

Séance à distance — TP3

SQL : clés et jointures (rappels), opérations ensemblistes, regroupements

— Version corrigée —

Déroulé conseillé : Partie 1, rappels sur les clés et les jointures (≈ 40 min) puis exercice 1 sur papier (≈ 20 min) ; Partie 2, nouvelles notions (≈ 40 min) puis exercice 2 sur papier (≈ 15 min) ; exercices 3, 4 et 5 sur machine (le reste de la séance, et à terminer pour la séance suivante).
Notebook : <https://capitale2.ac-paris.fr/web/c/c5ca-12201317/mcer>. **Exécutez d'abord la première cellule** (elle crée toutes les tables). La base est effacée à chaque fermeture du notebook : il faudra la réexécuter à chaque fois. Chaque exemple du cours a sa cellule dans le notebook : **exécutez-les au fil de la lecture.**

La base bibliotheque

Tous les exemples du cours utilisent la base ci-dessous, qui décrit les livres d'une petite bibliothèque, leurs auteurs, les adhérents et les emprunts. Elle est déjà présente dans le notebook.

Table auteurs

id	nom	prenom	nationalite	naissance
1	Hugo	Victor	française	1802
2	Verne	Jules	française	1828
3	Zola	Émile	française	1840
4	Camus	Albert	française	1913
5	Shelley	Mary	britannique	1797
6	Orwell	George	britannique	1903
7	Dumas	Alexandre	française	1802
8	Saint-Exupéry	Antoine de	française	1900
9	Christie	Agatha	britannique	1890
10	Proust	Marcel	française	1871

Table adherents

id	nom	prenom	ville
1	Martin	Léa	Saint-Étienne
2	Bernard	Hugo	Firminy
3	Petit	Sarah	Saint-Étienne
4	Durand	Tom	Roanne
5	Moreau	Inès	Firminy

Table livres

id	titre	annee	genre	id_auteur
1	Les Misérables	1862	roman	1
2	Notre-Dame de Paris	1831	roman	1
3	Vingt mille lieues sous les mers	1870	aventure	2
4	Le Tour du monde en 80 jours	1872	aventure	2
5	Germinal	1885	roman	3
6	L'Assommoir	1877	roman	3
7	L'Étranger	1942	roman	4
8	La Peste	1947	roman	4
9	Frankenstein	1818	science-fiction	5
10	1984	1949	science-fiction	6
11	La Ferme des animaux	1945	roman	6
12	Le Comte de Monte-Cristo	1844	aventure	7
13	Les Trois Mousquetaires	1844	aventure	7
14	Le Petit Prince	1943	conte	8
15	Le Crime de l'Orient-Express	1934	policier	9
16	Mort sur le Nil	1937	policier	9

Table emprunts

id	id_livre	id_adherent	date_emprunt
1	14	1	2026-09-02
2	7	1	2026-09-02
3	3	2	2026-09-05
4	10	3	2026-09-08
5	14	3	2026-09-10
6	1	4	2026-09-12
7	15	2	2026-09-15
8	8	1	2026-09-20
9	12	4	2026-09-22
10	10	2	2026-09-25
11	16	3	2026-09-28
12	4	4	2026-10-01

1 Rappels : tables, clés et jointures

1.1 Vocabulaire et schéma

Définition : table, attribut, schéma

Une base de données relationnelle est formée de **tables** (on dit aussi *relations*). Chaque colonne d'une table est un **attribut** ; ses valeurs appartiennent à un **domaine** (on dit aussi *type*) : entier (INTEGER), flottant (REAL) ou chaîne de caractères (TEXT). Chaque ligne est un **enregistrement**.

Le **schéma** d'une table est la donnée de son nom et de ses attributs avec leurs domaines, par exemple :

```
livres(id INTEGER, titre TEXT, annee INTEGER, genre TEXT, id_auteur INTEGER)
```

Pour écrire une requête, on n'a pas besoin de connaître toutes les lignes d'une base : **le schéma suffit** (à condition de savoir aussi comment les tables sont reliées, voir la suite).

1.2 Pourquoi plusieurs tables ?

Imaginons que la bibliothèque range tout dans une seule table :

titre	auteur	nationalite	naissance
Les Misérables	Victor Hugo	française	1802
Notre-Dame de Paris	Victor Hugo	française	1802
Quatrevingt-treize	V. Hugo	française	1820

Les informations sur Victor Hugo sont **répétées** pour chacun de ses livres. C'est encombrant, et surtout dangereux : à la troisième ligne, une faute de frappe rend la base **incohérente** (Hugo est-il né en 1802 ou en 1820 ?). On range donc les auteurs dans **leur propre table**, où chaque auteur n'apparaît qu'une fois, et chaque livre indique seulement *quel* auteur l'a écrit. Pour cela, il faut pouvoir désigner un auteur sans ambiguïté : c'est le rôle de la clé primaire.

1.3 Clé primaire

Définition : clé primaire

Une **clé primaire** d'une table est un attribut (ou un groupe d'attributs) dont la valeur **identifie chaque ligne de façon unique** : deux lignes différentes ne peuvent jamais avoir la même valeur de clé primaire.

Exemple 1 : des clés primaires dans la vie courante

Le numéro de sécurité sociale d'une personne, la plaque d'immatriculation d'une voiture, le numéro INE d'un élève, le code postal... non, pas le code postal : plusieurs petites communes partagent souvent le même code postal. Une clé primaire doit être unique *pour toutes les lignes possibles*, pas seulement pour celles qu'on a aujourd'hui.

Exemple 2 : les clés primaires de la base bibliotheque

Dans chacune des quatre tables, la clé primaire est l'attribut `id` : un numéro attribué à chaque ligne.

Pourquoi ne pas prendre le nom de l'auteur ? Parce que rien n'empêche deux auteurs

d'avoir le même nom, et même le même nom *et* le même prénom : Alexandre Dumas (l'auteur des *Trois Mousquetaires*) et son fils, Alexandre Dumas lui aussi (l'auteur de *La Dame aux camélias*). De même, deux livres peuvent avoir le même titre. Un numéro, attribué par la base, est sûr d'être unique.

1.4 Clé étrangère

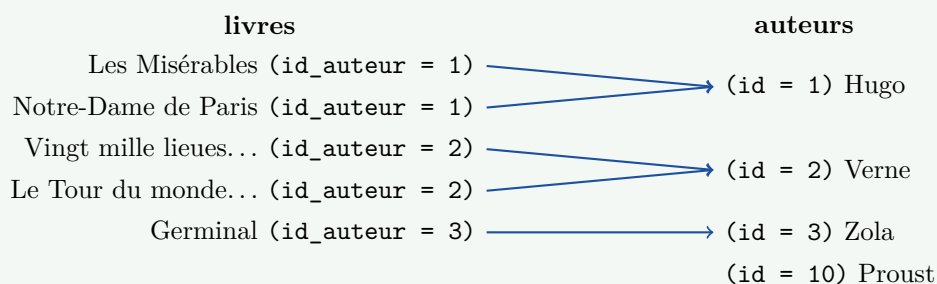
Définition : clé étrangère

Une **clé étrangère** est un attribut d'une table dont les valeurs sont des valeurs de la **clé primaire d'une autre table**. Elle sert à **relier** une ligne à une ligne d'une autre table : on dit qu'elle *fait référence* à cette clé primaire.

