus_recente
--------------+-------------
2009-09-08    | 2010-09-01
```

Le total et la moyenne des absences :

```sql
SELECT SUM(absences) AS total,
       AVG(absences) AS moyenne
FROM eleve;
```

```
total | moyenne
------+--------
36    | 3.0
```

36 demi-jours d'absence au total, pour 12 élèves : 3 en moyenne.

Arrondir avec ROUND

Calcule la moyenne des absences des élèves nés en 2010 :

```sql
SELECT AVG(absences) AS moyenne
FROM eleve
WHERE date_naissance >= '2010-01-01';
```

```
moyenne
-------
2.25
```

Pour afficher une seule décimale, entoure le calcul de `ROUND`, qui veut dire *arrondir*. Le deuxième nombre indique combien de décimales garder :

```sql
SELECT ROUND(AVG(absences), 1) AS moyenne
FROM eleve
WHERE date_naissance >= '2010-01-01';
```

```
moyenne
-------
2.3
```

Une fonction d'agrégation avec WHERE

La requête précédente contenait un `WHERE` (*où*). SQLite commence toujours par filtrer les lignes avec `WHERE`, puis il calcule sur les lignes qui restent.

Compte les élèves de Jodoigne :

```
SELECT COUNT(*) AS nb
FROM eleve
WHERE commune = 'Jodoigne';
```

```
nb
--
4
```

Quatre, alors que cinq élèves habitent Jodoigne. La faute de frappe `jodoigne` fausse le compte. Avec `LIKE`, qui ne fait pas la différence entre majuscules et minuscules pour les lettres sans accent, on retrouve les cinq :

```
SELECT COUNT(*) AS nb
FROM eleve
WHERE commune LIKE 'jodoigne';
```

```
nb
--
5
```

> 🧠 Une fonction d'agrégation calcule sur les lignes gardées par `WHERE`. Si le filtre rate une ligne, le calcul est faux, sans aucun avertissement.

Un résultat par groupe avec GROUP BY

Combien d'élèves habitent **chaque** commune ? Avec ce que tu connais, il faudrait écrire une requête par commune. `GROUP BY`, qui veut dire *regrouper par*, fait le travail en une fois :

```
SELECT commune, COUNT(*) AS nb_eleves
FROM eleve
GROUP BY commune;
```

```
commune   | nb_eleves
----------+----------
Hannut    | 2
Jodoigne  | 4
```

```
Perwez    | 2
Ramillies | 1
Wavre     | 2
jodoigne  | 1
```

SQLite range d'abord les lignes en paquets : un paquet par valeur de `commune`. Ensuite, il applique `COUNT(*)` à chaque paquet séparément, et renvoie une ligne par paquet.

```
Hannut    : Englebert, Jacques              → 2
Jodoigne  : Adam, Delvaux, Fontaine, Istas  → 4
Perwez    : Charlier, Hubert                → 2
Ramillies : Kevers                          → 1
Wavre     : Bastin, Gérard                  → 2
jodoigne  : Lambert                         → 1
```

> 📖 **NOUVELLE NOTION : LE REGROUPEMENT AVEC GROUP BY**
>
> `GROUP BY colonne` (*regrouper par*) range les lignes en **groupes** qui ont la même valeur dans cette colonne. Les fonctions d'agrégation sont alors calculées **pour chaque groupe**, et le résultat contient une ligne par groupe.
>
> En anglais : GB ***group by***.

Encore une fois, la faute de frappe se voit : `jodoigne` forme son propre groupe, avec une seule élève. Pour SQLite, c'est une commune à part.

Plusieurs calculs par groupe

Une même requête peut calculer plusieurs agrégats pour chaque groupe. Voici, par commune, le nombre d'élèves et la moyenne des absences :

```sql
SELECT commune,
       COUNT(*) AS nb_eleves,
       ROUND(AVG(absences), 1) AS moyenne
FROM eleve
GROUP BY commune
ORDER BY commune;
```

