1. Introduction

1.1. Projets MapReduce

Comme dans le premier TP, vous allez travailler avec Eclipse en éditant les sources sur le poste client et en compilant et exécutant sur le serveur Hadoop.

- Eclipse
  - Créez un dossier appelé `TP3` dans votre compte (pas dans `Documents` car les chemins ne sont pas prévus).
  - Téléchargez le fichier `ModeleMapReduce.tar.gz`.
  - Décompressez-le dans le dossier TP3.
  - Lancez Eclipse. Il va vous demander dans quel workspace travailler : `TP3`.
  - Importez le projet `ModeleMapReduce` dans votre Workspace : menu `File`, `import...`, dans le dialogue choisir l'onglet `General`, `Existing Projects into Workspace...`, en face de `Select root directory`, cliquez sur `Browse...` puis immédiatement sur `Valider`, ça va afficher la liste des projets du Workspace et normalement `ModeleMapReduce` est coché, pour finir cliquez en bas sur `Finish`.
  - Vous allez peut-être devoir ajouter les chemins d'accès aux jars Hadoop (`hadoop-common`, `hadoop-hdfs` et `hadoop-mapreduce-client-core`) si vous n'avez pas fait comme demandé dans le TP1 : il doit y avoir un dossier `~/lib/hadoop` contenant ces fichiers jar dans votre compte. Un tel problème se voit au point d'exclamation rouge sur le projet dans Eclipse. Menu `Properties` du projet, onglet `Java Build Path`, sous-onglet `Libraries`, `Remove` sur les mauvaises et `Add External JARs` sur les correctes.
  - Ensuite, pour créer un nouveau projet, il suffit de copier-coller le modèle puis d'ouvrir un terminal dans le dossier de ce nouveau projet et taper `make rename`. Cela renommer toutes les classes et fichiers en fonction du nom du projet.
- Hadoop
  - Ouvrez une session ssh : `ssh hadoop` et allez dans le dossier `TP3` puis dans le projet voulu. Vous devez voir un sous-dossier `src` et le fichier `Makefile`.
  - Les résultats d'exécution dépendent du projet, alors vous devrez éditer le fichier `Makefile` : mettez les bons paramètres pour `ENTREES` et `SORTIE` ; attention ça doit être des noms valides dans HDFS, par exemple `ENTREES=/share/paris/arbres.csv` et `SORTIE=resultats`.
  - Tapez `make` pour compiler, lancer l'exécution et afficher le résultat.
  - Les résultats sont mis dans un dossier HDFS défini par variable `SORTIE` du `Makefile`. Ils sont nommés `part-r-00000`, selon le nombre de *reducer* finaux. Ce dossier doit être supprimé avant chaque lancement, et c'est prévu dans le Makefile. Vous pouvez réafficher les résultats en tapant `make results`.

La commande `make` permet d'automatiser la séquence approximative suivante :

```
mkdir -p bin
javac -cp ~/lib/hadoop/*.jar src/*.java -d bin
jar cfe projet.jar ProjetDriver -C bin *.class
hdfs dfs -rm -r -f resultats
hadoop jar projet.jar entrees resultats
hdfs dfs -cat resultats/part-r-00000 | more
```

Vous verrez ces commandes à chaque exécution.

Pour tuer une application qui n'en finit pas, CTRL-C ne suffit pas. En fait, l'application continue en arrière-plan. Pour la tuer vraiment, il faut avoir son identifiant, par exemple `application_1455868731302_0020` et taper `yarn application -kill` suivi de son identifiant. On peut obtenir l'identifiant soit avec l'interface Web, soit dans les premières lignes affichées lors du lancement.

Avertissement

Il faudra attendre quelques secondes après une modification sur Eclipse avant de lancer l'exécution dans la fenêtre SSH. C'est à cause du serveur NFS qui tarde à propager les fichiers sur tout le réseau.

D'autre part, parfois Eclipse semble perdre des classes : elles sont visibles dans le projet, mais soulignées en rouge dans les sources. C'est probablement parce que la version de Java diffère entre Eclipse (Java 1.7) et Hadoop (Java 1.8). Eclipse doit probablement s'appuyer sur les fichiers .class du dossier bin, mais ils sont incompatibles. La solution consiste à nettoyer le projet : soit faire `make clean` dans la fenêtre SSH, soit utiliser le menu `Project/Clean...`.