Exemple 3 : la clé étrangère livres.id_auteur

Dans la table livres, l'attribut id_auteur est une clé étrangère qui fait référence à l'attribut id de la table auteurs. Le livre *Germinal* a id_auteur = 3 : son auteur est la ligne de auteurs dont l'id vaut 3, c'est-à-dire Émile Zola.

On peut représenter ce lien par des flèches (extrait) :



Chaque livre a **un seul** auteur (une seule flèche part de chaque livre), mais un auteur peut avoir **plusieurs** livres, ou aucun (Proust). On dit que c'est une association « un auteur – plusieurs livres ». La clé étrangère est toujours du côté « plusieurs » : c'est le livre qui indique son auteur, pas l'inverse (un auteur ayant plusieurs livres, il faudrait plusieurs cases).

Exemple 4 : une table de liaison

Un livre peut être emprunté par plusieurs adhérents, et un adhérent peut emprunter plusieurs livres : l'association est « plusieurs – plusieurs ». On ne peut pas la représenter par une simple clé étrangère dans livres ni dans adherents. On crée donc une table intermédiaire, **emprunts**, qui contient **deux clés étrangères** :

- emprunts.id_livre fait référence à livres.id;
- emprunts.id_adherent fait référence à adherents.id.

Chaque ligne de emprunts relie *un* livre à *un* adhérent. Par exemple la ligne (5, 14, 3, '2026-09-10') signifie : « le livre 14 (*Le Petit Prince*) a été emprunté par l'adhérent 3 (Sarah Petit) le 10 septembre ». L'association « plusieurs – plusieurs » est ainsi découpée en deux associations « un – plusieurs ».

Exemple 5 (plus difficile) : une clé primaire formée de plusieurs attributs

Une clé primaire peut être formée de **plusieurs** attributs : c'est alors la *combinaison* des valeurs qui doit être unique. Supposons que la table emprunts n'ait pas de colonne id, et qu'un adhérent ne puisse pas emprunter deux fois le même livre le même jour. Son schéma serait :

```
emprunts(id_livre INTEGER, id_adherent INTEGER, date_emprunt TEXT)
```

Aucun attribut seul ne convient comme clé primaire : `id_livre` n'est pas unique (*Le Petit Prince* a été emprunté deux fois), `id_adherent` non plus (Léa a fait trois emprunts), `date_emprunt` non plus (deux emprunts le 2 septembre). Mais le *triplet* (`id_livre`, `id_adherent`, `date_emprunt`) identifie chaque emprunt : c'est la clé primaire. Vous rencontrerez des clés de ce type dans les exercices 4 et 5.

1.5 La jointure

Les données sont maintenant bien rangées... mais dispersées. Pour afficher « le titre de chaque livre avec le nom de son auteur », il faut aller chercher l'information dans **deux** tables à la fois, en suivant la clé étrangère. C'est le rôle de la **jointure**.

Étape 1 : toutes les associations possibles (produit cartésien)

Pour comprendre, prenons seulement les deux premiers auteurs et les trois premiers livres. La requête

```
SELECT * FROM auteurs, livres;
```

(avec une virgule entre les deux tables) associe **chaque** ligne de **auteurs** à **chaque** ligne de **livres** : c'est le **produit cartésien**. Avec 2 auteurs et 3 livres, on obtient $2 \times 3 = 6$ lignes (on ne montre que quelques colonnes) :

auteurs.id	auteurs.nom	livres.id	livres.titre	livres.id_auteur	
1	Hugo	1	Les Misérables	1	oui
1	Hugo	2	Notre-Dame de Paris	1	oui
1	Hugo	3	Vingt mille lieues...	2	non
2	Verne	1	Les Misérables	1	non
2	Verne	2	Notre-Dame de Paris	1	non
2	Verne	3	Vingt mille lieues...	2	oui

La plupart de ces lignes n'ont aucun sens (« Verne a écrit Les Misérables »)! Les seules lignes *cohérentes* sont celles où `livres.id_auteur` est égal à `auteurs.id` (dernière colonne). Sur toute la base, le produit cartésien contient $10 \times 16 = 160$ lignes, dont seulement 16 sont cohérentes.

Étape 2 : ne garder que les lignes cohérentes (jointure)

Définition : jointure

La requête

```
SELECT ... FROM table1 JOIN table2 ON condition;
```

associe chaque ligne de `table1` à chaque ligne de `table2` **pour laquelle la condition est vraie**. Autrement dit : on forme toutes les associations possibles, et on ne garde que celles qui vérifient la condition. Le résultat est une « grande table » qui contient toutes les colonnes des deux tables.

En pratique, la condition est presque toujours

clé étrangère = clé primaire à laquelle elle fait référence.

Exemple 6 : la jointure livres – auteurs

```
SELECT livres.titre, auteurs.nom
FROM livres JOIN auteurs ON livres.id_auteur = auteurs.id;
```

Résultat :

titre	nom
Les Misérables	Hugo
Notre-Dame de Paris	Hugo
Vingt mille lieues sous les mers	Verne
...	... (16 lignes au total)

Chaque livre est « recollé » à son auteur. Hugo apparaît deux fois (une fois par livre). Proust n'apparaît pas : aucun livre n'a `id_auteur = 10`, la ligne de Proust ne trouve aucune correspondance.

La notation Table.Attribut

Dans une requête avec plusieurs tables, on désigne une colonne par

`table.attribut`

d'abord la table, puis l'attribut, séparés par un point : `livres.titre` se lit « dans la table `livres`, la colonne `titre` ». On va du plus général au plus précis, du contenant vers le contenu, exactement comme :

- en Python, `math.sqrt` : d'abord le module `math`, puis la fonction `sqrt` qu'il contient ;
- un chemin de fichier `Documents/tp3.sql` : d'abord le dossier, puis le fichier ;
- une adresse postale lue à l'envers : France, Saint-Étienne, rue...

Attention, erreur fréquente

On écrit `livres.titre`, et jamais ~~`titre.livres`~~. En SQL, c'est toujours la table d'abord, puis l'attribut.

Si on se trompe, SQLite comprend que `titre` est un nom de table et cherche dedans une colonne `livres`. Il répond par l'erreur :

```
no such column: titre.livres
```

Quand vous voyez ce message, vérifiez d'abord l'ordre table/attribut !

Pourquoi préfixer ? Les tables `livres` et `auteurs` ont toutes les deux une colonne `id`. La requête

```
SELECT id, titre FROM livres JOIN auteurs ON livres.id_auteur = auteurs.id;
```

provoque l'erreur `ambiguous column name: id` : SQLite ne sait pas de quel `id` on parle. Le préfixe est donc **obligatoire** dès qu'un nom de colonne existe dans plusieurs des tables utilisées (ici `livres.id` ou `auteurs.id`).

Méthode pour écrire une requête avec jointure

À retenir

1. **Quelles colonnes** faut-il afficher ou tester ? Dans **quelles tables** sont-elles ?
2. **Quel chemin** relie ces tables ? On le lit sur les clés étrangères (s'il faut passer par une table intermédiaire, on l'ajoute).
3. On écrit FROM avec la première table, puis **un JOIN par table ajoutée**, avec la condition clé étrangère = clé primaire.
4. On ajoute WHERE, ORDER BY... et on choisit les colonnes du SELECT.