```
commune   | nb_eleves | moyenne
----------+-----------+--------
Hannut    | 2         | 6.0
Jodoigne  | 4         | 3.8
Perwez    | 2         | 2.5
Ramillies | 1         | 1.0
Wavre     | 2         | 1.5
jodoigne  | 1         | 0.0
```

Trier les groupes

Pour ranger les communes de la plus peuplée à la moins peuplée, on trie sur le résultat du calcul. `ORDER BY` (*ordonner par*) peut utiliser le nom donné avec `AS` :

```sql
SELECT commune, COUNT(*) AS nb_eleves
FROM eleve
GROUP BY commune
ORDER BY nb_eleves DESC, commune;
```

```
commune   | nb_eleves
----------+----------
Jodoigne  | 4
Hannut    | 2
Perwez    | 2
Wavre     | 2
Ramillies | 1
jodoigne  | 1
```

`DESC` trie du plus grand au plus petit. La deuxième colonne de tri, `commune`, départage les communes qui ont le même nombre d'élèves.

Filtrer avant de regrouper

`WHERE` s'applique **avant** le regroupement. Voici, par commune, le nombre d'élèves qui ont au moins une absence :

```sql
SELECT commune, COUNT(*) AS nb_eleves
FROM eleve
WHERE absences > 0
GROUP BY commune
ORDER BY commune;
```

```
commune   | nb_eleves
----------+----------
Hannut    | 2
Jodoigne  | 3
Perwez    | 1
Ramillies | 1
Wavre     | 1
```

Le groupe `jodoigne` a disparu : sa seule élève n'a aucune absence, elle a été écartée par `WHERE` avant la formation des groupes.

L'ordre des clauses avec GROUP BY

Une requête est faite de **clauses**, des morceaux qui commencent chacun par un mot-clé. `GROUP BY` prend place entre `WHERE` et `ORDER BY` :

```sql
SELECT ...      -- ce que je veux voir
FROM ...        -- où sont les données
WHERE ...       -- quelles lignes garder avant de regrouper
GROUP BY ...    -- comment regrouper
ORDER BY ...    -- dans quel ordre
LIMIT ...;      -- combien de lignes
```

Placé ailleurs, `GROUP BY` provoque une erreur :

```sql
SELECT commune, COUNT(*)
FROM eleve
GROUP BY commune
WHERE absences > 0;
```

```
near "WHERE": syntax error
```

Autre erreur fréquente : vouloir filtrer sur un agrégat avec `WHERE`.

```sql
SELECT commune, COUNT(*)
FROM eleve
WHERE COUNT(*) >= 2
GROUP BY commune;
```

```
misuse of aggregate: COUNT()
```

Le message se lit *mauvaise utilisation de la fonction d'agrégation*. `WHERE` travaille ligne par ligne, **avant** que les groupes existent : à ce moment-là, il n'y a encore rien à compter.

> ℹ️ Pour filtrer les **GROUPES** après le calcul, SQL propose `HAVING`, qui veut dire *ayant*. Il se place juste après `GROUP BY`. Par exemple, `GROUP BY commune HAVING COUNT(*) >= 2` ne garde que les communes qui ont au moins deux élèves : Hannut, Jodoigne, Perwez et Wavre. `HAVING` n'est pas exigé à ce niveau, mais tu le rencontreras.

Les colonnes autorisées dans le SELECT

Que se passe-t-il si tu ajoutes le nom de l'élève dans une requête regroupée par commune ?

```sql
SELECT commune, nom, COUNT(*) AS nb_eleves
FROM eleve
GROUP BY commune;
```

```
commune   | nom       | nb_eleves
----------+-----------+----------
Hannut    | Englebert | 2
Jodoigne  | Adam      | 4
Perwez    | Charlier  | 2
Ramillies | Kevers    | 1
```

