Aujourd'hui, nous allons utiliser un autre schéma, le schéma `sakila` qui contient des informations utilisées par un magasin de location de DVD. Le schéma est visible sur Moodle (Sciences / UFR de math / Master ISIFAR / BD ). Vous pouvez aussi inspecter les tables comme d'habitude avec

```
bd_2017=# \d sakila.film
bd_2017=# \d sakila.actor
```

# 1 Requêtes SQL (encore plus) avancées

## 1.1 Requêtes imbriquées

Les requêtes imbriquées permettent d'utiliser le resultat d'une requête dans la clause `WHERE` d'une requête. On utilisera essentiellement les deux connecteurs suivants : `IN, EXISTS`. `IN` permet de tester la présence d'une valeur dans le resultat 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 films disponibles en allemand, on pourra écrire :

```
SELECT * FROM sakila.film
WHERE language_id IN (SELECT language_id
                      FROM sakila.language
                      WHERE name='German');
```

Pour connaître les acteurs qui ont joué dans au moins un film, on pourra écrire :

```
SELECT * FROM sakila.actor AS ac
WHERE EXISTS (SELECT *
              FROM sakila.film_actor as fa
              WHERE fa.actor_id = ac.actor_id);
```

## 1.2 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;
```

# 2 Exercices

## 2.1 Requêtes

1. Quels sont les langues qui ne sont officielles dans aucun pays ? Écrivez une version avec `NOT IN` et une autre avec `LEFT JOIN`.

**Solution:** Première version :

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

Deuxième version :

```
SELECT l1.language FROM countrylanguage as l1
                   LEFT JOIN countrylanguage as l
                   ON (l1.language = l.language and l.isofficial)
                   WHERE l.language IS NULL;
```

Troisième version :

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

En calcul relationnel

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

2. Quels sont les films qui n'ont jamais été loués ?

**Solution:** Là encore, plusieurs possibilités. Avec ce qu'on a vu la semaine dernière :

```
WITH DejaLoue AS
(SELECT film_id FROM sakila.rental NATURAL JOIN sakila.inventory),
NonLoue AS
(SELECT film_id FROM sakila.film EXCEPT SELECT * FROM DejaLoue)

SELECT title FROM film NATURAL JOIN NonLoue;
```

Avec les requêtes imbriquées :

```
SELECT title ,film_id FROM sakila.film
                      WHERE film_id NOT IN (SELECT film_id FROM
                       sakila.rental NATURAL JOIN sakila.inventory);
```

En calcul relationnel

$$\Big\{ f.\texttt{title} : \texttt{film}(f) \wedge \neg\big(\exists t, t_1 \quad \texttt{inventory}(t) \wedge \exists t_1 \quad \texttt{rental}(t_1) \wedge f.\texttt{film\_id} = t.\texttt{film\_id} \wedge t.\texttt{inventory\_id} = t_1.\texttt{inventory\_id}\big)\Big\}$$

Cette question est exactement du même type que la précédente. On y répond de la même de la même manière : pour trouver *dans la base* 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.

3. Quels sont les acteurs qui ont joués dans toutes les catégories de film ?

**Solution:**

```
WITH ActCat AS (SELECT actor_id, category_id FROM sakila.film_actor
                NATURAL JOIN sakila.film_category),
ActNot AS (SELECT actor_id FROM sakila.actor, sakila.category
    WHERE (actor_id, category_id) NOT IN (SELECT * FROM ActCat)),
ActId AS (SELECT actor_id FROM sakila.actor
    EXCEPT SELECT * FROM ActNot)

SELECT first_name, last_name FROM sakila.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 $\sigma, \pi, \times$.

$$\pi_{\texttt{actor\_id,category\_id}}(\texttt{film\_actor} \bowtie \texttt{film\_category}) \div \pi_{\texttt{category}}(\texttt{film\_category})$$