Exemple 7 : jointure et WHERE

« Les titres des livres d'aventure, avec le nom de leur auteur. »

1. On a besoin de livres.titre, livres.genre et auteurs.nom.
2. Lien : livres.id_auteur fait référence à auteurs.id.

```
SELECT livres.titre, auteurs.nom
FROM livres JOIN auteurs ON livres.id_auteur = auteurs.id
WHERE livres.genre = 'aventure';
```

Résultat :

Vingt mille lieues sous les mers (Verne), Le Tour du monde en 80 jours (Verne), Le Comte de Monte-Cristo (Dumas), Les Trois Mousquetaires (Dumas).

Exemple 8 : trois tables

« Qui a emprunté un roman policier ? Afficher le prénom et le nom de l'adhérent et le titre. »

1. On a besoin de adherents.prenom, adherents.nom, livres.titre et livres.genre.
2. Il n'y a pas de lien direct entre livres et adherents : on passe par emprunts, qui a une clé étrangère vers chacune des deux.

```
SELECT adherents.prenom, adherents.nom, livres.titre
FROM emprunts
JOIN livres ON emprunts.id_livre = livres.id
JOIN adherents ON emprunts.id_adherent = adherents.id
WHERE livres.genre = 'policier';
```

Résultat :

prenom	nom	titre
Hugo	Bernard	Le Crime de l'Orient-Express
Sarah	Petit	Mort sur le Nil

On part de la table de liaison **emprunts** ; chaque emprunt est recollé à son livre (premier JOIN) puis à son adhérent (second JOIN). On peut aussi écrire toutes les conditions dans un seul ON, reliées par AND :

```
SELECT adherents.prenom, adherents.nom, livres.titre
FROM emprunts JOIN livres JOIN adherents
ON emprunts.id_livre = livres.id AND emprunts.id_adherent = adherents.id
WHERE livres.genre = 'policier';
```

Les deux écritures sont équivalentes : il faut savoir lire et écrire les deux.

Attention, erreur fréquente

Relier les mauvaises colonnes. La requête suivante s'exécute *sans erreur* :

```
SELECT livres.titre, auteurs.nom  
FROM livres JOIN auteurs ON livres.id = auteurs.id;
```

mais elle associe le livre n°1 à l'auteur n°1, le livre n°2 à l'auteur n°2... : « Notre-Dame de Paris | Verne », « Vingt mille lieues sous les mers | Zola »... Le numéro d'un livre n'a rien à voir avec le numéro de son auteur. La condition relie toujours une **clé étrangère** à la **clé primaire qu'elle référence**. Une requête qui s'exécute n'est pas forcément juste : vérifiez toujours que le résultat a du sens !

1.6 Trois petits compléments

Exemple 9 : supprimer les doublons avec DISTINCT

SELECT nationalite FROM auteurs; renvoie 10 lignes (« française » sept fois, « britannique » trois fois). Avec DISTINCT, chaque ligne différente n'apparaît qu'**une seule fois** :

```
SELECT DISTINCT nationalite FROM auteurs;
```

Résultat :

française, britannique.

DISTINCT est souvent utile après une jointure, où une même valeur peut apparaître plusieurs fois.

Heures et dates. Dans nos bases, heures et dates sont des *textes* de la forme 'HH:MM' et 'AAAA-MM-JJ'. Comme les chiffres sont rangés du plus important au moins important, l'ordre alphabétique de ces textes coïncide avec l'ordre chronologique : WHERE Heure > '17:00' fonctionne (à condition d'écrire '09:00' et non '9:00').

Apostrophes. Les textes s'écrivent entre apostrophes simples : 'aventure'. Les guillemets doubles "... " servent, dans la norme SQL, à écrire des *noms de colonnes* : SQLite les accepte souvent pour des textes, mais ce n'est pas correct.

Hors programme (pour information)

Une case peut être vide : elle contient alors la valeur spéciale NULL. On la teste avec colonne IS NULL (et non = NULL, qui n'est jamais vrai). Cela sert dans une question de l'exercice 3.

2 Nouvelles notions

2.1 Opérations ensemblistes : UNION, INTERSECT, EXCEPT

Le résultat d'une requête est une table, que l'on peut voir comme un **ensemble de lignes**. On peut combiner les résultats de deux requêtes comme on combine des ensembles.

Définition : opérations ensemblistes

Si R1 et R2 sont deux requêtes SELECT qui renvoient **le même nombre de colonnes** :

R1 UNION R2	lignes présentes dans R1 ou dans R2	$R_1 \cup R_2$
R1 INTERSECT R2	lignes présentes dans R1 et dans R2	$R_1 \cap R_2$
R1 EXCEPT R2	lignes présentes dans R1 mais pas dans R2	$R_1 \setminus R_2$

Comme pour des ensembles, le résultat ne contient **pas de doublons**. Le point-virgule ne se met qu'à la toute fin.

Exemple 10 : UNION

Les genres des livres de Hugo *ou* de Verne :

```
SELECT livres.genre
FROM livres JOIN auteurs ON livres.id_auteur = auteurs.id
WHERE auteurs.nom = 'Hugo'
UNION
SELECT livres.genre
FROM livres JOIN auteurs ON livres.id_auteur = auteurs.id
WHERE auteurs.nom = 'Verne';
```

Résultat :

aventure, roman (et non « roman, roman, aventure, aventure »).

Ici, on aurait aussi pu écrire une seule requête avec OR (et un DISTINCT). Mais UNION permet aussi de réunir des résultats provenant de **tables différentes**, ce qu'un OR ne sait pas faire : SELECT nom FROM auteurs UNION SELECT nom FROM adherents; donne tous les noms de famille de la base.

Exemple 11 : INTERSECT et EXCEPT

Les genres écrits à la fois par des auteurs français *et* par des auteurs britanniques :

```
SELECT livres.genre
FROM livres JOIN auteurs ON livres.id_auteur = auteurs.id
WHERE auteurs.nationalite = 'française'
INTERSECT
SELECT livres.genre
FROM livres JOIN auteurs ON livres.id_auteur = auteurs.id
WHERE auteurs.nationalite = 'britannique';
```

Résultat :

roman.

En échangeant les deux nationalités et en remplaçant INTERSECT par EXCEPT, on obtient les genres écrits par des Britanniques *mais par aucun* Français : policier, science-fiction.

Attention, erreur fréquente

On **ne peut pas** remplacer cet INTERSECT par WHERE auteurs.nationalite = 'française' AND auteurs.nationalite = 'britannique'. Une condition WHERE porte sur *une seule ligne à la fois*, et aucune ligne n'a deux nationalités : le résultat serait vide. INTERSECT, lui, compare les résultats de *deux* requêtes.

2.2 Requêtes imbriquées

Le résultat d'une requête étant une table, on peut l'utiliser **à l'intérieur** d'une autre requête, entre parenthèses.

Exemple 12 : sous-requête après IN

WHERE valeur IN (SELECT ...) est vrai lorsque la valeur figure dans le résultat de la sous-requête (NOT IN : lorsqu'elle n'y figure pas). Les livres qui n'ont jamais été empruntés :

```
SELECT livres.titre FROM livres
WHERE livres.id NOT IN (SELECT emprunts.id_livre FROM emprunts);
```

Résultat :

Notre-Dame de Paris, Germinal, L'Assommoir, Frankenstein, La Ferme des animaux, Les Trois Mousquetaires.

