Atelier 1 : Cluster Spark avec Docker

Analyse des événements GDELT

Big Data - DIC3 Informatique
Groupe 2: **Awa Ndiaye, Mouhamed Lamine Faye**
Dialika Thiaw et Moussa Ndoye

10 décembre 2025

Table des matières

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

```python
class GDELTEventCounter:
    """Classe pour compter les evenements GDELT par pays."""

    def __init__(self, master: str = "local[*]",
                 app_name: str = "GDELTEventCounter",
                 country_col: int = 51,
                 has_header: bool = False):
        self.spark = SparkSession.builder \
            .appName(app_name) \
            .master(master) \
            .getOrCreate()
        self.country_col = country_col
        self.has_header = has_header
        self.df = None
        self.results = None

    @timed
    def load_data(self, input_path: str) -> DataFrame:
        """Charge les donnees GDELT depuis un fichier CSV."""
        self.df = self.spark.read.csv(
            input_path, sep='\t',
            header=self.has_header, inferSchema=True
        )
        return self.df

    @timed
    def count_by_country(self) -> DataFrame:
        """Compte les evenements par pays."""
        col_name = self.df.columns[self.country_col] \
            if self.has_header else f"_c{self.country_col}"
        country_df = self.df.withColumnRenamed(col_name,
           "CountryCode")
        self.results = country_df.filter(
            col("CountryCode").isNotNull()
        ).groupBy("CountryCode") \
         .agg(count("*").alias("EventCount")) \
         .orderBy(col("EventCount").desc())
        return self.results

    @timed
    def save_results(self, output_path: str) -> None:
```

```
        """Sauvegarde les resultats en CSV."""
        self.results.coalesce(1).write \
            .mode("overwrite").option("header", "true") \
            .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 :

