TP 1: Environnement de développement pour hadoop et structure d’un projet MapReduce

📌 Commandes Hadoop HDFS

1/ Démarrage des services Hadoop :

sudo service hadoop-hdfs-namenode start # Démarre le service NameNode
sudo service hadoop-hdfs-datanode start # Démarre le service DataNode

2/ Commandes de base du HDFS :

👉 Hadoop propose deux types de commandes pour interagir avec le système de fichiers :

- `hadoop fs`
- `hdfs dfs`

Les deux sont équivalentes.

Commande	Description
`hadoop fs -ls /` `hdfs dfs -ls /`	Affiche le contenu du répertoire racine de HDFS.
`hadoop fs -put /chemin/local/file.txt /chemin/hdfs/`	Upload d'un fichier local vers le HDFS.
`hadoop fs -get /chemin/hdfs/file.txt /chemin/local/`	Télécharge un fichier de HDFS vers le disque local.
`hadoop fs -tail /chemin/hdfs/file.txt`	Affiche les dernières lignes d'un fichier.
`hadoop fs -cat /chemin/hdfs/file.txt`	Affiche tout le contenu d'un fichier.
`hadoop fs -mv /chemin/hdfs/file.txt /chemin/hdfs/newfile.txt`	Renomme un fichier dans HDFS.
`hadoop fs -rm /chemin/hdfs/file.txt`	Supprime un fichier de HDFS.
`hadoop fs -mkdir /chemin/hdfs/myinput`	Crée un répertoire dans HDFS.
`hadoop fs -cat /chemin/hdfs/file.txt | less`	Affiche le contenu d'un fichier page par page.

📌 Commande pour exécuter une application Hadoop MapReduce

Pour exécuter votre application Hadoop MapReduce, utilisez la commande suivante :
hadoop jar <chemin/fichier.jar> <chemin/d'entrée> <répertoire/de/sortie>

Explication :

- `<chemin/fichier.jar>` : chemin vers votre fichier `.jar` contenant les classes compilées (Mapper, Reducer, Driver).
- `<chemin/d'entrée>` : chemin HDFS où se trouvent les fichiers d'entrée.
- `<répertoire/de/sortie>` : chemin HDFS où seront stockés les résultats. (⚠️ Il **ne doit pas déjà exister**.)

Exemple :
hadoop jar /home/user/WordCount.jar /input /output
Cela va :

- Utiliser le fichier `/home/user/WordCount.jar`
- Prendre les données d'entrée dans `/input`
- Écrire les résultats dans `/output`


```
import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class Explorer {

        public static void main(String[] args) throws IOException, URISyntaxException {
        // Définir l'URI du fichier dans HDFS
       // Tu définis l'adresse d'un fichier situé dans HDFS (Hadoop Distributed File System).
        URI uri = new URI("hdfs://quickstart.cloudera:8020/Dtp/test.txt");

        // Créer une configuration Hadoop
       //Tu crées une configuration Hadoop pour connecter ton programme au cluster.
        Configuration conf = new Configuration();

        // Obtenir le système de fichiers basé sur l'URI et la configuration
         //Tu récupères une instance de FileSystem, en te connectant au système de
        //fichiers HDFS avec l'uri et la conf.
```

```
FileSystem fs = FileSystem.get(uri, conf);

// Afficher les métadonnées du fichier
 //Tu récupères les informations du fichier (taille, permissions, date de modification,
//type...) et tu les affiches dans la console.
System.out.println(fs.getFileStatus(new Path(uri)));}}
```

Résumé rapide

Élément	Rôle
`URI`	Indique l'adresse du fichier dans HDFS.
`Configuration`	Paramètre la connexion Hadoop.
`FileSystem`	Interface pour interagir avec HDFS.
`getFileStatus`	Récupère les informations sur le fichier spécifié.

Syntaxe générale du Mapper :

```
public  class MonMapper
        extends Mapper<CléEntree, ValeurEntree, CléSortie, ValeurSortie> {

        public void map(CléEntree key, ValeurEntree value, Context context)
        throws IOException, InterruptedException {

        // Traitement
        // context.write(clé_sortie, valeur_sortie);
        }
}
```


🔵 **À savoir** :

- `Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>`
- `map(KEYIN key, VALUEIN value, Context context)`
- On lit une (clé, valeur) → on émet (clé, valeur) pour Reducer

Syntaxe générale du Reducer :

```
public  class MonReducer

        extends Reducer<CléEntrée, ListeDeValeursEntrée, CléSortie, ValeurSortie> {

        public void reduce(CléEntrée key, Iterable<ValeurEntrée> values, Context context)

        throws IOException, InterruptedException {
```

```
        // Parcourir les valeurs

        // Émettre une (clé_sortie, valeur_sortie)

        // context.write(clé_sortie, valeur_sortie);

        }
}
```

🔵 **À savoir** :

- `Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>`
- `reduce(KEYIN key, Iterable<VALUEIN> values, Context context)`

Résumé très rapide pour révision :

	Mapper	**Reducer**
Hérite de	`Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>`	`Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>`
Fonction principale	`map(KEYIN key, VALUEIN value, Context context)`	`reduce(KEYIN key, Iterable<VALUEIN> values, Context context)`
Objectif	Transformer (clé, valeur) en (clé, valeur) pour le Reducer	Combiner toutes les valeurs d'une même clé

