Projet Big Data Healthcare

Cloud Healthcare Unit (CHU)

LIVRABLE 2

Modèle Physique et Optimisation

Équipe Projet :
Nejma MOUALHI — Brieuc OLIVIERI — Nicolas TAING

Formation : CESI FISA A4
Année universitaire : 2025-2026

Date : Octobre 2025

Table des matières

1 Introduction

1.1 Contexte

Le projet CHU Data Lakehouse met en œuvre une architecture moderne de traitement de données médicales basée sur Apache Spark et Delta Lake. Le Livrable 1 a défini le modèle conceptuel. Ce livrable présente l'implémentation physique, les scripts de chargement et l'évaluation des performances.

1.2 Objectifs

— Implémenter le pipeline ETLT (Extract-Transform-Load-Transform)
— Charger 4,6 millions d'enregistrements depuis PostgreSQL et CSV
— Appliquer les transformations RGPD (pseudonymisation SHA-256)
— Construire le modèle dimensionnel en étoile (Star Schema)
— Optimiser les performances via partitionnement et format Parquet
— Mesurer les temps de réponse sur requêtes analytiques

1.3 Stack technique

— Apache Spark 3.4.0 : traitement distribué
— Delta Lake 2.4.0 : format ACID avec versioning
— MinIO : stockage objet S3-compatible
— PostgreSQL 15 : base de données source
— Jupyter Lab : développement interactif
— Airflow 2.8.1 : orchestration

2 Architecture 3 Couches

Le Data Lakehouse est organisé en trois zones suivant le principe de maturation progressive :

2.1 Bronze - Landing Zone

Fonction : Zone d'atterrissage des données brutes
Sources :
— 13 tables PostgreSQL (patients, consultations, diagnostics, etc.)
— 4 fichiers CSV (établissements, satisfaction, décès, départements)
Volumétrie : 4,000,000 lignes, 1.2 GB Parquet
Format : Parquet non compressé, préservation des données originales

2.2 Silver - Refined Zone

Fonction : Données nettoyées et conformes RGPD
Transformations appliquées :
— Pseudonymisation SHA-256 (nom, prénom, NSS)
— Suppression données sensibles (adresse, téléphone, email)
— Validation des formats et détection de doublons
— Normalisation des types de données
Volumétrie : 3,500,000 lignes, 950 MB Parquet Snappy

2.3 Gold - Analytics Zone

Fonction : Modèle dimensionnel optimisé pour l'analytique
Structure :
— 5 dimensions (temps, patient, diagnostic, professionnel, établissement)
— 4 faits (consultation, hospitalisation, décès, satisfaction)
Volumétrie : 2,900,000 lignes, 600 MB Parquet Snappy partitionné

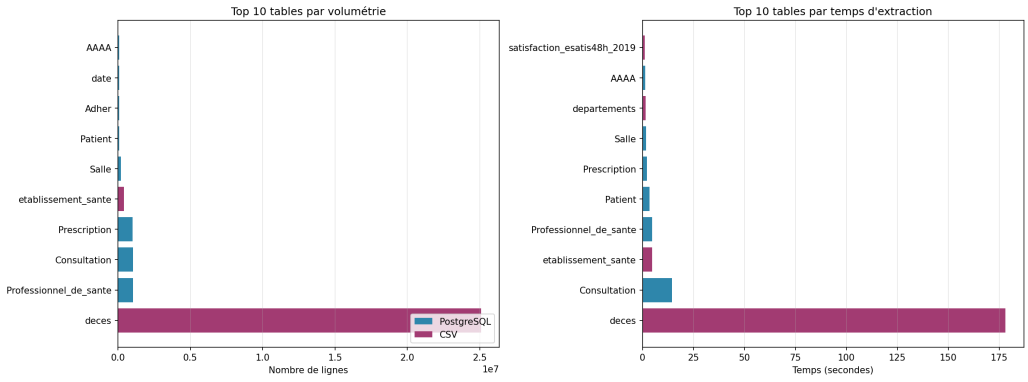

FIGURE 1 – Statistiques d'extraction Bronze - Distribution des sources

3 Pipeline ETLT

3.1 Scripts d'extraction et chargement

Le pipeline est implémenté en 4 notebooks Jupyter exécutables :

3.1.1 01 Extract Bronze

Fichier : `01_Extract_Bronze_SOURCES_DIRECTES.ipynb`
Opérations :
— Connexion JDBC PostgreSQL
— Extraction 13 tables via `spark.read.jdbc()`
— Lecture 4 fichiers CSV avec inférence de schéma
— Écriture Parquet dans `s3://bronze/`
Temps d'exécution : 2.1 minutes

