

## BDR - Devoir surveillé

21 mai 2021 - durée 2h

Documents autorisés. Appareils électroniques de communication interdits.

**Exercice 1 :** Dans cet exercice nous allons étudier des optimisations d'exécution de requêtes appelées *fusion de boucles et de conditions*. Pour cela, nous allons manipuler des flux **Python**.

Nous allons commencer par la fusion de condition. Pour rappel, la sélection (ou filtre) dans un flux s'implémente de la manière suivante :

```
def selection(table, pred):  
    """Construit le flux des éléments de ~table~ qui satisfont le prédicat  
    ~pred~ (fonction des tuples dans les booléens)."""  
    for tp in table:  
        if pred(tp):  
            yield tp
```

On peut alors réécrire l'expression :

```
selection(selection(tb,pred1), pred2)
```

en l'expression équivalente

```
selection(tb, lambda x: pred1(x) and pred2(x))
```

Remarquez que cette expression n'utilise qu'une seule fois la fonction **selection**. Cela évite de créer un flux intermédiaire et rend l'exécution plus efficace.

Question 1.1 : Nous allons passer à la fusion de boucles. La transformation de flux s'implémente de la manière suivante :

```
def transformation(table, f):  
    """Renvoie un flux obtenu en appliquant ~f~ à chacun des tuples composant  
    ~table~."""  
    for tp in table:  
        yield f(tp)
```

Écrivez une expression équivalente à :

```
transformation(transformation(tb,f),g)
```

de sorte qu'elle n'utilise qu'une seule fois la fonction **transformation**.

Question 1.2 : Afin de combiner les deux formes de réécriture que nous avons vues lors des questions précédentes, écrivez la fonction **selection\_transformation** décrite ci-dessous :

```
def selection_transformation(table, pred, f):  
    """Renvoie le flux des éléments de ~table~ qui satisfont le prédicat  
    ~pred~ et auxquels on a appliqué la fonction ~f~."""
```

Question 1.3 : Réécrivez l'expression suivante en une expression équivalente qui n'utilise qu'une seule fois la fonction `selection_transformation` :

```
transformation(transformation(selection(selection(table, pred1), pred2), f), g)
```

Question 1.4 : Pour rappel, voici le code de la fonction `union` :

```
def union(tbl1, tbl2):
    """construit un flux qui énumère les éléments de ~t1~ puis ceux de ~t2~."""
    for tp in tbl1:
        yield tp
    for tp in tbl2:
        yield tp
```

Supposons que les flux `tb1` et `tb2` soient identiques (ils énumèrent les mêmes tuples dans le même ordre). Est-ce que les expressions

```
union(selection(tb1, pred1), selection(tb2, pred2))
```

et

```
selection(tb1, lambda x: pred1(x) or pred2(x))
```

produisent les mêmes flux ? Si oui, expliquez pourquoi ; si non expliquez les différences.

**Exercice 2 :** Dans cet exercice, écrivez une fonction respectant la spécification ci-dessous :

```
def compte_minimum_table(table, col):
    """Renvoie le nombre d'occurrence de la plus petite des valeurs associées à
    l'attribut ~col~ dans ~table~. Si ~table~ est vide, renvoie 0.
    """
```

**NB :** veillez à ce que l'empreinte mémoire de votre fonction ne dépende pas de la taille du flux d'entrée `table`.

**Exercice 3 :** Le service météo du Royaume-Uni collecte les relevés de températures des stations météo de son territoire dans la table `temperatures` de sa base de données. Pour simplifier, cette table contient seulement trois colonnes :

1. une colonne `temperature`, la température du relevé en degrés Celsius,
2. une colonne `timestamp`, la date du relevé,
3. une colonne `station_id`, l'identifiant de la station météo du relevé.

Question 3.1 : Sachant que la colonne `temperature` est indexée, donnez une requête qui utilisera cet index et qui donne les stations dont la température était supérieure à 26 degrés Fahrenheit le 31 décembre 2020 à minuit.

Pour écrire cette requête, il vous faut savoir que :

1. si  $T$  est une température en degrés Celsius, alors  $\frac{9}{5}T + 32$  est cette température exprimée en degrés Fahrenheit,

2. minuit le 31 décembre 2020 est représenté par '2021-01-01 00:00:00' en sql.

Question 3.2 : On définit (timestamp, station\_id) comme clé primaire de la table `temperatures`. Est-ce que l'index créé sur cette clé primaire est utile pour les requêtes suivantes (justifier) :

```
SELECT * FROM temperatures
WHERE station_id = 63 ;
```

```
SELECT * FROM temperatures
WHERE timestamp between '2021-01-01 00:00:00' and '2021-02-01 00:00:00' ;
```

**Exercice 4 :** Cet exercice manipule des transactions sous postgresql. Pour mémoire, la norme SQL définit des niveaux d'isolement en fonction des anomalies dont on souhaite se prémunir.

