université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

25/10/2024

Traîtement Distribué des Données
Master 1 ISD

TP Noté

durée 2h15

Lire en entier les consignes suivantes

Le TP noté est sur 20 points et comportes 3 exercices. Il est conseillé de passer environ **40 minutes** par exercice. Il est aussi conseillé de laisser du code en commentaire si ce dernier ne compile pas ou qu'il donne un mauvais résultat, des points pourront tout de même être attribués.

La consultation de *documents de la page du cours* est autorisée. Les logiciels de messagerie et les IA génératives type ChatGPT, Copilot, ... ne sont pas des documents. De même les sites du style StackOverflow ou tout autre source d'information non présente sur la page du cours n'est pas un document de cours. Leur utilisation sera considérée comme une tentative de fraude. En cas de problème spécifique de syntaxe SQL, Python ou Java, vous pouvez poser des questions.

Description de l'archive

Télécharger sur le site du cours l'archive du TP noté. Décompresser l'archive. Cette dernière contient les fichiers et répertoires suivants :

`infos.txt` : un simple fichier texte **à compléter** avec votre nom, votre prénom et votre adresse email

`init.sh` : le script initialisant l'environnement du TP noté

`exo2.sh` : un script permettant de tester facilement l'exercice 2 (Hadoop)

`rendu.sh` : un script générant une archive contenant tout vos fichiers, à déposer via le formulaire disponible sur la page du cours

`TDD_TP_NOTE_2024_ACOMPLETER` : le projet Eclipse contenant les exercices 1 (SQL) et 2 (Hadoop)

`mostCentralCities.py` : le code à compléter de l'exercice 3 (Spark)

`res_exo3.txt` : le résultat attendu pour l'exercice 3

Informations

Renseigner votre nom, prénom, adresse email (de préférence `@etu-upsaclay.fr` sinon adresse personnelle).

Import du projet Eclipse

— Ouvrir Eclipse (laisser le workspace par défaut et faire OK)
— Dans le menu `File` choisir `Import ...`
— Choisir `General` puis `Existing Project into Workspace`
— Choisir `Select root directory` (attention, ce n'est pas comme en TP)
— Faire `Browse ...` et aller sélectionner le répertoire `TDD_TP_NOTE_2024_ACOMPLETER`

Il est **interdit** de renommer les classes et packages fournis. Vous pouvez cependant ajouter des classes et méthodes auxiliaires si besoin.

Rendu du projet

Se placer dans le répertoire contenant le script `rendu.sh` et l'exécuter :

```
./rendu.sh
```

Le script liste les fichiers ajoutés dans l'archive. L'archive se trouve dans le répertoire supérieur avec le nom `tdd_m1_isd_tp_note_2024_`d`_`h`.tar.gz` où d et h sont la date et l'heure de création de l'archive. Vous pouvez déposer l'archive créé via le formulaire sur la page du cours. Vous pouvez déposer autant de fois que vous voulez. Il est suggéré de soumettre après chaque exercice.

Exercice 1 : Transactions SQL (7 points)

On souhaite implémenter une procédure simple sur une table contenant des billets d'avions. Toute l'application est contenu dans une classe `PlaneTicketDB` du package `tdd.tpnote.exo1`. Cette dernière possède les méthodes suivantes :

`PlaneTicketDB(String host, String user, String base, String pw)` le constructeur, qui prend en argument les paramètres de connexion à la base de donnée (ne pas modifier).

`Connection connect()` qui renvoie un objet `Connection` sur lequel créer des `Statement` (ne pas modifier).

`void init ()` qui initialise une table

```
CREATE TABLE TICKETS_xxx (CITY1 VARCHAR(150), -- ville départ
                          CITY2 VARCHAR(150), -- ville d'arrivée
                          PRICE FLOAT, -- prix
                          QTITY INTEGER) -- quantité
```

représentant des billets d'avions (à lire mais ne pas modifier).

`void afficheInfos(String depart)` qui affiche sur la sortie standard les lignes de la table ayant pour ville de départ `depart` (ne pas modifier).

`String ajustePrixEtQtite(String depart)` qui ajuste le prix et la quantité de certains billets **(à compléter, cf. question 2)**.

`void test1()` qui effectue en séquence 5 ajustements de prix pour la ville `"London"` (en appelant la méthode `ajustePrixEtQtite("London")`, ne pas modifier) .

`void test2()` qui utiliser des `Threads` pour effectuer en parallèle 5 mises à jours de prix pour la ville `"London"` (en appelant la méthode `ajustePrixEtQtite("London")`, **à compléter, cf. question 3**).

`void main(String[] args` la méthode principale (`à modifier pour saisir les paramètres de connexion et tester les autres méthodes, cf. question 1`).

Questions

1. (0 point) **Très important :** modifier le constructeur de la classe `PlaneTicketDB` pour que l'attribut `TABLE_NAME` soit de la forme `TICKETS_prenom_nom` où `prenom` et `nom` sont votre prénom et votre nom. Utiliser uniquement des majuscules non accentuées, aucun espace et aucun symbole « - ». Toutes les sessions vont utiliser la même base SQL, il faut donc que les noms de tables soient unique. Dans toute la suite, vous construirez vos requêtes en utilisant l'attribut `TABLE_NAME` par exemple :

   ```
   /* ne pas oublier les espaces après FROM et avant WHERE */
   String query = "SELECT * FROM " + TABLE_NAME + " WHERE price >= 10";
   stmt.executeQuery(query);
   ```

2. (4.5 points) Compléter la méthode `ajustePrixEtQtite(String depart)`. Cette dernière doit réaliser la réservation de la manière suivante :
   - Commencer une transaction
   - Rechercher toutes les billets pour lequel `CITY1` vaut `depart`. La requête `SELECT` que vous écrivez doit forcément se terminer par `FOR UPDATE`. Par exemple, la chaîne Java construite devra être de la forme :

     ```
     "SELECT ... FROM ... WHERE ... FOR UPDATE"
     ```

     Pour chacun de ces billets :
     - si la quantité est strictement supérieure à 5, multiplier le prix du billet par 0.9 et diviser la quantité par 2
     - sinon, multiplier le prix du billet par 1.1 et multiplier la quantité par 2
   - Commiter la transaction

   En cas d'erreur après le début de la transaction, effectuer un `rollback`.

