Document technico-fonctionnel — Airflow × PySpark (Partie logs)

1) Objectif

Traiter automatiquement un fichier de logs au format :

timestamp @url

Pour chaque ligne, on doit :

- extraire Breakdown_Type (le nombre après slab),

- extraire Breakdown_Level (le nombre après 256),

- trier par timestamp (et à égalité par Breakdown_Type),

- écrire les résultats propres dans output/ et les lignes invalides dans bad_lines/.

Bonus : si les données d'entrée ne sont pas triées. Le pipeline doit gérer ce cas.

2) Entrées / Sorties

Entrée principale : data/break_down.txt (non trié à priori).

Sorties (générées par le job) :

- output/final.csv : résumé complet par type de panne,

- output/exo_output.csv : format réduit (type, niveaux),

- output/runs.csv : détails des séquences de pannes,

- output/exo_runs_output.csv : runs globaux successifs,

- bad_lines/* : lignes rejetées (format invalide).

Règles de parsing :

- Breakdown_Type = nombre après slab/

- Breakdown_Level = nombre après 256/

3) Architecture technique

- Airflow : Webserver + Scheduler dans des conteneurs Docker.

- Image Docker : apache/airflow:2.9.1-python3.11, Java 17, PySpark 3.5.1 offline via wheels.

- Volumes : dags/, jobs/, data/, output/, bad_lines/ montés dans les conteneurs.

- Port UI : http://localhost:8089

- Variables d'env utiles :

JAVA_HOME=/usr/lib/jvm/java-17-openjdk-amd64

PYSPARK_PYTHON=python

Le repo ne fixe pas de limites CPU/RAM/GPU. Spark utilise les valeurs par défaut du conteneur.

4) Qu'est-ce qu'un BashOperator ?

Dans Airflow, un BashOperator exécute une commande ou un script Bash (shell Unix). Utile pour lancer des commandes simples ou des scripts externes.

Dans ce projet, ils appellent python jobs/breakdown_job.py avec différents arguments (--stage read, --stage sort, etc.) pour les étapes du traitement.

5) DAG Airflow (réel)

DAG id: breakdown_pipeline.

Chaînage: validate_input → fix_pyspark_perms → Read → SortData → Compute → SaveResult.

Tâches :

1. validate_input (PythonOperator) : vérifie la présence/lecture du fichier. Si manquant → échec.

2. fix_pyspark_perms (BashOperator): crée/autorise stage/, output/, bad_lines/.

3. Read (BashOperator) : lit le brut, isole les lignes invalides.

4. SortData (BashOperator): trie par timestamp.

5. Compute (BashOperator): extrait Breakdown_Type et Breakdown_Level, calcule les séquences.

6. SaveResult (BashOperator): écrit les CSV finaux.

Planification: Manuel (Trigger DAG)

6) Exploitation

Pour lancer:

docker compose up airflow-init

docker compose up -d

Pour arrêter:

docker compose down

Redémarrer:

docker compose down && docker compose up -d

Déclencher un run :

UI : bouton Trigger DAG sur breakdown_pipeline

CLI : airflow dags trigger breakdown_pipeline

7) Autre workflow : arrêt propre

S'il y'a un autre workflow pour éviter des problèmes de confusion

UI :

- Pause le DAG

- Mark Failed ou Clear sur les tâches en cours

CLI :

airflow dags pause <dag_id>

airflow dags list-runs -d <dag_id>

airflow tasks state -d <dag_id> <task_id> <execution_date> failed

airflow tasks clear -d <dag_id> -s <start_date> -e <end_date> --yes

8) Gestion des erreurs

- Fichier manquant : échec

- Lignes invalides : bad_lines/

- Droits : fix_pyspark_perms

- Tâche en erreur : consulter logs

9) Commandes utiles

docker compose up airflow-init

docker compose up -d

docker compose logs -f

docker compose exec airflow-webserver bash

airflow dags list

airflow tasks list breakdown_pipeline

airflow dags trigger breakdown_pipeline

airflow dags pause breakdown_pipeline

airflow dags unpause breakdown_pipeline

10) Points à adapter

- Ajouter options Spark mémoire
- Ajuster ressources Docker
- Archiver les anciennes sorties