```
Wavre     | Bastin    | 2
jodoigne  | Lambert   | 1
```

SQLite accepte la requête, mais le résultat n'a pas de sens. Le groupe Jodoigne contient quatre élèves : pourquoi afficher Adam plutôt que Delvaux ? SQLite en a choisi un, sans règle sur laquelle tu peux compter.

> 🧠 Dans une requête avec `GROUP BY`, le `SELECT` ne doit contenir que les colonnes du regroupement et des fonctions d'agrégation.

> ℹ️ SQLite est tolérant sur ce point, d'autres logiciels non. MySQL, dans sa configuration habituelle, refuse cette requête avec un message d'erreur qui signale une colonne absente du `GROUP BY`. Une requête qui respecte la règle fonctionne partout.

🧪 Exercices

Pour chaque exercice, écris la requête, exécute-la, puis vérifie que le résultat répond vraiment à la question.

Exercice 1 — Compter et calculer ★☆☆☆☆

Écris une requête qui affiche, avec un nom de colonne parlant :

1. le nombre d'élèves nés en 2010 ;
2. le nombre d'élèves sans adresse email ;
3. le nombre d'élèves qui n'ont aucune absence ;
4. le total des absences des élèves de Wavre ;
5. la moyenne des absences des élèves nés en 2009, arrondie à une décimale.

Exercice 2 — Prédire le résultat ★★☆☆☆

Sans exécuter, écris le résultat de chaque requête. Exécute ensuite pour vérifier.

```sql
-- a)
SELECT COUNT(*), COUNT(email) FROM eleve WHERE commune = 'Hannut';

-- b)
SELECT COUNT(DISTINCT absences) FROM eleve;

-- c)
SELECT MIN(nom), MAX(nom) FROM eleve;

-- d)
SELECT COUNT(*) AS nb, AVG(absences) AS moyenne FROM eleve WHERE commune = 'Namur';
```

Pour la requête d), explique pourquoi les deux colonnes ne donnent pas le même genre de réponse.

Exercice 3 — Regrouper ★★☆☆☆

1. Affiche, pour chaque commune, le total des absences, de la commune la plus absente à la moins absente.
2. Affiche, pour chaque commune, le nombre d'élèves et le nombre d'adresses email connues.
3. Affiche la commune qui compte le plus d'élèves, et ce nombre, sur une seule ligne.
4. Dans les résultats des questions 1 et 2, combien de lignes obtiens-tu ? Combien de communes réelles ? Explique la différence.

Exercice 4 — Réparer des requêtes ★★★☆☆

Chacune de ces requêtes provoque une erreur ou ne répond pas à la question. Pour chacune : exécute-la, observe, explique ce qui ne va pas, puis corrige-la.

```sql
-- a) Le nombre d'élèves
SELECT COUNT(email) FROM eleve;

-- b) Le nombre d'élèves par commune
SELECT commune, COUNT(*) FROM eleve ORDER BY commune GROUP BY commune;

-- c) Le nombre d'élèves ayant au moins une absence, par commune
SELECT commune, COUNT(*) FROM eleve GROUP BY commune WHERE absences > 0;

-- d) Le nombre d'élèves par commune
SELECT commune, nom, COUNT(*) FROM eleve GROUP BY commune;
```

Exercice 5 — Mesurer l'effet d'une faute de frappe ★★★☆☆

1. Calcule la moyenne des absences des élèves de Jodoigne avec `WHERE commune = 'Jodoigne'`.
2. Calcule-la à nouveau avec `WHERE commune LIKE 'jodoigne'`.
3. Les deux résultats sont différents. Lequel est juste ? Explique d'où vient l'écart.
4. Un rapport envoyé à la commune de Jodoigne utilise la première requête. Quelle erreur contient-il ?

✍️ À retenir

- Une fonction d'agrégation résume un ensemble de lignes en une seule valeur : `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`.
- `COUNT(*)` compte les lignes ; `COUNT(colonne)` ne compte pas les `NULL` (ici 12 contre 9) ; `COUNT(DISTINCT colonne)` compte les valeurs différentes.
- `ROUND(valeur, n)` arrondit à `n` décimales.
- `WHERE` filtre les lignes **avant** le calcul.
- `GROUP BY colonne` (*regrouper par*) calcule les agrégats pour chaque groupe : une ligne par groupe.
- Ordre des clauses : `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `ORDER BY`, `LIMIT`.
- Avec `GROUP BY`, le `SELECT` ne contient que les colonnes regroupées et des agrégats.

Suite

Tu as maintenant tous les outils du niveau pour interroger une table. Pour les réviser en une fois, avec les définitions, la syntaxe et les pièges, ouvre la fiche à étudier.