```
def timed(func):
    """Decorateur pour mesurer le temps d'execution."""
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"[TIMER] {func.__name__}: {end - start:.2f}
           secondes")
        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
```

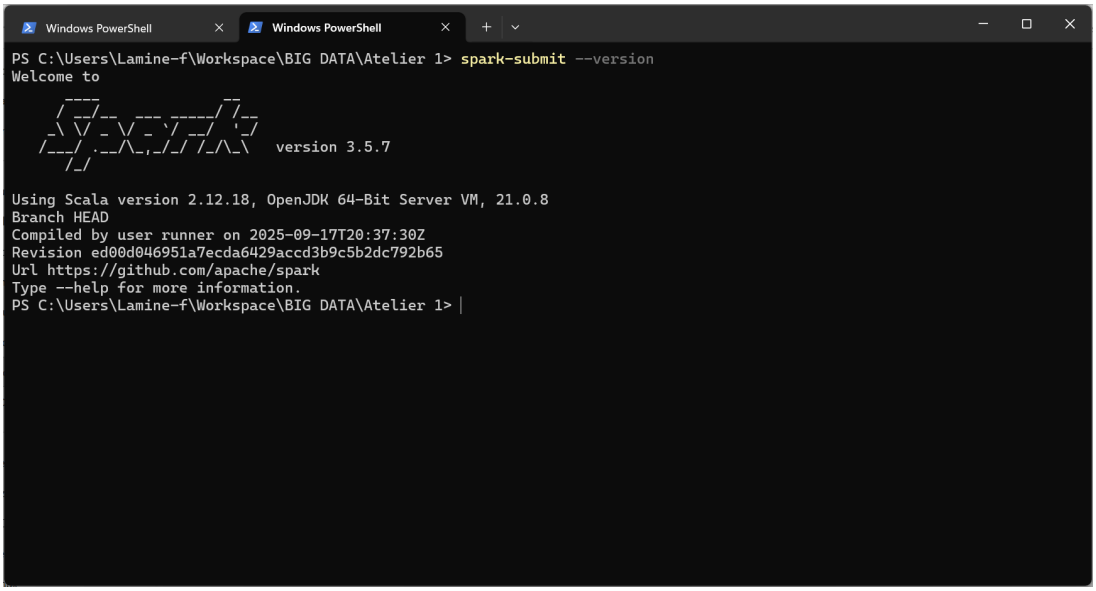


FIGURE 1 – Vérification de la version de Spark

3.3 Exécution locale

Pour lancer le programme en local :

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

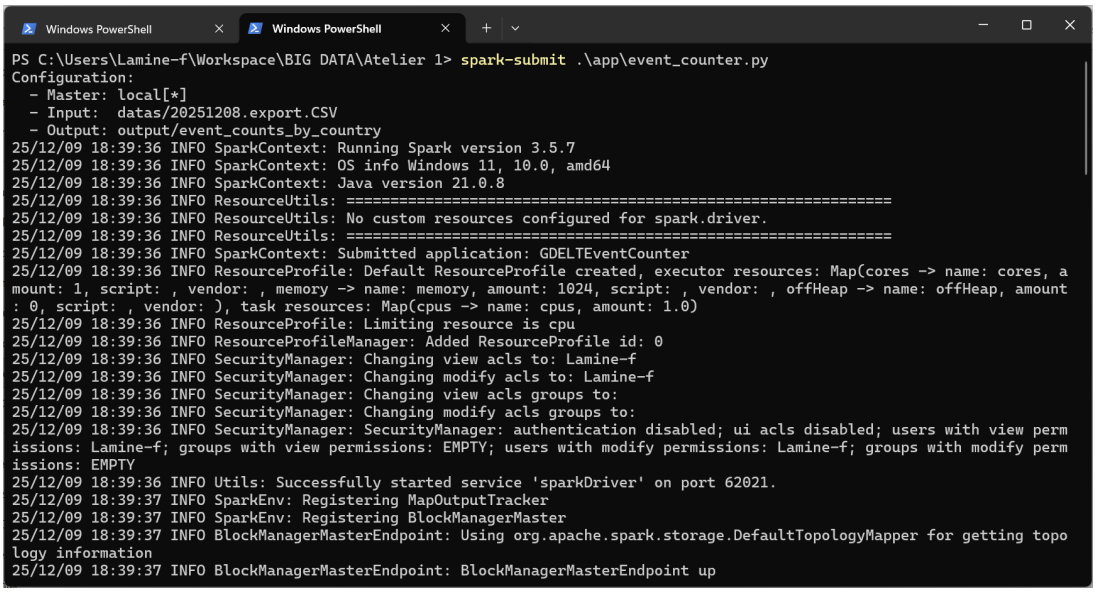
