TD 3 : SQL

SQL Interrogation de données, requêtes complexes

2025-10-10

L3 MIASHS
Université Paris Cité
Année 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
);
```

> 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`.

2. Quelles sont les régions où au moins deux pays ont la même forme de gouvernement ? (21 lignes)
3. Quels sont les films qui n'ont jamais été loués ? (42 lignes)

> **i** En calcul relationnel
>
> $$\Big\{ f.\texttt{title} : \texttt{film}(f) \wedge \neg(\exists t, t_1 \quad \texttt{inventory}(t) \wedge \exists t_1 \quad \texttt{rental}(t_1) \wedge f.\texttt{film\_id} = t.\texttt{film\_id} \wedge t.\texttt{inventory\_id} = t_1.\texttt{inventory\_id}) \Big\}$$

> **i** 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)

> **!** Cette requête réalise une opération sophistiquée de l'algèbre relationnelle la *division* ou $\div$. Il ne s'agit pas d'une opération primitive comme $\sigma, \pi, \times$.
>
> $$\pi_{\texttt{actor\_id,category\_id}}(\texttt{film\_actor} \bowtie \texttt{film\_category}) \div \pi_{\texttt{category}}(\texttt{film\_category})$$

5. Existe-t-il des acteurs qui ne jouent avec aucun autre acteur ? (0 ligne)
6. Nom, prénom des clients installés dans des villes sans magasin ? (599 lignes)
7. Lister les pays pour lesquels toutes les villes ont au moins un magasin. (1 ligne)
8. Déterminer la liste des films disponibles dans toutes les langues.

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.

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;
```

> **i** 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'.

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') ?

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 ?*

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

$$\begin{aligned}\pi_{\texttt{last\_name},\texttt{first\_name}} \Big(& \texttt{actor} \bowtie \\ & \Big(\pi_{\texttt{actor\_id}} (\texttt{film\_actor} \bowtie \texttt{film\_id\_cat}(\texttt{'Children'})) \bigcap \\ & \pi_{\texttt{actor\_id}} (\texttt{film\_actor} \bowtie \texttt{film\_id\_cat}(\texttt{'Horror'})) \Big) \Big)\end{aligned}$$