WordCount exo :
***/WordCountMapper**
// Importation des classes nécessaires pour la gestion d'exception et la manipulation de texte
import java.io.IOException;

```
import java.util.StringTokenizer;

// Importation des types Hadoop pour représenter les clés/valeurs
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

// Déclaration de la classe WordCountMapper
// Elle hérite de MapReduceBase et implémente l'interface Mapper
// Les types sont : clé d'entrée = LongWritable, valeur d'entrée = Text, clé de sortie = Text, valeur de sortie = IntWritable
public class WordCountMapper extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {

	// Constante qui représente la valeur 1 (on va émettre (mot, 1) pour chaque mot rencontré)
	private final static IntWritable one = new IntWritable(1);

	// Variable pour stocker temporairement chaque mot lu
	private Text word = new Text();

	// La méthode "map" est appelée pour traiter chaque ligne du fichier d'entrée
	public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> collector, Reporter reporter) throws IOException {

	// Convertir la ligne Hadoop (de type Text) en chaîne Java classique (String)
	String line = value.toString();

	// Utilisation d'un StringTokenizer pour découper la ligne en mots
	// Ici, on utilise " " (espace) comme séparateur correct
	StringTokenizer st = new StringTokenizer(line, " ");

	// Tant qu'on trouve encore des mots dans la ligne
	while (st.hasMoreTokens()) {
	// Prendre le prochain mot et le stocker dans "word"
	word.set(st.nextToken());

	// Émettre le couple (mot, 1) vers le Reducer
	collector.collect(word, one);}}}
```

***/WordCountReducer**

```
// Importation des classes nécessaires
import java.io.IOException;
import java.util.Iterator;
```

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;

// Déclaration de la classe WordCountReducer
// Elle hérite de MapReduceBase et implémente l'interface Reducer
// Types : clé d'entrée = Text, valeur d'entrée = IntWritable, clé de sortie = Text, valeur de sortie = IntWritable
public class WordCountReducer extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {

	// La méthode "reduce" est appelée pour traiter les listes de valeurs associées à une même clé (un mot ici)
	public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> outputCollector, Reporter reporter) throws IOException {

	// Variable pour stocker la somme des valeurs
	int sum = 0;

	// Parcours de toutes les valeurs associées à ce mot
	while (values.hasNext()) {
	// Ajout de la valeur actuelle à la somme
	sum += values.next().get();
	}

	// Émettre (mot, somme) : mot + nombre total d'occurrences
	outputCollector.collect(key, new IntWritable(sum));}
```


***/WordCount (Main)**

```
// Importations nécessaires pour Hadoop, HDFS et types de données
import java.net.URI;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.RunningJob;
import org.apache.hadoop.mapred.TextOutputFormat;

// Déclaration de la classe principale qui va configurer et lancer le Job Hadoop
public class WordCount {
```

```
        public static void main(String[] args) throws Exception {

        // Création de la configuration Hadoop
        Configuration conf = new Configuration();

        // Définir le chemin d'entrée (fichier texte) sur HDFS
        Path inputPath = new Path("hdfs://quickstart.cloudera:8020/tp1/test.txt");

        // ⚠️ Erreur corrigée ici : outputPath doit être différent de inputPath pour éviter de
l'écraser
        Path outputPath = new Path("hdfs://quickstart.cloudera:8020/tp1/output"); // corrigé

        // Création de la configuration du job MapReduce
        JobConf job = new JobConf(conf, WordCount.class);

        // Définir la classe principale contenant le main
        job.setJarByClass(WordCount.class);

        // Donner un nom au job (facilite la surveillance)
        job.setJobName("WordCounterJob");

        // Définir les chemins d'entrée et de sortie
        FileInputFormat.setInputPaths(job, inputPath);
        FileOutputFormat.setOutputPath(job, outputPath);

        // Définir les types de clé et de valeur en sortie
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        // Définir le format de sortie (texte simple ici)
        job.setOutputFormat(TextOutputFormat.class);

        // Définir le Mapper et le Reducer que l'on veut utiliser
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        // Connexion à HDFS
        FileSystem hdfs = FileSystem.get(URI.create("hdfs://localhost:9000"), conf);

        // Si le dossier de sortie existe déjà, le supprimer (obligatoire sinon Hadoop va
échouer)
        if (hdfs.exists(outputPath)) {
        hdfs.delete(outputPath, true);
        }

        // Exécuter le job et attendre la fin
        RunningJob runningJob = JobClient.runJob(job);
```

```
        // Afficher si le job s'est bien terminé
        System.out.print("Job is successful: " + runningJob.isSuccessful());
        }
}
```

TP 2:
Exo1:
***/client_map**

```
import java.io.IOException;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

// Définition de la classe client_map qui hérite de Mapper
public class client_map extends Mapper<LongWritable, Text, Text, FloatWritable> {

        // Fonction map exécutée pour chaque ligne du fichier d'entrée
        public void map(LongWritable ligne, Text ligneText, Context context) throws
IOException, InterruptedException {
        // Convertir la ligne lue depuis le fichier en String
        String line = ligneText.toString();

        // Récupérer l'identifiant du client (1ère colonne séparée par des virgules)
        String clientID = line.split(",")[0];

        // Récupérer le montant payé par le client (5ème colonne, indice 4)
        float montant = Float.parseFloat(line.split(",")[4]);

        // Envoyer la paire (clientID, montant) au contexte pour passer à l'étape Reduce
        context.write(new Text(clientID), new FloatWritable(montant));
        }
}
```

***/client_reduce**

```
import java.io.IOException;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

// Définition de la classe client_reduce qui hérite de Reducer
public class client_reduce extends Reducer<Text, FloatWritable, Text, FloatWritable> {

        // Fonction reduce exécutée pour chaque clientID avec tous ses montants
        public void reduce(Text clientID, Iterable<FloatWritable> paiements, Context context)
throws IOException, InterruptedException {

        // Initialiser la variable pour stocker le paiement maximum
```

```
        float maxPaiements = 0.0f;

        // Variable temporaire pour parcourir chaque paiement
        float tempo = 0.0f;

        // Boucle sur tous les montants du même clientID
        for (FloatWritable pay : paiements) {
        tempo = pay.get(); // Récupérer le montant courant

        // Si le montant actuel est supérieur au maximum enregistré, on met à jour
        if (tempo > maxPaiements) {
                maxPaiements = tempo;
        }
        }

