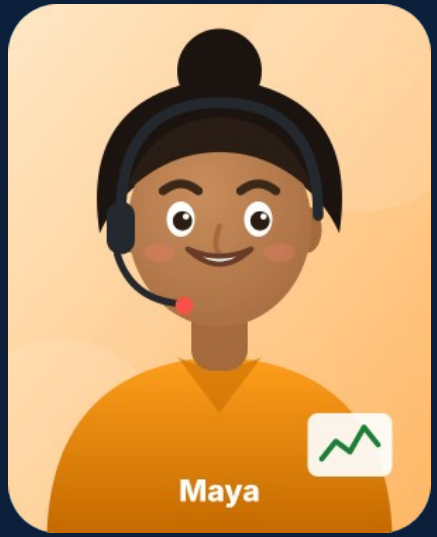
HEF-070 · MISSION 06 / 10

Jointures et agrégations

Les données sont dans plusieurs tables. Les jointures les rassemblent, les agrégations les résument.



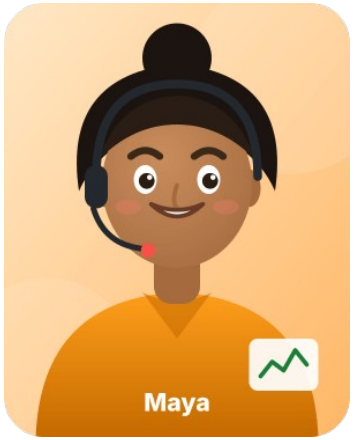


Badge : Assembleur de données

85 min · intermédiaire

L'histoire

Maya demande : « Combien de membres par cours ? Quels membres ne sont inscrits à rien ? Quel est le score moyen par ville, uniquement pour les villes de plus de 3 membres ? » Les données sont éparpillées dans quatre tables.



> *« C'est le moment d'apprendre la jointure, dit Sam, et de comprendre la différence entre ce qui est calculé ligne par ligne et ce qui est calculé par groupe. »*

Tu apprends les jointures, GROUP BY, HAVING, les sous-requêtes et les CTE.

Ta mission

À la fin, je serai capable de…

1. choisir entre INNER JOIN et LEFT JOIN selon la question
2. utiliser COUNT, SUM, AVG, MIN, MAX avec GROUP BY
3. distinguer WHERE et HAVING
4. écrire une sous-requête et une CTE
5. trouver les lignes « sans correspondance »

Défi d'échauffement

Défi d'échauffement

Tu as deux listes : les **élèves** (avec leur numéro) et les **notes** (numéro d'élève, matière, note). Certains élèves n'ont encore aucune note. (a) Pour un bulletin de **tous** les élèves, même sans note, comment assembles-tu les deux listes ? (b) Pour une liste des seuls élèves **qui ont des notes**, comment fais-tu ? (c) Comment trouver ceux qui n'ont **aucune** note ?

Le concept

La jointure

Une **jointure** (*JOIN*) combine des lignes de deux tables selon une condition.

```
SELECT m.nom, c.titre
FROM membres m
JOIN inscriptions i ON i.membre_id = m.id
JOIN cours c        ON c.code = i.cours_code;
```

Les types de jointures

Type	Résultat
`INNER JOIN` (ou `JOIN`)	seulement les lignes qui ont une correspondance des **deux** côtés
`LEFT JOIN`	**toutes** les lignes de gauche, avec NULL quand il n'y a pas de correspondance à droite

Le concept

Les types de jointures (suite)

Type	Résultat
`RIGHT JOIN`	l'inverse (peu utilisé : on inverse l'ordre des tables)
`FULL JOIN`	tout des deux côtés
`CROSS JOIN`	produit cartésien (toutes les combinaisons)

Pour trouver les « sans correspondance » : `LEFT JOIN` puis `WHERE droite.cle IS NULL`.

```
SELECT m.nom
FROM membres m
LEFT JOIN inscriptions i ON i.membre_id = m.id
WHERE i.membre_id IS NULL;           -- membres sans inscription
```

Le concept

Les agrégations

Fonction	Rôle
`COUNT(*)`	nombre de lignes
`COUNT(col)`	nombre de valeurs non NULL
`SUM`, `AVG`, `MIN`, `MAX`	calculs sur un ensemble

`GROUP BY` regroupe les lignes qui partagent une valeur ; l'agrégation s'applique **à chaque groupe**.

```
SELECT ville, COUNT(*) AS n, AVG(score) AS moyenne
FROM membres
GROUP BY ville;
```

Règle : toute colonne du `SELECT` doit être soit dans le `GROUP BY`, soit dans une fonction d'agrégation.