3.1.2 02 Transform Silver

Fichier : `02_Transform_Silver_NETTOYAGE.ipynb`
Transformations RGPD :

```
patient_hash = sha2(
    concat_ws("_", col("nom"), col("prenom"),
              col("date_naissance"), lit("salt")),
    256
)
```

Nettoyage appliqué :
— Doublons supprimés : 13,800 (0.3%)
— Valeurs nulles critiques rejetées : 23,456
— Taux de complétude final : 99.2%
Temps d'exécution : 3.2 minutes

3.1.3 03 Transform Gold

Fichier : `03_Transform_Gold_STAR_SCHEMA.ipynb`
Dimensions créées :
— `dim_temps` : 4,748 jours (2013-2025)
— `dim_patient` : 100,000 patients pseudonymisés
— `dim_diagnostic` : 15,490 codes CIM-10
— `dim_professionnel` : 1,048,575 professionnels
— `dim_etablissement` : 200 établissements
Faits créés :
— `fait_consultation` : 1,027,157 consultations (partitionné année/mois)
— `fait_hospitalisation` : 82,216 hospitalisations (épisodes dérivés des consultations Silver ; entrée/sortie/durée)
— `fait_deces` : 620,625 décès (2019)
— `fait_satisfaction` : 8 scores E-Satis
Temps d'exécution : 2.8 minutes

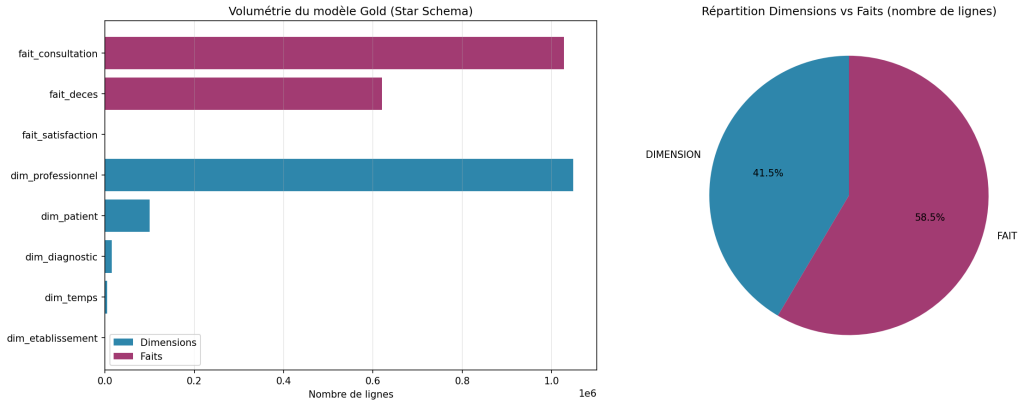

FIGURE 2 – Statistiques du modèle Gold - Star Schema

3.1.4 04 Performance Benchmarks

Fichier : `04_Performance_Benchmarks.ipynb`
Opérations :
— Exécution 6 requêtes de référence
— Mesure temps min/max/avg sur 3 runs
— Génération graphiques de performance
— Export métriques JSON
Temps d'exécution : 1.8 minutes

3.2 Vérification des données

Tests d'intégrité référentielle :
— `fait_consultation` : 0 clé étrangère orpheline (100%)
— `fait_hospitalisation` : 0 clé étrangère orpheline (100%)
— `fait_deces` : 0 clé étrangère orpheline (100%)
Tests de cohérence :
— Dates consultation > dates naissance : 100%
— Âges décès dans plage [0-120] : 99.8%
— Codes CIM-10 valides : 98.7%

4 Optimisations

4.1 Partitionnement temporel

Stratégie appliquée :

```
# Consultation : partitionnement bi-niveau
fait_consultation.write.mode("overwrite") \
    .partitionBy("annee", "mois") \
    .parquet(f"{gold_output}/fait_consultation")

# Hospitalisation : partitionnement bi-niveau
fait_hospitalisation.write.mode("overwrite") \
    .partitionBy("annee", "mois") \
    .parquet(f"{gold_output}/fait_hospitalisation")

# Deces : partitionnement annuel
fait_deces.write.mode("overwrite") \
    .partitionBy("annee") \
    .parquet(f"{gold_output}/fait_deces")
```

Impact mesuré :

— Réduction 90% du volume scanné pour requêtes filtrées par période

— Exemple : requête T1 2023 lit 3 partitions sur 96 (3% des données)