        // Envoyer (clientID, paiement maximum) au contexte pour écrire dans le fichier final
        context.write(clientID, new FloatWritable(maxPaiements));
        }
}
```

***/client_driver (main)**

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

// Définition de la classe client_driver qui contient le main
public class client_driver {

        public static void main(String[] args) throws Exception {

        // Créer une nouvelle instance de job
        Job job = Job.getInstance();

        // Définir la classe principale contenant le main
        job.setJarByClass(client_driver.class);

        // Définir le nom du job
        job.setJobName("Max_Paiement");

        // Spécifier le chemin d'entrée du fichier sur HDFS
        FileInputFormat.addInputPath(job, new Path("/dossier_input"));

        // Spécifier le chemin de sortie des résultats sur HDFS
        FileOutputFormat.setOutputPath(job, new Path("/dossier_output"));

        // Associer la classe Mapper au job
```

```
        job.setMapperClass(client_map.class);

        // Associer la classe Reducer au job
        job.setReducerClass(client_reduce.class);

        // Définir le type de la clé de sortie (identifiant du client)
        job.setOutputKeyClass(Text.class);

        // Définir le type de la valeur de sortie (montant payé)
        job.setOutputValueClass(FloatWritable.class);

        // Lancer le job et terminer le programme selon son succès
        System.exit(job.waitForCompletion(true) ? 0 : 1);
        }
}
```

✨ Schéma du traitement MapReduce pour **Max Paiement Client**

```
                [Fichier d'entrée HDFS]
                |
        Chaque ligne = "clientID, ..., ..., ..., montant"
                |
                v
        (Phase Map)
        +------------------+
        |       client_map      |
        +------------------+
                |
        Extrait (clientID, montant)
        Exemple : ("C1", 100.0)
                ("C1", 250.0)
                ("C2", 300.0)
                ("C1", 150.0)
                ("C2", 50.0)
                |
                v
        (Shuffle / Sort)
        --> Groupement par clé
        Exemple :
        "C1" -> [100.0, 250.0, 150.0]
        "C2" -> [300.0, 50.0]
                |
                v
        (Phase Reduce)
        +--------------------+
        |   client_reduce          |
        +--------------------+
                |
        Pour chaque clientID, cherche **le montant maximum**
```

"C1" -> 250.0
"C2" -> 300.0
|
v
[Fichier de sortie HDFS]
Résultat final :
---
clientID	Max Paiement
C1	250.0
C2	300.0

Exo 2 :

***/WordCategoryMapper**

```
// Importation des bibliothèques nécessaires
import java.io.IOException;

import org.apache.hadoop.io.IntWritable; // Type Hadoop représentant un entier
import org.apache.hadoop.io.Text; // Type Hadoop représentant une chaîne de caractères
import org.apache.hadoop.mapreduce.Mapper; // Classe de base pour créer un Mapper

// Définition de la classe WordCategoryMapper qui hérite de Mapper
public class WordCategoryMapper extends Mapper<Object, Text, Text, IntWritable> {

	// Valeur constante 1 utilisée pour compter les mots
	private final static IntWritable one = new IntWritable(1);

	// Variable pour stocker la catégorie du mot
	private Text category = new Text();

	// La méthode map() est appelée automatiquement pour chaque ligne du fichier d'entrée
	@Override
	public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
	// On divise la ligne de texte en mots séparés par des espaces
	String[] words = value.toString().split(" ");

	// On traite chaque mot individuellement
	for (String word : words) {
	// On compte le nombre de morceaux du mot en le découpant par tirets "-"
	int wordLength = word.split("-").length;

	// On détermine la catégorie du mot selon le nombre de parties
	if (wordLength == 1) {
		category.set("Catégorie 1"); // Mot simple, sans tiret
	} else if (wordLength >= 2 && wordLength <= 3) {
		category.set("Catégorie 2"); // Mot composé (2 à 3 morceaux)
```

```
        } else if (wordLength >= 4 && wordLength <= 10) {
                category.set("Catégorie 3"); // Mot très composé
        } else {
                category.set("Catégorie 4"); // Mot ultra long
        }

        // On envoie la paire (catégorie, 1) au système Hadoop
        context.write(category, one);
        }
        }
}
```

***/WordCategoryReducer**

```
// Importation des bibliothèques nécessaires
import java.io.IOException;

import org.apache.hadoop.io.IntWritable; // Entier Hadoop
import org.apache.hadoop.io.Text; // Chaîne de caractères Hadoop
import org.apache.hadoop.mapreduce.Reducer; // Classe de base pour le Reducer

// Définition du Reducer : prend une clé Text et une liste de IntWritable, retourne Text et
IntWritable
public class WordCategoryReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

        // La méthode reduce() est appelée pour chaque clé (chaque catégorie)
        @Override
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {

        int sum = 0; // Initialisation du compteur

        // On parcourt tous les 1 reçus pour cette catégorie
        for (IntWritable val : values) {
        sum += val.get(); // On additionne les valeurs
        }

        // On envoie le résultat : (Catégorie, Nombre total de mots)
        context.write(key, new IntWritable(sum));
        }
}
```

