DATA INTEGRATION
DATA LAKE

Construction Data Lake Data Warehouse

Kévin HEUGAS

Valentin MASSONNIERE

DE3 M1 Data Engineering & IA

SOMMAIRE

Objectif : Mettre en place une architecture de données complète en consommant les flux Kafka/ksqlDB (du TP 2) pour alimenter un Data Lake local et un Data Warehouse (MySQL).

I. Phase 1 : Design et Mise en Place

1. Architecture du Data Lake

○ 1.1. Définition de la structure des dossiers.

La structure du projet va se composer de la façon suivante :

- ***application/***
  - ***kafka_producer_transaction.py***
    - **Contenu :** Le script du TP2 à modifier pour le TP3.
    - **Rôle :** Augmenter le paramètre *NUM_MESSAGES* pour envoyer un grand volume de transactions brutes dans le topic *transaction_log* afin de simuler une charge de travail réaliste.
  - ***consumer_datalake.py***
    - **Contenu :** Un pipeline **Apache Beam**.
    - **Rôle :**
      - Lit les messages depuis tous les topics de ksqlDB (Streams *et* Tables).
      - Analyse le timestamp du message pour déterminer la date (*year*, *month*, *day*).
      - Analyse le nom du topic pour déterminer le chemin (ex: *streams/transactions_flat*).
      - Écrit les messages sous forme de fichiers (ex: JSON) dans le dossier *data_lake/* en respectant la structure de partitionnement.
  - ***consumer_datawarehouse.py***
    - **Contenu :** Un pipeline Apache Beam.
    - **Rôle :**
      - Lit les messages depuis les topics des Tables ksqlDB **uniquement** (ex: *TOTAL_DEPENSE_PAR_USER_TYPE*, *TRANSACTION_LIFECYCLE*).
      - Se connecte à la base de données MySQL *tp_data_warehouse*.
      - Traduit chaque message en requête SQL *INSERT ... ON DUPLICATE KEY UPDATE ...* pour s'assurer que les données du Data Warehouse reflètent l'état le plus récent de la table ksqlDB.

- ***orchestrator.py***
  - **Contenu :** Un script Python utilisant les bibliothèques *schedule* et *subprocess*.
  - **Rôle :** Il exécute une boucle *schedule.every(10).minutes.do(...)* qui, toutes les 10 minutes, lance les jobs Beam définis dans les scripts "consumer". Il transforme le pipeline de streaming en un pipeline "batch" (par lots).

- ***requirements.txt***
  - **Contenu :** Une liste des dépendances Python.
  - **Rôle :** Permet d'installer tout ce qui est nécessaire avec une seule commande (*pip install -r requirements.txt*). Il doit contenir :
    - kafka-python-ng
    - mysql-connector-python
    - apache-beam
    - schedule