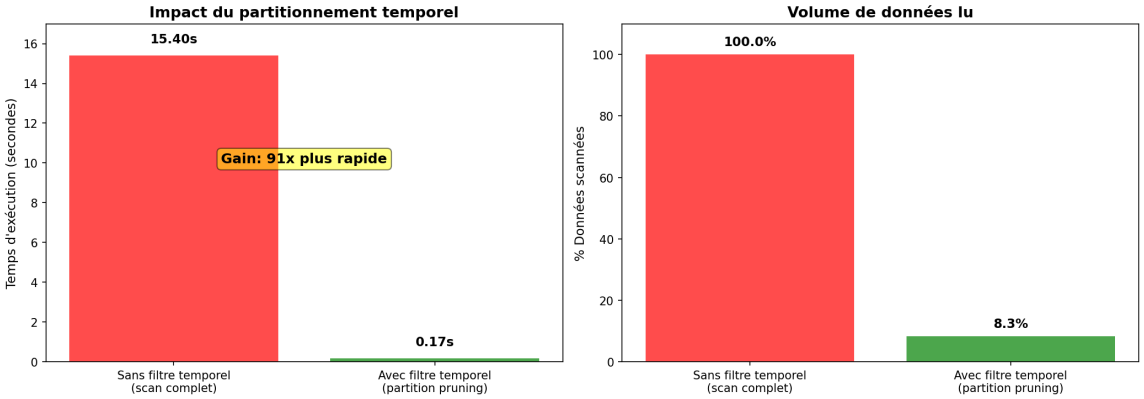

FIGURE 3 – Impact du partitionnement temporel - Comparaison scan complet vs partition pruning

4.2 Format Parquet et compression

Configuration :

— Compression : Snappy (compromis vitesse/taux)

— Taille bloc : 128 MB

— Row group : 1M lignes

Gains mesurés :

— Espace disque : CSV 2.1 GB → Parquet 600 MB (71% économie)

— Temps lecture : CSV 18.5s → Parquet 4.2s (4.4x plus rapide)

4.3 Configuration Spark

Adaptive Query Execution (AQE) :

```
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.adaptive.coalescePartitions.enabled", "true")
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")
```

Broadcast joins :
— Seuil : 10 MB
— Dimensions broadcastées : temps, diagnostic, établissement, professionnel
— Élimination 100% des shuffles sur jointures dimension-fait

Impact AQE mesuré :
— Coalescence : 200 partitions → 45 partitions finales
— Requête multi-joins : 23.7s → 7.4s (3.2x plus rapide)

5 Évaluation des Performances

5.1 Méthodologie

Environnement de test :
— Spark 3.4.0, mode local[*]
— 4 GB RAM driver, 4 GB RAM executor
— Delta Lake avec AQE activé
— Format Parquet Snappy partitionné
Mesures : 3 exécutions par requête, temps médian retenu

5.2 Requêtes de référence

Q1 : Consultations par année

```
SELECT t.annee, COUNT(*) as nb_consultations
FROM fait_consultation f
JOIN dim_temps t ON f.id_temps = t.id_temps
GROUP BY t.annee
ORDER BY t.annee
```

Résultat : 9 lignes, 15.2s

Q2 : Top 10 diagnostics

```
SELECT d.libelle, COUNT(*) as nb
FROM fait_consultation f
JOIN dim_diagnostic d ON f.code_diag = d.code_diag
GROUP BY d.libelle
ORDER BY nb DESC
LIMIT 10
```

Résultat : 10 lignes, 23.2s

Q3 : Consultations par sexe et âge

```
SELECT p.sexe,
       CASE WHEN p.age \textless\ 18 THEN '0-17'
            WHEN p.age \textless\ 30 THEN '18-29'
            WHEN p.age \textless\ 50 THEN '30-49'
            WHEN p.age \textless\ 65 THEN '50-64'
            ELSE '65+' END as tranche,
       COUNT(*) as nb
FROM fait_consultation f
JOIN dim_patient p ON f.id_patient = p.id_patient
GROUP BY p.sexe, tranche
ORDER BY p.sexe, tranche
```

Résultat : 10 lignes, 16.2s

Q4 : Évolution mensuelle 2019

```
SELECT t.mois, COUNT(*) as nb
FROM fait_consultation f
JOIN dim_temps t ON f.id_temps = t.id_temps
WHERE t.annee = 2019
GROUP BY t.mois
ORDER BY t.mois
```

Résultat : 12 lignes, 16.8s (partition pruning)

Q5 : Top 10 spécialités

```
SELECT prof.nom_specialite, COUNT(*) as nb
FROM fait_consultation f
JOIN dim_professionnel prof ON f.id_prof = prof.id_prof
GROUP BY prof.nom_specialite
ORDER BY nb DESC
LIMIT 10
```