***/WordCategory (main)**

```
// Importation des classes nécessaires pour configurer et exécuter un job Hadoop
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job; // Classe Job principale de la nouvelle API
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
```

```
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

// Classe principale contenant le point d'entrée du programme
public class WordCategory {

	// Méthode main, appelée au lancement du programme
	public static void main(String[] args) throws Exception {

	// Vérification des arguments (fichier d'entrée et fichier de sortie)
	if (args.length != 2) {
	System.err.println("Usage: WordCategory <input path> <output path>");
	System.exit(-1);
	}

	// Création d'un objet de configuration Hadoop
	Configuration conf = new Configuration();

	// Création d'un job avec cette configuration et un nom
	Job job = Job.getInstance(conf, "Word Category Count");

	// Indique la classe principale pour le job (celle qui contient le main)
	job.setJarByClass(WordCategory.class);

	// Définition des classes Mapper et Reducer
	job.setMapperClass(WordCategoryMapper.class);
	job.setReducerClass(WordCategoryReducer.class);

	// Définition des types de clés et de valeurs en sortie
	job.setOutputKeyClass(Text.class);
	job.setOutputValueClass(IntWritable.class);

	// Définition des chemins d'entrée et de sortie (à partir des arguments)
	FileInputFormat.addInputPath(job, new Path(args[0]));
	FileOutputFormat.setOutputPath(job, new Path(args[1]));

	// Lancement du job et attente de sa fin
	System.exit(job.waitForCompletion(true) ? 0 : 1);
	}
}
```


Résumé rapide du fonctionnement :

Étape	Que fait-on ?
Mapper	Pour chaque mot, on détecte combien il a de parties (avec les tirets "-") et on l'associe à une catégorie.

Shuffle/Sort	Hadoop groupe tous les mots par catégorie automatiquement.
Reducer	On compte le nombre de mots dans chaque catégorie.
Résultat	On obtient pour chaque catégorie le **nombre total** de mots qu'elle contient.

Exo 3 :

***/TemperatureMapper**

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

// Mapper qui émet deux paires par ligne : (ville_min, Tmin) et (ville_max, Tmax)
public class TemperatureMapper extends Mapper<Object, Text, Text, IntWritable> {

	@Override
	protected void map(Object key, Text value, Context context) throws IOException,
InterruptedException {

	// Découpe la ligne en champs à l'aide du séparateur ";"
	String[] fields = value.toString().split(";");

	// On s'assure qu'il y a bien 5 champs
	if (fields.length == 5) {
	String city = fields[1].trim(); // Récupère la ville
	int tMax = Integer.parseInt(fields[3].trim()); // Récupère la température maximale
	int tMin = Integer.parseInt(fields[4].trim()); // Récupère la température minimale

	// On émet deux paires distinctes avec des clés modifiées pour les différencier
	// La première clé est de la forme : "ville_max"
	context.write(new Text(city + "_max"), new IntWritable(tMax));

	// La seconde clé est de la forme : "ville_min"
	context.write(new Text(city + "_min"), new IntWritable(tMin));
	}
	}
}
```

***/TemperatureReducer**

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

// Reducer qui reçoit soit des Tmin, soit des Tmax selon la clé (_min ou _max)
```

```
public class TemperatureReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

	@Override
	protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {

	// Détermine si on cherche un min ou un max à partir de la clé
	boolean isMin = key.toString().endsWith("_min");

	int result = isMin ? Integer.MAX_VALUE : Integer.MIN_VALUE;

	// Parcours toutes les valeurs associées à la clé (min ou max) pour cette ville
	for (IntWritable val : values) {
	int temp = val.get();
	if (isMin && temp < result) {
		result = temp; // Si c'est un min, on garde le plus petit
	} else if (!isMin && temp > result) {
		result = temp; // Si c'est un max, on garde le plus grand
	}
	}

	// Envoie de la ville avec sa température (min ou max)
	context.write(key, new IntWritable(result));
	}
}
```

***/TemperatureAnalysis (Main)**

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class TemperatureAnalysis {

	public static void main(String[] args) throws Exception {
	if (args.length != 2) {
	System.err.println("Usage: TemperatureAnalysis <input path> <output path>");
	System.exit(-1);
	}

	// Création de la configuration du job Hadoop
	Configuration conf = new Configuration();
	Job job = Job.getInstance(conf, "Temperature Analysis");
	job.setJarByClass(TemperatureAnalysis.class);

	// Définir les classes Mapper et Reducer à utiliser
```

```
        job.setMapperClass(TemperatureMapper.class);
        job.setReducerClass(TemperatureReducer.class);

        // Définir les types de clés et valeurs en sortie
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        // Définir les chemins d'entrée et de sortie (arguments donnés au lancement du
programme)
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // Lancer le job
        System.exit(job.waitForCompletion(true) ? 0 : 1);
        }
}
```

Résumé rapide du fonctionnement :

Étape	Ce qu'on fait
Mapper	Pour chaque ligne du fichier, extrait la ville, la température maximale et minimale. Émet deux paires : `(ville_min, Tmin)` et `(ville_max, Tmax)`.
Shuffle/Sort	Hadoop trie automatiquement les données par clé (`ville_min`, `ville_max`) pour les grouper par ville.
Reducer	Pour chaque clé, calcule soit le minimum (si clé se termine par `_min`), soit le maximum (si `_max`) parmi les valeurs reçues.
Résultat	Génère pour chaque ville la température minimale et maximale rencontrée. Résultat final dans le fichier `part-r-00000`.

TP3: Système de gestion de base de données NoSQL -Hbase-

Instructions de lancement des services

sudo service hadoop-hdfs-namenode start
sudo service hadoop-hdfs-datanode start
sudo service zookeeper-server start
sudo service hbase-master start
sudo service hbase-regionserver start

Instructions de base du Shell HBase

hbase shell : Permet de lancer le Shell Hbase
status : Retourne l'état des RégionServers du système
version : Retourne la version courante de Hbase sur le système
table_help : Affiche l'aide des commandes manipulant les tables du système
whoami : Affiche des informations sur l'utilisateur courant

Instructions usuelles pour le langage de définition/manipulation de données

Création de table :
create 'NOMTABLE', 'FAMILLE1', 'FAMILLE2'...
create 'NOMTABLE',{NAME=>'FAM1'},{NAME=>'FAM2',VERSION=>3}
Destruction d'une table :
disable 'NOMTABLE'
drop 'NOMTABLE'
Ajout et suppression de n-uplets :
put 'NOMTABLE', 'CLE', 'FAM:COLONNE', 'VALEUR'
deleteall 'NOMTABLE', 'CLE', 'FAM:COLONNE' ,TIMESTAMP
Modifier (ajouter une famille de colonne…) :
Alter 'NOMTABLE',{NAME=>'NEW_FAM'}
Affichage de n-uplets :
get 'NOMTABLE', 'CLE','FAM:COLONNE',TIMESTAMP
Recherche de n-uplets :
scan 'NOMTABLE'
scan 'NOMTABLE',{CONDITIONS}

Rôles des Filtres HBase

Filtre	Rôle (Description originale)	Syntaxe Shell (Exemple)
`SingleColumnValueFilter`	*"Filter sur la valeur d'une colonne"* (Filtre les lignes où une colonne spécifique a une valeur donnée)	`scan 'ma_table', {FILTER => "SingleColumnValueFilter('famille', 'colonne', =, 'binary:valeur')"}`
`RowFilter`	*"Filter sur la row-key"* (Filtre les lignes par clé de ligne, via regex ou comparaison)	`scan 'ma_table', {FILTER => "RowFilter(=, 'regexstring:^valeur')"}`
`ColumnPrefixFilter`	*"Sélectionner des colonnes commençant par un prefixe"* (Filtre les colonnes par préfixe de nom)	`scan 'ma_table', {FILTER => "ColumnPrefixFilter('prefix')"}`

`PageFilter`	*"Limiter nombre de lignes retournées"* (Paginer les résultats)	`scan 'ma_table', {FILTER => "PageFilter(10)"}`

Exo 1 :

1. Création de la table avec deux familles de colonnes:

```
create 'facturation',
 {NAME => 'client', VERSIONS => 1},
 {NAME => 'details', VERSIONS => 3}
```

2. La famille `details` est déjà configurée pour conserver jusqu'à 3 versions dans la commande de création.
3. Insertion de factures fictives et mises à jour:

```
# Insertion initiale
put 'facturation', 'facture_001', 'client:nom', 'Client A'
put 'facturation', 'facture_001', 'client:email', 'clientA@example.com'
put 'facturation', 'facture_001', 'details:date', '2023-01-01'
put 'facturation', 'facture_001', 'details:montant', '1000'
put 'facturation', 'facture_001', 'details:statut', 'payé'

# Mise à jour du montant
put 'facturation', 'facture_001', 'details:montant', '1200'
put 'facturation', 'facture_001', 'details:statut', 'impayé'
```

4. Extraction des factures impayées:

```
scan 'facturation', {FILTER => "SingleColumnValueFilter('details', 'statut', =, 'binary:impayé')"}
```

5. Récupération des factures avec ID commençant par "facture_00":

```
scan 'facturation', {FILTER => "RowFilter(=, 'regexstring:facture_00.*')"}
```

6. Limites des filtres dans le Shell HBase:

- Syntaxe complexe et peu intuitive
- Performances dégradées sur de grandes tables
- Options de filtrage limitées comparées à SQL
- Difficulté à combiner plusieurs filtres

Exo 2 :

```
import java.io.IOException;
import java.util.Arrays;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.*;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;
```

```
public class FacturationHBase {
	public static void main(String[] args) throws IOException {
	// 1. Configuration HBase
	Configuration config = HBaseConfiguration.create();
	Connection connection = ConnectionFactory.createConnection(config);

	Admin admin = connection.getAdmin();

	TableName tableName = TableName.valueOf("facturation");

	// 2. Création de la table "facturation" avec deux familles : client et details
	if (!admin.tableExists(tableName)) {
	HTableDescriptor table = new HTableDescriptor(tableName);

	// Famille "client"
	HColumnDescriptor clientFamily = new HColumnDescriptor("client");
	table.addFamily(clientFamily);

	// Famille "details" avec 3 versions max
	HColumnDescriptor detailsFamily = new HColumnDescriptor("details");
	detailsFamily.setMaxVersions(3); // Conserver 3 versions max
	table.addFamily(detailsFamily);

	admin.createTable(table);
	System.out.println("Table créée : facturation");
	}

	Table table = connection.getTable(tableName);

	// 3. Insertion de plusieurs factures avec mises à jour (versions)
	// Exemple : facture_001
	String rowKey = "facture_001";

	// Insertion 1 : date, montant, statut
	Put put1 = new Put(Bytes.toBytes(rowKey));
	put1.addColumn(Bytes.toBytes("client"), Bytes.toBytes("nom"), Bytes.toBytes("Ali"));
	put1.addColumn(Bytes.toBytes("client"), Bytes.toBytes("email"),
Bytes.toBytes("ali@example.com"));
	put1.addColumn(Bytes.toBytes("details"), Bytes.toBytes("date"),
Bytes.toBytes("2025-04-01"));
	put1.addColumn(Bytes.toBytes("details"), Bytes.toBytes("montant"),
Bytes.toBytes("1000"));
	put1.addColumn(Bytes.toBytes("details"), Bytes.toBytes("statut"),
Bytes.toBytes("impayée"));
	table.put(put1);

	// Attendre pour simuler un autre timestamp (optionnel)
	try { Thread.sleep(1000); } catch (InterruptedException e) {}
```

```
        // Insertion 2 : mise à jour montant et statut (nouvelle version)
        Put put2 = new Put(Bytes.toBytes(rowKey));
        put2.addColumn(Bytes.toBytes("details"), Bytes.toBytes("montant"),
Bytes.toBytes("1200"));
        put2.addColumn(Bytes.toBytes("details"), Bytes.toBytes("statut"),
Bytes.toBytes("payée"));
        table.put(put2);

        System.out.println("Deux versions de la facture insérées.");