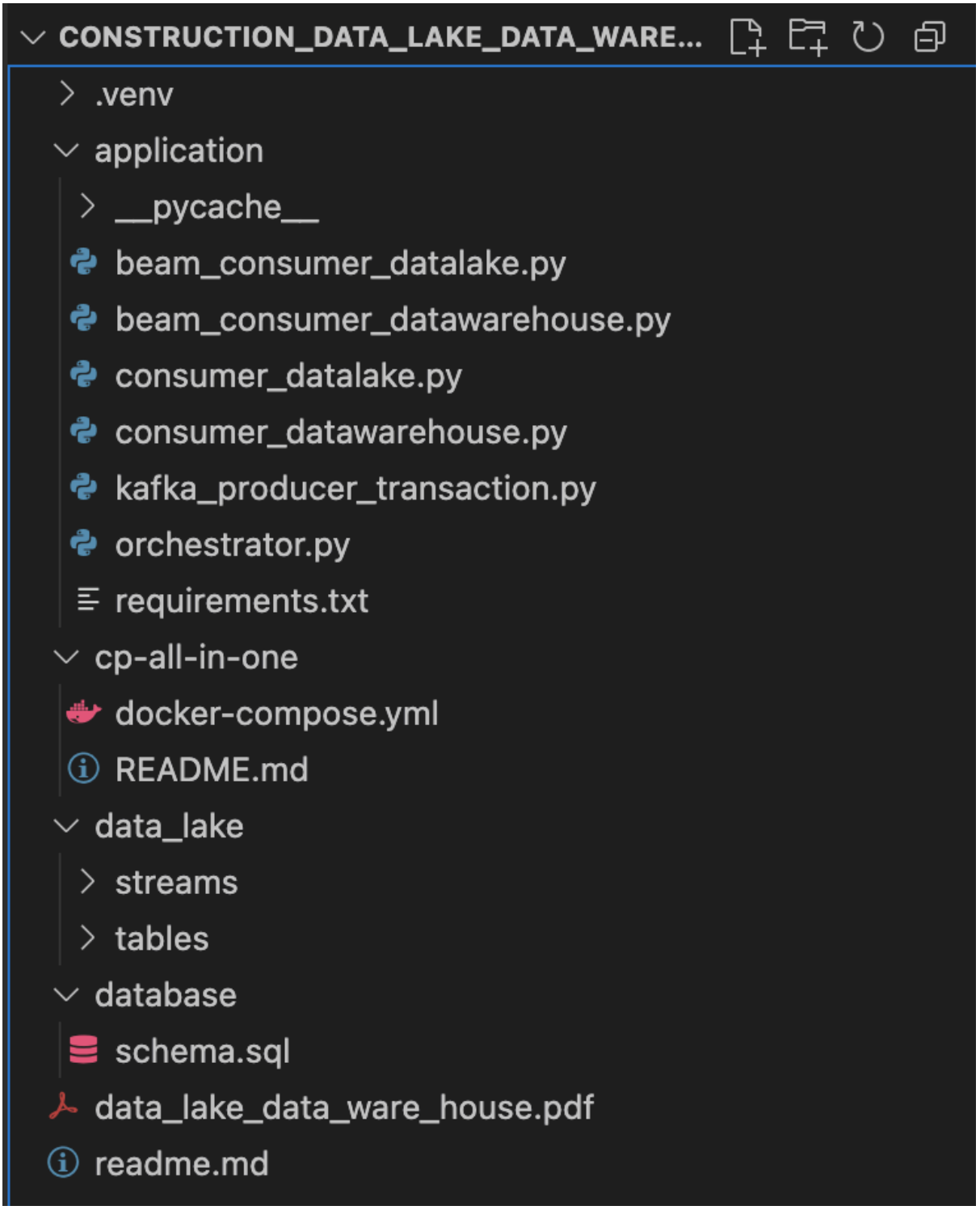
- ***data_lake/***
  C'est la **destination** des données brutes, le Data Lake local.

  - ***streams/*** et ***tables/***
    - **Contenu :** Ces dossiers seront vides au début.
    - **Rôle :** Ils seront peuplés par le script *consumer_datalake.py* au fil du temps. Les sous-dossiers (ex: *transactions_blacklisted/year=...*) seront créés dynamiquement par le script lors de l'écriture des données.

- ***database/***
  C'est le dossier qui définit la structure du Data Warehouse.

  - ***schema.sql***
    - **Contenu :** Les requêtes *CREATE TABLE* en SQL pour la base MySQL.
    - **Rôle :** Sert de documentation et de script d'installation pour le Data Warehouse, avec les schémas pour :
      - *total_depense_par_user_type*
      - *transaction_lifecycle*
      - *total_par_type_5min_sliding*
      - *user_permissions* (pour la partie Gouvernance)

- ***.gitignore***
  - **Contenu :** Fichier de configuration pour Git.
  - **Rôle :** Indique à Git les fichiers et dossiers à ignorer.