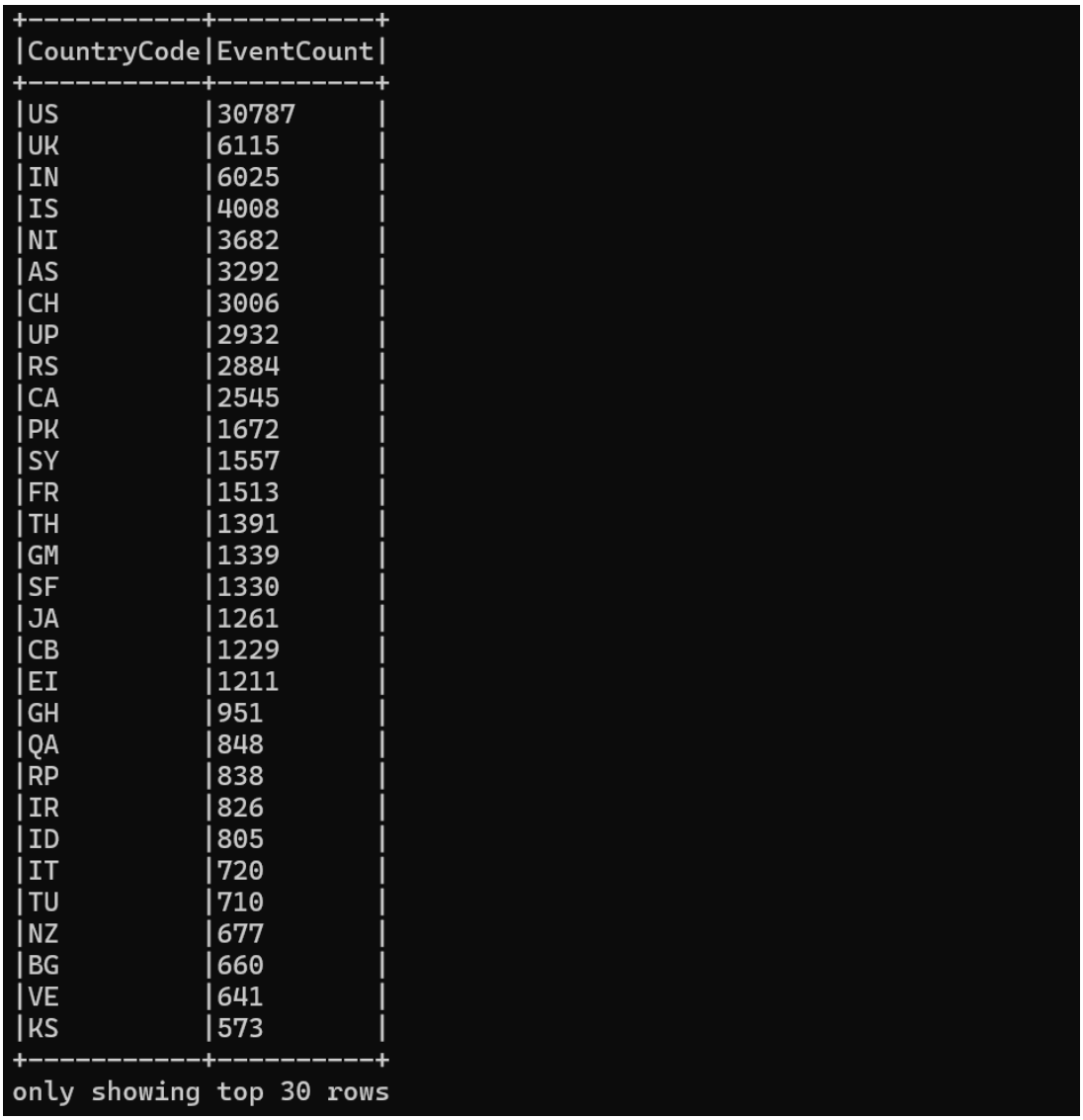


FIGURE 2 – Exécution locale sur le fichier 20251208.export.CSV



CountryCode	EventCount
US	30787
UK	6115
IN	6025
IS	4008
NI	3682
AS	3292
CH	3006
UP	2932
RS	2884
CA	2545
PK	1672
SY	1557
FR	1513
TH	1391
GM	1339
SF	1330
JA	1261
CB	1229
EI	1211
GH	951
QA	848
RP	838
IR	826
ID	805
IT	720
TU	710
NZ	677
BG	660
VE	641
KS	573

only showing top 30 rows

FIGURE 3 – Résultats de l'exécution locale

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