        // 4. Lecture avec plusieurs versions (jusqu'à 3)
        Get get = new Get(Bytes.toBytes(rowKey));
        get.readAllVersions();
        get.setMaxVersions(3);
        get.addColumn(Bytes.toBytes("details"), Bytes.toBytes("montant"));
        get.addColumn(Bytes.toBytes("details"), Bytes.toBytes("statut"));

        Result result = table.get(get);

        System.out.println("Versions de la facture :");
        for (Cell cell : result.rawCells()) {
        System.out.println("Colonne: " + Bytes.toString(cell.getQualifierArray(),
cell.getQualifierOffset(), cell.getQualifierLength()));
        System.out.println("Valeur: " + Bytes.toString(cell.getValueArray(),
cell.getValueOffset(), cell.getValueLength()));
        System.out.println("Timestamp: " + cell.getTimestamp());
        }

        // 5. Filtrage des factures "impayées"
        Scan scan = new Scan();
        SingleColumnValueFilter filter = new SingleColumnValueFilter(
                Bytes.toBytes("details"),
                Bytes.toBytes("statut"),
                CompareFilter.CompareOp.EQUAL,
                Bytes.toBytes("impayée")
        );
        scan.setFilter(filter);

        ResultScanner scanner = table.getScanner(scan);
        System.out.println("\nFactures impayées :");
        for (Result res : scanner) {
        System.out.println(Bytes.toString(res.getRow()));
        }

