# TD 3 : SQL

SQL Interrogation de données, requêtes complexes

2024-10-04

**Avec solutions**

- **L3 MIASHS/Ingémath**
- **Université Paris Cité**
- Année 2024-2025
- Course Homepage

- Moodle

Objectifs de la séance :

- requêtes imbriquées
- jointures externes
- vues
- fonctions SQL

En plus du schéma `world`, nous allons utiliser le schéma `pagila` qui contient des informations utilisées par un chaîne fictive de magasins de location de DVD.

Le schéma `pagila` est visible ici.

Sous `psql` ou `pgli`, vous pouvez aussi inspecter les tables comme d'habitude avec

```
bd_2023-24> \d pagila.film
bd_2023-24> \d pagila.actor
```

Quand on travaille sur plusieurs schémas (ici `world`, `pagila` et votre schéma personnel), il est bon
- d'ajuster `search_path` (sous `psql, pgcli`) : `set search_path to world, pagila, ... ;` (remplacer `...`par votre identifiant)
- de qualifier les noms de table pour indiquer le schéma d'origine : `world.country` versus `pagila.country`

## Requêtes imbriquées

Les requêtes imbriquées permettent d'utiliser le résultat d'une requête dans la clause `WHERE`.

On utilisera essentiellement les opérateurs suivants : `IN, EXISTS, ALL, ANY`.

`IN` permet de tester la présence d'une valeur dans le résultat d'une requête.

`EXISTS` renvoie `True` si la requête donnée est non-vide et `False` sinon. On peut les combiner avec `NOT` pour inverser leur comportement : `NOT IN` et `NOT EXISTS`. Par exemple, pour connaître les régions sans monarchie, on pourra écrire :

```
SELECT DISTINCT region
FROM world.country
WHERE region NOT IN (
  SELECT region
```

```
  FROM world.country
  WHERE governmentform like '%Monarchy%'
);
```

Pour connaître les régions qui ont au moins une langue officielle, on pourra écrire :

```
SELECT DISTINCT region
FROM world.country AS co
WHERE EXISTS (
  SELECT *
  FROM world.countrylanguage AS cl
  WHERE co.countrycode = cl.countrycode AND
    cl.isofficial
);
```

Remarquez que dans ce dernier exemple, la sous-requête fait intervenir des attributs de la requête principale, c'est pourquoi on parle de requêtes imbriquées.

`ANY` et `ALL` sont deux autres opérateurs. Par exemple

```
SELECT *
FROM table
WHERE col < ALL(
  requete
)
```

sélectionnera les lignes de `table` telles que la valeur de `col` est plus petite que toutes les valeurs retournées par la requête `requete`. Ainsi, la requête

```
SELECT *
FROM world.country
WHERE population_country >= ALL(
  SELECT population_country
  FROM world.country
);
```

retournera la liste des pays les plus peuplés.

```
SELECT *
FROM table
WHERE col < ANY(
  requete
)
```

sélectionnera les lignes de `table` telles que la valeur de `col` est strictement plus petite qu'au moins une des valeurs retournées par la requête `requete`.

Pour connaître les régions où l'on ne trouve qu'une seule forme de gouvernement, on pourra écrire :

```
SELECT DISTINCT region
FROM world.country as c1
WHERE c1.governmentform = ALL(
  SELECT c2.governmentform
  FROM world.country as c2
  WHERE c2.countrycode!=c1.countrycode AND
    c2.region=c1.region
);
```

> **i** On remarque que dans `EXISTS` ou `IN` on peut utiliser des attributs de notre requête globale, ce qui les rend plus *puissants* que
>
> ```
> WITH ... AS (
>   ...
> )
> ```

## Jointure externe

La jointure externe est une jointure un peu particulière. On a vu la semaine dernière que lorsqu'on faisait une jointure, les lignes de la table de droit étaient recollées aux lignes de la table de gauche. Si une ligne a gauche ne pouvaient pas être recollée, elle disparaissait de la jointure. La jointure extérieure permet de garder ces lignes-là malgré tout.

