

# TP : Data Warehouse Moderne

SCD Type 2, Data Vault 2.0, Architecture Lakehouse avec PySpark et Delta Lake

## Table des Matières

---

- 1. Introduction et Préparation
- 2. Installation PostgreSQL
- 3. Installation Python et PySpark
- 4. Connexion PySpark-PostgreSQL
- 5. Slowly Changing Dimensions
- 6. Data Vault 2.0

7. Architecture Lakehouse

8. Projet Final

# 1. Introduction et Vue d'Ensemble

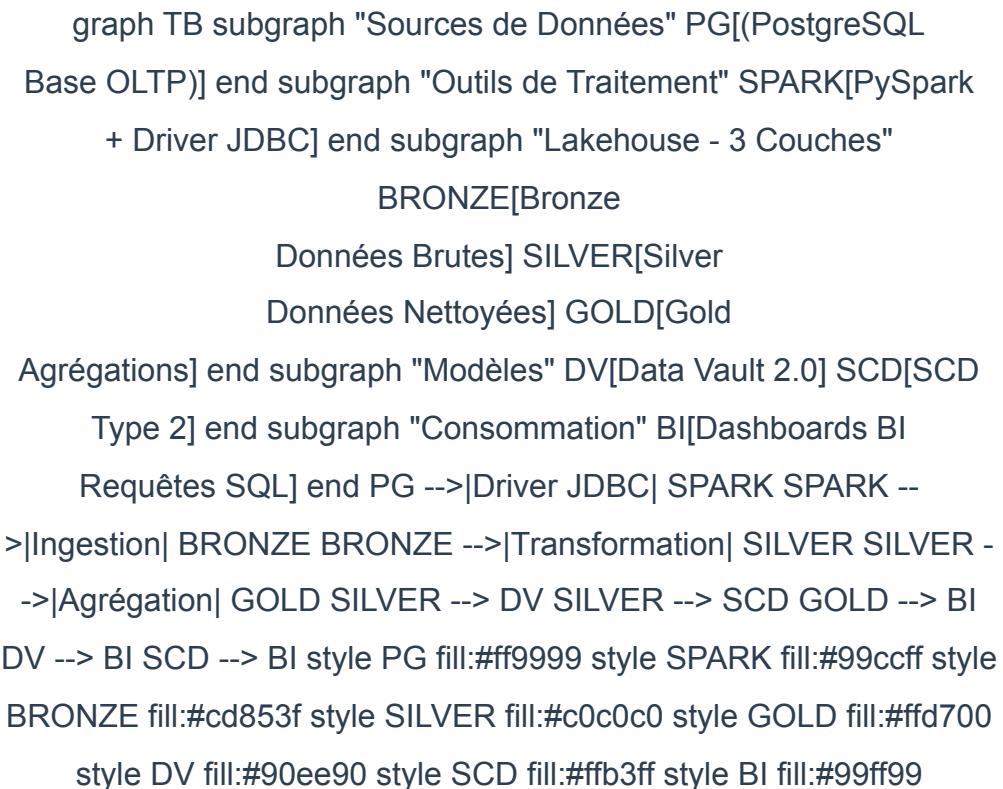
## 1.1 Objectifs du TP

Ce travail pratique vous permet de construire un Data Warehouse moderne complet, depuis l'installation des outils jusqu'à la création d'un pipeline de données bout-en-bout. Vous allez apprendre à :

- Installer et configurer un environnement de Data Warehousing professionnel
- Comprendre comment PySpark se connecte à PostgreSQL (avec explications détaillées)
- Implémenter des Slowly Changing Dimensions pour gérer l'historique des données
- Créer un modèle Data Vault 2.0 pour une architecture agile
- Construire un Lakehouse avec les couches Bronze, Silver et Gold
- Utiliser Delta Lake pour des transactions ACID et le Time Travel

## 1.2 Architecture Globale du Projet

# Vue d'Ensemble : Du système source au BI



## Explication du diagramme

### Flux de données :

1. **PostgreSQL (source)** : Contient vos données transactionnelles (clients, produits, ventes)
2. **Driver JDBC** : Permet à PySpark de se connecter à PostgreSQL (nous allons l'installer)
3. **PySpark** : Lit les données, les transforme et les écrit dans le Lakehouse
4. **Bronze** : Stockage des données brutes telles quelles arrivent

- 5. **Silver** : Nettoyage, validation, enrichissement des données
- 6. **Gold** : Tables agrégées optimisées pour les requêtes analytiques

## 1.3 Prérequis

Catégorie	Requis	Vérification
<b>Système</b>	Windows 10+, macOS 11+, ou Linux (Ubuntu 20.04+)	N/A
<b>Espace disque</b>	Au moins 10 GB libres	<code>df -h</code> (Linux/Mac) ou Explorateur de fichiers (Windows)
<b>RAM</b>	8 GB minimum (16 GB recommandé)	Gestionnaire de tâches
<b>Connexion Internet</b>	Requise pour téléchargements	N/A