        // Fermeture des ressources
        scanner.close();
        table.close();
```

```
connection.close();
}
```

Exo 3 :

Explication avant le début de la correction de l'exo :

1. Modèle Clé/Valeur

- **Stockage** : une **clé unique** pointe vers une **valeur opaque** (pas de structure interne visible).
- **Exemples** : Redis, DynamoDB.
- **Avantages** :
    - Très **rapide** pour retrouver une valeur à partir d'une clé.
- **Inconvénients** :
    - **Pas adapté** pour faire des recherches complexes.
    - **Erreur sur la clé = donnée introuvable**.
- **Utilisation** :
    - Cas où seule **l'identification rapide** est importante (ex: retrouver une place de cinéma avec son numéro).

2. Modèle Orienté Colonne

- **Stockage** : Données par **colonnes** (et non par lignes comme dans une base relationnelle).
- **Exemples** : HBase, Cassandra.
- **Avantages** :
    - **Ajout facile** de nouvelles colonnes.
    - **Bon pour l'analytique** (traverser de larges ensembles de données sur une colonne).
- **Inconvénients** :
    - Moins pratique si tu dois lire **tout** d'un coup plusieurs colonnes différentes.
- **Utilisation** :

    - **Grands volumes de données analytiques**.
    - Cas où **les données évoluent souvent** (ex: ajouter de nouveaux champs).

3. Modèle Orienté Document

- **Stockage** : Chaque **document** est un objet autonome, souvent en **JSON**, **BSON** ou **XML**.
- **Exemples** : MongoDB, CouchDB.
- **Avantages** :
    - **Flexible** : un document peut avoir des champs différents d'un autre.
    - Parfait pour des **données semi-structurées**.
- **Inconvénients** :
    - Pas optimal pour les **requêtes analytiques massives**.
- **Utilisation** :
    - **Applications évolutives**, avec des données très **hétérogènes** (ex: sites web, catalogues, bibliothèques en ligne).

4. Modèle Orienté Graphe

- **Stockage** : Données sous forme de **nœuds** et **relations** (arêtes).
- **Exemples** : Neo4j, JanusGraph.
- **Avantages** :
    - **Parfait** pour modéliser les **relations complexes** (réseaux sociaux, recommandation).
- **Inconvénients** :
    - Moins efficace si tu n’as **pas beaucoup de relations** entre données.
- **Utilisation** :
    - Quand **les relations sont aussi importantes** que les données elles-mêmes.

5. Modèle Relationnel (bases classiques)

- **Stockage** : **Tables** avec des lignes et colonnes fixes, fortement structurées.
- **Exemples** : MySQL, PostgreSQL, Oracle.
- **Avantages** :
    - **Très bonne cohérence** (**ACID**).
    - **Langage SQL puissant** pour requêter.
- **Inconvénients** :
    - **Rigidité** : difficile de faire évoluer le schéma.
- **Utilisation** :
    - Cas où il y a un besoin fort de **cohérence** et de **relations bien définies**.

📋 Résumé tableau rapide :

Modèle	Avantages	Inconvénients	Cas typiques
Clé/Valeur	Très rapide	Clé obligatoire, aucune recherche complexe	Cache, recherches ultra rapides
Colonne	Flexible, bon pour analytique	Mauvais si données très liées	Big Data, analytique
Document	Flexible, naturel pour données semi-structurées	Pas top pour l'analytique lourd	Applications Web, CMS
Graphe	Relations complexes très efficaces	Pas utile si peu de relations	Réseaux sociaux, graphes
Relationnel	Cohérence forte (ACID)	Rigidité du schéma	Systèmes critiques (banques, stocks)

cas 1 :

- Dans une base **clé/valeur**, la modification est complexe car il faut retrouver précisément la clé.
  Une petite erreur dans la clé peut casser la cohérence de la base.
  ➔ La cohérence dépend entièrement du modèle clé/valeur.

- **Base orientée graphe** : pas adaptée ici, car il n'y a pas beaucoup de relations entre les données.

- **Base orientée colonne** : plus adaptée, car on peut facilement ajouter de nouvelles informations sans perturber les colonnes existantes.
  ➔ Contrairement au modèle **document**, où une modification redéploie tout le document.

cas 2 :

- **Numériser** : avoir son document disponible sous forme électronique.

- **Cataloguer** : donner accès aux documents en ligne (ex: lire un livre en ligne).

- **Modèle orienté colonne** :

    - On peut stocker les métadonnées (titre, auteur, etc.).

    - Le fichier numérisé lui-même est stocké ailleurs (dépôt externe) ; la **colonne contient juste l'URL**.

- **Modèle orienté document** :

    - Plus naturel, car on peut intégrer directement tout ce qui concerne le document dans un seul enregistrement.

cas 3 :

- **Modèle clé/valeur** :

    - Recherche très rapide basée uniquement sur le **numéro de siège**.

    - **Pas besoin** d'une base orientée colonne, car aucune évolution complexe n'est prévue.
      ➔ Ici, **seule la vitesse est recherchée**.

cas 4 :

- **Modèle orienté colonne** :

    - Mieux adapté qu'un modèle document, car il y a un **besoin d'analytique** (profil précis à récupérer).

    - **Modèle relationnel** possible aussi si la quantité de données est faible.

cas 5 :

- **Modèle orienté graphe** :
    - Dans un cas réel complexe (beaucoup de relations entre utilisateurs), ce serait idéal.
- **Mais** ici, avec peu de paramètres et si la vitesse est prioritaire, **clé/valeur** reste suffisant.

cas 6 :

- Le critère principal est **la haute cohérence**.
- Il faudrait alors utiliser :
    - Soit une base **relationnelle**,
    - Soit éventuellement un **modèle orienté colonne** (qui peut garantir de la cohérence).

cas 7 :

- **Modèle orienté colonne** est adapté.

cas 8 :

- **Non**, une base relationnelle n'est pas idéale ici.
- Car les **données sont variées** et évolutives, donc mieux gérées par :
    - **Une base orientée document**,
    - **Ou éventuellement une base orientée colonne**.

cas 9 :

- **Pas clé/valeur** : il y a trop de données différentes.
- **Pas colonne** : difficulté de gérer des champs parfois absents (beaucoup de NULL).
- **Oui document** :
    - Les données sont **semi-structurées**,
    - Ajouter de nouvelles infos ne casse pas l'existant.

TP4: Initiation au Cloud Computing

Cas 1 (Startup)

- **Modèle de service Cloud** : **PaaS** (Platform as a Service)
  Justification : La startup a besoin d'un environnement pour développer, tester et déployer rapidement des applications sans gérer l'infrastructure sous-jacente. PaaS fournit des outils et frameworks prêts à l'emploi (ex: Google App Engine, Heroku).
- **Type de cloud** : **Public**
  Justification : Le cloud public offre une scalabilité immédiate, des coûts réduits (pas d'investissement initial), et convient aux startups en croissance rapide.

Cas 2 (Compagnie de télécommunication)

- **Modèle de service Cloud** : **IaaS** (Infrastructure as a Service)
  Justification : Le traitement de grands volumes de données en temps réel nécessite une infrastructure flexible (ex: Hadoop/Spark sur AWS EC2 ou Google Compute Engine).
- **Type de cloud** : **Hybride**
  Justification : Les données sensibles peuvent être traitées sur un cloud privé, tandis que les pics de demande réseau sont gérés par le cloud public pour l'élasticité.

Cas 3 (Banque internationale)

- **Modèle de service Cloud** : **IaaS** ou **Cloud Privé**
  Justification : La banque a besoin d'un contrôle total sur les données sensibles et la conformité réglementaire (ex: OpenStack ou VMware en cloud privé).
- **Type de cloud** : **Privé**
  Justification : Sécurité maximale, conformité aux régulations financières (ex: RGPD, Bâle III), malgré un coût plus élevé.

Cas 4 (Plateforme de e-commerce)

- **Modèle de service Cloud** : **SaaS** (Software as a Service) pour les outils analytiques (ex: Salesforce CRM) + **IaaS** pour l'infrastructure élastique (ex: AWS Lambda).
- **Type de cloud** : **Public**
  Justification : Besoin d'élasticité pour gérer les pics de trafic (ex: Black Friday) et de solutions clés en main pour l'analyse en temps réel.

Cas 5 (Université)

- **Modèle de service Cloud** : **SaaS**
  Justification : Utilisation de plateformes éducatives clés en main (ex: Moodle, Google Classroom) sans gestion d'infrastructure.
- **Type de cloud** : **Public**
  Justification : Coût réduit, accès global pour les étudiants, et maintenance externalisée.

Cas 6 (Organisme de santé)

- **Modèle de service Cloud** : **IaaS** pour Hadoop/HDFS
  Justification : Stockage et analyse de données médicales volumineuses nécessitent une infrastructure scalable (ex: Hadoop sur Azure HDInsight).
- **Type de cloud** : **Hybride**
  Justification : Stockage des données sensibles sur un cloud privé pour la conformité (ex: HIPAA), avec analyse distribuée sur le cloud public pour la puissance de calcul.

Synthèse des choix pour Hadoop

- **Modèle de service** : **IaaS** (ex: AWS EMR, Google Dataproc)
  Pourquoi : Hadoop nécessite un contrôle fin sur les nœuds de stockage (HDFS) et de calcul (MapReduce), ce que IaaS permet.
- **Type de cloud** :
  - **Public** : Si les données ne sont pas sensibles (coût réduit, scalabilité).
  - **Hybride** : Pour des données sensibles (ex: santé), avec le traitement sur le public et le stockage sur le privé.

Avantages des architectures proposées

1. **Coût** :
   - Public : Pas d'investissement initial, paiement à l'usage.
   - Privé : Coût élevé mais justifié pour la sécurité (banque, santé).
2. **Agilité** :
   - PaaS/SaaS permettent un déploiement rapide (startup, université).
3. **Scalabilité** :
   - Elasticité du cloud public pour les pics de demande (e-commerce, télécoms).

Gestion des Big Data :

- IaaS + Hadoop permet une analyse distribuée et élastique.
- Les solutions SaaS (ex: Google BigQuery) simplifient l'analyse sans gestion d'infrastructure.

TP5: Traitement de données avec Apache Spark

1. Transformations vs Actions

✅ a) Transformations

Ce sont des opérations paresseuses (lazy) qui définissent un nouveau RDD à partir d'un autre, sans l'exécuter immédiatement.

Elles ne déclenchent aucun calcul tant qu'aucune action n'est demandée.

Exemples :

map(), filter(), flatMap(), union(), distinct(), groupByKey(), etc.

🧠 Elles construisent une chaîne logique d'opérations, mais aucune donnée n'est réellement traitée à ce stade.

✅ b) Actions

Ce sont des opérations terminales qui déclenchent réellement l'exécution du DAG et le traitement des données.

Elles renvoient un résultat (ou déclenchent un effet, comme écrire un fichier).

Exemples :

collect(), count(), first(), take(n), reduce(), saveAsTextFile(), etc.

💥 Elles forcent Spark à exécuter toutes les transformations nécessaires pour produire un résultat.

🔷 2. DAG – Directed Acyclic Graph

📌 Définition

Le DAG (graphe orienté acyclique) est une représentation des étapes de calcul dans Spark.

Chaque nœud du DAG représente une transformation.

Les arêtes indiquent la dépendance entre les opérations.

Il est acyclique : pas de boucle infinie.

🔧 Fonctionnement

Quand on applique des transformations, Spark construit le DAG, sans rien exécuter.

Lorsqu'une action est appelée, Spark optimise et découpe le DAG en stages et tasks.

Ensuite, il exécute les stages dans l'ordre défini par le DAG.

🎯 Exemple :