On utilisera `LEFT JOIN` et `RIGHT JOIN`. Par exemple, la requête suivante renvoie la liste des pays et leur langages. Les pays qui ne se trouvent pas dans la table `countrylanguage` (il y en a, l'Antarctique par exemple) seront listés quand même et les informations manquantes seront remplies avec des valeurs `NULL`.

```
SELECT *
FROM world.country AS p LEFT JOIN
  world.countrylanguage AS l ON
  p.countrycode = l.countrycode;
```

On peut utiliser cette requête pour trouver les pays qui n'ont pas de langue officielle par exemple :

```
SELECT *
FROM world.country as p LEFT JOIN
  world.countrylanguage AS l ON
  p.countrycode = l.countrycode AND l.isofficial
WHERE l.countrycode IS NULL;
```

## Requêtes

1. Quels sont les langues qui ne sont officielles dans aucun pays ? (355 lignes)

Écrivez une version avec `EXCEPT`, une avec `NOT IN` et une autre avec `LEFT JOIN`.

**Solution**

```
(
  SELECT DISTINCT language
  FROM  world.countrylanguage
)

EXCEPT

(
  SELECT language
  FROM world.countrylanguage
  WHERE isofficial
);
```

Première version

**Solution**

```
SELECT DISTINCT language
FROM world.countrylanguage
WHERE language NOT IN
      (SELECT language
       FROM world.countrylanguage
       WHERE isofficial);
```

Deuxième version :

**Solution**

```
SELECT DISTINCT l1.language
FROM world.countrylanguage AS l1
     LEFT JOIN world.countrylanguage AS l
     ON (l1.language = l.language AND l.isofficial)
WHERE l.language IS NULL;
```

Troisième version :

**Solution**

```
SELECT DISTINCT cl.language
FROM world.countrylanguage cl
WHERE  NOT EXISTS (
  SELECT cl1.language
  FROM world.countrylanguage cl1
  WHERE cl1.language=cl.language AND
  cl1.isofficial
);
```

En calcul relationnel

$$\Big\{ l.\texttt{language} : \texttt{countrylanguage}(l) \wedge \neg(\exists t \quad \texttt{countrylanguage}(t) \wedge l.\texttt{language} = t.\texttt{language} \wedge t.\texttt{isofficial}) \Big\}$$

2. Quelles sont les régions où au moins deux pays ont la même forme de gouvernement ? (21 lignes)

**Solution**

```
SELECT DISTINCT region
FROM world.country AS c1
WHERE c1.governmentform = ANY(
  SELECT c2.governmentform
  FROM world.country AS c2
  WHERE c2.countrycode!=c1.countrycode AND c2.region=c1.region
);
```

**Solution**

```
SELECT DISTINCT c1.region
FROM world.country AS c1 JOIN world.country AS c2
  ON c1.region=c2.region AND
    c1.countrycode!=c2.countrycode AND
    c1.governmentform=c2.governmentform;
```

3. Quels sont les films qui n'ont jamais été loués ? (42 lignes)

**Solution**

Là encore, plusieurs possibilités. Avec ce que l'on sait déjà :

```
WITH DejaLoue AS (
  SELECT film_id
  FROM pagila.rental JOIN pagila.inventory USING (inventory_id)
), NonLoue AS (
  SELECT film_id
  FROM pagila.film
  EXCEPT
  SELECT *
  FROM DejaLoue
)

SELECT title
FROM pagila.film NATURAL JOIN NonLoue;
```

Avec les requêtes imbriquées :

```
SELECT title,film_id FROM pagila.film
                     WHERE film_id NOT IN (
                      SELECT film_id
                      FROM pagila.rental JOIN pagila.inventory USING (inventory_id)
);
```

En calcul relationnel

$$\Big\{ f.\texttt{title} \mid \texttt{film}(f) \wedge \neg(\exists t, t_1 \quad \texttt{inventory}(t) \wedge \exists t_1 \quad \texttt{rental}(t_1) \wedge f.\texttt{film_id} = t.\texttt{film_id} \wedge t.\texttt{inventory_id} = t_1.\texttt{inventory_id}) \Big\}$$

Cette question est exactement du même type que la précédente. On y répond de la même manière : pour trouver *1* les objets d'un certain type qui ne possèdent pas une propriété, on cherche dans la base tous les objets de ce type et on fait la différence avec l'ensemble des objets de ce type qui possèdent la propriété dans la base.