## 1.4 Durée Estimée

**Total : 12 à 16 heures** réparties comme suit :

- Sections 1-3 (Installation) : 2-3 heures
- Section 4 (Connexion PySpark) : 2-3 heures
- Section 5 (SCD) : 2-3 heures
- Section 6 (Data Vault) : 3-4 heures

- Section 7 (Lakehouse) : 3-4 heures

## Avant de Commencer

Assurez-vous d'avoir :

- Un éditeur de texte ou IDE (VS Code, PyCharm, ou Notepad++)
- Les droits administrateur sur votre machine
- Un bloc-notes pour noter les mots de passe que vous créerez

## 2. Installation et Configuration de PostgreSQL

---

### 2.1 Téléchargement de PostgreSQL

1

#### Télécharger PostgreSQL

##### Pour Windows

1. Allez sur <https://www.postgresql.org/download/windows/>
2. Cliquez sur "Download the installer"
3. Choisissez la version 15 ou 16 (64-bit)
4. Téléchargez le fichier `postgresql-16.x-windows-x64.exe`

##### Pour macOS

1. Option 1 - Homebrew (recommandé) :

```
brew install postgresql@16
```

2. Option 2 - Installateur graphique :

- Téléchargez depuis

<https://www.postgresql.org/download/macosx/>

## Pour Linux (Ubuntu/Debian)

```
sudo apt update sudo apt install postgresql postgresql-contrib
```

2

## Installation de PostgreSQL (Windows - détaillé)

### Étape par étape :

#### 1. Lancez l'installateur

- Double-cliquez sur le fichier téléchargé
- Cliquez sur "Next" pour commencer

#### 2. Répertoire d'installation

- Laissez le chemin par défaut : `C:\Program Files\PostgreSQL\16`
- Cliquez sur "Next"

#### 3. Composants à installer

- Cochez TOUTES les cases :

- PostgreSQL Server (serveur de base de données)
- pgAdmin 4 (interface graphique - **IMPORTANT**)
- Stack Builder (pour extensions)
- Command Line Tools (outils en ligne de commande)

◦ Cliquez sur "Next"

#### 4. Répertoire des données

- Laissez : `C:\Program Files\PostgreSQL\16\data`
- Cliquez sur "Next"

#### 5. Mot de passe superutilisateur (**TRÈS IMPORTANT**)

- Créez un mot de passe FORT pour l'utilisateur `postgres`
- Exemple : `PostgreSQL2025!`
- **NOTEZ CE MOT DE PASSE** - vous en aurez besoin constamment
- Confirmez le mot de passe
- Cliquez sur "Next"

#### 6. Port

- Laissez le port par défaut : `5432`
- Cliquez sur "Next"

#### 7. Locale

- Choisissez "Default locale" ou "French, France"
- Cliquez sur "Next"

#### 8. Installation

- Vérifiez le résumé

- Cliquez sur "Next" pour lancer l'installation
- Attendez 3-5 minutes

## 9. Fin d'installation

- Décochez "Stack Builder" (pas nécessaire maintenant)
- Cliquez sur "Finish"

3

## Vérifier que PostgreSQL fonctionne

### Méthode 1 : Via pgAdmin (Interface Graphique)

1. Dans le menu Démarrer Windows, cherchez "pgAdmin 4"
2. Lancez pgAdmin 4
3. À gauche, cliquez sur "Servers"
4. Cliquez sur "PostgreSQL 16"
5. Entrez le mot de passe que vous avez créé
6. Si ça se connecte : **PostgreSQL fonctionne !**

### Méthode 2 : Via ligne de commande

Ouvrez l'invite de commandes (CMD) et tapez :

```
psql -U postgres
```

Entrez votre mot de passe. Vous devriez voir :

```
postgres=#
```

Pour quitter :

\q

## Problèmes fréquents

### Problème 1 : "psql n'est pas reconnu comme commande"

Solution : Ajoutez PostgreSQL au PATH Windows

1. Cherchez "Variables d'environnement" dans Windows
2. Cliquez sur "Variables d'environnement..."
3. Dans "Variables système", trouvez "Path" et cliquez "Modifier"
4. Cliquez "Nouveau" et ajoutez : `C:\Program Files\PostgreSQL\16\bin`
5. Cliquez OK partout
6. Fermez et rouvrez l'invite de commandes

### Problème 2 : "Le service PostgreSQL ne démarre pas"

Solution :

1. Ouvrez "Services" (services.msc dans Exécuter)
2. Trouvez "postgresql-x64-16"
3. Clic droit > Démarrer

## Checkpoint Section 2

Vérifiez que :

- PostgreSQL est installé
- Le service PostgreSQL est démarré
- Vous pouvez vous connecter avec pgAdmin
- Vous avez noté votre mot de passe

## 3. Installation Python, PySpark et Dépendances

---

### 3.1 Installation de Python

4

#### Installer Python 3.10 ou supérieur

##### Windows

1. Allez sur <https://www.python.org/downloads/>
2. Téléchargez Python 3.10 ou 3.11 (64-bit)
3. Lancez l'installateur
4. **IMPORTANT : Cochez "Add Python to PATH"**
5. Cliquez sur "Install Now"
6. Attendez la fin de l'installation

##### Vérification

Ouvrez un nouveau terminal et tapez :

```
{ python --version
```

Vous devriez voir : Python 3.10.x ou Python 3.11.x

## 5   Créer un environnement virtuel

### Pourquoi un environnement virtuel ?

Un environnement virtuel isole les packages Python de ce projet. Cela évite les conflits avec d'autres projets Python sur votre machine.