```
rdd1 = sc.textFile("fichier.txt")          # RDD 1

rdd2 = rdd1.filter(lambda x: "error" in x)   # RDD 2

rdd3 = rdd2.map(lambda x: (x, 1))     # RDD 3

result = rdd3.count()              # ACTION
```

🔽 Ce code construit ce DAG :

textFile → filter → map → count

Seule la dernière ligne (count()) déclenche le calcul de tout le graphe.

help(map) : appel à l'aide sur la fonction map de Python, non utile ici dans le contexte Spark.

rdd.collect() : utile pour voir le contenu mais à éviter sur de grands fichiers (risque de surcharge mémoire).

Des erreurs de syntaxe étaient présentes (if x<y : x else y) → corrigé avec max() et min() directement.

map(lambda x: x.split(",")[0], ...) → fait plusieurs fois sans filtrage, a mené à des transformations inutiles.

Exercice 1 : Température MIN et MAX par station

🎯 **Objectif :**

- Récupérer la température MAX pour chaque station.
- Récupérer la température MIN pour chaque station.
- Associer les deux températures pour chaque station.

```
# 1. Charger les données

rdd = sc.textFile("ExoTemperature.csv")

# 2. Filtrer les températures MAX

rddMAX = rdd.filter(lambda x: "MAX" in x)
```

```
# 3. Mapper les lignes MAX sous forme (Station_ID, température)
rddFiltreMax = rddMAX.map(lambda x: (
        x.split(",")[0],            # Station_ID
        int(x.split(",")[3])        # Température en dixièmes de degré
))
# 4. Trouver la température MAX par station
rddMaxParStation = rddFiltreMax.reduceByKey(lambda x, y: max(x, y))
# 5. Idem pour MIN
rddMIN = rdd.filter(lambda x: "MIN" in x)

rddFiltreMin = rddMIN.map(lambda x: (
        x.split(",")[0],
        int(x.split(",")[3])
))
rddMinParStation = rddFiltreMin.reduceByKey(lambda x, y: min(x, y))
# 6. Joindre les résultats MIN et MAX
rddMinMax = rddMinParStation.join(rddMaxParStation)
# 7. Afficher le résultat
rddMinMax.collect()
```


✅ Exercice 2 : Analyse des températures

1. Moyenne mensuelle par station

```
# Extraire les températures avec station et mois comme clé
rddTemp = rdd.map(lambda x: (
        (x.split(",")[0], x.split(",")[1][4:6]),  # (Station_ID, Mois)
        int(x.split(",")[3]) / 10.0           # Température en °C
))
```

```
# Calcul de la moyenne avec aggregateByKey

rddMoyMensuelle = rddTemp \

        .aggregateByKey((0, 0),                         # (somme, compte)

                lambda acc, val: (acc[0]+val, acc[1]+1),

                lambda acc1, acc2: (acc1[0]+acc2[0], acc1[1]+acc2[1])) \

        .mapValues(lambda acc: acc[0]/acc[1])        # moyenne = somme / compte
```

🔹 **2. Détection d'anomalies (écart > 20%)**

```
# Mapper à nouveau pour joindre les moyennes

rddTempParDate = rdd.map(lambda x: (

        (x.split(",")[0], x.split(",")[1][4:6]),        # (Station_ID, mois)

        (x.split(",")[1], float(x.split(",")[3]) / 10.0) # (Date, Température)

))

# Jointure avec les moyennes mensuelles

rddJoin = rddTempParDate.join(rddMoyMensuelle)  # ((Station_ID, mois), ((date, temp), moyenne))

# Détection des anomalies : écart > 20%

rddAnomalies = rddJoin.filter(lambda x: abs(x[1][0][1] - x[1][1]) > 0.2 * x[1][1])

# Compter les anomalies par station

rddAnomaliesParStation = rddAnomalies.map(lambda x: (x[0][0], 1))
```

🔹 **3. Histogramme des anomalies par station**

```
import matplotlib.pyplot as plt

# Réduction

resultats = rddAnomaliesParStation.reduceByKey(lambda x, y: x + y).collect()

# Données pour matplotlib

stations = [x[0] for x in resultats]

anomalies = [x[1] for x in resultats]
```

```
# Affichage de l’histogramme
plt.figure(figsize=(10, 6))
plt.bar(stations, anomalies, color="tomato")
plt.xlabel("Station")
plt.ylabel("Nombre d’anomalies")
plt.title("Anomalies par station (écart > 20%)")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```