

Atelier 1 : Cluster Spark avec Docker

Analyse des événements GDELT

Big Data - DIC3 Informatique

Groupe 2: **Awa Diop, Mouhamed Lamine Faye**

Dialika Thiaw et Moussa Ndoeye

9 décembre 2025

Table des matières

1	Introduction - Projet GDELT	3
1.1	Présentation de GDELT	3
1.2	Objectif de l'atelier	3
1.3	Source des données	3
1.4	Fichier utilisé	3
1.5	Identification de la colonne pays	3
2	Code Source	3
2.1	Architecture du code	3
2.1.1	Structure des fichiers	4
2.2	Classe GDELTEventCounter	4
2.3	Réutilisabilité et flexibilité	5
2.4	Module timer.py	5
3	Environnement d'exécution local	5
3.1	Configuration Windows	5
3.1.1	Prérequis	5
3.1.2	Téléchargement de winutils	6
3.1.3	Téléchargement de Spark 3.5.7	6
3.2	Vérification de l'installation	6
3.3	Exécution locale	6
4	Déploiement Docker	7
4.1	Architecture du cluster	7
4.2	Fichier docker-compose.yml	7
4.3	Démarrage du cluster	7
4.4	Interfaces Web	8
4.4.1	Spark Master UI (port 8080)	8
4.4.2	Spark Worker 1 UI (port 8081)	9
4.4.3	Spark Worker 2 UI (port 8082)	9
4.5	Exécution sur le cluster	9

5	Comparaison des performances	9
5.1	Fichier de test	9
5.2	Exécution locale	10
5.3	Exécution sur le cluster	11
5.4	Tableau comparatif	11
6	Conclusion	11
6.1	Points clés	12
6.2	Avantages du cluster	12
6.3	Perspectives	12

1 Introduction - Projet GDELT

1.1 Présentation de GDELT

Le projet **GDELT** (Global Database of Events, Language, and Tone) est une initiative qui surveille les médias du monde entier en temps réel. Il analyse les actualités diffusées dans plus de 100 langues et identifie les personnes, lieux, organisations, thèmes et événements qui façonnent notre monde.

GDELT est la plus grande base de données ouverte sur le comportement humain à travers le monde, couvrant des événements depuis 1979 jusqu'à aujourd'hui.

1.2 Objectif de l'atelier

L'objectif de cet atelier est de **compter le nombre d'événements par pays** à partir des données GDELT, en utilisant Apache Spark pour le traitement distribué.

1.3 Source des données

En exploitant le projet GDELT, nous nous sommes rendus sur le site officiel pour récupérer les fichiers de données :

<http://data.gdeltproject.org/events/index.html>

Nous utilisons le format **GDELT 2.0 Events** qui contient 58 colonnes par événement.

1.4 Fichier utilisé

Pour l'étude et les tests en local, nous avons téléchargé le fichier :

- **Nom** : 20251208.export.CSV.zip
- **Taille** : 41.4 Mo (compressé)
- **Événements** : 111,373 événements

1.5 Identification de la colonne pays

Après analyse du schéma GDELT 2.0, nous avons identifié la **colonne 51** (index à partir de 0) comme étant la colonne **ActionGeo_CountryCode**. Cette colonne contient les codes pays FIPS à 2 lettres (ex : US, UK, FR, IN).

Ces codes étant uniques par pays, ils sont idéaux pour effectuer le comptage des événements.

2 Code Source

2.1 Architecture du code

Le code est organisé de manière orientée objet avec une classe principale **GDELTEventCounter** qui encapsule toute la logique de traitement Spark.

2.1.1 Structure des fichiers

```
app/  
    __init__.py          # Initialisation du package  
    event_counter.py     # Classe principale  
    timer.py            # Module de mesure du temps
```