CONSTRUCTION_DATA_LAKE_DATA_WARE...
.venv
application
__pycache__
beam_consumer_datalake.py
beam_consumer_datawarehouse.py
consumer_datalake.py
consumer_datawarehouse.py
kafka_producer_transaction.py
orchestrator.py
requirements.txt
cp-all-in-one
docker-compose.yml
README.md
data_lake
streams
tables
database
schema.sql
data_lake_data_ware_house.pdf
readme.md

○ 1.2. Stratégie de partitionnement (par topic et par date).

La stratégie de partitionnement est divisée par type de source, les streams et les tables, et la partition est en fonction de la date. Il s'agit du format "ruche" / hive.

- *streams/* pour les flux d'événements.
- *tables/* pour les états agrégés.
- Nom du topic : selon le topic Kafka d'où proviennent les données.
- Partition par date : *year=YYYY/month=MM/day=DD* pour mieux visualiser les partitions.

○ 1.3. Choix des modes de stockage (Append vs. Overwrite) pour les Streams et les Tables ksqlDB.

Les choix des modes de stockage pour les streams et les tables ksqlDB se portent sur les suivants :

- Pour les streams ksqlDB : Mode Append (ajout)
  - **Quels streams** : *transactions_full*, *transactions_flat*, *transactions_processed*, *transactions_blacklisted*, *transactions_completed*, etc.
  - **Pourquoi** : Un stream est un enregistrement d'événements, c'est un historique. Les anciennes données doivent être gardées et les nouvelles données sont ajoutées à l'historique. C'est immuable.
  - **Consumer** : Le consumer lira un lot de message et les ajoutera sous forme de nouveaux fichiers dans le dossier de la journée, en .json.

- Pour les tables ksqlDB : Mode Overwrite (écrasement)
  - **Quelles tables :** *total_par_transaction_type*, *product_purchase_counts*, *transaction_lifecycle*, *total_depense_par_user_type*, *total_par_type_5min_sliding*.
  - **Pourquoi :** Une table ksqlDB n'est pas un historique, c'est un état actuel.
  - **Consumer :** Lorsque le job s'exécute (toutes les 10 minutes), il doit supprimer le fichier *snapshot_complet.json* précédent et le remplacer par un nouveau fichier contenant l'état le plus récent de la table.

2. Architecture du Data Warehouse

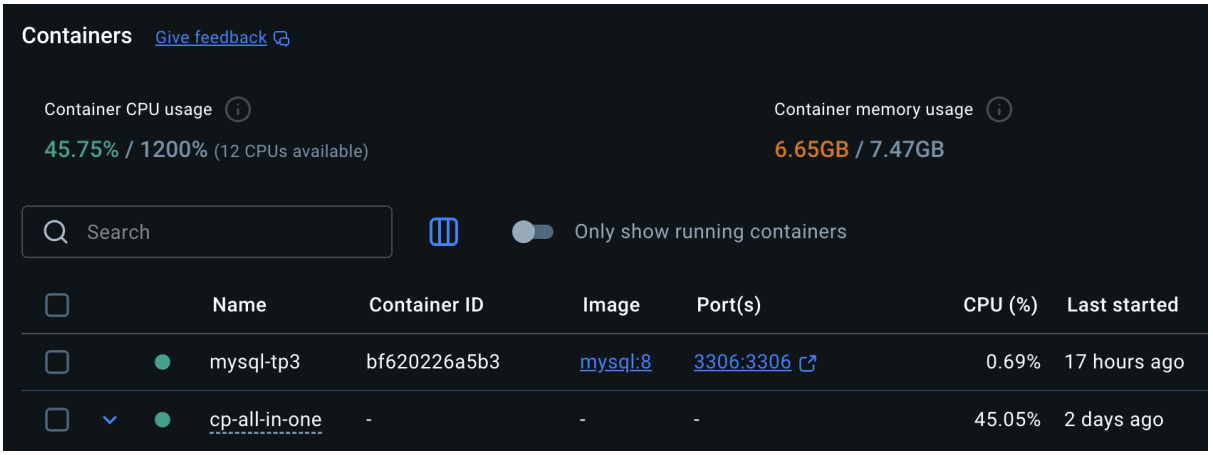
○ 2.1. Lancement du conteneur MySQL (Docker).