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Dans un outil de simulation d'un supermarché, on dispose d'une table mémorisant les caisses ouvertes et fermées. Cette table a pour schéma `caisse(id, ouverte)`. On suppose que pour cet exercice, il existe trois caisses nommées *A*, *B* et *C*.

Question 4.1 : Dans l'ordonnancement suivant,  $T_1$  lit l'état de la caisse *A*, et  $T_2$  modifie la valeur de cet état (de ouverte=true à ouverte=false).

$T_1$	$T_2$
begin (lire, A) <i>A caisse ouverte</i>  (lire, A) <i>A caisse ouverte</i>  (lire, A) <i>A caisse fermée</i> commit	begin  (lire, A) <i>A caisse ouverte</i> (ecrire, A) <i>A caisse fermée</i>  commit

1. Quel problème est mis en évidence ?
2. En déduire dans quel(s) niveau(x) d'isolement ces transactions peuvent se dérouler.

Le schéma dispose aussi d'une procédure stockée `fermerCaisse` qui ferme une caisse à condition qu'il en reste au moins 2 d'ouvertes. Cette procédure est exécutée dans une transaction :

```
create or replace procedure fermerCaisse (numCaisse integer) AS $$
DECLARE  nb_ouvertes integer ;
BEGIN
    select count(*) into nb_ouvertes from Caisse where ouverte = true;
```

```

if (nb_ouvertes > 2) then
    update Caisse set ouverte = false where id = numCaisse ;
    commit;
end if;
END; $$
LANGUAGE plpgsql ;

```

L'instruction select de la procédure fermerCaisse se traduit au niveau transactionnel par (lire, A)(lire ,B)(lire, C).

Sachant que les 3 caisses sont ouvertes, voici l'ordonnancement de deux transactions concurrentes  $T_1$  et  $T_2$ , l'une qui ferme la caisse A, l'autre qui ferme la caisse B :

$T_1$	$T_2$
begin	begin
(lire, A)	
(lire ,B)	
(lire, C)	
	(lire, A)
	(lire, B)
	(lire, C)
(ecrire, A)	
	(ecrire, B)
commit	commit

Question 4.2 : Quel est, avec cet ordonnancement, le nombre de caisses ouvertes constaté par  $T_1$  et celui constaté par  $T_2$  au début de la procédure ?

Question 4.3 : Quel est le nombre de caisses ouvertes à la fin, quand  $T_1$  et  $T_2$  ont validé ?

Question 4.4 : Est-ce que cet ordonnancement est sérialisable ? Justifiez votre réponse en comparant avec les exécutions en série ( $T_1 ; T_2$ ) et ( $T_2 ; T_1$ ).

**Exercice 5 :** Une université souhaite mettre à disposition des professeurs et des élèves un service de bases de données PostgreSQL. La direction de l'université a demandé aux personnels techniques de mettre en place ce service.

Pour les questions qui suivent, vous trouverez des informations générales et des éléments de syntaxe d'instructions SQL dans les annexes A et B.

Question 5.1 : Un personnel technique est en charge d'administrer le SGBD (créer les rôles, créer les bases, etc.). Donnez la/les commande(s) SQL pour créer cet utilisateur **bda** avec les privilèges nécessaires.

L'administrateur **bda** va devoir créer beaucoup d'élèves et de professeurs. Les utilisateurs ainsi créés devront respecter les contraintes suivantes :

- Chaque professeur ou élève a accès à une base de données dont le nom est son nom d'utilisateur du SGBD, et peut créer des tables dans cette base. Ils ne peuvent cependant pas créer de nouvelles bases de données.
- Le SGBD contient une base de données spéciale – appelons la **edt** – avec une table **emplois** qui servira à indiquer les emplois du temps. Les élèves n'y auront accès qu'en lecture tandis que les professeurs pourront modifier les données qu'elle contient. Ni les élèves ni les professeurs ne peuvent créer d'autres tables dans cette base.
- Aucun professeur ni élève ne doit pouvoir se connecter à la base de données d'un autre professeur ou élève.

Question 5.2 : On suppose que **bda** a créé la base **edt** et la table **emplois**. Donnez les instructions SQL qu'il doit exécuter pour - tout en respectant les contraintes ; créer les rôles génériques **prof** et **eleve** ; créer deux utilisateurs : un professeur **prof1** et un étudiant **elev1**, avec chacun une base à son nom, et avec les privilèges nécessaires. Faites en sorte que l'administrateur ait le moins de travail possible à la création d'un nouvel utilisateur, en factorisant la gestion des droits au niveau des rôles génériques, autant que possible.