#### Étapes :

1. Créez un dossier pour le projet :

```
{ mkdir C:\TP_DataWarehouse cd C:\TP_DataWarehouse
```

2. Créez l'environnement virtuel :

```
{ python -m venv venv
```

Cela crée un dossier `venv` dans votre répertoire

3. Activez l'environnement virtuel :

**Windows :**

```
venv\Scripts\activate
```

macOS/Linux :

```
source venv/bin/activate
```

Votre invite de commande devrait maintenant afficher `(venv)`

## 6

## Installer PySpark et les dépendances

Avec l'environnement virtuel activé, installez les packages :

```
pip install pyspark==3.5.0 pip install delta-spark==3.0.0 pip install psycopg2-binary==2.9.9 pip install pandas==2.1.4
```

### Que fait chaque package ?

Package	Rôle
<b>pyspark</b>	Framework de traitement distribué de données (Apache Spark pour Python)
<b>delta-spark</b>	Extension pour Delta Lake (transactions ACID, Time Travel)
<b>psycopg2-binary</b>	Driver Python pour PostgreSQL (pour scripts Python classiques)

**pandas**

Bibliothèque de manipulation de données

## Vérification de l'installation :

Créez un fichier `test_installation.py` :

```
import pyspark
import delta
import psycopg2
import pandas

print("PySpark version:", pyspark.__version__)
print("Delta Lake version:", delta.__version__)
print("Psycopg2 version:", psycopg2.__version__)
print("Pandas version:", pandas.__version__)
print("\nToutes les bibliothèques sont installées !")
```

Exécutez :

```
{ python test_installation.py }
```

Vous devriez voir les versions affichées sans erreur.

## 4. Connexion PySpark à PostgreSQL - Explications Détaillées

---

## 4.1 Comprendre la Connexion JDBC

### Comment PySpark communique avec PostgreSQL ?

PySpark ne peut pas se connecter directement à PostgreSQL. Il a besoin d'un **driver JDBC** (Java Database Connectivity).

#### Schéma de connexion :

PySpark (Python) → Driver JDBC (Java) → PostgreSQL

#### Pourquoi du Java ?

Spark est écrit en Scala (qui tourne sur la JVM Java). Quand vous utilisez PySpark, votre code Python appelle des fonctions Spark qui s'exécutent réellement en Java/Scala.

### Architecture de Connexion Détailée

```
sequenceDiagram participant P as Script Python participant PS as PySpark participant JDBC as Driver JDBC PostgreSQL participant PG as PostgreSQL
P->>PS: spark.read.format("jdbc")
PS->>JDBC: Charge le driver .jar
JDBC->>PG: Se connecte (host, port, user, password)
PG-->>JDBC: Connexion établie
JDBC-->>PS: Prêt à lire les données
PS->>JDBC: SELECT * FROM table
JDBC->>PG: 
```

Exécute la requête SQL PG-->JDBC: Retourne les résultats JDBC-->>PS: Données au format Spark PS-->>P: DataFrame PySpark

## 4.2 Télécharger le Driver JDBC PostgreSQL

7

### Télécharger le driver JDBC

#### Étapes détaillées :

1. Allez sur <https://jdbc.postgresql.org/download/>
2. Trouvez la section "Current Version" (par exemple 42.7.1)
3. Cliquez sur le fichier `postgresql-42.7.1.jar`
4. Le fichier .jar se télécharge (environ 1 MB)
5. Placez ce fichier dans un dossier dédié :

```
mkdir C:\TP_DataWarehouse\drivers # Déplacez le  
fichier .jar téléchargé dans ce dossier
```