2.2 Classe GDELTEventCounter

```
1 class GDELTEventCounter:  
2     """Classe pour compter les evenements GDELT par pays."""  
3  
4     def __init__(self, master: str = "local[*]",  
5                  app_name: str = "GDELTEventCounter",  
6                  country_col: int = 51,  
7                  has_header: bool = False):  
8         self.spark = SparkSession.builder \  
9             .appName(app_name) \  
10            .master(master) \  
11            .getOrCreate()  
12        self.country_col = country_col  
13        self.has_header = has_header  
14        self.df = None  
15        self.results = None  
16  
17        @timed  
18        def load_data(self, input_path: str) -> DataFrame:  
19            """Charge les donnees GDELT depuis un fichier CSV."""  
20            self.df = self.spark.read.csv(  
21                input_path, sep='\\t',  
22                header=self.has_header, inferSchema=True  
23            )  
24            return self.df  
25  
26        @timed  
27        def count_by_country(self) -> DataFrame:  
28            """Compte les evenements par pays."""  
29            col_name = self.df.columns[self.country_col] \  
30                if self.has_header else f"_c{self.country_col}"  
31            country_df = self.df.withColumnRenamed(col_name,  
32                "CountryCode")  
33            self.results = country_df.filter(  
34                col("CountryCode").isNotNull()  
35            ).groupBy("CountryCode") \  
36                .agg(count("*").alias("EventCount")) \  
37                .orderBy(col("EventCount").desc())  
38            return self.results  
39  
40        @timed  
41        def save_results(self, output_path: str) -> None:
```

```

41     """Sauvegarde les resultats en CSV."""
42     self.results.coalesce(1).write \
43         .mode("overwrite").option("header", "true") \
44         .csv(output_path)

```

2.3 Réutilisabilité et flexibilité

Le code est conçu pour être **réutilisable** et **flexible** grâce aux paramètres de ligne de commande :

Argument	Description	Défaut
-input	Chemin du fichier GDELT	datas/20251208.export.CSV
-output	Chemin de sortie CSV	output/event_counts
-master	URL du Spark Master	local[*]
-country-col	Index de la colonne pays	51
-header	Fichier avec en-tête	False

TABLE 1 – Arguments de ligne de commande

2.4 Module timer.py

Le décorateur `@timed` permet de mesurer automatiquement le temps d'exécution de chaque méthode :

```

1 def timed(func):
2     """Décorateur pour mesurer le temps d'exécution."""
3     @wraps(func)
4     def wrapper(*args, **kwargs):
5         start = time.time()
6         result = func(*args, **kwargs)
7         end = time.time()
8         print(f"[TIMER] {func.__name__}: {end - start:.2f}
9               secondes")
10        return result
    return wrapper

```

3 Environnement d'exécution local

3.1 Configuration Windows

L'environnement local est configuré sur Windows avec les composants suivants :

3.1.1 Prérequis

- **Java** : JDK 11 ou 17
- **Apache Spark** : Version 3.5.7
- **Hadoop** : Binaires Windows (winutils)

3.1.2 Téléchargement de winutils

Pour que Spark puisse écrire des fichiers sur Windows, nous avons téléchargé les binaires Hadoop depuis :

<https://github.com/cdarlint/winutils/tree/master/hadoop-3.3.6/bin>

Les fichiers ont été placés dans `C:\hadoop\bin\`.

3.1.3 Téléchargement de Spark 3.5.7

Apache Spark a été téléchargé depuis :

<https://www.apache.org/dyn/closer.lua/spark/spark-3.5.7/spark-3.5.7-bin-hadoop3.tgz>

3.2 Vérification de l'installation

```
spark-submit --version
```

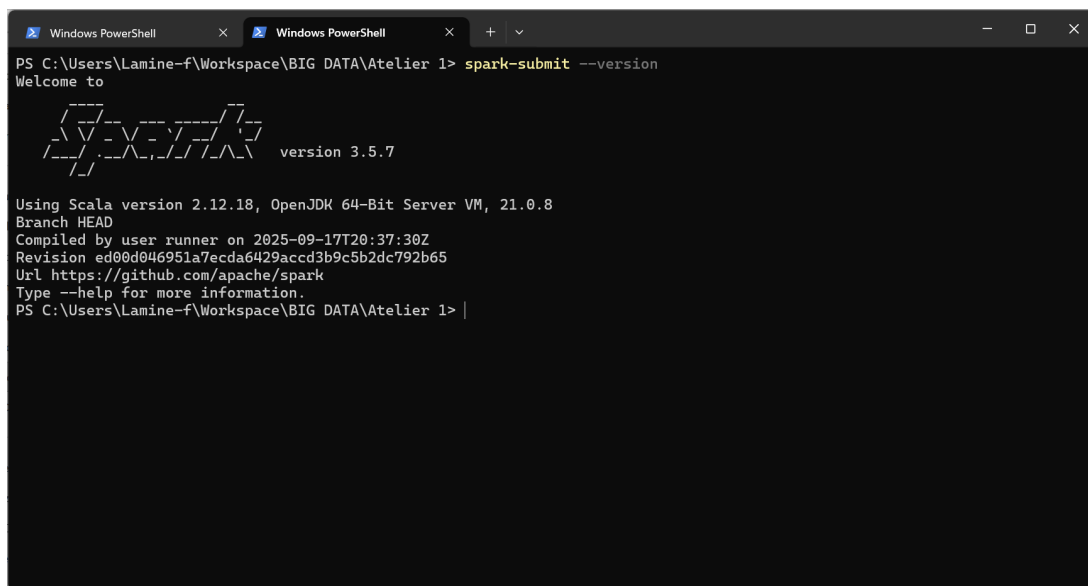


FIGURE 1 – Vérification de la version de Spark

3.3 Exécution locale

Pour lancer le programme en local :

```
spark-submit app/event_counter.py --input datas/20251208.export.CSV
```