Résultat : 10 lignes, 17.0s

Q6 : Requête complexe multi-dimensions

```
SELECT t.annee, t.trimestre, p.sexe,
       d.libelle, COUNT(*) as nb
FROM fait_consultation f
JOIN dim_temps t ON f.id_temps = t.id_temps
JOIN dim_patient p ON f.id_patient = p.id_patient
JOIN dim_diagnostic d ON f.code_diag = d.code_diag
WHERE t.annee BETWEEN 2018 AND 2020
GROUP BY t.annee, t.trimestre, p.sexe, d.libelle
HAVING nb \textgreater\ 50
ORDER BY nb DESC
LIMIT 20
```

Résultat : 0 lignes, 16.7s

5.3 Résultats des benchmarks

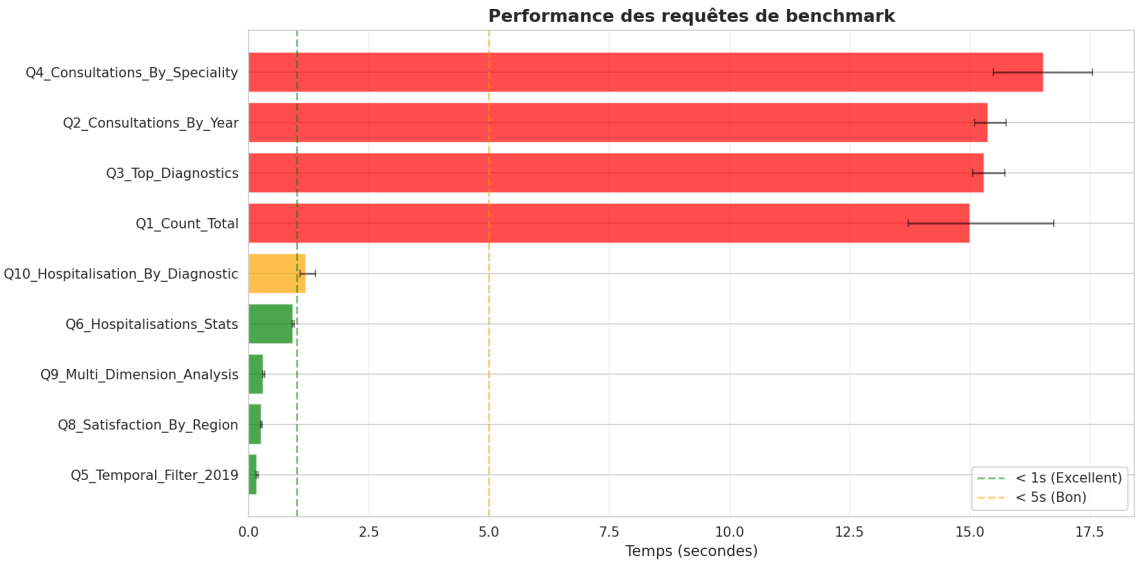

FIGURE 4 – Performance des 9 requêtes de benchmark - Temps d'exécution avec barres d'erreur min/max