4. Quels sont les acteurs qui ont joué dans toutes les catégories de film ? (11 lignes)

**Solution**

```
WITH ActCat AS (SELECT actor_id, category_id FROM pagila.film_actor fa
                  JOIN pagila.film_category fc ON (fa.film_id=fc.film_id)),
ActNot AS (SELECT actor_id FROM pagila.actor,pagila.category
    WHERE (actor_id,category_id) NOT IN (SELECT * FROM ActCat)),
ActId AS (SELECT actor_id FROM pagila.actor
    EXCEPT SELECT * FROM ActNot)

SELECT first_name,last_name FROM pagila.actor NATURAL JOIN ActId  ;
```

Cette requête réalise une opération sophistiquée de l'algèbre relationnelle la *division* ou $\div$. Il ne s'agit pas d'une opération primitive comme $\sigma, \pi, \times$.

$$\pi_{\texttt{actor_id,category_id}}(\texttt{film_actor} \bowtie \texttt{film_category}) \div \pi_{\texttt{category}}(\texttt{film_category})$$

**Solution**

La version suivante calcule le même résultat, et suit fidèlement le plan d'exécution le plus élémentaire pour réaliser la division.

```
WITH
  ActCat AS (
    SELECT actor_id, category_id
    FROM pagila.film_actor fa JOIN pagila.film_category fc ON (fa.film_id=fc.film_id)),
  ActCrosCat AS (
    SELECT  actor_id, category_id
    FROM  pagila.actor, pagila.category),
  ActNotCat AS (
    SELECT *
    FROM ActCrosCat
    EXCEPT
    SELECT *
    FROM ActCat),
  ActId AS (
    SELECT actor_id
    FROM pagila.actor
    EXCEPT
    SELECT actor_id
    FROM ActNotCat)

SELECT first_name,last_name
FROM pagila.actor NATURAL JOIN ActId  ;
```

En comptant le nombre $n$ de catégories de films dans une première requête, on peut aussi sélectionner les acteurs qui apparaissent dans au moins $n$ catégories de film.

5. Existe-t-il des acteurs qui ne jouent avec aucun autre acteur ? (0 ligne)

**Solution**

```
WITH Copain AS
  (SELECT
    R1.actor_id
   FROM
    pagila.film_actor as R1
  JOIN
    pagila.film_actor as R2
  ON
    (R1.film_id = R2.film_id AND
     R1.actor_id != R2.actor_id)
)

SELECT
  actor_id
FROM
  pagila.actor
WHERE
  actor_id NOT IN (
    SELECT *
    FROM Copain
);
```

ou avec `NOT EXISTS`

```
SELECT actor_id
FROM
  pagila.actor a
WHERE
  NOT EXISTS (
    SELECT fa2.actor_id
    FROM
      pagila.film_actor fa1
    JOIN
      pagila.film_actor fa2
    ON
     (fa1.actor_id=a.actor_id AND
      fa2.actor_id<> a.actor_id AND
      fa1.film_id=fa2.film_id)
)
```

**Solution**

ChatGPT *fecit* :

```
WITH ActorFilmCounts AS (
    -- Pour chaque acteur, chaque film, nombre de co-acteurs dans le film
    SELECT
        fa1.actor_id,
        fa1.film_id,
        COUNT(fa2.actor_id) AS other_actors_count
    FROM
        film_actor fa1
    LEFT JOIN
        film_actor fa2
    ON
        fa1.film_id = fa2.film_id
        AND fa1.actor_id <> fa2.actor_id
    GROUP BY
        fa1.actor_id, fa1.film_id
)
SELECT
    a.actor_id,
    a.first_name,
    a.last_name
FROM
    ActorFilmCounts afc
JOIN
    actor a
ON
    afc.actor_id = a.actor_id
GROUP BY
    a.actor_id, a.first_name, a.last_name
HAVING
    -- garder les acteurs qui n'ont pas de co-acteurs
    SUM(afc.other_actors_count) = 0;
```

**Une erreur**

☢ La requête suivante ne répond pas à la question posée :

```
SELECT
    fa.actor_id
FROM
    film_actor fa
LEFT JOIN
    film_actor fa2
ON
  (fa.film_id = fa2.film_id AND fa.actor_id<>fa2.actor_id)
WHERE
  fa2.actor_id IS NULL;
```

Elle liste les identifiants d'acteurs qui ont joué au moins une fois dans un film ne comportant qu'un seul acteur, alors qu'on ne cherche les acteurs qui n'ont joué que dans des films ne comportant qu'un seul acteur.
Cette requête renvoie 31 lignes.