Le concept

WHERE et HAVING

	WHERE	HAVING
Quand	**avant** le regroupement, sur les lignes	**après** le regroupement, sur les groupes
Exemple	`WHERE score > 10`	`HAVING COUNT(*) > 3`

Sous-requêtes et CTE

Une **sous-requête** est une requête dans une requête.

```
SELECT nom FROM membres
WHERE score > (SELECT AVG(score) FROM membres);
```

Le concept

Sous-requêtes et CTE (suite)

Une **CTE** (*Common Table Expression*, clause `WITH`) nomme un résultat intermédiaire, ce qui rend les requêtes complexes lisibles.

```
WITH par_membre AS (
    SELECT membre_id, COUNT(*) AS nb FROM inscriptions GROUP BY membre_id
)
SELECT m.nom, p.nb FROM membres m JOIN par_membre p ON p.membre_id = m.id WHERE p.
   nb >= 2;
```

Le piège de la multiplication des lignes

Joindre deux tables « un-à-plusieurs » puis agréger peut **compter plusieurs fois** la même chose. Toujours se demander : *combien de lignes y a-t-il après la jointure ?*

Le concept

Le piège de la multiplication des lignes (suite)

🧠 ***Astuce*** *: écris la jointure sans agrégation d'abord, regarde quelques lignes, puis ajoute le GROUP BY.*

Exemple pas à pas

Exemple pas à pas

Problème : répondre aux questions de Maya.

```
import sqlite3

bd = sqlite3.connect(":memory:")
bd.executescript('''
CREATE TABLE membres (id INTEGER PRIMARY KEY, nom TEXT, ville TEXT, score INTEGER)
   ;
CREATE TABLE cours (code TEXT PRIMARY KEY, titre TEXT);
CREATE TABLE inscriptions (membre_id INTEGER, cours_code TEXT);
INSERT INTO membres (nom, ville, score) VALUES ('Ada','Lyon',90), ('Bob','Lille',4
   5), ('Cléo','Lyon',60),
                                              ('Dan','Nice',10), ('Eve','Lyon',3
   0), ('Félix','Lyon',75);
INSERT INTO cours VALUES ('HEF-000','Mentalité'), ('HEF-010','Algorithmes'), ('HEF
   -020','Structures');
INSERT INTO inscriptions VALUES (1,'HEF-000'), (1,'HEF-010'), (2,'HEF-000'), (3,'H
   EF-000'), (3,'HEF-010'), (3,'HEF-020');
```

Exemple pas à pas

Exemple pas à pas (suite)

```
''')
q = lambda sql: bd.execute(sql).fetchall()

# Combien de membres par cours (y compris les cours sans inscrit) ?
print(q('''SELECT c.code, COUNT(i.membre_id) AS nb
           FROM cours c LEFT JOIN inscriptions i ON i.cours_code = c.code
           GROUP BY c.code ORDER BY c.code'''))

# Membres sans aucune inscription
print(q('''SELECT m.nom FROM membres m LEFT JOIN inscriptions i ON i.membre_id = m
   .id
           WHERE i.membre_id IS NULL ORDER BY m.nom'''))

# Villes de plus de 3 membres et leur score moyen
print(q('''SELECT ville, COUNT(*) AS n, ROUND(AVG(score), 1) FROM membres GROUP BY
    ville HAVING COUNT(*) > 3'''))

# Membres au-dessus de la moyenne
print(q("SELECT nom FROM membres WHERE score > (SELECT AVG(score) FROM membres) OR
```

Exemple pas à pas

Exemple pas à pas (suite)

```
   DER BY nom"))

# CTE : membres inscrits à au moins 2 cours
print(q('''WITH nb AS (SELECT membre_id, COUNT(*) AS n FROM inscriptions GROUP BY
   membre_id)
           SELECT m.nom, nb.n FROM membres m JOIN nb ON nb.membre_id = m.id WHERE
   nb.n >= 2 ORDER BY m.nom'''))
```

Lecture :

- `LEFT JOIN` + `COUNT(i.membre_id)` : `COUNT(colonne)` ignore les NULL, donc un cours sans inscrit affiche **0** (avec `COUNT(*)` il afficherait 1).
- `HAVING COUNT(*) > 3` filtre sur les **groupes** : seule Lyon (4 membres) reste.
- La sous-requête calcule la moyenne une fois, puis on compare chaque score.