6. Le chemin final devrait être :

```
C:\TP_DataWarehouse\drivers\postgresql-42.7.1.jar
```

8

### Tester la connexion PySpark-PostgreSQL

Créez un fichier `test_connexion_pyspark.py` avec ce code COMMENTÉ

LIGNE PAR LIGNE :

```
# =====#
# Script : Test de connexion PySpark vers PostgreSQL
# Description : Vérifie que PySpark peut lire des données
# =====#

# ÉTAPE 1 : Importer les bibliothèques nécessaires
from pyspark.sql import SparkSession
from delta import configure_spark_with_delta_pip

print("Bibliothèques importées avec succès")

# =====#
# ÉTAPE 2 : Configurer Spark avec le driver JDBC
# =====#

# Créer un builder Spark (constructeur de session)
builder = SparkSession.builder \
    .appName("Test Connexion PostgreSQL") \
    .master("local[*]")

# EXPLICATION DE .master("local[*]") :
# - "local" = Spark tourne sur votre machine (pas en cluster)
# - "[*]" = Utilise tous les cœurs CPU disponibles

# Ajouter le driver JDBC PostgreSQL au classpath
# IMPORTANT : Remplacez ce chemin par le vôtre
builder = builder.config(
    "spark.jars",
    "C:/TP_DataWarehouse/drivers/postgresql-42.7.1.jar"
)

# EXPLICATION :
# - spark.jars dit à Spark "charge ce fichier .jar au démarrage"
# - Spark va pouvoir utiliser le code Java dans ce .jar pour lire les données

# Configurer Delta Lake (optionnel pour ce test, mais nécessaire)
spark = configure_spark_with_delta_pip(builder).getOrCreate()
```

```
# EXPLICATION DE .getOrCreate() :
# - Si une session Spark existe déjà, la réutilise
# - Sinon, en crée une nouvelle

print("Session Spark créée avec succès")
print(f"Version de Spark : {spark.version}")

# =====
# ÉTAPE 3 : Configurer les paramètres de connexion PostgreSQL
# =====

# Dictionnaire contenant toutes les infos de connexion
postgres_config = {
    "url": "jdbc:postgresql://localhost:5432/postgres",
    # EXPLICATION de l'URL :
    # - jdbc:postgresql:// = préfixe obligatoire pour JDBC
    # - localhost = serveur (votre machine)
    # - 5432 = port PostgreSQL (par défaut)
    # - postgres = nom de la base de données

    "dbtable": "pg_database",
    # EXPLICATION :
    # - pg_database est une table système PostgreSQL (toujours présente)
    # - On l'utilise juste pour tester la connexion

    "user": "postgres",
    # EXPLICATION :
    # - Nom d'utilisateur PostgreSQL
    # - "postgres" est le superutilisateur créé à l'installation

    "password": "VOTRE_MOT_DE_PASSEICI",
    # IMPORTANT : Remplacez par le mot de passe que vous avez choisi

    "driver": "org.postgresql.Driver"
    # EXPLICATION :
    # - Nom de la classe Java du driver PostgreSQL
    # - Cette classe est dans le fichier .jar qu'on a téléchargé
}

print("Configuration PostgreSQL définie")
```

```
# =====
# ÉTAPE 4 : Lire les données depuis PostgreSQL
# =====

print("\nTentative de connexion à PostgreSQL...")

try:
    # Lire la table avec Spark
    df = spark.read \
        .format("jdbc") \
        .options(**postgres_config) \
        .load()

    # EXPLICATION DÉTAILLÉE :
    # 1. spark.read = commence une lecture de données
    # 2. .format("jdbc") = dit à Spark "utilise JDBC pour
    # 3. .options(**postgres_config) = passe tous les paramètres
    #     Le ** "déplie" le dictionnaire :
    #     {url: "...", user: "..."} devient url="...", user...
    # 4. .load() = exécute la lecture

    print("Connexion réussie !")
    print(f"Nombre de lignes lues : {df.count()}")

    # Afficher le schéma (structure) des données
    print("\nSchéma de la table :")
    df.printSchema()

    # Afficher quelques lignes
    print("\nAperçu des données :")
    df.show(5)

    print("\n" + "="*70)
    print("TEST RÉUSSI : PySpark peut lire les données depuis PostgreSQL")
    print("="*70)

except Exception as e:
    print("\nERREUR lors de la connexion :")
    print(str(e))
    print("\nVérifiez :")
    print("1. PostgreSQL est démarré")
    print("2. Le mot de passe est correct")
```

```
    print("3. Le chemin vers le .jar est correct")
    print("4. Le port 5432 est ouvert")

# =====
# ÉTAPE 5 : Arrêter la session Spark proprement
# =====

spark.stop()
print("\nSession Spark arrêtée")
```

## Exécution du test :

1. Modifiez le mot de passe dans le code
2. Modifiez le chemin du .jar si nécessaire
3. Exécutez :

```
python test_connexion_pyspark.py
```

## Résultat attendu :

```
Bibliothèques importées avec succès
Session Spark créée avec succès
Version de Spark : 3.5.0
Configuration PostgreSQL définie

Tentative de connexion à PostgreSQL...
Connexion réussie !
Nombre de lignes lues : X

Schéma de la table :
root
| -- datname: string (nullable = true)
| -- ...

Aperçu des données :
+-----+...
```

```
| datname      | ...  
+-----+...  
=====  
TEST RÉUSSI : PySpark peut lire les données depuis PostgreSQL  
=====  
Session Spark arrêtée
```

## Erreurs Fréquentes et Solutions

### Erreur 1 : "java.lang.ClassNotFoundException: org.postgresql.Driver"

**Cause :** Le fichier .jar n'est pas trouvé ou le chemin est incorrect

**Solution :**

- Vérifiez que le fichier .jar existe à l'emplacement spécifié
- Utilisez des / au lieu de \ dans le chemin (même sous Windows)
- Utilisez un chemin absolu complet