Pour le lancement du conteneur MySQL, il suffit de lancer un terminal et exécuter la commande suivante :

```
kzm@KZMs-MacBook-Pro application % docker run -d \
    --name mysql-tp3 \
    -p 3306:3306 \
    -e MYSQL_ROOT_PASSWORD=admin123 \
    -e MYSQL_DATABASE=tp3 \
    mysql:8
```

Je modifie le mot de passe et je lance.

Le conteneur est bien installé et tourne dans Docker.



○ 2.2. Modélisation des données : création d'un diagramme Entité-Association simple pour les tables issues de ksqlDB.

Le projet TP3 n'est pas un entrepôt relationnel complexe, c'est un ensemble de vues matérialisées des tables ksqlDB.

De ce fait, il n'y a pas de clés étrangères ou de relations complexes entre elles, ce sont des tables de fait indépendantes. Le diagramme Entité - Association représente 4 entités non connectées :

Entité 1 : *total_depense_par_user_type*

- Attributs : *user_id* (PK), *transaction_type* (PK), *total_depense_usd*

Entité 2 : *transaction_lifecycle*

- Attributs : *transaction_id* (PK), *status_history*, *last_update_time*

Entité 3 : *total_par_type_5min_sliding*

- Attributs : *window_start_time* (PK), *transaction_type* (PK), *total_amount_5min*

Entité 4 : *user_permissions* (Pour la partie Gouvernance)

- Attributs : *user_id* (PK), *folder_path* (PK), *permission*

- 2.3. Définition des schémas SQL (*CREATE TABLE*) avec clés primaires.

database > schema.sql

```sql
USE tp3;

-- ------------------------------------------------------------------------
-- Montant dépensé par utilisateur et type de transaction
-- ------------------------------------------------------------------------
CREATE TABLE IF NOT EXISTS total_depense_par_user_type (
    user_id VARCHAR(255) NOT NULL,
    transaction_type VARCHAR(255) NOT NULL,
    total_depense_usd DECIMAL(15, 2),
    last_updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id, transaction_type)
);


-- ------------------------------------------------------------------------
-- Cycle de vie (statuts) d'une transaction
-- ------------------------------------------------------------------------
CREATE TABLE IF NOT EXISTS transaction_lifecycle (
    transaction_id VARCHAR(255) NOT NULL,
    status_history JSON,
    last_update_time TIMESTAMP,
    last_updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY (transaction_id)
);


-- ------------------------------------------------------------------------
-- Total par type sur une fenêtre glissante de 5min
-- ------------------------------------------------------------------------
CREATE TABLE IF NOT EXISTS total_par_type_5min_sliding (
    window_start_time TIMESTAMP NOT NULL,
    transaction_type VARCHAR(255) NOT NULL,
    total_amount_5min DECIMAL(15, 2),
    last_updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY (window_start_time, transaction_type)
);

```

3. Gouvernance

○ 3.1. Modélisation et création de la table user_permissions dans MySQL.

```sql
-- ------------------------------------------------------------------------
-- Modélisation et création de la table user_permissions dans MySQL.
-- ------------------------------------------------------------------------
CREATE TABLE IF NOT EXISTS user_permissions (
    user_id VARCHAR(255) NOT NULL,
    folder_path VARCHAR(512) NOT NULL,
    permission ENUM('read', 'write', 'none') DEFAULT 'none',
    PRIMARY KEY (user_id, folder_path)
);
```

II. Phase 2 : Développement des Consumers Python

Cette phase consiste à écrire les applications Python capables de consommer les données en temps réel depuis Kafka et de les acheminer vers leurs destinations respectives : le Data Lake et le Data Warehouse.