La sous-requête renvoie la colonne des numéros de livres empruntés ; on garde les livres dont l'id n'y est pas. Une sous-requête utilisée avec IN doit renvoyer **une seule colonne**.

Exemple 13 : sous-requête après FROM

Une sous-requête peut aussi remplacer une table après FROM. On doit alors lui donner un nom avec AS. C'est utile pour faire une jointure *après* un INTERSECT ou un EXCEPT. Les livres empruntés à la fois par Léa et par Sarah :

```
SELECT livres.titre
FROM (SELECT emprunts.id_livre
      FROM emprunts JOIN adherents ON emprunts.id_adherent = adherents.id
      WHERE adherents.prenom = 'Léa'
      INTERSECT
      SELECT emprunts.id_livre
      FROM emprunts JOIN adherents ON emprunts.id_adherent = adherents.id
      WHERE adherents.prenom = 'Sarah') AS communs
JOIN livres ON communs.id_livre = livres.id;
```

Résultat :

Le Petit Prince.

Pourquoi intersecter des numéros et pas directement des titres ? Ici cela marcherait aussi. Mais en général, deux lignes différentes peuvent avoir le même nom (deux arrêts de bus « Victor Hugo » dans deux villes différentes, deux auteurs nommés Alexandre Dumas...) : seule la clé primaire est sûre d'identifier une ligne.

2.3 Fonctions d'agrégation, GROUP BY et HAVING

Définition : fonctions d'agrégation

Une **fonction d'agrégation** calcule **une seule valeur** à partir de toute une colonne :

COUNT(*)	nombre de lignes
COUNT(DISTINCT col)	nombre de valeurs <i>différentes</i> dans la colonne
MIN(col), MAX(col)	plus petite, plus grande valeur
SUM(col), AVG(col)	somme, moyenne

Exemple 14 : agrégation sur toute la table

```
SELECT COUNT(*) AS nb, MIN(naissance) AS plus_ancien,
      MAX(naissance) AS plus_recent, AVG(naissance) AS moyenne
FROM auteurs;
```

Résultat :

nb	plus_ancien	plus_recent	moyenne
10	1797	1913	1854.6

De même, `SELECT COUNT(DISTINCT genre) FROM livres;` renvoie 5 (il y a 5 genres différents).

Et si l'on veut le nombre de livres *de chaque genre*? Il faut d'abord **regrouper** les livres par genre, puis compter dans chaque groupe.

Définition : regroupement avec GROUP BY

```
SELECT col, fonction_d_agregation(...) FROM ... GROUP BY col;
```

1. Le SGBD range les lignes en **paquets** : un paquet par valeur de `col`.
2. Chaque paquet produit **une seule ligne** du résultat, dans laquelle les fonctions d'agrégation sont calculées sur les lignes *du paquet*.

Règle : après `SELECT`, chaque colonne qui n'est pas dans une fonction d'agrégation doit figurer dans le `GROUP BY`. On peut regrouper selon plusieurs colonnes : `GROUP BY col1, col2`.

Exemple 15 : nombre de livres par genre

```
SELECT genre, COUNT(*) AS nb_livres
FROM livres
GROUP BY genre;
```

Les paquets formés :

aventure	Vingt mille lieues..., Le Tour du monde..., Le Comte de Monte-Cristo, Les Trois Mousq.
conte	Le Petit Prince
policier	Le Crime de l'Orient-Express, Mort sur le Nil
roman	Les Misérables, Notre-Dame... (7 livres)
science-fiction	Frankenstein, 1984

Résultat :

genre	nb_livres
aventure	4
conte	1
policier	2
roman	7
science-fiction	2

Exemple 16 : GROUP BY après une jointure

Nombre d'emprunts par ville d'adhérent :

```
SELECT adherents.ville, COUNT(*) AS nb_emprunts
FROM emprunts JOIN adherents ON emprunts.id_adherent = adherents.id
GROUP BY adherents.ville;
```

Résultat :

Firminy : 3, Roanne : 3, Saint-Étienne : 6.

La jointure est faite en premier (une ligne par emprunt, avec la ville de l'adhérent), puis les lignes sont regroupées par ville.

Définition : filtrer les groupes avec HAVING

`HAVING condition`, placé après `GROUP BY`, ne garde que les **groupes** qui vérifient la condition. La condition porte en général sur une fonction d'agrégation.

WHERE	filtre les lignes , avant le regroupement. Ne peut pas contenir <code>COUNT</code> , <code>MAX</code> ...
HAVING	filtre les groupes , après le regroupement.

Exemple 17 : HAVING, et WHERE + HAVING

Les genres représentés par au moins 3 livres :

```
SELECT genre, COUNT(*) AS nb_livres
FROM livres
GROUP BY genre
HAVING COUNT(*) >= 3;
```

Résultat :

aventure : 4, roman : 7.

Les genres représentés par au moins 2 livres *parus après 1900* :

```
SELECT genre, COUNT(*) AS nb_livres
FROM livres
WHERE annee > 1900
GROUP BY genre
HAVING COUNT(*) >= 2;
```

Résultat :

policier : 2, roman : 3.

Le **WHERE** enlève d'abord les livres parus avant 1900 ; les paquets ne contiennent donc plus que des livres récents (le paquet « roman » ne contient plus que L'Étranger, La Peste et La Ferme des animaux). Écrire **WHERE COUNT(*) >= 2** provoque une erreur (**misuse of aggregate**) : au moment du **WHERE**, les paquets n'existent pas encore.

Exemple 18 : deux astuces utiles

« **Pour toutes les lignes du groupe** » avec **MAX** ou **MIN**. Les auteurs dont *tous* les livres sont parus avant 1880 : cela revient à dire que leur livre le plus récent est paru avant 1880.

```
SELECT auteurs.nom, MAX(livres.annee) AS dernier_livre
FROM livres JOIN auteurs ON livres.id_auteur = auteurs.id
GROUP BY auteurs.id, auteurs.nom
HAVING MAX(livres.annee) < 1880;
```

Résultat :

Hugo (1862), Verne (1872), Shelley (1818), Dumas (1844).

On regroupe par **auteurs.id** (deux auteurs différents peuvent avoir le même nom), et on ajoute **auteurs.nom** au **GROUP BY** parce qu'on l'affiche (règle ci-dessus).

Compter des valeurs différentes dans chaque groupe avec COUNT(DISTINCT ...). Les adhérents qui ont emprunté des livres d'au moins 3 genres différents :

```
SELECT adherents.prenom, COUNT(DISTINCT livres.genre) AS nb_genres
FROM emprunts
JOIN livres ON emprunts.id_livre = livres.id
JOIN adherents ON emprunts.id_adherent = adherents.id
GROUP BY adherents.id, adherents.prenom
HAVING COUNT(DISTINCT livres.genre) >= 3;
```

Résultat :

Hugo : 3, Sarah : 3.

Exemple 19 : un agrégat dans une requête imbriquée

Quel est le genre le plus représenté ? On ne peut pas écrire **MAX(COUNT(*))** : on ne met pas une fonction d'agrégation dans une autre. On calcule d'abord le nombre de livres de chaque genre dans une sous-requête, puis le maximum de ces nombres :

```
SELECT genre, COUNT(*) AS nb_livres
FROM livres
GROUP BY genre
HAVING COUNT(*) = (SELECT MAX(t.nb)
                    FROM (SELECT COUNT(*) AS nb FROM livres GROUP BY genre) AS t);
```

Résultat :

roman : 7.