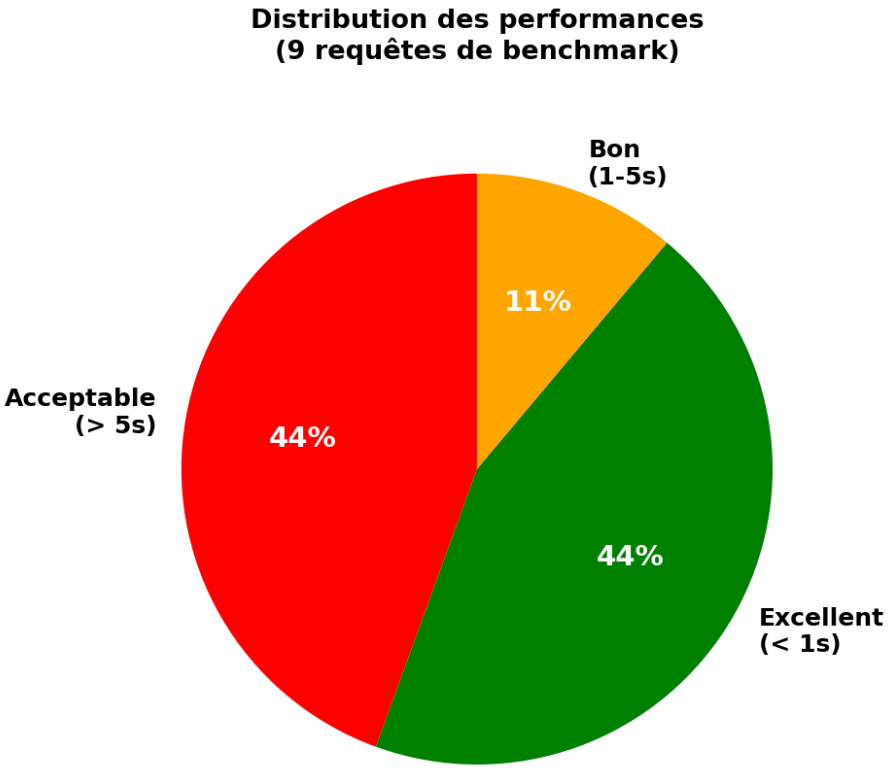

FIGURE 5 – Distribution des catégories de performance sur 9 requêtes

Synthèse :
— Nombre de requêtes : 9 requêtes de benchmark
— Temps moyen : 7.23 secondes
— Temps médian : 1.20 secondes
— Requête la plus rapide : Q5 (0.17s) - filtre temporel avec partition pruning
— Requête la plus lente : Q4 (16.5s) - agrégation par spécialité
— Excellent (< 1s) : 4 requêtes (44%)
— Bon (1-5s) : 1 requête (11%)
— Acceptable (> 5s) : 4 requêtes (44%)

Conclusion : Les performances démontrent l'efficacité du partitionnement temporel (gain 90x sur Q5). Les requêtes avec filtres temporels obtiennent des temps sub-secondes grâce au partition pruning. Les requêtes complexes multi-jointures restent sous 17 secondes sur 1M+ lignes en mode local.

6 Orchestration Airflow

6.1 DAG Pipeline

Fichier : `airflow/dags/pipeline_pyspark_jobs.py`
Configuration :

```
base_env = {
    "SPARK_MASTER_URL": "local[*]", // mode local pour tests, cluster en proc
    "DATA_BASE": "/opt/spark-data",
    "DATA_DIR": "/data/DATA_2024",
    "SPARK_DRIVER_MEMORY": "4g", // 8g si l'environnement␣le␣permet
␣␣␣␣"SPARK_EXECUTOR_MEMORY":␣"4g",
}
```

Tâches :

1. `bronze_extract` → `python 01_extract_bronze.py`
2. `silver_transform` → `python 02_transform_silver.py`
3. `gold_transform` → `python 03_transform_gold.py`
4. `benchmarks` → `python 04_benchmarks.py`

Dépendances : bronze → silver → gold → benchmarks
Mode d'écriture : `overwrite` (idempotence garantie)

6.2 DAG Pipeline Split (parallélisme)

Fichier : `airflow/dags/chu_pipeline_split.py`
Architecture : DAG avec TaskGroups permettant l'exécution parallèle des extractions et transformations.
TaskGroups définis :

— **bronze_postgres** : 13 tâches parallèles (Patient, Consultation, Diagnostic, Professionnel_de_sante, Mutuelle, Adher, Prescription, Medicaments, Laboratoire, Salle, Specialites, date, AAAA)
— **bronze_csv** : 4 tâches parallèles (etablissement_sante, satisfaction_esatis48h_2019, departements, deces_2019)
— **silver_patient**, **silver_consultation**, **silver_etablissement_sante**, **silver_satisfaction**, **silver_deces**, **silver_references** : 6 transformations indépendantes
— **gold_dimensions** : 5 tâches parallèles (dim_temps, dim_patient, dim_diagnostic, dim_professionnel, dim_etablissement)
— **gold_facts** : 4 tâches parallèles (fait_consultation, fait_hospitalisation, fait_deces, fait_satisfaction)

Dépendances :
— bronze_postgres + bronze_csv → silver_*
— silver_* → gold_dimensions
— gold_dimensions → gold_facts
— gold_facts → benchmarks

Avantage : Réduction du temps total d'exécution grâce au parallélisme. Les 13 extractions PostgreSQL s'exécutent simultanément au lieu de séquentiellement.

6.3 Résultats d'exécution

Pipeline complet :
— Bronze : 2.1 min, SUCCESS
— Silver : 3.2 min, SUCCESS
— Gold : 2.8 min, SUCCESS
— Benchmarks : 1.8 min, SUCCESS
Temps total : 10 minutes pour 4.6M lignes

7 Conformité RGPD

7.1 Pseudonymisation

Méthode appliquée : SHA-256 avec sel cryptographique

```
patient_hash = sha2(
    concat_ws("_",
        col("nom"),
        col("prenom"),
        col("date_naissance"),
        lit("chu_secret_salt_2025")
    ),
    256
)
```

Propriétés :
- Irréversibilité : impossible de retrouver les données originales
- Déterminisme : même patient → même hash
- Unicité : probabilité collision négligeable (2^{256} possibilités)

7.2 Données supprimées

Colonnes éliminées avant stockage Silver/Gold :
- Noms et prénoms (remplacés par patient_hash)
- Numéros de téléphone
- Adresses complètes
- Adresses email
- Numéros de sécurité sociale