   Encore une fois ne pas oublier de rajouter à la fin de la requête `SELECT` la clause `FOR UPDATE`. Cette dernière permet d'indiquer au SGBD que les lignes sélectionnées vont servir à une mise à jour dans la même transaction. En effet, nous sommes ici dans un cas très particulier :

— Une requête `SELECT` est effectuée.
— Pour chaque résultat (« pendant que le `SELECT` s'effectue encore »), on exécute un ordre `UPDATE`.
Ce modèle de calcul est différent des exemples vus (un `SELECT` complet suivi d'un `UPDATE`) et nécessite la directive `FOR UPDATE`.

Vous pouvez tester vos résultats avec la méthode `test1()` (cf. la méthode `main`). Le fichier disponible `prixajustes.txt`, disponible dans le répertoire `resources` du projet `Eclipse` donne la sortie attendue.

3. (2.5 points) Compléter la méthode `test2()`. Pour la tester, vous devez commenter l'appel à `test1()` dans le `main()` et dé-commenter celui à `test2()`. Les deux méthodes de tests doivent renvoyer les mêmes résultats.

Exercice 2 : Hadoop (6 points)

Toutes les classes à modifier se trouvent dans le package `tdd.tpnote.exo2`. La transformation MapReduce demandée travaille sur le fichier `capitals_distances.txt` consultable dans le répertoire `resources` du projet Eclipse. Ce fichier est une suite de lignes de la forme :

```
PAYS1;CAPITALE1;PAYS2;CAPITALE2;DISTANCE
```

(Les noms de pays et de capitales sont en anglais, la distance est un nombre entier positif et représente la distance « à vol d'oiseau » entre les deux capitales). De plus, le fichier est organisé de manière à ce que `CAPITALE1` soit toujours une chaîne plus petite (au sens de la comparaison des chaînes) que `CAPITALE2`. Par exemple, on aura une ligne :

```
Germany;Berlin;France;Paris;875
```

mais pas de ligne

```
France;Paris;Germany;Berlin;875
```

car la chaîne `"Paris"` est plus grande que `"Berlin"`.

Pour la transformation demandée ci-dessous, l'exercice est faisable en laissant inchangés les types de clé et de valeurs du `map` et du `reduce`. Si vous les modifiez, n'oubliez pas de modifier aussi le `main` de la classe `Driver`. On rappelle qu'une façon simple d'utiliser plusieurs valeurs comme clé de sortie ou valeur de sortie d'un `map` est de créer une chaîne de caractères contenant ces deux valeurs séparées par un caractère spécial (par exemple `:`) puis de séparer cette chaîne dans la méthode `reduce`. On rappelle enfin que la méthode statique `Integer.parseInt(s)` permet de transformer une chaîne de caractères en entiers, et que l'expression `i + ""` permet de convertir un entier en chaîne.

Question : Calculer pour chaque ville la ville la plus proche et la ville la plus éloignée.

Compléter les méthodes `map` et `reduce` se trouvant dans la classe `DistMinMax`. Votre `map` prend en entrée les lignes du fichier décrit ci-dessus (une par une) et le `reduce` doit produire une sortie de la forme suivante $(v, v_{min} : d_{min}, v_{max} : d_{max})$ où v est une ville, v_{min} la ville la plus proche de v (et d_{min} la distance associée) et v_{max} la ville la plus éloignée de v (et d_{max} la distance associée).

Vous pouvez tester votre transformation en lançant dans le terminal le script `./exo2.sh` (assurez vous d'avoir sauvegardé le fichier dans Eclipse). La sortie de la transformation doit être similaire (à l'ordre près) à celle donnée dans le fichier `DistMinMax.txt` se trouvant dans le répertoire `resources`.

Exercice 3 : Spark/Python (7 points)

Pour cet exercice, vous pouvez fermer Eclipse et travailler avec VSCode.

Compléter le fichier `mostCentralCities.py`. Celui-ci charge le fichier de distances des capitales (cf. Exercice 2) dans un RDD des lignes du fichier. Le programme doit afficher dans la console les 5 villes les plus proches de leurs villes voisines. Plus précisément, on souhaite afficher les 5 villes pour lesquelles la moyenne des distances aux autres villes est la plus petite.

Indications : on pourra calculer pour chaque ville la moyenne des distances de toutes les autres villes. Une fois ce résultat obtenu, on pourra trier par distance moyenne et enfin prendre les premiers résultats du RDD avec l'action finale `.take(5)`, et afficher la liste des 5 villes obtenues dans la console.

Attention, le calcul doit être symétrique : Paris doit être comptée dans les villes distante de moins de 1000km de Berlin, mais Berlin doit aussi être compté dans les villes distantes de moins de 1000km de Paris, bien que le fichier contienne uniquement la ligne :

```
Germany;Berlin;France;Paris;875
```

et pas la ligne

```
France;Paris;Germany;Berlin;875
```

Vous pouvez tester votre programme simplement avec :

```
python3 mostCentralCities.py
```

Le fichier `res_exo3.txt` donne la sortie demandée pour cet exercice.