On pourrait aussi trier (ORDER BY COUNT(*) DESC LIMIT 1), mais en cas d'égalité entre deux genres, on n'en obtiendrait qu'un seul.

À retenir

SQLite est plus permissif que la norme SQL. Certaines requêtes s'exécutent dans le notebook mais sont incorrectes selon la norme (celle du programme). Respectez toujours ces règles :

- avec GROUP BY, toute colonne du SELECT hors fonction d'agrégation figure dans le GROUP BY;
- pas de fonction d'agrégation dans un WHERE, ni dans une autre fonction d'agrégation;
- dans HAVING, on réécrit la fonction (HAVING COUNT(*) >= 3) plutôt que d'utiliser le nom donné par AS (HAVING nb_livres >= 3, accepté par SQLite mais pas par la norme);
- une sous-requête placée après FROM reçoit un nom avec AS;
- les textes s'écrivent entre apostrophes simples.

2.4 Pour aller plus loin : utiliser deux fois la même table

Exemple 20 : auto-jointure

On peut donner un **nom provisoire** à une table dans une requête avec AS : FROM auteurs AS a1. Cela permet d'utiliser *deux fois* la même table, comme s'il y en avait deux copies : c'est une **auto-jointure**. Les paires d'auteurs nés la même année :

```
SELECT a1.nom, a2.nom, a1.naissance
FROM auteurs AS a1 JOIN auteurs AS a2 ON a1.naissance = a2.naissance
WHERE a1.id < a2.id;
```

Résultat :

Hugo, Dumas, 1802.

Sans le WHERE, chaque auteur serait associé à lui-même (Hugo, Hugo, 1802), et chaque paire apparaîtrait deux fois (Hugo, Dumas et Dumas, Hugo). C'est cette technique qui peut servir dans la question n) de l'exercice 5.

À retenir

Ordre d'écriture des clauses d'une requête (seules SELECT et FROM sont obligatoires) :

```
SELECT ... FROM ... JOIN ... ON ...
WHERE ... GROUP BY ... HAVING ... ORDER BY ... LIMIT ... OFFSET ...;
```

Ordre dans lequel le SGBD les traite : FROM/JOIN (construire la table) → WHERE (filtrer les lignes) → GROUP BY (faire les paquets) → HAVING (filtrer les paquets) → SELECT (choisir les colonnes) → ORDER BY → LIMIT.

Pour combiner deux requêtes : UNION, INTERSECT, EXCEPT (même nombre de colonnes).

3 Exercices

Exercice 1 : clés et jointures, à la main

On utilise la base `bibliotheque` (tables page 1).

1. Dans la table `adherents`, pourquoi l'attribut `nom` ne conviendrait-il pas comme clé primaire? Et le couple (`nom`, `prenom`)?
2. Quelles sont les clés étrangères de la base? À quoi chacune fait-elle référence?
3. Combien de lignes contient le résultat de `SELECT * FROM auteurs, livres;`? Et celui de `SELECT * FROM livres JOIN auteurs ON livres.id_auteur = auteurs.id;`? Justifier.
4. Un élève écrit la requête suivante pour afficher chaque titre avec le nom de son auteur.

```
SELECT titre.livres, nom.auteurs
FROM livres JOIN auteurs ON id_auteur.livres = id.auteurs;
```

Quel message d'erreur obtient-il? Corriger la requête.

5. Un autre élève écrit `... FROM livres JOIN auteurs ON livres.id = auteurs.id;`. Sa requête s'exécute. Donner les trois premières lignes du résultat et le nombre total de lignes. Quelle est l'erreur?
6. Donner le résultat de la requête suivante, puis la traduire par une phrase en français.

```
SELECT adherents.nom, emprunts.date_emprunt
FROM emprunts JOIN adherents ON emprunts.id_adherent = adherents.id
WHERE emprunts.id_livre = 10;
```

7. Écrire une requête qui affiche les titres des livres empruntés par Tom Durand. Quelles tables faut-il? Pourquoi `auteurs` n'est-elle pas nécessaire?

Correction

1. Deux adhérents peuvent porter le même nom (deux membres d'une même famille, par exemple) : la clé primaire ne serait plus unique. Le couple (`nom`, `prenom`) ne convient pas non plus : deux personnes peuvent avoir les mêmes nom et prénom (Alexandre Dumas père et fils!). C'est pour cela qu'on numérote les lignes.
2. `livres.id_auteur` fait référence à `auteurs.id`; `emprunts.id_livre` fait référence à `livres.id`; `emprunts.id_adherent` fait référence à `adherents.id`.
3. Produit cartésien : $10 \times 16 = 160$ lignes. Jointure : 16 lignes, car chaque livre a exactement un auteur (sa valeur `id_auteur` correspond à une seule ligne de `auteurs`).
4. `no such column: titre.livres` : SQLite cherche une table `titre`. On écrit `table.attribut` :

```
SELECT livres.titre, auteurs.nom
FROM livres JOIN auteurs ON livres.id_auteur = auteurs.id;
```

5. « Les Misérables | Hugo », « Notre-Dame de Paris | Verne », « Vingt mille lieues sous les mers | Zola ». Il y a 10 auteurs, donc seuls les livres d'id 1 à 10 trouvent une correspondance : 10 lignes. On a relié deux clés primaires sans rapport ; il faut relier la clé étrangère `livres.id_auteur` à `auteurs.id`.
6. Le livre 10 est *1984* (emprunts 4 et 10) : « Petit | 2026-09-08 » et « Bernard | 2026-09-25 ». La requête affiche le nom des adhérents qui ont emprunté *1984*, avec la date de chaque emprunt.
7. Il faut `livres` (pour le titre), `adherents` (pour le nom) et `emprunts` pour les relier. L'auteur n'intervient ni dans ce qu'on affiche, ni dans la condition. Résultat : Les Misérables, Le Comte de Monte-Cristo, Le Tour du monde en 80 jours.

```
SELECT livres.titre
FROM emprunts
JOIN livres    ON emprunts.id_livre    = livres.id
JOIN adherents ON emprunts.id_adherent = adherents.id
WHERE adherents.prenom = 'Tom' AND adherents.nom = 'Durand';
```

Exercice 2 : ensembles et regroupements, à la main

1. Donner le résultat de la requête :

```
SELECT nationalite, COUNT(*) AS nb FROM auteurs GROUP BY nationalite;
```

2. Donner le résultat de chacune des deux requêtes suivantes, puis les traduire par une phrase en français.

```
SELECT id_adherent FROM emprunts WHERE id_livre = 14
UNION
SELECT id_adherent FROM emprunts WHERE id_livre = 10;
```

```
SELECT id_adherent FROM emprunts WHERE id_livre = 14
INTERSECT
SELECT id_adherent FROM emprunts WHERE id_livre = 10;
```

3. Pourquoi la requête suivante renvoie-t-elle un résultat vide ? La corriger.

```
SELECT id_adherent FROM emprunts WHERE id_livre = 14 AND id_livre = 10;
```

4. Un élève veut le nombre de livres de chaque auteur, mais seulement pour les auteurs ayant écrit au moins 2 livres. Il écrit :

```
SELECT id_auteur, COUNT(*) FROM livres WHERE COUNT(*) >= 2 GROUP BY id_auteur;
```

Que se passe-t-il ? Corriger la requête.