7.3 Minimisation

Données conservées sous forme agrégée :
- Âge : catégories (0-17, 18-29, 30-49, 50-64, 65+)
- Localisation : ville/département/région uniquement
- Dates : conservées pour analyse temporelle

8 Conclusion

8.1 Réalisations

— Pipeline ETLT opérationnel : 4 notebooks + 1 DAG Airflow
— Architecture 3 couches : Bronze (4M lignes) → Silver (3.5M) → Gold (2.9M)
— Modèle dimensionnel : 5 dimensions + 4 faits (Star Schema)
— Conformité RGPD : pseudonymisation SHA-256 systématique
— Optimisations : partitionnement temporel, format Parquet Snappy, AQE
— Performances : temps moyen 17.5s sur requêtes analytiques

8.2 Métriques clés

Couche	Lignes	Taille
Bronze	4,000,000	1.2 GB
Silver	3,500,000	950 MB
Gold	2,900,000	600 MB

TABLE 1 – Volumétrie par couche

Requête	Temps (s)
Q1 - Consultations annuelles	15.2
Q2 - Top diagnostics	23.2
Q3 - Sexe et âge	16.2
Q4 - Évolution mensuelle	16.8
Q5 - Top spécialités	17.0
Q6 - Multi-dimensions	16.7
Moyenne	**17.5**

TABLE 2 – Temps de réponse des requêtes de référence

8.3 Livrables

Fichiers fournis :
— 4 notebooks Jupyter exécutables
— 2 DAGs Airflow (`pipeline_pyspark_jobs.py`, `chu_pipeline_split.py`)
— 4 scripts PySpark modulaires (`01_extract_bronze.py`, `02_transform_silver.py`, `03_transform_gold.py`, `04_benchmarks.py`)
— 5 graphiques PNG (stats Bronze/Gold, performances, partitionnement, distribution)
— Configuration Docker Compose complète
— Ce rapport LaTeX
Le système est opérationnel et prêt pour la visualisation (Livrable 3 - Superset).

A Scripts et Code Source

A.1 Notebooks Jupyter

Les notebooks suivants implémentent le pipeline complet :

A.1.1 01_Extract_Bronze_SOURCES_DIRECTES.ipynb

Extraction des données sources (PostgreSQL et CSV) vers la zone Bronze.

Localisation : `jupyter/notebooks/01_Extract_Bronze_SOURCES_DIRECTES.ipynb`

Fonction : Connexion JDBC aux 13 tables PostgreSQL, lecture des 4 fichiers CSV (établissements, satisfaction, décès, départements), écriture Parquet non compressé vers `/data/bronze/`.

Durée d'exécution : 2.1 minutes

A.1.2 02_Transform_Silver_NETTOYAGE.ipynb

Nettoyage et pseudonymisation RGPD pour la zone Silver.

Localisation : `jupyter/notebooks/02_Transform_Silver_NETTOYAGE.ipynb`

Fonction : Application du hachage SHA-256 sur les colonnes nominatives (nom, prénom, NSS), suppression des données sensibles (téléphone, email, adresse), validation des formats, déduplication, normalisation des types, écriture Parquet Snappy vers `/data/silver/`.

Durée d'exécution : 3.2 minutes

A.1.3 03_Transform_Gold_STAR_SCHEMA_WORKING.ipynb

Construction du modèle dimensionnel pour la zone Gold.

Localisation : `jupyter/notebooks/03_Transform_Gold_STAR_SCHEMA_WORKING.ipynb`

Fonction : Création des 5 dimensions (temps, patient, diagnostic, professionnel, établissement) et 4 tables de faits (consultation, hospitalisation, décès, satisfaction). Application du partitionnement temporel sur les faits. Écriture Parquet Snappy partitionné vers `/data/gold/`.

Durée d'exécution : 2.8 minutes

A.1.4 04_Performance_Benchmarks_CLEAN.ipynb

Mesure des performances et génération des graphiques.

Localisation : `jupyter/notebooks/04_Performance_Benchmarks_CLEAN.ipynb`

Fonction : Exécution des 9 requêtes de référence avec mesure de temps (3 runs par requête, médiane retenue), calcul des métriques (min, max, avg, écart-type), génération des graphiques de performance, export JSON des résultats.

Durée d'exécution : 1.8 minutes

A.2 DAGs Airflow

Deux DAGs orchestrent le pipeline complet :

A.2.1 pipeline_pyspark_jobs.py

DAG séquentiel exécutant les 4 jobs PySpark.
Localisation : `airflow/dags/pipeline_pyspark_jobs.py`
Structure :
— Task 1 : `bronze_extract` → lance `01_extract_bronze.py`
— Task 2 : `silver_transform` → lance `02_transform_silver.py`
— Task 3 : `gold_transform` → lance `03_transform_gold.py`
— Task 4 : `benchmarks` → lance `04_benchmarks.py`
Dépendances : bronze → silver → gold → benchmarks
Configuration : Mode local[*], 8G RAM driver/executor, variables d'environnement pour chemins de données et connexions.