Question 5.3 : La machine sur laquelle tourne le serveur sera considérée comme sûre et seul le personnel technique y a accès. 30 machines sont à disposition des élèves et professeurs à l'université, aux adresses 10.134.16.1 jusqu'à 10.134.16.30. Sur ces postes chaque élève et professeur a un compte, et le nom de compte est identique à celui de la base de données créée pour eux. En vous aidant de l'annexe C, proposez une configuration qui fasse en sorte que : (a) n'importe qui peut se connecter sur n'importe quelle base depuis la machine sur laquelle tourne le serveur ; (b) les élèves et professeurs peuvent se connecter à leur base et à la base **edt** sans mot de passe depuis les machine de TP ; et (c) les élèves et professeurs peuvent se connecter à leur base et à la base **edt** depuis n'importe où en indiquant leur mot de passe.

## Annexes pour l'exercice 5

### A Privilèges

Nous rappelons que tout objet (base, table, etc.) a un propriétaire, que ce propriétaire a tous les droits sur l'objet (le détruire, créer des objets dans celui-ci, donner des droits à n'importe qui sur l'objet, etc.), et que par défaut le propriétaire d'un objet est le rôle ayant exécuté l'instruction **CREATE**. Syntaxe pour donner des privilèges à un rôle :

**GRANT** <liste de privilèges> **ON** <objet> **TO** <liste de rôles>.

Les privilèges pertinents pour le sujet :

- **SELECT**, **INSERT**, **UPDATE**, **DELETE** : quand appliqués sur une table, donnent le droit d'exécuter les instructions SQL associées.
- **CREATE** : quand appliqué sur une base de données, donne le droit de créer des tables dans cette base.
- **CONNECT** : quand appliqué sur une base de données, donne le droit de s'y connecter.

## B Rôles

Nous rappelons qu'un rôle peut correspondre soit à un utilisateur, soit à un groupe d'utilisateurs. Syntaxe pour créer un rôle : `CREATE ROLE rolename <liste d'attributs>`. Syntaxe pour ajouter l'appartenance d'un rôle à un autre rôle :

`GRANT group_role TO <liste de rôles>`.

Les attributs pertinents pour le sujet :

- `CREATEDB` : permet de créer des bases de données. Pour rappel, la syntaxe pour créer une base est `CREATE DATABASE dbname`.
- `CREATEROLE`
- `LOGIN` : indique si le rôle a le droit de se connecter au SGBD.
- `PASSWORD 'string'` : le rôle est protégé par un mot de passe (sous condition que la politique d'authentification du SGBD soit correctement configurée).

## C Authentification

Le fichier `pg_hba.conf` contient des enregistrements du type

`local database role auth-method`

et

`host database role address auth-method,`

un par ligne. Lors d'une tentative de connexion en tant qu'un utilisateur sur une base (par exemple avec `psql -U utilisateur -d base`), la première ligne qui correspond au type de connexion est utilisée pour faire l'authentification et si celle-ci échoue les autres lignes ne sont pas considérées. Nous rappelons que pour se connecter à une base, il faut également avoir le privilège `CONNECT` sur celle-ci. Informations pertinentes pour l'exercice :

- `database` : indique le nom de la base en question. La valeur `all` indique que l'enregistrement concerne toutes les bases de données. La valeur `sameuser` indique que l'enregistrement coïncide si la base de données demandée a le même nom que l'utilisateur demandé.
- `role` : indique les utilisateurs de bases de données auxquels cet enregistrement correspond. La valeur `all` indique qu'il concerne tous les utilisateurs. Dans le cas contraire, il s'agit soit du nom d'un utilisateur spécifique de bases de données ou d'un nom de groupe précédé par un `+`. Par exemple `+vendeur` s'applique à tout utilisateur qui a le rôle `vendeur`.
- `address` : indique l'adresse IP ou la plage d'adresses IP à laquelle correspond cet enregistrement. On rappelle que `0.0.0.0` représente toutes les adresses possibles.
- `auth-method` :
  - `trust` : autorise la connexion sans condition.
  - `md5` : réalise une authentification par mot de passe.
  - `ident` : récupère le nom de l'utilisateur en contactant le serveur d'identification sur le poste client, et vérifie que cela correspond au nom d'utilisateur demandé. En d'autres termes et pour simplifier, si le login UNIX du client sur la machine est le même que le nom d'utilisateur demandé, l'authentification réussit (sans demander de mot de passe).