### Erreur 2 : "authentication failed for user postgres"

**Cause :** Mot de passe incorrect

**Solution :**

- Vérifiez le mot de passe (attention majuscules/minuscules)
- Testez le mot de passe avec pgAdmin ou psql

### Erreur 3 : "Connection refused"

**Cause :** PostgreSQL n'est pas démarré ou port incorrect

### Solution :

- Vérifiez que le service PostgreSQL est démarré
- Vérifiez le port (5432 par défaut)

## Checkpoint Section 4

Vérifiez que :

- Le driver JDBC est téléchargé et placé dans un dossier connu
- Le script de test s'exécute sans erreur
- PySpark peut lire les données depuis PostgreSQL
- Vous comprenez comment fonctionne la connexion JDBC

**Si tout est OK, vous êtes prêt pour la suite !**

## 5. Slowly Changing Dimensions (SCD Type 2)

### 5.1 Comprendre les SCD

#### Qu'est-ce qu'une Slowly Changing Dimension ?

Imaginez que vous avez un client nommé "Jean Dupont" qui habite à Paris.

Dans votre base de données, vous avez :

client_id	nom	ville
1	Jean Dupont	Paris

Un jour, Jean déménage à Lyon. Que faites-vous ?

- **Option 1 (SCD Type 1)** : Remplacer "Paris" par "Lyon" → PROBLÈME : Vous perdez l'historique
- **Option 2 (SCD Type 2)** : Créer une nouvelle ligne → SOLUTION : Vous gardez l'historique complet

**Résultat avec SCD Type 2 :**

client_key	client_id	nom	ville	date_debut	date_fin	est_
1	1	Jean Dupont	Paris	2023-01-01	2024-12- 31	FA
2	1	Jean Dupont	Lyon	2025-01-01	NULL	TRU

Ainsi, vous pouvez savoir que Jean habitait à Paris jusqu'au 31/12/2024, puis à Lyon depuis le 01/01/2025.

## 5.2 Créez la base de données et les tables sources

9

## Créer la base de données pour le TP

Ouvrez pgAdmin ou psql et exécutez :

```
-- Créer la base de données
CREATE DATABASE retailpro_dwh;

-- Se connecter à la nouvelle base
-- Dans pgAdmin : clic droit sur la base > Query Tool
-- Dans psql : \c retailpro_dwh
```

## Créer les tables sources

Créez un fichier `01_create_tables_sources.sql` :

```
-- =====
-- Script : Création des tables sources OLTP
-- Base : retailpro_dwh
-- Description : Simule un système transactionnel
-- =====

-- Table des clients
CREATE TABLE clients_source (
    client_id SERIAL PRIMARY KEY,           -- ID auto-incré
    nom VARCHAR(100) NOT NULL,
    prenom VARCHAR(100) NOT NULL,
    email VARCHAR(200) UNIQUE NOT NULL,
    ville VARCHAR(100),
    segment VARCHAR(50),                  -- Bronze, Silver
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Table des produits
CREATE TABLE produits_source (
    produit_id SERIAL PRIMARY KEY,
```

```
nom_produit VARCHAR(200) NOT NULL,
categorie VARCHAR(100),
prix_unitaire DECIMAL(10,2) NOT NULL,
cout_achat DECIMAL(10,2),
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Table des ventes
CREATE TABLE ventes_source (
    vente_id SERIAL PRIMARY KEY,
    client_id INTEGER REFERENCES clients_source(client_id),
    produit_id INTEGER REFERENCES produits_source(produit_id),
    date_vente TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    quantite INTEGER NOT NULL,
    montant_total DECIMAL(10,2) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Insérer des données de test
INSERT INTO clients_source (nom, prenom, email, ville, sexe)
VALUES ('Dupont', 'Jean', 'jean.dupont@email.com', 'Paris', 'Male'),
       ('Martin', 'Marie', 'marie.martin@email.com', 'Lyon', 'Female'),
       ('Bernard', 'Pierre', 'pierre.bernard@email.com', 'Marseille', 'Male');

INSERT INTO produits_source (nom_produit, categorie, prix_achat, prix_vente)
VALUES ('Laptop HP', 'Informatique', 899.99, 650.00),
       ('Souris Logitech', 'Informatique', 29.99, 15.00),
       ('Clavier Mécanique', 'Informatique', 149.99, 80.00);

INSERT INTO ventes_source (client_id, produit_id, quantite)
VALUES (1, 1, 1, 899.99),
       (2, 2, 2, 59.98),
       (3, 3, 1, 149.99);

-- Vérification
SELECT 'Clients:' as table_name, COUNT(*) as nb_lignes FROM clients_source
UNION ALL
SELECT 'Produits:', COUNT(*) FROM produits_source
UNION ALL
SELECT 'Ventes:', COUNT(*) FROM ventes_source;
```

Exécutez ce script dans pgAdmin (Query Tool) ou avec psql :

```
psql -U postgres -d retailpro_dwh -f  
01_create_tables_sources.sql
```