5. La requête `SELECT genre, titre, COUNT(*) FROM livres GROUP BY genre;` s'exécute sans erreur en SQLite. Pourquoi son résultat n'a-t-il pourtant pas de sens ? Quelle règle ne respecte-t-elle pas ?

Correction

1. Deux paquets : britannique : 3 ; française : 7.
2. Le livre 14 (Le Petit Prince) a été emprunté par les adhérents 1 et 3, le livre 10 (1984) par les adhérents 3 et 2. Avec UNION : 1, 2, 3 (sans doublon) : les adhérents ayant emprunté le Petit Prince *ou* 1984. Avec INTERSECT : 3 : les adhérents ayant emprunté le Petit Prince *et* 1984 (Sarah Petit).
3. La condition est testée ligne par ligne : une ligne de `emprunts` ne peut pas avoir à la fois `id_livre = 14` et `id_livre = 10`. Il faut intersecter deux requêtes : c'est la deuxième requête de la question 2.
4. Erreur misuse of aggregate : le WHERE est traité avant le regroupement.

```
SELECT id_auteur, COUNT(*) FROM livres GROUP BY id_auteur HAVING COUNT(*) >= 2;
```

5. Chaque paquet donne une seule ligne, mais un paquet contient plusieurs titres : SQLite en affiche un au hasard (par exemple « aventure | Vingt mille lieues sous les mers | 4 »), ce qui ne veut rien dire. La colonne `titre` n'est ni dans une fonction d'agrégation ni dans le `GROUP BY` : la requête est incorrecte selon la norme, et d'autres SGBD la refusent.

Où travailler ? Dans le notebook : <https://capytale2.ac-paris.fr/web/c/c5ca-12201317/mcer>. Toutes les bases (bibliotheque, pizza, notes et lignes_STAS) sont créées par la première cellule. Pour chaque question, écrire la requête dans la cellule prévue, l'exécuter et vérifier que le résultat a du sens. Une seule requête par cellule.

Exercice 3 : la pizzeria

La base pizza a pour schéma :

```
Client(idClient INTEGER, Nom TEXT, Prénom TEXT, Adresse TEXT)
Produit(idProduit INTEGER, Designation TEXT, Prix REAL)
Livreur(idLivreur INTEGER, Nom TEXT, Prénom TEXT)
Véhicule(idVéhicule INTEGER, Type TEXT, Kms INTEGER)
Commande(id INTEGER, idClient INTEGER, idProduit INTEGER, idLivreur INTEGER,
          idVéhicule INTEGER, Date TEXT, Heure TEXT, Livraison TEXT)
```

Clés primaires : Client.idClient, Produit.idProduit, Livreur.idLivreur, Véhicule.idVéhicule et Commande.id. Les attributs idClient, idProduit, idLivreur et idVéhicule de Commande sont des clés étrangères, qui font référence à l'attribut de même nom dans l'autre table. Attention aux accents dans les noms de colonnes (Prénom, idVéhicule)! Commencer par afficher le contenu de chaque table avec `SELECT * FROM ...;`.

- Obtenir la liste des prénoms des clients.
- Obtenir la liste des pizzas qui coûtent moins de 10 euros.
- Obtenir la liste des pizzas dont le prix est entre 10 euros et 12 euros inclus.
- Obtenir le prix de la pizza végétarienne.
- Obtenir la liste de tous les noms de famille (client ou livreur). [UNION]
- Obtenir la liste des scooters qui ont parcouru plus de 10 000 km.
- Obtenir la liste des commandes qui n'ont pas été livrées (heure de livraison NULL). [hors programme : IS NULL]
- Obtenir la liste des commandes passées après 17 h.
- Obtenir la liste des noms de pizzas commandées par le client de code idClient 1. [JOIN]
- Obtenir la liste des noms et prénoms des livreurs ayant livré une pizza chez le client de code idClient 1. [JOIN]

Correction

```
-- a) Albus, Fred, Minerva, Tom, Georges, Cédric
SELECT Client.Prénom FROM Client;
-- b) Royale (9 euros)
SELECT Produit.Designation, Produit.Prix FROM Produit WHERE Produit.Prix < 10;
-- c) Forézienne, Bretonne, Océane, Végétarienne
SELECT Produit.Designation, Produit.Prix FROM Produit
WHERE Produit.Prix >= 10 AND Produit.Prix <= 12;
-- d) 11 euros
SELECT Produit.Prix FROM Produit WHERE Produit.Designation = 'Végétarienne';
-- e) 7 noms : Weasley n'apparaît qu'une fois, bien qu'il y ait
--      deux clients et un livreur de ce nom
SELECT Client.Nom FROM Client
UNION
SELECT Livreur.Nom FROM Livreur;
-- f) véhicules 1 (25 000 km) et 3 (17 000 km)
SELECT Véhicule.idVéhicule, Véhicule.Kms FROM Véhicule
WHERE Véhicule.Type = 'Scooter' AND Véhicule.Kms > 10000;
-- g) commandes 3 et 5
SELECT Commande.id FROM Commande WHERE Commande.Livraison IS NULL;
-- h) commandes 1 (19:00) et 2 (18:00)
SELECT Commande.id, Commande.Heure FROM Commande WHERE Commande.Heure > '17:00';
```

```
-- i) Canicatti, Océane
SELECT Produit.Designation
FROM Commande JOIN Produit ON Commande.idProduit = Produit.idProduit
WHERE Commande.idClient = 1;
-- j) Harry Potter, Hermione Granger
SELECT DISTINCT Livreur.Nom, Livreur.Prénom
FROM Commande JOIN Livreur ON Commande.idLivreur = Livreur.idLivreur
WHERE Commande.idClient = 1;
```

Pour les requêtes sur une seule table, le préfixe **Table.** n'est pas obligatoire (**SELECT Prénom FROM Client;** est correct). En j), le **DISTINCT** évite d'afficher deux fois un livreur qui serait venu deux fois.

Exercice 4 : une base de notes

On veut une base de données des notes d'élèves. Chaque élève a un numéro étudiant, un prénom et un nom. Les devoirs ont un numéro d'identification unique, un titre et une matière. Le schéma est :

```
Eleves(idEleve INTEGER, Prenom TEXT, Nom TEXT)
Devoirs(idDevoir INTEGER, Titre TEXT, Matiere TEXT)
Resultats(idEleve INTEGER, idDevoir INTEGER, note INTEGER, rang INTEGER)
```

1. Analyse du schéma (sur papier).

- Indiquer la clé primaire de chaque table. Attention, pour l'une des tables, il faut deux attributs (voir l'exemple 5).
- Identifier les clés étrangères, et dire à quoi elles font référence.
- Quel est le rôle de la table **Resultats** vis-à-vis des tables **Eleves** et **Devoirs** (voir l'exemple 4) ?

2. Remplissage (sur papier). Écrire le contenu des trois tables à partir des données ci-dessous. Les élèves sont numérotés de 1 à 5 dans l'ordre du tableau; les devoirs portent les numéros 3 et 7. Le **rang** (1 pour la meilleure note, etc.) est à calculer.