4 Déploiement Docker

4.1 Architecture du cluster

Le cluster Spark est déployé avec Docker Compose et comprend :

- **1 Spark Master** : Coordonne les tâches
- **2 Spark Workers** : Exécutent les tâches

Chaque worker est configuré avec :

- 2 Go de mémoire
- 2 cœurs CPU

4.2 Fichier docker-compose.yml

```
1 services:
2   spark-master:
3     image: apache/spark:3.5.0
4     container_name: spark-master
5     ports:
6       - "8080:8080"    # Web UI Master
7       - "7077:7077"    # Port Spark Master
8       - "4040:4040"    # Application UI
9     volumes:
10      - ./datas:/data
11      - ./app:/app
12      - ./output:/output
13
14   spark-worker-1:
15     image: apache/spark:3.5.0
16     container_name: spark-worker-1
17     environment:
18       - SPARK_WORKER_MEMORY=2G
19       - SPARK_WORKER_CORES=2
20     ports:
21       - "8081:8081"
22
23   spark-worker-2:
24     image: apache/spark:3.5.0
25     container_name: spark-worker-2
26     environment:
27       - SPARK_WORKER_MEMORY=2G
28       - SPARK_WORKER_CORES=2
29     ports:
30       - "8082:8081"
31
32 networks:
33   spark-net:
34     driver: bridge
```