6. Nom, prénom des clients installés dans des villes sans magasin ? (599 lignes)

**Solution**

```
WITH
  CustomerCity AS (
    SELECT
      cu.first_name,
      cu.last_name,
      cu.customer_id,
      ad.city_id
    FROM
      pagila.customer cu
    JOIN
      pagila.address ad
    ON
      (cu.address_id=ad.address_id)),
  StoreCity AS (
    SELECT
      ad.city_id
    FROM
      pagila.store st
    JOIN p
      agila.address ad
    ON
      (st.address_id= ad.address_id)
)

SELECT
  first_name,
  last_name
FROM
  CustomerCity
WHERE
  city_id NOT IN (
      SELECT * FROM StoreCity
  );
```

7. Lister les pays pour lesquels toutes les villes ont au moins un magasin. (1 ligne)

**Solution**

```
SELECT country_id from pagila.country C
WHERE NOT EXISTS (
  SELECT *
  FROM pagila.city C2
  WHERE C.country_id=C2.country_id AND C2.city_id NOT IN (
    SELECT address.city_id
    FROM pagila.store
    JOIN pagila.address USING (address_id)
    )
);
```

8. Déterminer la liste des films disponibles dans toutes les langues.

**Solution**

*Comme pour les acteurs "toutes catégories"*, il s'agit d'une *division*. Dans la base installée, le résultat est vide.

Un même *dvd* (`inventory_id`) peut bien sûr être loué plusieurs fois, mais pas simultanément. Proposer

une requête qui vérifie que les dates de location d'un *dvd* donné sont compatibles.

**Solution**

`SQL` en général et `PostGres` en particulier proposent beaucoup de types et d'opérations sophistiquées pour représenter et manipuler les données temporelles. Plusieurs types de données permettent de représenter les instants (timestamp), les dates, les intervalles de temps, les durées, et de calculer sur le temps (ajouter une durée à une date, extraire une information calendaire d'une date ou d'un instant, …). Même si l'API varie d'un cadre à l'autre, on retrouve ces types et ces opérations dans tous les environnements de sciences des données : , 

## Vues

Les *vues* permettent de donner un nom à une requête afin de pouvoir l'appeler plus tard sans la réécrire à chaque fois. Une vue s'enregistre dans un schéma. Par exemple, dans le schéma `World`, on pourrait créer une vue `VillesRepublic` qui contient toutes les villes de la table `city` qui sont dans une république.

On crée une vue avec `CREATE VIEW nom AS requete`. Étant donné que vous ne pouvez écrire que dans votre schéma personnel, il faudra nommer vos vues `entid.nom` où `entid` est votre identifiant ENT. Ainsi

```
CREATE VIEW entid.VillesRepublic AS
  SELECT
    B.*
  FROM
      world.country as A
  NATURAL JOIN
      world.city as B
  WHERE
      A.governmentform like '%Republic%';
```

crée une vue dans votre schéma personnel. Désormais, si on veut sélectionner les villes qui sont dans une république et dont la population est supérieure à 1000000, on pourra simplement écrire :

```
SELECT *
FROM
    entid.VillesRepublic
WHERE
    population_city>=1000000;
```

Remarquez la différence entre `WITH` et une vue. `WITH` nomme une requête temporairement, seulement à l'échelle de la requête courante tandis qu'une vue est enregistrée de façon permanente. Cependant, chaque fois que vous appelez votre vue, elle est réévaluée par le système de base de données.

Notez aussi que SQL n'est pas sensible à la casse. La vue `entid.VillesRepublic` peut être aussi désignée par `entid.villesrepublic`.

Pour supprimer une vue existante on utilise la commande `DROP VIEW` suivie du nom de la vue à supprimer. Par exemple l'instruction

```
DROP VIEW entid.VillesRepublic ;
```

supprime la vue créée précédemment.

Dans votre schéma personnel (qui porte le nom de votre identifiant ENT), écrire une vue `film_id_horror` qui renvoie la liste des films de catégorie 'Horror'.