1. Installation des prérequis

Le fichier install requirements.txt est créé avec le contenu suivant :

- **kafka-python-ng** : Le client Kafka pour Python, utilisé pour se connecter aux topics et consommer les messages.
- **mysql-connector-python** : Le pilote officiel pour connecter les scripts Python à la base de données MySQL.
- **apache-beam** : Le modèle de programmation unifié qui sera utilisé en Phase 3 pour définir les pipelines de traitement de données (ETL).
- **schedule** : L'orchestrateur de tâches léger, utilisé en Phase 3 pour déclencher les pipelines à intervalle régulier.

Les packages sont installés avec la commande suivante

- pip install -r application/requirements.txt

2. Consumer 1 : Alimentation du Data Lake avec le script Python consumer_datalake.py

Ce consumer va capturer l'intégralité des données sortant de ksqlDB, donc les streams et les tables, et va les stocker de manière brute dans le Data Lake.

La logique de ce script est la suivante :

- **Connexion Multi-Topic** : Le consumer s'abonne à une liste exhaustive de tous les topics générés par ksqlDB (ex: TRANSACTIONS_PROCESSED, TRANSACTION_LIFECYCLE, etc.).

- **Partitionnement Dynamique** : Pour chaque message reçu, le script extrait le timestamp du message Kafka pour construire dynamiquement le chemin de stockage. Il respecte la structure "Hive" définie en Phase 1 (ex: ../data_lake/streams/TOPIC_NAME/year=YYYY/month=MM/day=DD/).

- **Logique d'écriture (Append)** : Conformément au design du Data Lake (mode "append"), chaque message est traité comme un événement immuable. Il est écrit dans un fichier JSON unique (nommé d'après son *offset* Kafka) pour préserver l'historique complet, sans jamais écraser de données.

3. Consumer 2 : Alimentation du Data Warehouse avec le script Python consumer_datawarehouse.py

Le second consumer, consumer_datawarehouse.py, est plus ciblé. Son rôle est de maintenir une réplique à jour des tables d'agrégation ksqlDB (les vues matérialisées) dans la base de données relationnelle MySQL.

Sa logique de fonctionnement est distincte de celle du Data Lake :

- **Connexion Ciblée** : Le consumer s'abonne uniquement aux topics correspondant aux tables ksqlDB (ex: TOTAL_DEPENSE_PAR_USER_TYPE, TRANSACTION_LIFECYCLE), car le Data Warehouse ne stocke que les résultats finaux agrégés.

- **Lecture Clé-Valeur** : Ce script doit lire à la fois la clé (Key) et la valeur (Value) du message Kafka. La clé (ex: user_id, transaction_id) est utilisée pour la clé primaire de la table MySQL.

- **Logique d'écriture (Upsert)** : Pour chaque message, le script exécute une requête INSERT ... ON DUPLICATE KEY UPDATE .... Cette commande "UPSERT" insère une nouvelle ligne si la clé n'existe pas, ou met à jour la ligne existante si elle est déjà présente. Cela garantit que le Data Warehouse reflète l'état le plus récent des agrégats ksqlDB.

- **Transformation de Données** : Une logique de conversion a été implémentée pour gérer les types de données, notamment le "parsing" (analyse) des timestamps au format ISO 8601 (ex: ...Z) venant de ksqlDB en objets datetime compatibles avec les colonnes TIMESTAMP de MySQL.

III. Phase 3 : Orchestration et Tests

L'objectif de cette phase est de faire évoluer notre architecture d'un modèle "streaming" (où les consumers tournent en continu, comme en Phase 2) vers un modèle "batch" orchestré. Ce modèle, plus traditionnel pour l'alimentation de Data Lakes et Data Warehouses, consiste à exécuter des jobs à intervalle régulier pour traiter les données par lots.

1. Génération de Données

Pour valider cette nouvelle architecture batch, il est nécessaire de simuler un volume de données significatif en attente de traitement.