Élève	DS 3 (informatique) programmation impérative	DS 7 (physique) énergie mécanique
Jack Sparrow	4	20
Peter Pan	2	18
Sherlock Holmes	16	10
Thomas Anderson	20	4
Ellen Ripley	17	15

Les tables sont déjà créées et remplies dans le notebook : afficher leur contenu avec **SELECT *** **FROM ...**; et comparer avec vos tableaux.

3. Interrogation (sur machine).

- Afficher la liste des prénoms des élèves.
- Afficher la liste des matières traitées.
- Afficher la liste de toutes les notes obtenues. À l'aide de **SELECT DISTINCT**, obtenir de même la liste des notes obtenues, sans répétition.
- Afficher la liste des matières où au moins un élève a eu moins de 3. [JOIN]
- Afficher la liste des noms des élèves ayant eu moins de 10 au devoir numéro 3. [JOIN]
- Afficher les prénoms des élèves qui se sont classés premiers à l'un au moins des devoirs.
- Afficher les numéros étudiants des élèves s'étant classés dans les 3 premiers aux deux devoirs. [INTERSECT, ou GROUP BY + HAVING]
- Afficher la liste des élèves (prénom et nom) qui étaient dans les 3 premiers au DS 3 mais pas au DS 7. [EXCEPT + requête imbriquée]

Correction

1.a) Clés primaires : Eleves.idEleve, Devoirs.idDevoir, et le couple (idEleve, idDevoir) pour Resultats : un élève a plusieurs notes et un devoir a plusieurs notes, donc aucun des deux attributs seul n'est unique, mais un élève n'a qu'une note par devoir.

1.b) Resultats.idEleve fait référence à Eleves.idEleve ; Resultats.idDevoir fait référence à Devoirs.idDevoir.

1.c) Un élève passe plusieurs devoirs et un devoir est passé par plusieurs élèves : l'association est « plusieurs – plusieurs ». Resultats est la table de liaison, comme emprunts dans la bibliothèque (avec en plus la note et le rang).

2.

idEleve	Prenom	Nom
1	Jack	Sparrow
2	Peter	Pan
3	Sherlock	Holmes
4	Thomas	Anderson
5	Ellen	Ripley

idDevoir	Titre	Matiere
3	programmation impérative	informatique
7	énergie mécanique	physique

idEleve	idDevoir	note	rang	idEleve	idDevoir	note	rang
1	3	4	4	1	7	20	1
2	3	2	5	2	7	18	2
3	3	16	3	3	7	10	4
4	3	20	1	4	7	4	5
5	3	17	2	5	7	15	3

3.

```
-- a) Jack, Peter, Sherlock, Thomas, Ellen
SELECT Eleves.Prenom FROM Eleves;
-- b) informatique, physique
SELECT Devoirs.Matiere FROM Devoirs;
-- c) 10 notes, puis 8 notes distinctes (4 et 20 apparaissent deux fois)
SELECT Resultats.note FROM Resultats;
SELECT DISTINCT Resultats.note FROM Resultats;
-- d) informatique (le 2 de Peter Pan)
SELECT DISTINCT Devoirs.Matiere
FROM Resultats JOIN Devoirs ON Resultats.idDevoir = Devoirs.idDevoir
WHERE Resultats.note < 3;
-- e) Sparrow, Pan
SELECT Eleves.Nom
FROM Resultats JOIN Eleves ON Resultats.idEleve = Eleves.idEleve
WHERE Resultats.idDevoir = 3 AND Resultats.note < 10;
-- f) Thomas, Jack
SELECT DISTINCT Eleves.Prenom
FROM Resultats JOIN Eleves ON Resultats.idEleve = Eleves.idEleve
WHERE Resultats.rang = 1;
-- g) 5 (Ellen Ripley). Première méthode : intersection
SELECT Resultats.idEleve FROM Resultats
WHERE Resultats.idDevoir = 3 AND Resultats.rang <= 3
INTERSECT
SELECT Resultats.idEleve FROM Resultats
WHERE Resultats.idDevoir = 7 AND Resultats.rang <= 3;
-- Deuxième méthode : le plus mauvais rang de l'élève est <= 3
SELECT Resultats.idEleve FROM Resultats
GROUP BY Resultats.idEleve
HAVING MAX(Resultats.rang) <= 3;
-- h) Sherlock Holmes, Thomas Anderson
SELECT Eleves.Prenom, Eleves.Nom
FROM (SELECT Resultats.idEleve FROM Resultats
WHERE Resultats.idDevoir = 3 AND Resultats.rang <= 3
```

```

EXCEPT
SELECT Resultats.idEleve FROM Resultats
WHERE Resultats.idDevoir = 7 AND Resultats.rang <= 3) AS t
JOIN Eleves ON t.idEleve = Eleves.idEleve;

```

Exercice 5 : le réseau STAS

La base `lignes_STAS` décrit les lignes de bus et de tram de l'agglomération stéphanoise. Son schéma :

```

Arrets(id INTEGER, nom TEXT, commune TEXT)
Routes(ligne TEXT, ordre_arret INTEGER, id_arret INTEGER)

```

La clé primaire de `Arrets` est `id`; celle de `Routes` est le couple (`ligne`, `ordre_arret`). Dans `Routes`, la ligne ('10', 3, 279) signifie : « le 3^e arrêt de la ligne 10 est l'arrêt d'id 279 ». Le numéro de ligne est un **texte** (il y a des lignes 'T1', 'T2'...) : on écrit donc `Routes.ligne = '10'`, avec des apostrophes. Certaines lignes sont décrites dans les deux sens, un même arrêt peut donc apparaître deux fois sur une ligne : pensez à `DISTINCT`.

- Étudier la structure de la base : afficher les 10 premières lignes de chaque table. Quelle est la clé étrangère ? À quoi fait-elle référence ? Pourquoi `ligne` seul ne peut-il pas être la clé primaire de `Routes` ?
- Afficher la liste des lignes disponibles (sans répétitions).
- Afficher la liste des villes desservies par le réseau (toujours sans répétitions).
- Afficher tous les noms d'arrêts de la commune La Fouillouse.
- Combinaison des deux tables en une seule contenant toutes les informations de façon cohérente. *[JOIN]*
- Afficher le plan de la ligne 10 (arrêts desservis et communes), dans l'ordre de la ligne.
- Afficher l'ensemble des lignes desservant l'arrêt Grand Gonnet.
- Afficher le nom de l'arrêt de départ de chaque ligne.
- Afficher les lignes partant de Saint-Étienne.
- Afficher l'ensemble des lignes qu'on peut prendre pour aller de Grand Gonnet à Terrasse (c'est-à-dire qui desservent ces deux arrêts). *[INTERSECT, ou GROUP BY + HAVING]*
- Afficher les arrêts communs des lignes 2 et 3. *[INTERSECT + requête imbriquée, exemple 13]*
- Afficher les arrêts desservis par la ligne 2 mais pas par la ligne 1 ni la ligne 3. *[EXCEPT]*
- Afficher l'ensemble des arrêts accessibles sans correspondance depuis le Lycée Fauriel. *[WHERE ... IN (SELECT ...), exemple 12]*
- (Bonus) Afficher l'ensemble des arrêts accessibles avec au plus 1 correspondance depuis le lycée. *[requêtes imbriquées, ou exemple 20]*
- Le maire décide de changer le nom de la commune de Saint-Héand en Saint-Héant. Expliquer les modifications à effectuer. La structure de la base de données est-elle bien conçue ? Proposer un nouveau schéma minimisant la redondance d'information (voir la partie « Pourquoi plusieurs tables ? »).