**Solution**

```
CREATE VIEW entid.film_id_horror
AS
( SELECT pagila.film_id
  FROM
    pagila.film_category JOIN
    pagila.category USING(category_id)
  WHERE
    category.name='Horror'
) ;
```

## Fonctions SQL

Dans votre schéma personnel (qui porte le nom de votre identifiant ENT), écrire une fonction SQL `film_id_cat` qui prend en paramètre une chaîne de caractère `s` et renvoie la liste des films de catégorie `s`. On rappelle la syntaxe :

```
CREATE OR REPLACE FUNCTION entid.film_id_cat(s TEXT)
RETURNS TABLE(film_id INTEGER)
LANGUAGE 'sql' AS
$$
requete
$$
```

et l'usage

```
CREATE OR REPLACE FUNCTION
  entid.film_id_cat(s text)
RETURNS TABLE(film_id smallint) AS
$$
    SELECT fc.film_id
    FROM
        pagila.film_category fc
  JOIN
        pagila.category ca
  ON (fc.category_id=ca.category_id)
    WHERE
    ca.name=s ;
$$ LANGUAGE sql ;
```

Utilisez votre fonction pour écrire les requêtes suivantes :

Quels sont les acteurs qui ont déjà joué dans un film d'horreur (catégorie 'Horror') ?

**Solution**

Les solutions sont données en utilisant la fonction suivante

```
CREATE OR REPLACE FUNCTION
  entid.film_id_cat(s pagila.category.name%TYPE)
RETURNS TABLE(film_id pagila.film.film_id%TYPE)
AS $$
SELECT fc.film_id
FROM
    pagila.film_category fc
JOIN
    pagila.category ca
ON (fc.category_id=ca.category_id)
WHERE
  ca.name=s ;
$$ LANGUAGE sql;
```

**Solution**

```
SELECT DISTINCT ac.*
FROM
    pagila.actor ac
NATURAL JOIN
  (SELECT fa.actor_id
     FROM
    pagila.film_actor fa
   WHERE
    fa.film_id IN (
      SELECT *
          FROM entid.film_id_cat('Horror')
    )
  );
```

ou

**Solution**

```
SELECT DISTINCT ac.*
FROM
  pagila.actor ac
JOIN
  pagila.film_actor fa ON (ac.actor_id=fa.actor_id)
NATURAL JOIN
  entid.film_id_cat('Horror') ;
```

(156 tuples renvoyés).

Quels sont les acteurs qui n'ont jamais joué dans une comédie (`Comedy`) ? (53 lignes)

**☢ Attention ! Cette requête ne répond pas à la question :**

```
SELECT DISTINCT ac.*
FROM pagila.actor ac NATURAL JOIN
     (SELECT * FROM pagila.film_actor
      WHERE film_id NOT IN
        (SELECT * FROM pagila.film_id_cat('Comedy') )
      ) as X;
```

Elle répond à la question : *Quels sont les acteurs qui ont joué dans un film qui n'est pas une comédie ?*

**Solution**

Une réponse correcte est

```
SELECT DISTINCT last_name, first_name
FROM
  pagila.actor A1
WHERE
  NOT EXISTS
  (SELECT *
  FROM
    pagila.film_actor A2,
    (pagila.film_category JOIN pagila.category using (category_id)) C
  WHERE
      A1.actor_id=A2.actor_id AND
      name='Comedy' AND
      A2.film_id=C.film_id
  );
```

ou encore

**Solution**

```
SELECT
  DISTINCT ac.last_name, ac_first_name
FROM
  pagila.actor ac
WHERE NOT EXISTS
  (SELECT *
   FROM
    pagila.film_actor fa
   WHERE film_id IN
     (SELECT *
      FROM
        entid.film_id_cat('Comedy')
     ) AND
    fa.actor_id = ac.actor_id
  )  ;
```

En calcul relationnel, en considérant `film_id_cat('Comedy')` comme une relation (ce qui est cohérent avec la définition de la fonction) cette requête s'exprime

$$\{a.\texttt{last_name}, a.\texttt{first_name} : \texttt{actor}(a) \wedge \neg(\exists \text{fa} \quad \texttt{film_actor}(\text{fa}) \wedge \text{fa}.\texttt{actor_id} = a.\texttt{actor_id} \wedge \texttt{film_id_cat}('\text{Comedy}')(\text{fa}.\texttt{film_id}))\}$$