4.3 Démarrage du cluster

```
# Demarrer le cluster
docker-compose up -d

# Verifier les conteneurs
docker ps
```

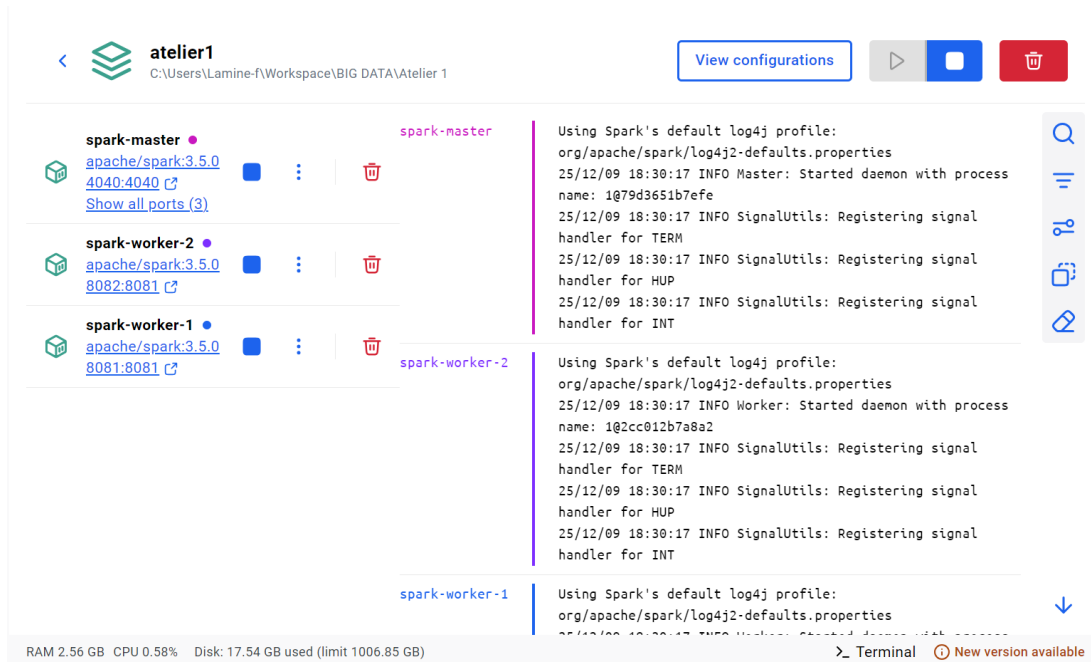


FIGURE 2 – Docker Desktop - Conteneurs du cluster Spark

4.4 Interfaces Web

4.4.1 Spark Master UI (port 8080)

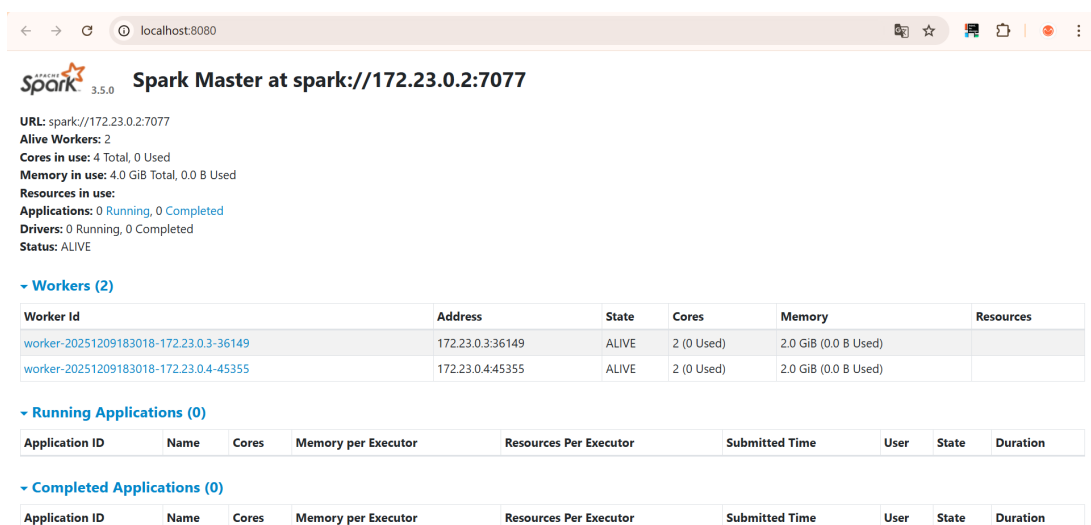


FIGURE 3 – Interface Web du Spark Master

4.4.2 Spark Worker 1 UI (port 8081)



FIGURE 4 – Interface Web du Worker 1

4.4.3 Spark Worker 2 UI (port 8082)



FIGURE 5 – Interface Web du Worker 2

4.5 Exécution sur le cluster

```
docker exec spark-master /opt/spark/bin/spark-submit \  
  --master spark://spark-master:7077 \  
  /app/event_counter.py \  
  --input /data/20251208.export.CSV \  
  --output /output/results
```

5 Comparaison des performances

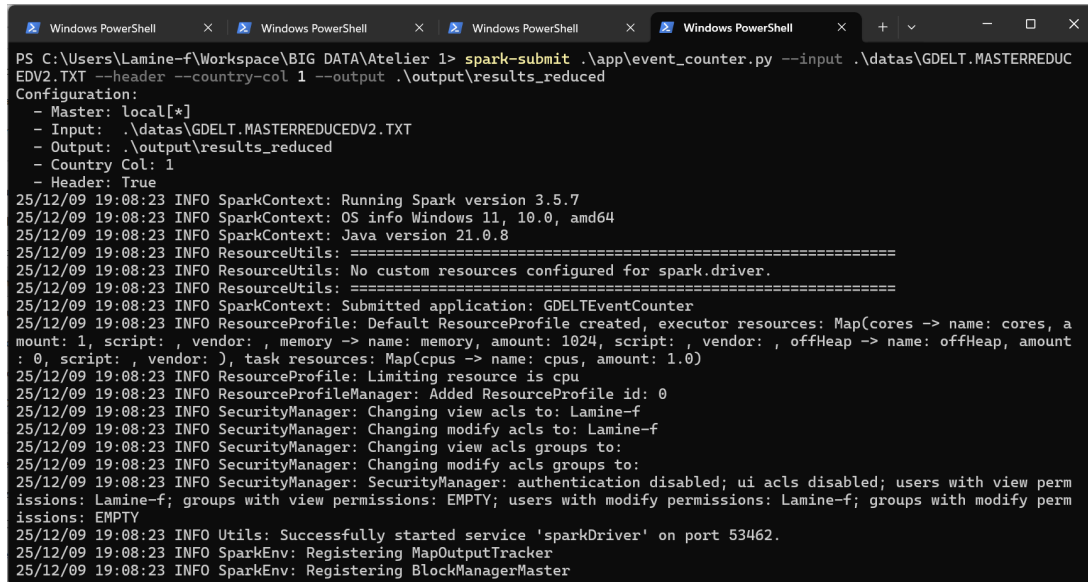
Avec un cluster de 3 nœuds (1 master + 2 workers), nous avons testé les performances avec un fichier volumineux.