2. Activité du cluster

Cette partie sera à faire après avoir lancé une application *MapReduce*.

Quand votre programme ne fonctionne pas, il faut examiner les informations retournées par Hadoop. Il y a deux types de compte-rendus à analyser : les affichages lors du lancement du job MapReduce et les logs de Yarn.

2.1. Analyse des résultats d'exécution

Juste après avoir tapé `make` pour lancer un projet, Yarn affiche beaucoup de lignes.

- Repérez les suivantes :

```
2019-03-27 10:44:05 [main] input.FileInputFormat: Total input paths to process : 1
2019-03-27 10:44:06 [main] mapreduce.JobSubmitter: number of splits:1
```

Elles indiquent que vos données sont composées d'un seul fichier et que du coup, 2e ligne, il ne lance qu'un seul Mapper.

- La ligne suivante donne l'identifiant de l'application :

```
2019-03-27 ... impl.YarnClientImpl: Submitted application application_1515425519656_0423
```

- Le mode « uber » est une optimisation quand les données sont très petites : il ne lance qu'un seul Mapper, sur une seule machine.

```
2019-03-27 mapreduce.Job: Job job_1515425519656_0423 running in uber mode : true
```

- Ensuite, vous aurez un état d'avancement des tâches map et reduce :

```
2019-03-27 10:44:11,359 INFO  [main] mapreduce.Job:  map 100% reduce 100%
```

- À la fin du job, vous allez voir ces informations très importantes, le nombre de paires (clé, valeur) en entrée et en sortie du map :

```
Map-Reduce Framework
    Map input records=30067
    Map output records=28761
```

Par exemple, ici, il y a eu 30067 lignes traitées et seulement 28761 ont généré une paire à destination du reducer. Les autres lignes ont donc été rejetées, soit par un test, soit par une exception. Quand `Map output records` vaut zéro, c'est sûrement que votre mapper ne parvient pas à travailler du tout ; vous aurez à analyser les logs sur Yarn, voir plus bas.

- Il y a ensuite le nombre de paires (clé, valeur) traitées par le combiner, ici aucune car pas de combiner :

```
Combine input records=0
Combine output records=0
```

- Pour finir, il y a le nombre de clés différentes ainsi que les paires traitées par le reducer :

```
Reduce input groups=2
Reduce input records=28761
Reduce output records=2
```

Ici, il y a deux clés distinctes parmi les (clé, valeur) produites par les mappers, 28761 paires en tout en entrée et seulement 2 en sortie (après le reduce).

2.2. Interface web

Lorsque votre job a planté, vous devez chercher la cause, probablement une exception : `NullPointerException`, `ClassCastException`, `NumberFormatException`...

Ouvrez IceWeasel, puis configurez le proxy comme dans le TP1 : préférences, avancé, réseau, paramètres, configuration manuelle du proxy 192.168.100.99, port 80, DNS distant, utiliser pour tous les protocoles. Cette reconfiguration permet d'accéder aux machines internes du cluster. Pour remettre le proxy comme avant, il suffira de cocher `pas de proxy`.

Ensuite, ouvrez le lien http://master, c'est la page d'accueil du cluster.

1. Ouvrez le premier lien de la page d'accueil `YARN applications MapReduce`. Cette page affiche toutes les applications MapReduce lancées sur YARN.
2. Repérez votre application dans la liste (il y a votre login dans la colonne User). Cliquez sur son lien. La nouvelle page affiche les informations de votre application. En bas, vous verrez les tentatives (attempts) et sur quel(s) esclave(s) elle a tourné, mais vous ne pourrez cliquer dessus que si Iceweasel est en mode « cluster », voir le TP1.

3. Vous pouvez cliquer sur le lien `Logs` tout en bas à droite, attendez deux petites secondes que les logs apparaîssent à la place du message d'erreur, car il faut les télécharger des esclaves vers master.
   - *stdout* contient tous les messages écrits par vos instructions `System.out.println`,
   - *stderr* contient tous les messages écrits par `System.err.println`, pensez à décommenter les instructions `e.printStackTrace()` dans les map et reduce.
   - *syslog* contient les messages émis par Hadoop, c'est là que vous trouverez les exceptions que vous n'avez pas intercepté. Cliquez sur `Click here for the full log` sauf s.
4. Si vous voulez plus d'informations, cliquez sur le lien `History` à côté de `Tracking URL`. La nouvelle page détaille les tâches *Map* et *Reduce*. Au début, vous n'en aurez qu'un de chaque, mais les derniers projets sont nettement plus gros.
5. Cliquez sur les liens `Map` tout en bas, puis au retour sur les `Reduce`, puis sur le lien de chaque tâche qui vous intéresse. Il y a des liens `Logs` pour avoir les messages de chaque tâche.