1.1 Modification du Producer

Le script kafka_producer_transaction.py a été modifié pour générer une charge de travail plus importante. Les paramètres de "threading" et de nombre de messages ont été augmentés pour produire un total de **5 000 messages** (10 threads x 500 messages). Un court sleep de 10ms a été ajouté entre les envois pour ne pas saturer le broker.

1.2. Simulation du flux

Ce script a été exécuté **une seule fois**. Cela a eu pour effet de créer un **"backlog"** (une file d'attente) de 5 000 messages dans le topic transaction_log, qui ont ensuite été traités par ksqlDB et mis en attente dans les topics de sortie (ex: TRANSACTIONS_PROCESSED, TOTAL_DEPENSE_PAR_USER_TYPE, etc.).

2. Mise en place de l'orchestration "Batch"

2.1. Adaptation en Pipelines Apache Beam

Pour conserver une trace claire des deux approches, les scripts consumer_datalake.py et consumer_datawarehouse.py de la Phase 2 ont été conservés intacts.

Deux **nouveaux** fichiers ont été créés pour implémenter la logique de la Phase 3 :

- beam_consumer_datalake.py
- beam_consumer_datawarehouse.py

La logique interne de ces scripts a été modifiée :

- **Fin de la boucle infinie :** La boucle while True a été supprimée.
- **Consommation par lot :** Les scripts utilisent désormais consumer.poll() avec un *timeout* (ex: 5 secondes) pour récupérer un lot de tous les messages disponibles sur les topics depuis la dernière exécution. Un group_id Kafka est utilisé pour que Kafka gère les "offsets" (marque-pages) et ne renvoie que les nouveaux messages à chaque cycle.
- **Intégration Beam :** Ce lot de messages est ensuite passé à un pipeline **Apache Beam** (beam.Create(message_list)).
- **Traitement (DoFn) :** La logique d'écriture (création de fichiers pour le Data Lake, UPSERT pour le Data Warehouse) a été déplacée dans des classes DoFn (ex: WriteToFile, WriteToMySQL).
- **Fin du script :** Une fois le pipeline Beam exécuté, le script se termine. Chaque script expose une fonction run_pipeline() qui exécute l'ensemble de ce processus.

2.2. Création du Script Maître (orchestrator.py)

Le script orchestrator.py sert de chef d'orchestre. Il utilise la bibliothèque schedule pour automatiser l'exécution de ces nouveaux pipelines Beam.

Son fonctionnement est le suivant :

- **Importation :** Le script importe les fonctions run_pipeline() de beam_consumer_datalake.py et beam_consumer_datawarehouse.py.
- **Planification :** Il définit une tâche récurrente : schedule.every(10).minutes.do(run_all_jobs).
- **Exécution :** La fonction run_all_jobs appelle séquentiellement les deux pipelines (d'abord le Data Lake, puis le Data Warehouse).
- **Test initial :** Au démarrage, l'orchestrateur a été configuré pour lancer les jobs une première fois immédiatement afin de traiter le backlog de 5 000 messages.
- **Mode "schedule" :** Une fois le premier cycle terminé, le script entre dans sa boucle principale, attendant 10 minutes avant de relancer le cycle.

Les logs ont confirmé le succès de cette approche : le premier cycle a traité les 5 000 messages en attente, et le script s'est ensuite mis en attente, prêt pour le prochain intervalle de 10 minutes.

```
2025-10-30 21:59:35,496 - [Orchestrator] - INFO - pipeline done! (job='job-002[job]')
2025-10-30 21:59:35,497 - [Orchestrator] - INFO - pipeline completed job-002[job]
2025-10-30 21:59:35,497 - [Orchestrator] - INFO - terminating job-002[job]
2025-10-30 21:59:35,498 - [Orchestrator] - INFO - Job state changed to DONE
2025-10-30 21:59:35,498 - [Orchestrator] - INFO - No more requests from control plane
2025-10-30 21:59:35,498 - [Orchestrator] - INFO - SDK Harness waiting for in-flight requests to complete
2025-10-30 21:59:35,498 - [Orchestrator] - INFO - Closing all cached grpc data channels.
2025-10-30 21:59:35,498 - [Orchestrator] - INFO - Closing all cached gRPC state handlers.
2025-10-30 21:59:35,499 - [Orchestrator] - INFO - Done consuming work.
2025-10-30 21:59:35,499 - [Orchestrator] - INFO - Pipeline BATCH Data Warehouse terminé.
2025-10-30 21:59:35,502 - [Orchestrator] - INFO - --- Fin du job Data Warehouse ---
2025-10-30 21:59:35,502 - [Orchestrator] - INFO - Cycle terminé. Prochaine exécution dans 10 minutes.
2025-10-30 21:59:35,502 - [Orchestrator] - INFO - Premier cycle terminé. Passage en mode schedule (toutes les 10 min).
^C2025-10-30 22:00:36,270 - [Orchestrator] - INFO - Arrêt de l'orchestrateur.
```