Correction

a) `Routes.id_arret` est une clé étrangère qui fait référence à `Arrets.id`. Une ligne de bus dessert plusieurs arrêts : la valeur '10' apparaît dans 91 lignes de `Routes`, elle n'identifie donc pas une ligne de la table. Le couple (`ligne`, `numéro d'ordre`), lui, est unique.

```

SELECT * FROM Arrets LIMIT 10;
SELECT * FROM Routes LIMIT 10;
-- b) 63 lignes : '1', '10', '11', ..., 'T1', 'T2', 'T3'
SELECT DISTINCT Routes.ligne FROM Routes;

```

```

-- c) 36 communes
SELECT DISTINCT Arrets.commune FROM Arrets;
-- d) 11 arrêts : La Porchère, Les Molineaux, Le Vernay, ...
SELECT Arrets.nom FROM Arrets WHERE Arrets.commune = 'La Fouillouse';
-- e) 2596 lignes : une par passage d'une ligne de bus à un arrêt
SELECT * FROM Routes JOIN Arrets ON Routes.id_arret = Arrets.id;
-- f) 91 lignes : aller puis retour, départ et arrivée
--   à Place Jean Jaurès
SELECT Routes.ordre_arret, Arrets.nom, Arrets.commune
FROM Routes JOIN Arrets ON Routes.id_arret = Arrets.id
WHERE Routes.ligne = '10'
ORDER BY Routes.ordre_arret;
-- g) 10, T1, T2 (T1 et T2 apparaissent deux fois sans DISTINCT :
--   les trams sont décrits dans les deux sens)
SELECT DISTINCT Routes.ligne
FROM Routes JOIN Arrets ON Routes.id_arret = Arrets.id
WHERE Arrets.nom = 'Grand Gonnet';
-- h) 63 lignes : ('1', 'Bellevue'), ('10', 'Place Jean Jaurès'), ...
SELECT Routes.ligne, Arrets.nom
FROM Routes JOIN Arrets ON Routes.id_arret = Arrets.id
WHERE Routes.ordre_arret = 1;
-- i) 28 lignes
SELECT Routes.ligne
FROM Routes JOIN Arrets ON Routes.id_arret = Arrets.id
WHERE Routes.ordre_arret = 1 AND Arrets.commune = 'Saint-Étienne';
-- j) T1, T2. Première méthode : intersection
SELECT Routes.ligne
FROM Routes JOIN Arrets ON Routes.id_arret = Arrets.id
WHERE Arrets.nom = 'Grand Gonnet'
INTERSECT
SELECT Routes.ligne
FROM Routes JOIN Arrets ON Routes.id_arret = Arrets.id
WHERE Arrets.nom = 'Terrasse';
-- Deuxième méthode : regroupement, on garde les lignes
-- qui passent par deux arrêts différents parmi les deux
SELECT Routes.ligne
FROM Routes JOIN Arrets ON Routes.id_arret = Arrets.id
WHERE Arrets.nom = 'Grand Gonnet' OR Arrets.nom = 'Terrasse'
GROUP BY Routes.ligne
HAVING COUNT(DISTINCT Arrets.nom) = 2;
-- k) 9 arrêts : Palais de Justice, Georges Teissier, ...
SELECT Arrets.nom
FROM (SELECT Routes.id_arret FROM Routes WHERE Routes.ligne = '2'
      INTERSECT
      SELECT Routes.id_arret FROM Routes WHERE Routes.ligne = '3') AS communs
JOIN Arrets ON communs.id_arret = Arrets.id;
-- l) 43 arrêts : Gare de Firminy, Rue Verdier, ...
SELECT Arrets.nom
FROM (SELECT Routes.id_arret FROM Routes WHERE Routes.ligne = '2'
      EXCEPT
      SELECT Routes.id_arret FROM Routes WHERE Routes.ligne = '1'
      EXCEPT
      SELECT Routes.id_arret FROM Routes WHERE Routes.ligne = '3') AS t
JOIN Arrets ON t.id_arret = Arrets.id;
-- m) 65 arrêts (le lycée est desservi par les lignes 9, 11 et 70)
SELECT DISTINCT Arrets.nom
FROM Routes JOIN Arrets ON Routes.id_arret = Arrets.id
WHERE Routes.ligne IN (SELECT Routes.ligne
                      FROM Routes JOIN Arrets ON Routes.id_arret = Arrets.id

```

```
WHERE Arrêts.nom = 'Lycée Fauriel');
```

n) On « remonte » en trois étapes : les lignes du lycée → les arrêts de ces lignes → les lignes passant par ces arrêts. Les arrêts recherchés sont ceux de ces dernières lignes.

```
SELECT DISTINCT Arrêts.nom
FROM Routes JOIN Arrêts ON Routes.id_arret = Arrêts.id
WHERE Routes.ligne IN
  (SELECT R2.ligne FROM Routes AS R2 WHERE R2.id_arret IN
    (SELECT R1.id_arret FROM Routes AS R1
     WHERE R1.ligne IN
      (SELECT Routes.ligne
       FROM Routes JOIN Arrêts ON Routes.id_arret = Arrêts.id
       WHERE Arrêts.nom = 'Lycée Fauriel'))));
```

Autre solution, avec des auto-jointures (exemple 20) : on construit une chaîne lycée → arrêt de la même ligne → même arrêt sur une autre ligne → arrêt de cette 2^e ligne.

```
SELECT DISTINCT A4.nom
FROM Arrêts AS A1
JOIN Routes AS R1 ON R1.id_arret = A1.id
JOIN Routes AS R2 ON R2.ligne = R1.ligne
JOIN Routes AS R3 ON R3.id_arret = R2.id_arret
JOIN Routes AS R4 ON R4.ligne = R3.ligne
JOIN Arrêts AS A4 ON A4.id = R4.id_arret
WHERE A1.nom = 'Lycée Fauriel';
```

Les deux requêtes donnent 467 noms d'arrêts (sur 971 noms différents dans le réseau).

o) Le nom de la commune est écrit **dans chaque arrêt** de la commune : il faut modifier les 9 lignes de Arrêts concernées (on les compte avec SELECT COUNT(*) FROM Arrêts WHERE Arrêts.commune = 'Saint-Héand';). L'information est **redondante** : si l'on oublie une ligne, ou si l'on fait une faute de frappe, la base devient incohérente (la commune aurait deux noms). C'est exactement le problème des auteurs vu dans la partie « Pourquoi plusieurs tables? », et la solution est la même : créer une table des communes, et ne garder dans Arrêts qu'une clé étrangère.

Communes(id INTEGER, nom TEXT)	clé primaire : id
Arrêts(id INTEGER, nom TEXT, id_commune INTEGER)	clé primaire : id
Routes(ligne TEXT, ordre_arret INTEGER, id_arret INTEGER)	clé primaire : (ligne, ordre_arret)

avec Arrêts.id_commune qui fait référence à Communes.id. Le changement de nom ne modifie alors qu'une seule ligne, dans Communes.