NB: quand vous revenez en arrière à la page listant les jobs, pensez à rafraîchir `ctrl-R` car ce n'est pas automatique.

2.3. En ligne de commande

La même chose en ligne de commande :

- Repérez l'identifiant de votre application quand vous la lancez, c'est un code comme `application_1455868731302_0020`
- `yarn logs -applicationId application_1455868731302_0020 | less` affiche tous les logs de cette application : *stdout*, *stderr* et *syslog* pour chacune des tâches.

Vous voyez que tout cet ensemble est extrêmement complexe. Habituez-vous à cette quantité, à ne chercher que ce qui vous concerne et à repérer les exceptions inattendues.

Pour arrêter un job de force, avant qu'il soit fini :

```
yarn application -kill application_1455868731302_0020
```

3. Déroulement du TP

Important : ce TP va durer 3 semaines et il y a toutes sortes d'exercices, faciles et difficiles. Alors commencez par tous les exercices faciles, vous ferez les difficiles la prochaine fois. Si vous commencez par des difficiles et que vous n'avez aucune expérience, vous allez avoir des problèmes. Vous faites comme vous voulez, c'est vous qui voyez...

Pour chacun de ces projets, si vous n'arrivez pas à faire tout ce qui est demandé, essayez de vous en rapprocher. La notation tient compte de vos tentatives.

À la fin de chaque séance, vous devrez déposer l'état actuel sur Moodle. Ne rien rendre vous expose à un zéro pour la séance. Allez voir § 8, page 12 pour savoir comment rendre le travail.

4. Arbres remarquables de Paris

Vous allez écrire quelques programmes *MapReduce* sur le fichier des arbres remarquables de Paris. Ce fichier a été placé dans `/share/paris/arbres.csv` sur HDFS.

Repartez de la classe `Arbre.java` que vous aviez programmée dans le TP1. Sinon, téléchargez `Arbre.java`. Cette classe regroupe toutes les fonctions de découpage et de conversion des lignes CSV en champs. Vous devrez ajouter les méthodes nécessaires pour les traitements demandés.

D'autre part, dans chaque *mapper* sur ce fichier, il faudra ignorer la première ligne. Sa clé `cleE` vaut 0 et pour ignorer cette ligne, on écrit `if (cleE.get() == 0L) return;`.

4.1. Arrondissements contenant des arbres (très facile)

Dupliquez le modèle de projet sous le nom `NombreArrondissements`, puis tapez `make rename` dans son dossier, puis `F5` une ou deux fois dans Eclipse.

Écrire un programme *MapReduce* qui affiche la liste des arrondissements distincts contenant des arbres dans ce fichier. Forcément, c'est vingt ou moins arrondissements différents, mais combien précisément ?

Il vous suffit de placer l'arrondissement en tant que clé et de mettre une valeur quelconque ou `NullWritable` en tant que valeur, en sortie du *mapper*. Le *reducer* devra se contenter de sortir les clés et valeurs qu'il reçoit, il n'a même pas de boucle à faire sur les valeurs puisqu'il les ignore.

4.2. Nombre d'arbres par genre (facile)

Dupliquez le modèle de projet sous le nom `NombreGenreArbres`, puis tapez `make rename` dans son dossier, puis `F5` une ou deux fois dans Eclipse.

Écrire un programme *MapReduce* qui calcule le nombre d'arbres de chaque genre. Par exemple, il y a 3 *Acer*, 19 *Platanus*, etc.

Comment allez-vous définir les clés et les valeurs transmises entre le *mapper* et le *reducer* ? Le *mapper* doit extraire le genre d'arbre. Le *reducer* récupère les paires (cléI, valeurI) ayant la même clé, donc il faut que cette clé soit le genre d'arbre ; la valeur étant le nombre d'arbre de ce genre.

Comme précédemment, il faut ignorer la première ligne, les titres des colonnes.

4.3. Hauteur maximale par genre d'arbre (moyen)

Dupliquez le modèle de projet sous le nom `MaxHauteurArbres`, puis tapez `make rename` dans son dossier.