## 5.3 Créer la table dimension avec SCD Type 2

### 10   Créer la dimension client avec SCD Type 2

Créez un fichier `02_create_dim_client_scd2.sql` :

```
-- =====  
-- Table : dim_client avec SCD Type 2  
-- Description : Dimension client avec historisation complète  
-- =====  
  
CREATE TABLE dim_client (  
    -- Clé surrogate (technique) - unique pour chaque VERSION  
    client_key SERIAL PRIMARY KEY,  
  
    -- Clé naturelle (métier) - peut avoir plusieurs versions  
    client_id INTEGER NOT NULL,  
  
    -- Attributs descriptifs  
    nom VARCHAR(100),  
    prenom VARCHAR(100),  
    email VARCHAR(200),  
    ville VARCHAR(100),  
    segment VARCHAR(50),  
    ...  
);
```

```

-- Colonnes SCD Type 2
date_debut DATE NOT NULL DEFAULT CURRENT_DATE,
date_fin DATE,                                -- NULL = version actuelle
est_courant BOOLEAN DEFAULT TRUE,   -- TRUE = version actuelle
version INTEGER DEFAULT 1,                  -- Numéro de version

-- Métadonnées
date_chargeMENT TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

-- Contraintes
CONSTRAINT unique_client_version UNIQUE (client_id, version);

-- Index pour performance
CREATE INDEX idx_dim_client_id ON dim_client(client_id);
CREATE INDEX idx_dim_client_courant ON dim_client(est_courant);

SELECT 'Table dim_client créée avec succès' as résultat;

```

Exécutez :

```

psql -U postgres -d retailpro_dwh -f
02_create_dim_client_scd2.sql