Le calcul relationnel traduit presque littéralement la démarche que nous suivons lorsqu'il faut construire le résultat à la main : pour trouver les `actor_id` des acteurs qui n'ont jamais joué dans une comédie, nous examinons toutes les valeurs $a$ de `actor_id` présentes dans la table `actor` (ou `film_actor`), et pour chacune de ces valeurs, nous verifions qu'il n'existe pas de tuple de la table `film_actor` où l'attribut `actor_id` soit égal à $a$ et où l'attribut `film_id` désigne un film qui apparaît dans le résultat de `film_id_cat('Comedy')`.
Nous *décrivons/explicitons* ainsi les propriétés du résultat de la requête *Quels sont les acteurs qui n'ont jamais joué dans une comédie ('Comedy') ?*.
Si maintenant nous cherchons à *1* ce résultat, nous pouvons d'abord calculer la liste des `actor_id` des acteurs qui ont joué dans une comédie, calculer la liste de tous les `actor_id` connus dans le schema et faire la différence, en algèbre relationnelle, cela se résume à

$$\pi_{\texttt{actor_id}}(\texttt{film_actor}) \setminus \pi_{\texttt{actor_id}}(\texttt{film_actor} \bowtie \texttt{film_id_cat}(\texttt{'Comedy'}))$$

Quels sont les acteurs qui ont joué dans un film d'horreur ('Horror') et dans un film pour enfant ('Children')? (130 lignes)

Ici l'erreur la plus fréquente consiste à écrire

```
SELECT
  actor_id
FROM
  pagila.film_actor AS fa
WHERE
  fa.film_id IN (
    SELECT *
    FROM entid.film_id_cat('Children')
  ) AND
  fa.film_id IN (
    SELECT *
    FROM entid.film_id_cat('Horror')
  );
```

Le résultat est vide et la requête ne correspond pas à la question posée.
Elle calcule les `actor_id` des acteurs qui ont dans au moins un film qui relève simultanément des catégories `Horror` et `Children` (ce genre de film est assez rare).
Pour calculer un résultat correct, il faut pour chaque valeur $a$ de `actor_id` rechercher deux tuples (pas nécessairement distincts) de `film_actor` où l'attribut `actor_id` vaut $a$ et ou dans un cas `film_id` désigne un film pour enfants et dans l'autre un film d'horreur. En calcul relationnel, cela donne

$$\begin{aligned}\{a.\texttt{last_name}, a.\texttt{first_name} : \texttt{actor}(a) \wedge \\ (\exists \text{fa} \quad \texttt{film_actor}(\text{fa}) \wedge \text{fa}.\texttt{actor_id} = a.\texttt{actor_id} \\ \wedge \texttt{film_id_cat}('\text{Children}')(\text{fa}.\texttt{film_id})) \\ (\exists \text{fa} \quad \texttt{film_actor}(\text{fa}) \wedge \text{fa}.\texttt{actor_id} = a.\texttt{actor_id} \\ \wedge \texttt{film_id_cat}('\text{Horror}')(\text{fa}.\texttt{film_id}))\}\end{aligned}$$

En algèbre relationnelle

$$\begin{aligned}\pi_{\texttt{last_name},\texttt{first_name}} \Big( & \texttt{actor} \bowtie \\ & \Big( \pi_{\texttt{actor_id}} (\texttt{film_actor} \bowtie \texttt{film_id_cat}(\texttt{'Children'})) \bigcap \\ & \pi_{\texttt{actor_id}} (\texttt{film_actor} \bowtie \texttt{film_id_cat}(\texttt{'Horror'})) \Big) \Big)\end{aligned}$$

**Solution**

En SQL, cela peut donner

```
SELECT DISTINCT a.first_name, a.last_name  FROM pagila.actor a
WHERE EXISTS (SELECT film_id
              FROM film_actor AS fa1 NATURAL JOIN
                   entid.film_id_cat('Children')
              WHERE  fa1.actor_id = a.actor_id) AND
      EXISTS (SELECT film_id
              FROM film_actor AS fa2 NATURAL JOIN
                   entid.film_id_cat('Horror')
              WHERE  fa2.actor_id = a.actor_id) ;
```

qui renvoie 129 tuples.