Écrire un programme *MapReduce* qui calcule la hauteur du plus grand arbre de chaque genre. Par exemple, le plus grand *Acer* fait 16m, le plus grand *Platanus* fait 45m, etc.

4.4. Arrondissement contenant le plus vieil arbre (difficile)

Dupliquez le modèle de projet sous le nom `ArrondissementVieilArbre`, puis tapez `make rename` dans son dossier.

Écrire un programme *MapReduce* qui affiche l'arrondissement où se trouve le plus vieil arbre.

Le *mapper* doit extraire l'âge et l'arrondissement de chaque arbre. Le problème, c'est que ces informations ne peuvent pas être utilisées en tant que clés et valeurs (pourquoi ?). Vous devrez définir une sous-classe de `Writable` pour contenir les deux informations, voir le cours n°2 et aussi

la première étude de cas du cours n°3. Le *reducer* doit regrouper tous ces données et ne sortir que l'arrondissement.

4.5. La taille dépend-elle de l'âge ? (difficile)

Dupliquez le modèle de projet sous le nom `CorrelationArbres`, puis tapez `make rename` dans son dossier.

On voudrait savoir si la taille des arbres d'un même genre est corrélée avec leur âge. Pour cela, on va calculer un coefficient de corrélation entre la taille et l'âge. Voici les calculs à faire, en posant x=âge et y=taille. C'est un peu long mais pas compliqué. La notation x_i signifie l'âge de l'arbre n°i avec i variant parmi les arbres concernés.

1. Calculer les sommes
   a. Calculer le nombre d'arbres (ne pas continuer si $n < 2$) :
      $n = \sum_i 1$
   b. Calculer la somme des âges :
      $S_x = \sum_i x_i$
   c. Calculer la somme des carrés des âges :
      $S_{x2} = \sum_i x_i^2$
   d. Calculer la somme des tailles :
      $S_y = \sum_i y_i$
   e. Calculer la somme des carrés des tailles :
      $S_{y2} = \sum_i y_i^2$
   f. Calculer la somme des produits des âges par les tailles :
      $S_{xy} = \sum_i x_i * y_i$
2. Calculer les moyennes
   a. Calculer la moyenne des âges :
      $M_x = \frac{S_x}{n}$
   b. Calculer la moyenne des carrés des âges :
      $M_{x2} = \frac{S_{x2}}{n}$
   c. Calculer la moyenne des tailles :
      $M_y = \frac{S_y}{n}$
   d. Calculer la moyenne des carrés des tailles :
      $M_{y2} = \frac{S_{y2}}{n}$
   e. Calculer la moyenne des produits taille*âge :
      $M_{xy} = \frac{S_{xy}}{n}$
3. Calculer les variances et le coefficient de régression
   a. Calculer la variance des âges :
      $V_x = M_{x2} - M_x * M_x$
   b. Calculer la variance des tailles :
      $V_y = M_{y2} - M_y * M_y$
   c. Calculer la covariance des âges et tailles :
      $V_{xy} = M_{xy} - M_x * M_y$
   d. Calculer le coefficient de corrélation r :
      $denom = \sqrt{V_x * V_y}$

si $|V_{xy}| \approx 0$ et $|denom| \approx 0$ alors $r = 1$ sinon $r = \frac{V_{xy}}{denom}$

e. Calculer la pente de la droite (pas utile ici) :
$pente = \frac{V_{xy}}{V_x}$

f. Calculer l'ordonnée à l'origine (pas utile ici) :
$ordo0 = M_y - pente * M_x$

Si le coefficient est proche de 1, c'est qu'il y a corrélation, la taille dépend de l'âge. Si le coefficient est faible, alors la taille est indépendante de l'âge. Si le coefficient est proche de -1, alors la taille varie inversement de l'âge.