5.1 Fichier de test

— Fichier : GDELT.MASTERREDUCEDV2.1979-2013.zip

- Taille : 6.12 Go (compressé)
- Événements : 87,298,047 événements
- Période : 1979 à 2013

5.2 Exécution locale

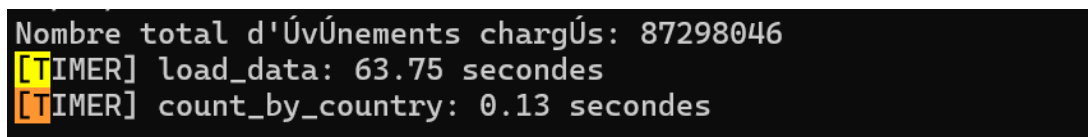


```

PS C:\Users\Lamine-f\Workspace\BIG DATA\Atelier 1> spark-submit .\app\event_counter.py --input .\datas\GDELT.MASTERREDUCEDV2.TXT --header --country-col 1 --output .\output\results_reduced
Configuration:
- Master: local[*]
- Input: .\datas\GDELT.MASTERREDUCEDV2.TXT
- Output: .\output\results_reduced
- Country Col: 1
- Header: True
25/12/09 19:08:23 INFO SparkContext: Running Spark version 3.5.7
25/12/09 19:08:23 INFO SparkContext: OS info Windows 11, 10.0, amd64
25/12/09 19:08:23 INFO SparkContext: Java version 21.0.8
25/12/09 19:08:23 INFO ResourceUtils: =====
25/12/09 19:08:23 INFO ResourceUtils: No custom resources configured for spark.driver.
25/12/09 19:08:23 INFO ResourceUtils: =====
25/12/09 19:08:23 INFO SparkContext: Submitted application: GDELTEventCounter
25/12/09 19:08:23 INFO ResourceProfile: Default ResourceProfile created, executor resources: Map(cores -> name: cores, amount: 1, script: , vendor: , memory -> name: memory, amount: 1024, script: , vendor: , offHeap -> name: offHeap, amount: 0, script: , vendor: ), task resources: Map(cpus -> name: cpus, amount: 1.0)
25/12/09 19:08:23 INFO ResourceProfile: Limiting resource is cpu
25/12/09 19:08:23 INFO ResourceProfileManager: Added ResourceProfile id: 0
25/12/09 19:08:23 INFO SecurityManager: Changing view acls to: Lamine-f
25/12/09 19:08:23 INFO SecurityManager: Changing modify acls to: Lamine-f
25/12/09 19:08:23 INFO SecurityManager: Changing view acls groups to:
25/12/09 19:08:23 INFO SecurityManager: Changing modify acls groups to:
25/12/09 19:08:23 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Lamine-f; groups with view permissions: EMPTY; users with modify permissions: Lamine-f; groups with modify permissions: EMPTY
25/12/09 19:08:23 INFO Utils: Successfully started service 'sparkDriver' on port 53462.
25/12/09 19:08:23 INFO SparkEnv: Registering MapOutputTracker
25/12/09 19:08:23 INFO SparkEnv: Registering BlockManagerMaster

```

FIGURE 6 – Lancement du script en local - Gros fichier

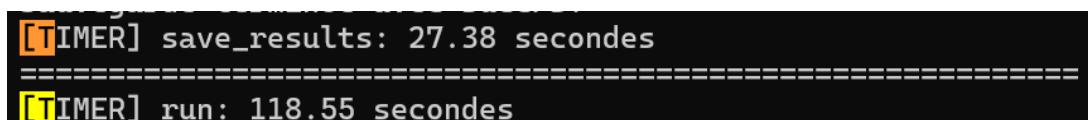


```

Nombre total d'Événements chargés: 87298046
[TIMER] load_data: 63.75 secondes
[TIMER] count_by_country: 0.13 secondes

```

FIGURE 7 – Temps de chargement et comptage - Local



```

[TIMER] save_results: 27.38 secondes
=====
[TIMER] run: 118.55 secondes