À toi, avec aide

Avec les tables de l'exemple, écris :

1. le nombre de cours suivis par chaque membre, **y compris** ceux qui n'en suivent aucun (0) ;
2. la liste des cours que **personne** n'a suivis ;
3. le membre qui suit le plus de cours (utilise une CTE) ;
4. la ville dont le score moyen est le plus élevé.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Pour 1, `LEFT JOIN` depuis `membres` et `COUNT(i.cours_code)`.

Indice 2

Pour 2, `LEFT JOIN` depuis `cours` puis `WHERE i.membre_id IS NULL`.

Indice 3

Pour 3, une CTE qui compte, puis `ORDER BY ... DESC LIMIT 1`.

À toi, avec aide · réponse

Réponse

```
import sqlite3

bd = sqlite3.connect(":memory:")
bd.executescript('''
CREATE TABLE membres (id INTEGER PRIMARY KEY, nom TEXT, ville TEXT, score INTEGER)
   ;
CREATE TABLE cours (code TEXT PRIMARY KEY, titre TEXT);
CREATE TABLE inscriptions (membre_id INTEGER, cours_code TEXT);
INSERT INTO membres (nom, ville, score) VALUES ('Ada','Lyon',90), ('Bob','Lille',4
   5), ('Cléo','Lyon',60), ('Dan','Nice',10), ('Eve','Lyon',30), ('Félix','Lyon',7
   5);
INSERT INTO cours VALUES ('HEF-000','Mentalité'), ('HEF-010','Algorithmes'), ('HEF
   -020','Structures'), ('HEF-030','Complexité');
INSERT INTO inscriptions VALUES (1,'HEF-000'), (1,'HEF-010'), (2,'HEF-000'), (3,'H
   EF-000'), (3,'HEF-010'), (3,'HEF-020');
''')
q = lambda s: bd.execute(s).fetchall()
print(q("SELECT m.nom, COUNT(i.cours_code) FROM membres m LEFT JOIN inscriptions i
    ON i.membre_id = m.id GROUP BY m.id ORDER BY m.id"))
```

À toi, avec aide · réponse

Réponse (suite)

```
print(q("SELECT c.code FROM cours c LEFT JOIN inscriptions i ON i.cours_code = c.c
   ode WHERE i.membre_id IS NULL"))
print(q('''WITH nb AS (SELECT membre_id, COUNT(*) AS n FROM inscriptions GROUP BY
   membre_id)
           SELECT m.nom, nb.n FROM membres m JOIN nb ON nb.membre_id = m.id ORDER
   BY nb.n DESC LIMIT 1'''))
print(q("SELECT ville, AVG(score) AS moy FROM membres GROUP BY ville ORDER BY moy
   DESC LIMIT 1"))
```

À toi, en solo

Pièges des jointures. Avec les tables `membres`, `inscriptions` et une table `paiements(membre_id, montant_centimes)` (un membre peut avoir plusieurs paiements et plusieurs inscriptions) :

1. Écris une requête naïve « total payé et nombre d'inscriptions par membre » avec **deux jointures** et un seul `GROUP BY`. Montre qu'elle **gonfle** les résultats (chaque paiement est compté autant de fois qu'il y a d'inscriptions).
2. Corrige avec des sous-requêtes (ou des CTE) qui agrègent **avant** de joindre.
3. Vérifie la correction avec un calcul en Python sur les données brutes.