Pour faire ce calcul sur YARN, vous allez programmer au moins quatre classes :

- La classe `Regression` qui implémente l'interface `Writable`, voir la classe `Variance` du cours n°3. C'est elle qui est échangée entre les *mappers* et les *reducers*. Elle gère six variables `double`. Ce sont les sommes partielles des informations du premier point de l'algorithme : n, S_x, S_{x2}, S_y, S_{y2} et S_{xy}. Cette classe rajoute trois méthodes :
  - `void set(double x, double y)` initialise les variables de l'algorithme : $n = 1$, $S_x = x$, etc. On l'appellera du *mapper* en fournissant x=âge et y=taille.
  - `void add(Regression autre)` ajoute les n, S_x... de `autre` à ceux de `this`. On l'appellera dans le *reducer* et dans le *combiner*.
  - `double getCoeffCorrelation()` calcule le coefficient de corrélation en déroulant les points 2. et 3. de l'algorithme. On l'appellera dans le *reducer*.
- Le *Mapper* doit produire une instance de `Regression` par genre d'arbre. Choisissez bien la clé afin qu'ils soient correctement regroupés par le *reducer*. Le fichier ne contient pas l'âge des arbres, mais leur année de plantation. Il ne faut rien produire si l'une des informations manque (exception).
- Le *Combiner* avance les calculs des sommes partielles mais ne calcule ni les moyennes ni le coefficient de régression. Il faut se rappeler que son lancement est optionnel.
- Le *Reducer* va recevoir toutes ces instances de `Regression` et finir les calculs : terminer les sommes (comme le *combiner*), puis calculer les moyennes, les variances et le coefficient de corrélation.

Lisez bien toute la première partie du cours n°3.

Pour la mise au point, il est recommandé d'ajouter des instructions `System.err.println(...)` pour afficher des informations cruciales comme les données lues, afin de savoir ce qui se passe quand les calculs sont faux. Les messages sont mis dans les logs.

Faire tourner le programme puis analyser le résultat.

Y aurait-il une meilleure corrélation entre l'âge et la circonférence, ou entre la hauteur et la circonférence ? Si vous avez bien programmé, ça sera très rapide à essayer.

4.6. Arrondissement contenant le plus d'arbres (très difficile)

Dupliquez le modèle de projet sous le nom `ArrondissementNbMaxArbres`, puis tapez `make rename` dans son dossier.

Écrire un programme *MapReduce* qui affiche l'arrondissement qui contient le plus d'arbres.

Le problème est que le programme va certainement afficher une liste de couples (n° d'arrondissement, nombre d'arbres) non classée par nombre. Si on appliquant ce programme à de vraies données

massives non limitées aux arrondissements de Paris, on récupérerait une liste énorme, inutilisable et le classement pourrait prendre des heures. Comment faire pour garder seulement la meilleure réponse ?

La solution passe probablement par une seconde phase *MapReduce*, dans laquelle le *Mapper* récupère les paires (arrondissement, nombre) issues de la première phase, et en fait des paires dont la clé est sans importance (`NullWritable`) et les valeurs sont elles-mêmes les paires d'entrée. Le *Reducer* les reçoit toutes et doit garder la meilleure paire. Voir la seconde étude de cas, CM3 pour comprendre comment enchaîner deux jobs dans un même programme.

Faites un bilan pour vous-même : comment auriez-vous programmé ce traitement en Java classique ? Quelle approche préférez-vous ?

5. Stations météo

Dans cette partie, vous allez traiter le fichier `/share/noaa/isd-history.txt` sur HDFS qui décrit les stations météo. Voir le TP1 pour la description des champs. Attention, il y a un autre fichier, avec presque le même nom, sauf `.csv`, qui n'a pas du tout la même structure.

5.1. Répartition des stations (facile)

Dupliquez le modèle de projet sous le nom `RepartitionStations`, puis tapez `make rename` dans son dossier.

Écrire un programme *MapReduce* qui calcule le nombre de stations dans l'hémisphère nord ($latitude \geq 0$) et dans l'hémisphère sud ($latitude < 0$). En principe, il doit afficher approximativement `Sud 4014, Nord 24090`, selon les conditions que vous avez mises. À vous de deviner quels sont les types de données à échanger entre le *mapper* et le *reducer*, et les calculs à faire.

Commencez par reprendre la classe `Station` du projet `NomPaysAltStations` du tp2, ou téléchargez `Station.java`. Cette classe contient les accesseurs facilitant la lecture du fichier `isd-history.txt`. Vous allez devoir ajouter celui qui permet d'obtenir la latitude.

Dans le *mapper*, il faut ignorer les 22 premières lignes du fichier car ce ne sont pas des données. Malheureusement dans le *mapper*, on ne reçoit pas le numéro de la ligne mais son offset (nombre d'octets depuis le début) dans `cleE`. La ligne 22 commence à l'octet 1001. Donc tant que `cleE` est inférieure à 1001, il faut ignorer la ligne.