IV. Phase 4 : Gestion et Évolution

1. Création du Flux (ksqlDB)

La première étape consiste à créer le nouveau flux de données dans ksqlDB, ce qui génère un nouveau topic Kafka. *Exemple :* CREATE STREAM TRANSACTIONS_HORS_UE AS SELECT ...;

2. Mise à jour du Consumer (Python)

Pour que ce nouveau topic soit aspiré dans le Data Lake, **une seule modification de code** est nécessaire.

Il faut modifier le fichier application/beam_consumer_datalake.py et ajouter le nom du nouveau topic au dictionnaire TOPIC_CONFIG.

3. Réutilisation du Travail (Accélération)

C'est là que l'architecture montre sa force : **99% du travail est réutilisé**.

- **Aucune nouvelle logique de pipeline** n'est nécessaire. L'orchestrateur (orchestrator.py) et le pipeline Beam (beam_consumer_datalake.py) sont réutilisés tels quels.

- **Le partitionnement est automatique.** La fonction get_partition_path est réutilisée. Elle lira le nom du nouveau topic (TRANSACTIONS_HORS_UE) et le timestamp de ses messages pour créer automatiquement la bonne arborescence de dossiers : data_lake/streams/TRANSACTIONS_HORS_UE/year=2025/month=10/day=30/

- **L'écriture est automatique.** La classe WriteToFile (le DoFn de Beam) est réutilisée. Elle écrira les nouveaux messages dans le bon dossier sans aucune modification.

4. Sécurité, mécanisme de suppression des données historiques ?

Un mécanisme de suppression des données (ou "purge") est essentiel pour la gouvernance, afin de réduire les coûts de stockage et de garantir la conformité réglementaire (comme le RGPD, qui impose une durée de conservation limitée des données).

La stratégie de suppression diffère entre le Data Lake et le Data Warehouse

1. Suppression dans le Data Lake (Fichiers)

La suppression dans le Data Lake est effectuée au niveau du dossier, ce qui est très efficace.

- **Stratégie** : Un script automatisé s'exécute périodiquement (ex: une fois par mois).
- **Mécanisme** : Le script scanne les dossiers et supprime récursivement les partitions de date qui sont plus anciennes que la politique de rétention (ex: 3 ans).
- **Avantage** : Cette méthode est extrêmement rapide car elle supprime des dossiers entiers au lieu d'inspecter des millions de fichiers individuels.

2. Suppression dans le Data Warehouse (MySQL)

La suppression dans le Data Warehouse est effectuée au niveau de la ligne à l'aide de requêtes SQL.

- **Stratégie** : Une tâche planifiée (ex: une procédure stockée MySQL) s'exécute périodiquement (ex: une fois par semaine).
- **Mécanisme** : Le script exécute des requêtes DELETE basées sur les colonnes de timestamp. Par exemple :
    - Pour la table transaction_lifecycle, il supprime les enregistrements où last_update_time est antérieur à 3 ans.
    - Pour les tables éphémères comme total_par_type_5min_sliding, il supprime les enregistrements de plus de 7 jours.