```

## 5.4 Script Python de chargement SCD Type 2 - EXPLIQUÉ LIGNE PAR LIGNE

11

### Créer le script de chargement SCD Type 2

Créez un fichier `03_load_scd_type2.py` :

```
# =====#
# Script : Chargement SCD Type 2 pour dim_client
# Description : Charge et historise les clients depuis la
# =====#
#
# =====#
# PARTIE 1 : IMPORTS
# =====#
import psycopg2 # Bibliothèque pour se connecter à PostgreSQL
from datetime import datetime, date

# EXPLICATION :
# - psycopg2 est un driver Python NATIF pour PostgreSQL
# - Différent de JDBC (qui est pour Java/Spark)
# - Ici on utilise Python pur, pas PySpark

print("Bibliothèques importées")

# =====#
# PARTIE 2 : CONFIGURATION DE LA CONNEXION
# =====#
# Paramètres de connexion à PostgreSQL
DB_CONFIG = {
    'host': 'localhost',          # Serveur PostgreSQL (votre
    'port': 5432,                 # Port par défaut de PostgreSQL
    'database': 'retailpro_dwh',  # Nom de la base de données
    'user': 'postgres',           # Utilisateur PostgreSQL
    'password': 'VOTRE_MOT_DE_PASSE' # IMPORTANT : Mettez votre mot de passe ici
}

# EXPLICATION :
# Ce dictionnaire contient toutes les informations nécessaires
# pour que Python se connecte à PostgreSQL

print(f"Configuration : connexion à {DB_CONFIG['database']}")

# =====#
# PARTIE 3 : FONCTION DE CONNEXION
# =====#
```

```
def creer_connexion():
    """
    Crée une connexion à PostgreSQL
    Returns: objet connexion ou None si erreur
    """
    try:
        # Tentative de connexion avec les paramètres
        conn = psycopg2.connect(**DB_CONFIG)
        # Le ** "déplie" le dictionnaire comme arguments
        # Équivalent à : psycopg2.connect(host='localhost'
                                           port=5432,
                                           database='clients',
                                           user='postgres',
                                           password='password')

        print("✓ Connexion à PostgreSQL réussie")
        return conn
    except Exception as e:
        # Si erreur (mot de passe incorrect, PostgreSQL absent)
        print(f"✗ Erreur de connexion : {e}")
        return None

# =====
# PARTIE 4 : FONCTION DE LECTURE DES CLIENTS SOURCE
# =====

def lire_clients_source(conn):
    """
    Lit tous les clients depuis la table source
    Args: conn = connexion PostgreSQL
    Returns: liste de tuples (client_id, nom, prenom, email, ville, segment)
    """
    # Créer un curseur pour exécuter des requêtes
    cur = conn.cursor()
    # EXPLICATION :
    # Un curseur est comme un pointeur qui permet d'exécuter des requêtes

    # Requête SQL pour lire les clients
    sql = """
        SELECT client_id, nom, prenom, email, ville, segment
        FROM clients_source
        ORDER BY client_id
    """

    # Exécuter la requête
    cur.execute(sql)
```

```
# EXPLICATION :
# execute() envoie la requête SQL à PostgreSQL

# Récupérer tous les résultats
clients = cur.fetchall()

# EXPLICATION :
# fetchall() récupère TOUTES les lignes retournées par la requête
# Retourne une liste de tuples : [(1, 'Dupont', 'Jean')]

cur.close()

print(f"✓ {len(clients)} clients lus depuis la source")
return clients

# =====
# PARTIE 5 : VÉRIFIER SI UN CLIENT EXISTE DÉJÀ
# =====

def client_existe_dans_dimension(conn, client_id):
    """
    Vérifie si un client existe déjà dans la dimension
    Args:
        conn = connexion PostgreSQL
        client_id = ID du client à chercher
    Returns:
        tuple (données client) si trouvé
        None si pas trouvé
    """
    cur = conn.cursor()

    # Chercher la version COURANTE du client
    sql = """
        SELECT client_key, client_id, nom, prenom, email,
        FROM dim_client
        WHERE client_id = %s
        AND est_courant = TRUE
    """
    # EXPLICATION :
    # - %s est un placeholder (emplacement) pour éviter les injections SQL
    # - WHERE client_id = %s : cherche ce client spécifique
    # - AND est_courant = TRUE : seulement la version active
```

```
        cur.execute(sql, (client_id,))
    # EXPLICATION :
    # - Le tuple (client_id,) remplace le %s dans la requête
    # - La virgule après client_id est IMPORTANTE pour créer un tuple

    resultat = cur.fetchone()
    # EXPLICATION :
    # fetchone() récupère UNE SEULE ligne (la première)
    # Retourne None s'il n'y a pas de résultat

    cur.close()
    return resultat

# =====
# PARTIE 6 : DÉTECTER LES CHANGEMENTS
# =====

def detecter_changement(client_source, client_dimension):
    """
    Compare les données source et dimension pour détecter les changements.

    Args:
        client_source = tuple depuis clients_source
        client_dimension = tuple depuis dim_client

    Returns:
        True si changement détecté, False sinon
    """
    # Extraire les valeurs à comparer
    # client_source : (client_id, nom, prenom, email, ville, segment)
    # Indices :          0           1           2           3           4

    source_email = client_source[3]
    source_ville = client_source[4]
    source_segment = client_source[5]

    # client_dimension : (client_key, client_id, nom, prenom, email, ville, segment)
    # Indices :          0           1           2           3           4           5           6

    dim_email = client_dimension[4]
    dim_ville = client_dimension[5]
    dim_segment = client_dimension[6]

    # Comparer les attributs
```

```
if source_email != dim_email:
    print(f"  Changement détecté : email {dim_email} -")
    return True

if source_ville != dim_ville:
    print(f"  Changement détecté : ville {dim_ville} -")
    return True

if source_segment != dim_segment:
    print(f"  Changement détecté : segment {dim_segment} -")
    return True

# Aucun changement
return False

# =====
# PARTIE 7 : INSÉRER UN NOUVEAU CLIENT (Version 1)
# =====

def inserer_nouveau_client(conn, client_data):
    """
    Insère un nouveau client dans la dimension (version 1)

    Args:
        conn = connexion PostgreSQL
        client_data = tuple (client_id, nom, prenom, email, ville, segment,
                             date_debut, date_fin, est_courant, version)
    """
    cur = conn.cursor()

    sql = """
        INSERT INTO dim_client