Pour connaître cet offset, il suffit de rajouter `System.out.println(cleE+": "+valeurE);` au début du *mapper* puis d'aller voir dans les logs.

N'oubliez pas d'intercepter les exceptions et de ne rien faire quand il y en a une.

6. Relevés météo

Maintenant, on va travailler avec les relevés météo téléchargés durant le TP1. Les données sont dans `/share/noaa/data`. Il y a un dossier par station et dedans il y a un fichier par année. Il faut que vous alliez voir quels fichiers il y a :

```
hdfs dfs -ls /share/noaa/data
```

C'est un dossier qui contient plein de sous-dossiers, un par station. Repérez le numéro de l'un de ceux que vous avez téléchargé durant le TP1 et faites :

```
hdfs dfs -ls /share/noaa/data/NUMERO
```

Vous allez voir tous les fichiers de mesures, un par année. Il y en a un peu plus de 70 et chacun contient en général une ligne de mesures par heure.

Pour voir un extrait de l'un de ces fichiers :

```
hdfs dfs -cat /share/noaa/data/NUMERO/NUMERO-99999-2019 | head
```

Voici une partie de leur structure :

offset	taille	exemple	signification
4	6	`332130`	USAF weather station identifier
10	5	`99999`	WBAN weather station identifier
15	8	`19500101`	observation date YYYYMMDD
23	4	`0300`	observation time HHMM
28	6	`+51317`	latitude (degrees x 1000)
34	7	`+028783`	longitude (degrees x 1000)
46	5	`+0171`	elevation (meters)
60	3	`320`	wind direction (degrees)
63	1	`1`	wind direction quality code
65	4	`0072`	wind speed (meters per second x 10)
69	1	`1`	wind speed quality code
70	5	`00450`	sky ceiling height (meters)
75	1	`1`	sky ceiling height quality code
78	6	`010000`	visibility distance (meters)
84	1	`1`	visibility distance quality code
87	1	`-`	air temperature sign
88	4	`0128`	air temperature (degrees Celsius x 10)
92	1	`1`	air temperature quality code
93	1	`-`	dew point temperature sign
94	4	`0139`	dew point temperature (degrees Celsius x 10)
98	1	`1`	dew point temperature quality code
99	5	`10268`	atmospheric pressure (hectopascals x 10)
104	1	`1`	atmospheric pressure quality code

Remarques :

- Certains champs sont multipliés par 10, ça veut dire par exemple que lorsqu'on lit `0128`, il faudra comprendre que c'est 12,8.
- Lorsque la mesure est composée uniquement de 9, c'est qu'elle est invalide.
- Il ne faut tenir compte que des mesures dont le `quality code` est l'un des caractères de `"01459ACIMPRU"` ; les autres signes indiquent des erreurs de mesure.
- Il faut gérer le signe à part lors de la conversion en int (`Integer.parseInt`) car elle émet une exception s'il y a un `+` devant le nombre.

Voici par exemple comment on extrait l'identifiant WBAN de la station qui a fait la mesure :

```
public static String getWBAN()
{
    return ligne.substring(10, 10+5);
}
```

Et voici comment on programme l'extraction de la température de condensation (dew point) :

```
public static double getDewPoint() throws Exception
{
    // qualité
    char quality = ligne.charAt(98);
    if ("01459ACIMPRU".indexOf(quality) < 0) throw new Exception();
    // valeur absolue
    String temp = ligne.substring(94, 94+4);
    if ("9999".equals(temp)) throw new Exception();
    // signe et conversion en nombre
    char signe = ligne.charAt(93);
    return Integer.parseInt(temp) * (signe=='-'?-0.1:0.1);
}
```

Attention, dans la suite, ce sont les identifiants USAF et les températures d'air qui nous intéressent, donc vous aurez à recopier et modifier ces deux fonctions.

6.1. Nombre de mesures par station (facile)

Dupliquez le modèle de projet sous le nom `NombreMesuresStations`, puis tapez `make rename` dans son dossier. Éditez le `Makefile` et mettez `ENTREES=/share/noaa/data/NUMERO` avec le n° de la station que vous aviez téléchargée.

Créez la classe `Mesure.java` comme les précédents projets et programmez les accesseurs nécessaires dans la suite, par exemple `getTemperature()`.