À toi, en solo · réponse

Pistes de correction

```
import sqlite3

bd = sqlite3.connect(":memory:")
bd.executescript('''
CREATE TABLE membres (id INTEGER PRIMARY KEY, nom TEXT);
CREATE TABLE inscriptions (membre_id INTEGER, cours_code TEXT);
CREATE TABLE paiements (membre_id INTEGER, montant_centimes INTEGER);
INSERT INTO membres (nom) VALUES ('Ada'), ('Bob');
INSERT INTO inscriptions VALUES (1,'A'), (1,'B'), (1,'C'), (2,'A');
INSERT INTO paiements VALUES (1, 1000), (1, 500), (2, 2000);
''')
naif = bd.execute('''SELECT m.nom, SUM(p.montant_centimes), COUNT(i.cours_code)
                     FROM membres m JOIN inscriptions i ON i.membre_id = m.id JOI
   N paiements p ON p.membre_id = m.id
                     GROUP BY m.id ORDER BY m.id''').fetchall()
juste = bd.execute('''WITH ins AS (SELECT membre_id, COUNT(*) AS nb FROM inscripti
   ons GROUP BY membre_id),
                           pay AS (SELECT membre_id, SUM(montant_centimes) AS tota
   l FROM paiements GROUP BY membre_id)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
                    SELECT m.nom, pay.total, ins.nb FROM membres m
                    LEFT JOIN ins ON ins.membre_id = m.id LEFT JOIN pay ON pay.m
   embre_id = m.id ORDER BY m.id''').fetchall()
print("naïf :", naif)       # Ada: 4500 (au lieu de 1500) et 6 (au lieu de 3)
print("juste:", juste)
assert juste == [("Ada", 1500, 3), ("Bob", 2000, 1)]
```

Ada a 3 inscriptions et 2 paiements : la jointure produit 3 × 2 = 6 lignes, ce qui multiplie les sommes. **Règle** : agréger chaque « côté plusieurs » séparément, puis joindre.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Joindre deux « un-à-plusieurs » puis agréger	Résultats gonflés	Agréger avant de joindre
`COUNT(*)` avec LEFT JOIN	Compte 1 pour les lignes sans correspondance	`COUNT(colonne_de_droite)`
`WHERE` au lieu de `HAVING`	Erreur ou mauvais filtre sur les groupes	WHERE = lignes, HAVING = groupes
Colonne non agrégée dans `SELECT`	Résultat arbitraire ou erreur	Dans GROUP BY ou dans une fonction

Mini-boss

Mini-boss

Rapport d'activité de la plateforme. Avec les tables `membres`, `cours`, `missions`, `inscriptions`, `missions_terminees(membre_id, mission_id, xp, terminee_le)` (génère 40 membres, 4 cours, 30 missions et environ 300 missions terminées), écris :

1. le classement des 10 membres avec le plus d'XP, avec leur rang et le nombre de missions terminées ;
2. pour chaque cours, le taux de complétion moyen (missions terminées / missions du cours) des inscrits ;
3. les membres inscrits à un cours mais qui n'ont **aucune** mission terminée dans ce cours ;
4. l'évolution mensuelle de l'XP total ;
5. les membres au-dessus de la moyenne de leur propre ville (sous-requête corrélée).

Vérifie chaque résultat avec un calcul Python indépendant sur les mêmes données.

Mini-boss · réponse

Pistes de correction

Un bon travail génère les données avec une graine fixe, écrit chaque requête avec des CTE nommées (`pts`, `nb_missions`, `terminees`), traite les `LEFT JOIN` et `COALESCE` pour les membres sans activité, agrège avant de joindre pour éviter la multiplication des lignes, utilise `substr(terminee_le, 1, 7)` pour le mois et `WHERE xp_membre > (SELECT AVG(xp_membre) FROM ... WHERE ville = m.ville)` pour la sous-requête corrélée. La vérification Python recalcule chaque résultat avec des dictionnaires et compare.

Quiz éclair

Quiz éclair

1. Un LEFT JOIN renvoie :

A. uniquement les lignes avec correspondance
B. toutes les lignes de gauche, avec NULL s'il n'y a pas de correspondance
C. les lignes de droite seulement

2. HAVING filtre :

A. les lignes avant le regroupement
B. les colonnes
C. les groupes après le GROUP BY

3. COUNT(colonne) :

A. ignore les valeurs NULL

Quiz éclair

Quiz éclair (suite)

B. compte aussi les NULL
C. est identique à COUNT(*)

4. 🔁 *Révision mission 05* : Une contrainte CHECK :

A. crée un index
B. impose une condition sur les valeurs
C. relie deux tables

5. 🔁 *Révision mission 05* : ON DELETE CASCADE signifie :

A. refuser la suppression
B. mettre la référence à NULL
C. supprimer aussi les lignes qui référencent la ligne supprimée

Quiz éclair · réponse

Réponses

1. **B**.
2. **C**.
3. **A**.
4. **B**.
5. **C**.