```
services:
  spark-master:
    image: apache/spark:3.5.0
```

```
    container_name: spark-master
    ports:
      - "8080:8080"  # Web UI Master
      - "7077:7077"  # Port Spark Master
      - "4040:4040"  # Application UI
    volumes:
      - ./datas:/data
      - ./app:/app
      - ./output:/output

  spark-worker-1:
    image: apache/spark:3.5.0
    container_name: spark-worker-1
    environment:
      - SPARK_WORKER_MEMORY=2G
      - SPARK_WORKER_CORES=2
    ports:
      - "8081:8081"

  spark-worker-2:
    image: apache/spark:3.5.0
    container_name: spark-worker-2
    environment:
      - SPARK_WORKER_MEMORY=2G
      - SPARK_WORKER_CORES=2
    ports:
      - "8082:8081"

networks:
  spark-net:
    driver: bridge
```

4.3 Démarrage du cluster

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

# Verifier les conteneurs
docker ps
```

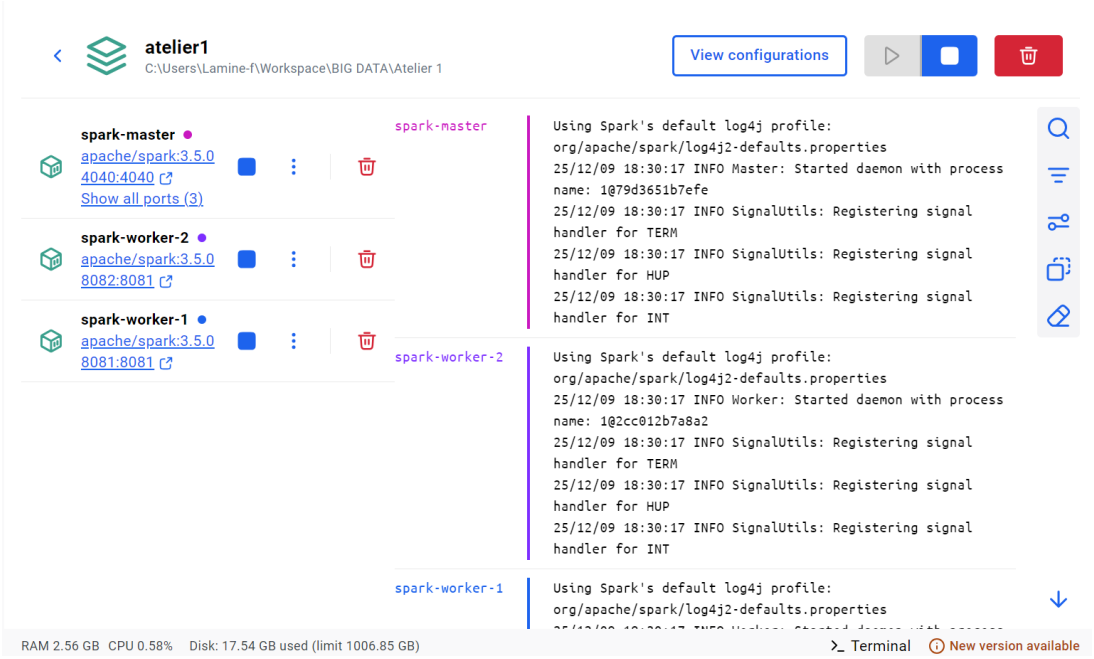


FIGURE 4 – Docker Desktop - Conteneurs du cluster Spark

4.4 Interfaces Web

4.4.1 Spark Master UI (port 8080)

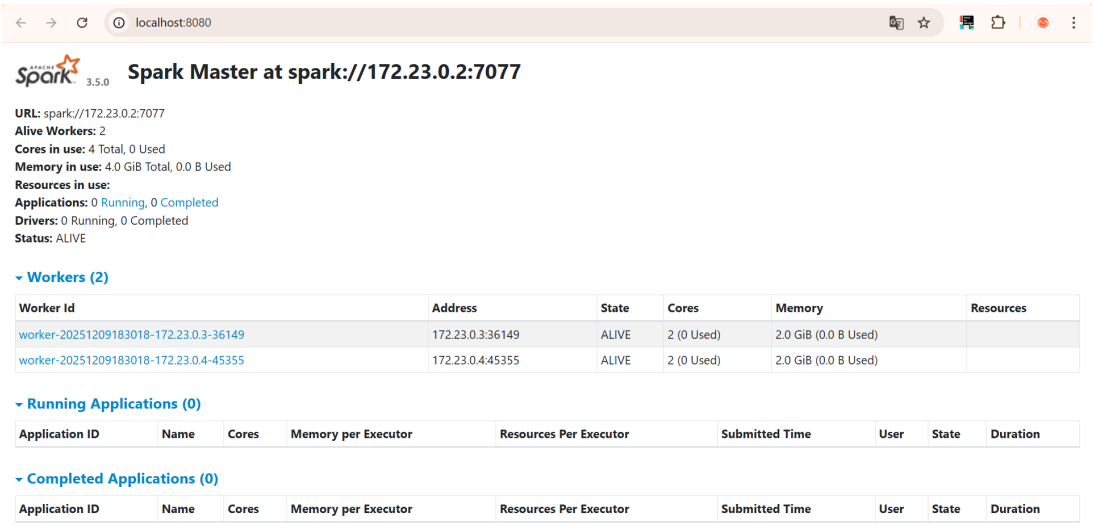


FIGURE 5 – Interface Web du Spark Master

4.4.2 Spark Worker 1 UI (port 8081)

FIGURE 6 – Interface Web du Worker 1

4.4.3 Spark Worker 2 UI (port 8082)