On voudrait savoir combien il y a de mesures de température en tout par station météo parmi les données de `/share/noaa/data`. Au lieu de ne traiter qu'un seul fichier, on va traiter une multitude dans des sous-dossiers et d'autre part, ils sont compressés. Cependant c'est totalement transparent. Lorsqu'on indique un dossier de dossiers contenant des fichiers comme `/share/noaa/data`, automatiquement tous les fichiers vont être traités et décompressés si besoin. Il faut seulement rajouter la ligne suivante dans le *driver*.

```
job.setInputDirRecursive(true);
```

Vous pouvez rajouter facilement un *combiner*, car c'est directement le *reducer*. Est-ce qu'il permet de gagner du temps ?

Ça risque de bouchonner pas mal à l'exécution car les machines ne sont pas très rapides, vous en avez pour environ 20 minutes. Au moins vous verrez l'avancement des tâches *map* et *reduce*... Faites autre chose en attendant.

6.2. Moyenne mobile (difficile)

Dupliquez le projet précédent sous le nom `TemperatureMoyenneAnnees`, puis tapez `make rename` dans son dossier.

On voudrait savoir si les températures sont vraiment en train d'augmenter lentement de manière globale. Pour cela, on se propose de calculer la température moyenne sur une fenêtre de quelques années et voir s'il y a une évolution.

Une moyenne mobile est une succession de moyennes calculées sur une fenêtre de valeurs. Par exemple, soient les valeurs [t1,t2,t3,t4,t5...]. Si on prend une fenêtre de 3 valeurs, alors les moyennes mobiles successives seront (t1+t2+t3)/3, (t2+t3+t4)/3, (t3+t4+t5)/3, etc. Une moyenne mobile permet de lisser les valeurs.

Soit une succession de mesures (année, température). On veut calculer la moyenne des températures avec une fenêtre de $\pm N$ années (ça fait $2N+1$ années par fenêtre) par un procédé *MapReduce*. Vous savez déjà que pour rendre le calcul associatif, il faut gérer des couples (somme, nombre) et non pas le rapport entre les deux. Mais comment faire pour ne repasser qu'une seule fois sur les données et calculer toutes les moyennes de fenêtres en même temps ?

C'est simple : à partir de chaque mesure (année, température), le *mapper* va générer plusieurs paires (A, (température,1)) pour A allant de $année - N$ à $année + N$. Mettre N à 1 ou 2, pas plus pour les premiers essais. Il faudra prévoir un *combiner* pour réduire localement le nombre de paires à gérer. Ainsi, les mesures venant de plusieurs fenêtres seront réduites ensemble avant d'être envoyées au *reducer*.

Le résultat est une liste de couples (année, moyenne de la température sur les $\pm N$ années autour). Il faudra ignorer les N premières et N dernières années car elles ne correspondent pas à de vraies données, mais seulement à la largeur de la fenêtre.

Vous pourrez lancer le programme sur la totalité des stations en configurant `ENTREES=/share/noaa/data` dans le `Makefile` mais 1) vous risquez de finir le TP après-demain matin, même si vous mettez une petite valeur de N et 2) en fait, vous allez mélanger des données venant de stations différentes et parfois ne couvrant pas les mêmes années. Imaginez que vous ayez des données entre 1930 et 1950 de Fairbanks et entre 1940 et 1980 d'Addis-Abeba, comment voulez-vous calculer une température moyenne qui ait du sens ?

Alors ce qui est proposé, c'est de générer des moyennes par année et par station. La clé de sortie du *reducer* sera l'identifiant USAF de la station, et il affichera l'année et la température. Quant à la sortie du *mapper*, à vous de réfléchir afin qu'il y ait à la fois l'année et l'identifiant de la station. Il n'est pas nécessaire que l'année soit gérée sous forme d'un entier.

Si vous voulez dessiner les données, alors essayez get_plot_temp.sh. Il faut configurer ses deux premières lignes, la valeur N et le dossier HDFS contenant les résultats.

Vous ne constatez pas vraiment la hausse des températures moyennes dont tout le monde parle ? Mais en fait, quelles sont les données que vous avez traitées ? Est-ce qu'elles sont régulièrement espacées dans le temps et l'espace, n'y-a-t-il pas des données manquantes, irrégulières, biaisées, mal réparties sur le globe qui faussent totalement les calculs ? L'analyse des données est bien plus cruciale que leur collecte. Il faut impérativement avoir des données signifiantes. Voici par exemple la position des stations dont vous avez téléchargé les données dans le TP1.

