université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

20/10/2023

Traîtement Distribué des Données
Master 1 ISD

TP Noté

durée 2h00

Consignes

Télécharger sur le site du cours le projet Eclipse pour ce TP noté et le projet Spark. Les trois exercices sont indépendants. En fin de TP, vous devez rendre sur eCampus :

— Le projet Eclipse sous forme d'une archive `.tar.gz` contenant les réponses aux exercices 1 et 2
— Le fichier `cityLessThan1000.py` se trouvant dans le projet `Spark`

Pour exporter votre projet depuis Eclipse (Exercice 1 et 2) :

— Cliquer avec le bouton droit sur le répertoire du projet
— Choisir `Export`
— Choisir `General` → `Archive File`
— Cocher le répertoire du projet
— Cocher l'option `Save in tar format`
— Cliquer sur `Browse`
— Choisir un emplacement et un nom de fichier **en remplaçant l'extension .zip en .tar.gz**

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

Pour l'exercice 3, il est recommandé de fermer Eclipse et de travailler avec VS Code.

Remarque la consultation de *documents* est autorisée. Les logiciels de messagerie et les IA génératives type ChatGPT, Copilot, . . .ne sont pas des documents et leur utilisation sera considérée comme une tentative de fraude.

Exercice 1 : transactions (8 points, environ 45 minutes)

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 INTEGER, -- prix
                          QTITY INTEGER) -- quantité
```

représentant des billets d'avions (à lire mais ne pas modifier). Ici, `xxx` correspond à votre login Unix. Lors de l'initialisation un message sera affiché dans la console avec le nom de la table. **Vous devez utiliser ce nom pour toutes vos requêtes**.

`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. (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`. 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.
2. (3 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, environ 35 minutes)

Toutes les classes à modifier se trouvent dans le package `tdd.tpnote.exo2`. Au début de l'exercice, vous pouvez sélectionner la classe `StartHadoop` et l'exécuter avec `Eclipse`. Cette dernière démarre Hadoop et upload les fichiers dans le HDFS. Vous n'avez aucune manipulation à faire en console.

La transformation MapReduce demandée travaille sur le fichier `capitals_distances.txt` consultable dans le répertoire `resources` du projet. Ce fichier est une suite de ligne 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 orthodromique ou « 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 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 sélectionnant la classe `RunDriverHistoDistance` et en l'exécutant dans Eclipse (il faut avoir démarré une fois `Hadoop` au préalable, avec la classe `StartHadoop`). 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`.

Il y a 3 points par méthode. Même si votre code ne fonctionne pas, du code partiel ou un commentaire expliquant votre approche peut vous permettre d'avoir des points.

Exercice 3 : Spark/Python (6 points, environ 35 minutes)

Extraire l'archive `tdd_tp_note_2023_spark.tar.gz` vous y trouverez un sous-répertoire `scripts`. On suppose que vous êtes dans un terminal, dans le répertoire `tdd_tp_note_2023_spark`.

— Lancer une première fois Hadoop et Spark
`scripts/start-spark.sh`
Le script prend une ou deux minutes à s'exécuter. Il stoppe Hadoop et Spark si ces derniers étaient démarrés, et les relance. Il dépose aussi les fichiers textes sur le HDFS.

— Lancer VS Code dans le répertoire courant :
`code .`
Vous pouvez ensuite tester votre code en exécutant le script :
`python3 cityLessThan1000.py`

Compléter le fichier `cityLessThan1000.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, pour chaque ville le nombre de villes situées à 1000km ou moins. Le résultat doit être trié par nombre de villes décroissant de ville. 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
```