FIGURE 7 – 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

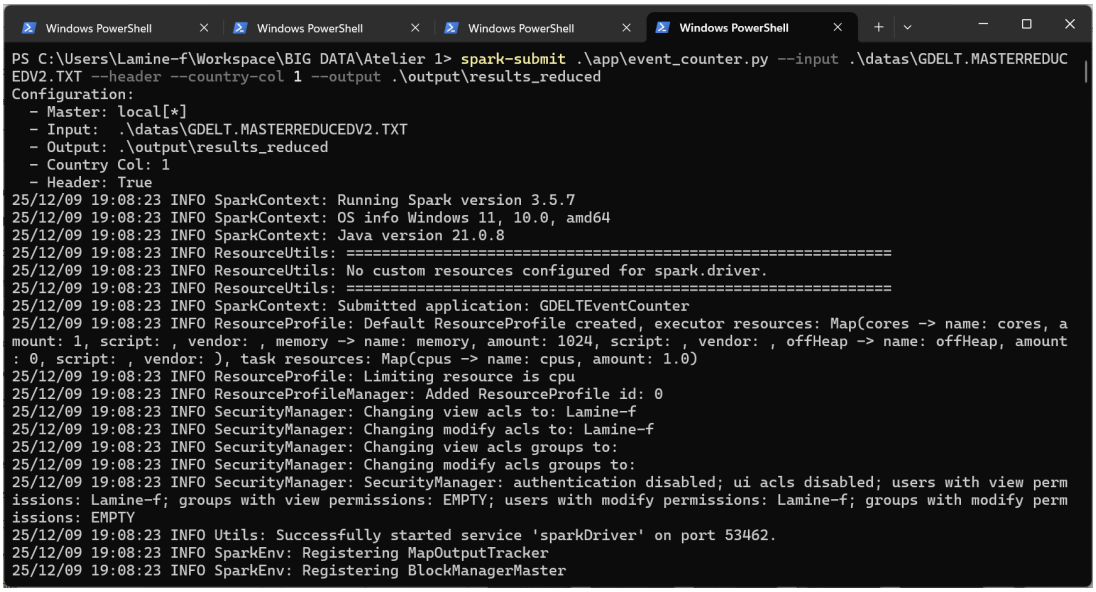


FIGURE 8 – Lancement du script en local - Gros fichier

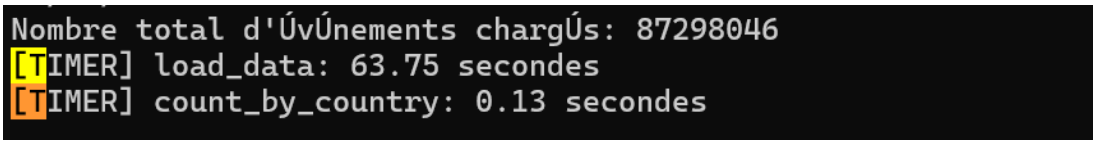


FIGURE 9 – Temps de chargement et comptage - Local

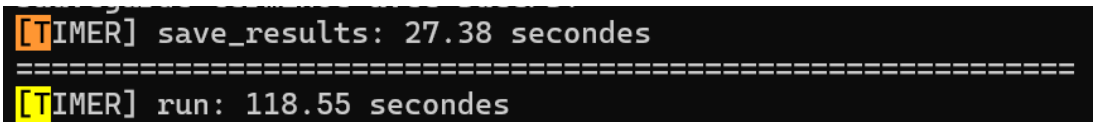


FIGURE 10 – Temps de sauvegarde et total - Local

5.3 Exécution sur le cluster

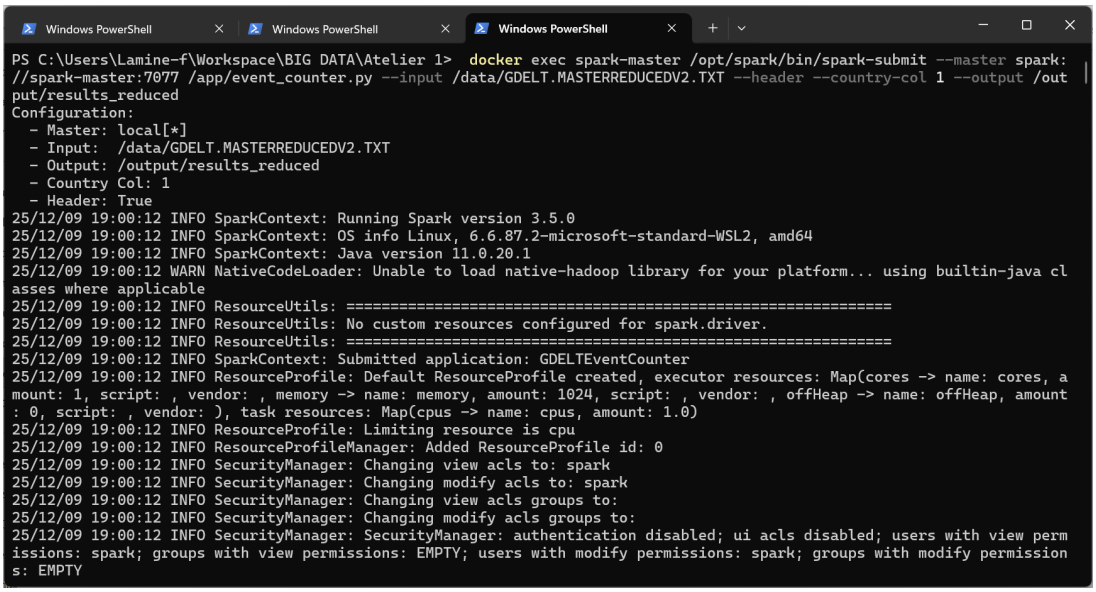


FIGURE 11 – Lancement du script sur le cluster - Gros fichier

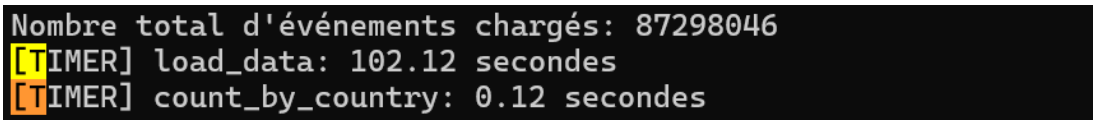


FIGURE 12 – Temps de chargement et comptage - Cluster

FIGURE 13 – 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