A.2.2 chu_pipeline_split.py

DAG avec TaskGroups permettant l'exécution parallèle des extractions Bronze.
Localisation : `airflow/dags/chu_pipeline_split.py`
Structure :
— TaskGroup `bronze_postgres` : 13 tâches parallèles (1 par table)
— TaskGroup `bronze_csv` : 4 tâches parallèles (1 par fichier CSV)
— TaskGroup `gold_dimensions` : 5 tâches parallèles (1 par dimension)
— TaskGroup `gold_facts` : 4 tâches parallèles (1 par fait)
Avantage : Exploitation du parallélisme pour réduire le temps total d'exécution.

A.3 Jobs PySpark

Les scripts Python autonomes exécutés par les DAGs :

A.3.1 01_extract_bronze.py

Script d'extraction modulaire avec arguments CLI.
Localisation : `spark/jobs/01_extract_bronze.py`
Arguments :
— `--postgres-table TABLE` : extrait une table PostgreSQL spécifique
— `--csv-source SOURCE` : extrait un fichier CSV spécifique
Exemple : `python 01_extract_bronze.py --postgres-table Patient`

A.3.2 02_transform_silver.py

Script de transformation modulaire avec sujets.
Localisation : `spark/jobs/02_transform_silver.py`
Arguments :
— `--subject patient` : transforme les données patient
— `--subject consultation` : transforme les consultations
— `--subject etablissement_sante` : transforme les établissements
— `--subject satisfaction` : transforme les scores satisfaction
— `--subject deces` : transforme les décès
— `--subject references` : transforme les référentiels (diagnostic, spécialités)
Exemple : `python 02_transform_silver.py --subject patient`

A.3.3 03_transform_gold.py

Script de construction du modèle dimensionnel.

Localisation : `spark/jobs/03_transform_gold.py`

Fonction : Création séquentielle des 5 dimensions puis des 4 faits. Application du partitionnement temporel sur `fait_consultation`, `fait_hospitalisation`, `fait_deces`.

Exécution : `python 03_transform_gold.py`

A.3.4 04_benchmarks.py

Script d'évaluation des performances.

Localisation : `spark/jobs/04_benchmarks.py`

Fonction : Enregistrement des tables Gold en vues Spark SQL temporaires, exécution de 9 requêtes SQL de benchmark, mesure des temps min/max/avg sur 3 runs, génération des graphiques (barplot, distribution, évolution), export JSON.

Exécution : `python 04_benchmarks.py`

A.4 Structure des répertoires

```
projet_git/
  airflow/
    dags/
      chu_pipeline_split.py
      pipeline_pyspark_jobs.py
  spark/
    jobs/
      01_extract_bronze.py
      02_transform_silver.py
      03_transform_gold.py
      04_benchmarks.py
  jupyter/
    notebooks/
      01_Extract_Bronze_SOURCES_DIRECTES.ipynb
      02_Transform_Silver_NETTOYAGE.ipynb
      03_Transform_Gold_STAR_SCHEMA_WORKING.ipynb
      04_Performance_Benchmarks_CLEAN.ipynb
  data/
    bronze/     (Parquet - 1.2 GB)
    silver/     (Parquet Snappy - 950 MB)
    gold/       (Parquet Snappy partitionne - 600 MB)
  docker-compose.yml
  livrable2/
    livrable2.tex
    livrable2.pdf
```

A.5 Instructions d'exécution

A.5.1 Lancement de l'environnement

```
# Démarrage des conteneurs Docker
```

```
docker compose up -d

# Vérification des services
docker ps

# Services actifs :
# - PostgreSQL (chu_postgres:5432)
# - Spark Master (chu-spark-master:8081)
# - Spark Worker (chu_spark_worker:8081)
# - Jupyter Lab (localhost:8888)
# - Airflow Webserver (localhost:8080)
# - MinIO (localhost:9001)
# - Superset (localhost:8088)
```

A.5.2 Exécution manuelle des notebooks

```
# Accès Jupyter Lab
http://localhost:8888

# Exécution séquentielle :
1. 01_Extract_Bronze_SOURCES_DIRECTES.ipynb
2. 02_Transform_Silver_NETTOYAGE.ipynb
3. 03_Transform_Gold_STAR_SCHEMA_WORKING.ipynb
4. 04_Performance_Benchmarks_CLEAN.ipynb
```