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

```
WITH ActCat AS (SELECT actor_id, category_id FROM
                sakila.film_actor NATURAL JOIN sakila.film_category),
  ActCrosCat AS (SELECT  actor_id, category_id FROM  sakila.actor,
                                                     sakila.category),
  ActNotCat AS (SELECT * FROM ActCrosCat
                EXCEPT SELECT * FROM ActCat),
  ActId AS (SELECT actor_id FROM sakila.actor EXCEPT
                SELECT actor_id FROM ActNotCat)

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

4. Existe-t-il des acteurs qui ne jouent avec aucun autre acteur ?

**Solution:**

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

```
SELECT actor_id FROM sakila.actor
    WHERE actor_id not in (SELECT * FROM Copain);
```

ou avec `NOT EXISTS`

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

Cette question ressemble beaucoup aux questions 1 et 2.

5. Nom, prénom des clients installés dans des villes sans magasins ?

**Solution:**

```
WITH CustomerCity AS
(SELECT first_name,last_name,customer_id, city_id
FROM sakila.customer NATURAL JOIN sakila.address),

StoreCity AS
(SELECT city_id FROM sakila.store NATURAL JOIN sakila.address)

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

6. Lister les pays pour lesquels toutes les villes ont au moins un magasin.

**Solution:** *Chaque fois qu'on lit dans la questions « tous » ou « toutes », on est tenté de voir une division. Ici ce n'est pas le cas. Il faut d'abord déterminer les villes sans magasins, puis leur pays et retirer ces pays de la liste de tous les pays.*

```
WITH city_without_store AS (
    SELECT city_id FROM sakila.city
    EXCEPT
    SELECT DISTINCT c.city_id
        FROM sakila.store NATURAL JOIN sakila.address
                          NATURAL JOIN sakila.city c
)

SELECT * FROM sakila.country co
WHERE NOT EXISTS (SELECT * FROM
                  sakila.country NATURAL JOIN city_without_store);
```

## 2.2 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
$$
requete
$$
```

**Solution:**

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

Utilisez votre fonction pour écrire les requêtes suivantes :

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

**Solution:**

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

ou

```
SELECT DISTINCT ac.*
FROM sakila.actor ac NATURAL JOIN
      sakila.film_actor NATURAL JOIN
      entid.film_id_cat('Horror') ;
```

(156 tuples renvoyés).

2. Quels sont les acteur qui n'ont jamais joué dans une comédie ('Comedy') ?

**Solution:**

Attention ! Cette requête ne répond pas à la question :

```
SELECT DISTINCT ac.*
FROM sakila.actor ac NATURAL JOIN
      (SELECT * FROM sakila.film_actor
       WHERE film_id NOT IN
          (SELECT * FROM entid.film_id_cat('Comedy')
          )
       ) fa ;
```

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

Une réponse correcte est

```
SELECT DISTINCT ac.last_name, ac_first_name
FROM sakila.actor ac
WHERE NOT EXISTS
      (SELECT * FROM sakila.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 acteur qui n'ont jamais joué dans une comédie ('Comedy') ?*.

Si maintenant nous cherchons à *calculer* 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

$$\pi_{\texttt{actor\_id}}(\texttt{film\_actor}) \setminus \pi_{\texttt{actor\_id}}(\texttt{film\_actor} \bowtie \texttt{film\_id\_cat('Comedy')})$$

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

**Solution:** Ici l'erreur la plus fréquente consiste à écrire

```
SELECT actor_id FROM sakila.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

$$\{a.\texttt{last\_name}, a.\texttt{first\_name} : \texttt{actor}(a) \wedge (\exists \text{fa} \quad \texttt{film\_actor}(\text{fa}) \wedge \text{fa}.\texttt{actor\_id} = a.\texttt{actor\_id} \wedge \texttt{film\_id\_cat}(\text{'Children'})(\text{fa}.\texttt{film\_id})) (\exists \text{fa} \quad \texttt{film\_actor}(\text{fa}) \wedge \text{fa}.\texttt{actor\_id} = a.\texttt{actor\_id} \wedge \texttt{film\_id\_cat}(\text{'Horror'})(\text{fa}.\texttt{film\_id}))\}$$

En algèbre relationnelle

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

En SQL, cela peut donner

```
SELECT DISTINCT a.first_name, a.last_name FROM sakila.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.