Je réfléchis

Je réfléchis

1. Quelle question de ta vie nécessiterait de croiser deux listes (qui a fait quoi, qui n'a rien fait) ?
2. Pourquoi vérifier le nombre de lignes après une jointure est-il un bon réflexe ?

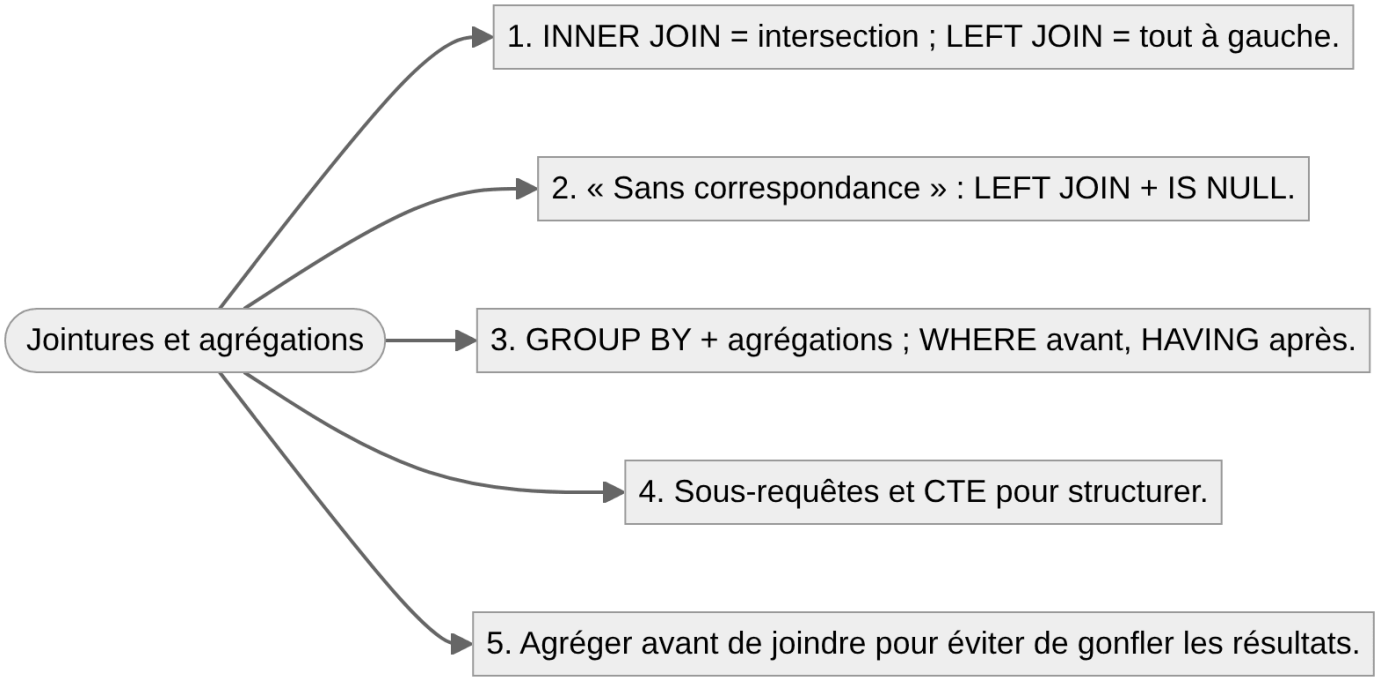
À retenir

À retenir

1. INNER JOIN = intersection ; LEFT JOIN = tout à gauche.
2. « Sans correspondance » : LEFT JOIN + IS NULL.
3. GROUP BY + agrégations ; WHERE avant, HAVING après.
4. Sous-requêtes et CTE pour structurer.
5. Agréger avant de joindre pour éviter de gonfler les résultats.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Assembleur de données

Prochaine mission : Index et plans d'exécution