```

FIGURE 8 – Temps de sauvegarde et total - Local

5.3 Exécution sur le cluster

```

PS C:\Users\Lamine-f\Workspace\BIG DATA\Atelier 1> docker exec spark-master /opt/spark/bin/spark-submit --master spark://spark-master:7077 /app/event_counter.py --input /data/GDELT.MASTERREDUCEDV2.TXT --header --country-col 1 --output /output/results_reduced
Configuration:
- Master: local[*]
- Input: /data/GDELT.MASTERREDUCEDV2.TXT
- Output: /output/results_reduced
- Country Col: 1
- Header: True
25/12/09 19:00:12 INFO SparkContext: Running Spark version 3.5.0
25/12/09 19:00:12 INFO SparkContext: OS info Linux, 6.6.87.2-microsoft-standard-WSL2, amd64
25/12/09 19:00:12 INFO SparkContext: Java version 11.0.20.1
25/12/09 19:00:12 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
25/12/09 19:00:12 INFO ResourceUtils: =====
25/12/09 19:00:12 INFO ResourceUtils: No custom resources configured for spark.driver.
25/12/09 19:00:12 INFO ResourceUtils: =====
25/12/09 19:00:12 INFO SparkContext: Submitted application: GDELTEventCounter
25/12/09 19:00:12 INFO ResourceProfile: Default ResourceProfile created, executor resources: Map(cores -> name: cores, amount: 1, script: , vendor: , memory -> name: memory, amount: 1024, script: , vendor: , offHeap -> name: offHeap, amount: 0, script: , vendor: ), task resources: Map(cpus -> name: cpus, amount: 1.0)
25/12/09 19:00:12 INFO ResourceProfile: Limiting resource is cpu
25/12/09 19:00:12 INFO ResourceProfileManager: Added ResourceProfile id: 0
25/12/09 19:00:12 INFO SecurityManager: Changing view acls to: spark
25/12/09 19:00:12 INFO SecurityManager: Changing modify acls to: spark
25/12/09 19:00:12 INFO SecurityManager: Changing view acls groups to:
25/12/09 19:00:12 INFO SecurityManager: Changing modify acls groups to:
25/12/09 19:00:12 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: spark; groups with view permissions: EMPTY; users with modify permissions: spark; groups with modify permissions: EMPTY

```

FIGURE 9 – Lancement du script sur le cluster - Gros fichier

```

Nombre total d'événements chargés: 87298046
[TIMER] load_data: 102.12 secondes
[TIMER] count_by_country: 0.12 secondes

```

FIGURE 10 – Temps de chargement et comptage - Cluster

```

[TIMER] save_results: 43.12 secondes
=====
[TIMER] run: 184.44 secondes

```

FIGURE 11 – Temps de sauvegarde et total - Cluster

5.4 Tableau comparatif

Métrique	Local	Cluster (3 nœuds)
Chargement des données	63.75 s	102.12 s
Comptage par pays	0.13 s	0.12 s
Sauvegarde des résultats	27.38 s	43.12 s
Temps total	118.55 s	184.44 s

TABLE 2 – Comparaison des temps d'exécution - Fichier GDELT Reduced (87M événements)

6 Conclusion

Cet atelier a permis de mettre en place une infrastructure de traitement distribué avec Apache Spark pour analyser les données GDELT.

6.1 Points clés

- **Flexibilité du code** : Le programme est paramétrable et réutilisable pour différents formats de fichiers GDELT.
- **Scalabilité** : Le déploiement Docker permet d'ajouter facilement des workers pour augmenter la capacité de traitement.
- **Mesure des performances** : Le module timer permet de comparer objectivement les temps d'exécution.

6.2 Analyse des résultats

Les résultats montrent que l'exécution locale est plus rapide pour ce jeu de données (118.55 s vs 184.44 s). Cela s'explique par :

- **Overhead de distribution** : La sérialisation et le transfert des données entre le master et les workers ajoutent un délai significatif.
- **Taille du fichier** : Avec 87 millions d'événements (2.5 Go), le fichier n'est pas assez volumineux pour amortir le coût de la distribution.
- **Environnement Docker** : L'exécution dans des conteneurs sur une même machine ajoute une couche de virtualisation.

Le cluster distribué devient avantageux lorsque :

- Le volume de données dépasse plusieurs dizaines de Go
- Les workers sont sur des machines physiques distinctes
- Le traitement est plus complexe (jointures, agrégations multiples)

6.3 Perspectives

Ce travail pourrait être étendu pour :

- Analyser d'autres dimensions des données GDELT (acteurs, types d'événements)
- Déployer sur un cluster cloud (AWS EMR, Google Dataproc)
- Implémenter un traitement en temps réel avec Spark Streaming