Voir la figure 1, page 13.

7. Parcmètres de Paris (optionnel)

Maintenant un mini-projet sans aucune indication. Le dossier `/share/paris/horodateurs-transactions` contient plusieurs fichiers, ce sont les listes de toutes les transactions des parcmètres de Paris en 2014. Ils proviennent de cette page.

Dans ces fichiers, les champs sont séparés par un `;` et contiennent ceci :

champ	exemple	signification
0	57320104	numéro (identifiant) du parcmètre
1	2014-06-25T16:01:17+02:00	date de la transaction
2	Résident	type d'usager
3	Paris Carte	moyen de paiement
4	0.65	montant en €
5	10.0	nombre d'heures payées (fractionnaire)
6	2014-06-25T16:01:17+02:00	début du stationnement
7	2014-06-26T16:01:17+02:00	fin du stationnement

Votre travail consiste à écrire différents programmes (projets) qui calculent :

- Le montant total par année et par mois des paiements,
- Le prix moyen par heure de stationnement (montant total en € / nombre total d'heures payées),
- La proportion de paiements avec « Paris Carte » par rapport aux paiements « CB ».

8. Travail à rendre chaque semaine

Rappel : créez un fichier appelé `IMPORTANT.txt` pour signaler toute anomalie en séance comme des problèmes de connexion ou des plantages indépendants de votre volonté. Indiquez aussi si vous avez été absent(e) avec justificatif à l'une des séances. Ça permettra au correcteur de compenser votre note.

Ouvrez un shell dans le dossier `TP3` (le workspace du TP). Tapez ces commandes shell : 

```
for d in */Makefile ; do pushd $(dirname "$d") ; make clean ; popd ; done
tar cfz tp3.tar.gz --ignore-failed-read *.txt $(for d in */Makefile ; do dirname "$d" ; d
tar tfvz tp3.tar.gz
```

Il y aura un avertisssement si vous n'avez pas de fichier `IMPORTANT.txt`, c'est normal donc ignorez-le. Vérifiez que l'archive contient bien les projets du TP, ainsi que le fichier `IMPORTANT.txt`.

Déposez le fichier `tp3.tar.gz` sur la page Moodle dédiée (il y en a une par semaine, pour enregistrer l'avancement car chaque séance est notée, la dernière contiendra la totalité de vos travaux).

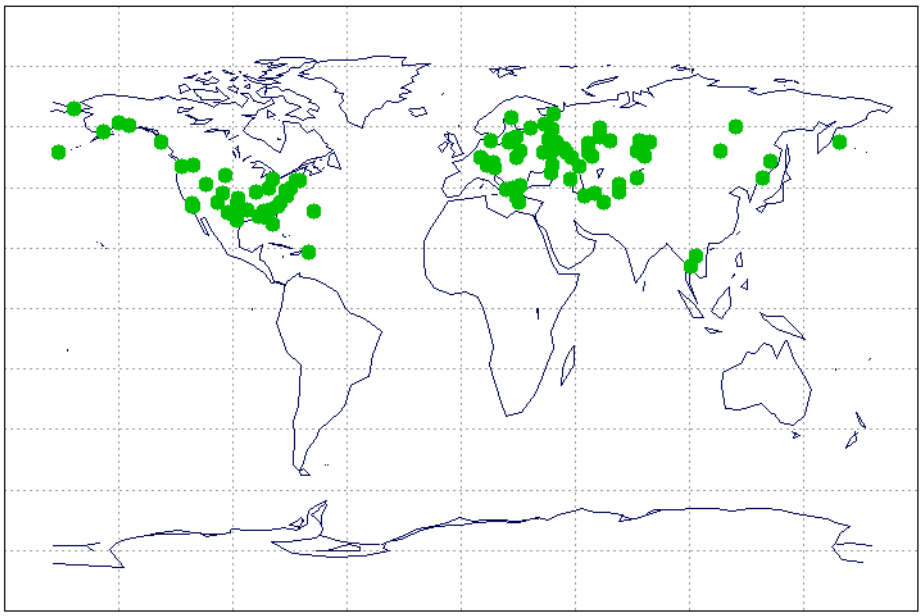

Figure 1: Répartition des stations