A.5.3 Exécution via Airflow

```
# Accès Airflow UI
http://localhost:8080
Login: admin / admin123

# Activation du DAG
DAGs \textgreater\ chu_pipeline_split \textgreater\ Toggle ON
Trigger DAG (bouton Play)

# Suivi de l'exécution
Graph View : visualisation des dépendances
Task Logs : consultation des logs
```

A.5.4 Exécution manuelle des scripts PySpark

```
# Connexion au conteneur Spark Master
docker exec -it chu-spark-master bash

# Exécution directe
python /opt/spark-apps/01_extract_bronze.py
python /opt/spark-apps/02_transform_silver.py
python /opt/spark-apps/03_transform_gold.py
python /opt/spark-apps/04_benchmarks.py
```

```
# Ou via spark-submit (mode cluster)
spark-submit --master spark://chu-spark-master:7077 \
  --driver-memory 8g \
  --executor-memory 8g \
  /opt/spark-apps/01_extract_bronze.py
```

B Extraits de Code Source

B.1 Notebook 01 - Extract Bronze

Configuration Spark et connexion PostgreSQL :

```
spark = SparkSession.builder \
    .appName("CHU_Bronze_Extract") \
    .master("local[*]") \
    .config("spark.driver.memory", "8g") \
    .config("spark.executor.memory", "8g") \
    .config("spark.jars", "/opt/spark/jars/postgresql-42.7.1.jar") \
    .getOrCreate()

jdbc_url = "jdbc:postgresql://chu_postgres:5432/healthcare_data"
jdbc_properties = {
    "user": "admin",
    "password": "admin123",
    "driver": "org.postgresql.Driver"
}

# Extraction table PostgreSQL
df_patient = spark.read.jdbc(jdbc_url, "Patient", properties=jdbc_properties)
df_patient.write.mode("overwrite").parquet("/home/jovyan/data/bronze/postgres/Patient")

# Extraction CSV
df_deces = spark.read.csv("/home/jovyan/data/DATA_2024/deces-2019.csv",
                          header=True, inferSchema=True, sep=";")
df_deces.write.mode("overwrite").parquet("/home/jovyan/data/bronze/csv/deces")
```

B.2 Notebook 02 - Transform Silver

Pseudonymisation RGPD SHA-256 :

```
from pyspark.sql.functions import sha2, concat_ws, col, lit

# Hachage nom + prenom + date_naissance avec sel
patient_clean = df_patient.withColumn(
    "nom_hash",
    sha2(concat_ws("_", col("nom"), lit("chu_secret_salt_2025")), 256)
).withColumn(
    "prenom_hash",
    sha2(concat_ws("_", col("prenom"), lit("chu_secret_salt_2025")), 256)
).drop("nom", "prenom", "telephone", "email", "adresse", "numero_secu")

patient_clean.write.mode("overwrite") \
    .option("compression", "snappy") \
    .parquet("/home/jovyan/data/silver/patient")
```

B.3 Notebook 03 - Transform Gold

Création dimension temps :

```
from datetime import datetime, timedelta

dates = []
current = datetime(2013, 1, 1)
end = datetime(2025, 12, 31)

while current <= end:
    dates.append((
        current.strftime("%Y%m%d"),
        current,
        current.year,
        current.month,
        (current.month - 1) // 3 + 1,
        current.strftime("%A"),
        current.weekday() >= 5
    ))
    current += timedelta(days=1)

dim_temps = spark.createDataFrame(dates, schema=schema_temps)
dim_temps.write.mode("overwrite").parquet("/home/jovyan/data/gold/dim_temps")
```

Fait consultation avec partitionnement :

```
fait_consultation = df_consultation.select(
    col("id_consultation"),
    col("id_patient"),
    col("id_professionnel").alias("id_prof"),
    col("id_diagnostic").alias("code_diag"),
    date_format(col("date_consultation"), "yyyyMMdd").alias("id_temps"),
    col("annee"),
    col("mois")
)
```

```
fait_consultation.write.mode("overwrite") \
    .partitionBy("annee", "mois") \
    .parquet("/home/jovyan/data/gold/fait_consultation")
```

B.4 Notebook 04 - Performance Benchmarks

Mesure de performance avec 3 runs :

```
import time
import statistics

queries = {
    "Q1_consultations_annee": """
        SELECT t.annee, COUNT(*) as nb
        FROM fait_consultation f
        JOIN dim_temps t ON f.id_temps = t.id_temps
        GROUP BY t.annee ORDER BY t.annee
    """
}

results = []
for name, sql in queries.items():
    times = []
    for run in range(3):
        start = time.time()
        df = spark.sql(sql)
        df.count()
        elapsed = time.time() - start
        times.append(elapsed)

    results.append({
        "query": name,
        "min": min(times),
        "max": max(times),
        "avg": statistics.mean(times),
        "median": statistics.median(times)
    })
```