        (client_id, nom, prenom, email, ville, segment,
         date_debut, date_fin, est_courant, version)
        VALUES (%s, %s, %s, %s, %s, %s, CURRENT_DATE, NULL,
                TRUE)
        RETURNING client_key
    """
    # EXPLICATION :
    # - INSERT INTO dim_client : insérer dans la table
    # - VALUES (%s, %s, ...) : 6 placeholders pour les 6 variables
    # - CURRENT_DATE : fonction PostgreSQL qui donne la date actuelle
    # - NULL : date_fin est NULL car c'est la version courante
    # - TRUE : est_courant = TRUE car c'est la version active
    # - 1 : version = 1 car c'est la première version
```

```
# - RETURNING client_key : PostgreSQL retourne l'ID ajouté

        cur.execute(sql, client_data)
# client_data contient les 6 valeurs qui remplacent les %s

        client_key = cur.fetchone()[0]
# EXPLICATION :
# fetchone() récupère le client_key retourné par RETURNING
# [0] prend le premier élément du tuple

        conn.commit()
# EXPLICATION :
# commit() valide la transaction dans PostgreSQL
# Sans commit(), les changements sont perdus !

        cur.close()
print(f"✓ Nouveau client inséré : client_id={client_id}")
return client_key

# =====
# PARTIE 8 : FERMER LA VERSION COURANTE
# =====

def fermer_version_courante(conn, client_key):
    """
    Ferme la version courante d'un client
    Args:
        conn = connexion
        client_key = clé de la version à fermer
    """
    cur = conn.cursor()

    sql = """
        UPDATE dim_client
        SET date_fin = CURRENT_DATE - INTERVAL '1 day',
            est_courant = FALSE
        WHERE client_key = %s
    """
# EXPLICATION :
# - UPDATE dim_client : modifier la table
# - SET date_fin = CURRENT_DATE - INTERVAL '1 day' :
#   mettre date_fin à hier (la nouvelle version commence à
#   aujourd'hui)
```

```
# - est_courant = FALSE : cette version n'est plus courante
# - WHERE client_key = %s : seulement pour cette version

        cur.execute(sql, (client_key,))
        conn.commit()
        cur.close()

        print(f"✓ Version fermée : client_key={client_key}")

# =====
# PARTIE 9 : CRÉER UNE NOUVELLE VERSION
# =====

def creer_nouvelle_version(conn, client_data, ancienne_version):
    """
    Crée une nouvelle version d'un client existant

    Args:
        conn = connexion
        client_data = nouvelles données
        ancienne_version = numéro de l'ancienne version
    """

    cur = conn.cursor()

    nouvelle_version = ancienne_version + 1

    sql = """
        INSERT INTO dim_client
        (client_id, nom, prenom, email, ville, segment,
         date_debut, date_fin, est_courant, version)
        VALUES (%s, %s, %s, %s, %s, %s, CURRENT_DATE, NULL)
        RETURNING client_key
    """
    # EXPLICATION :
    # Même chose que l'insertion, mais avec version = ancienne_version + 1

    # Créer le tuple avec la nouvelle version
    data_avec_version = client_data + (nouvelle_version,)
    # EXPLICATION :
    # Si client_data = (1, 'Dupont', 'Jean', ...)
    # Alors client_data + (2,) = (1, 'Dupont', 'Jean', ..., 2)

    cur.execute(sql, data_avec_version)
```

```
client_key = cur.fetchone()[0]
conn.commit()
cur.close()

print(f"✓ Nouvelle version créée : client_id={client_id}")
return client_key

# =====
# PARTIE 10 : FONCTION PRINCIPALE DE TRAITEMENT
# =====

def traiter_scd_type2():
    """
    Fonction principale qui orchestre tout le processus SCD type 2
    """
    print("\n" + "="*70)
    print("DÉBUT DU PROCESSUS SCD TYPE 2")
    print("=".join(["="]*70) + "\n")

    # Étape 1 : Se connecter à PostgreSQL
    conn = creer_connexion()
    if not conn:
        print("Impossible de continuer sans connexion")
        return

    # Étape 2 : Lire les clients depuis la source
    clients_source = lire_clients_source(conn)

    # Compteurs pour statistiques
    nb_nouveaux = 0
    nb_nouvelles_versions = 0
    nb_inchanges = 0

    # Étape 3 : Traiter chaque client
    for client in clients_source:
        client_id = client[0]
        print(f"\nTraitement du client {client_id}...")

        # Vérifier si le client existe déjà
        client_dim = client_existe_dans_dimension(conn, client_id)

        if client_dim is None:
```

```

        # CAS 1 : Nouveau client
        print(f"→ Nouveau client")
        inserer_nouveau_client(conn, client)
        nb_nouveaux += 1

    else:
        # CAS 2 : Client existant - vérifier changement
        if detecter_changement(client, client_dim):
            print(f"→ Changement détecté")
            # Fermer l'ancienne version
            fermer_version_courante(conn, client_dim[0])
            # Créer la nouvelle version
            creer_nouvelle_version(conn, client, client_dim[1])
            nb_nouvelles_versions += 1
        else:
            print(f"→ Aucun changement")
            nb_inchanges += 1

    # Étape 4 : Afficher les statistiques
    print("\n" + "="*70)
    print("RÉSUMÉ DU CHARGEMENT")
    print("="*70)
    print(f"Nouveaux clients insérés : {nb_nouveaux}")
    print(f"Nouvelles versions créées : {nb_nouvelles_versions}")
    print(f"Clients inchangés : {nb_inchanges}")
    print(f"Total traité : {len(clients_source)}")
    print("="*70)

    # Fermer la connexion
    conn.close()
    print("\nConnexion fermée")

# =====
# PARTIE 11 : POINT D'ENTRÉE DU SCRIPT
# =====

if __name__ == "__main__":
    # EXPLICATION :
    # Cette ligne vérifie si le script est exécuté directement
    # (et non importé comme module)

    traiter_scd_type2()

```

```
    print("\n✓ Script terminé avec succès")
```

## Exécution :

1. Modifiez le mot de passe dans DB\_CONFIG
2. Exécutez :

```
python 03_load_scd_type2.py
```

## Résultat attendu :

```
Bibliothèques importées
Configuration : connexion à retailpro_dwh sur localhost
=====
DÉBUT DU PROCESSUS SCD TYPE 2
=====

✓ Connexion à PostgreSQL réussie
✓ 3 clients lus depuis la source

Traitement du client 1...
→ Nouveau client
✓ Nouveau client inséré : client_id=1, client_key=1

Traitement du client 2...
→ Nouveau client
✓ Nouveau client inséré : client_id=2, client_key=2

Traitement du client 3...
→ Nouveau client
✓ Nouveau client inséré : client_id=3, client_key=3

=====
RÉSUMÉ DU CHARGEMENT
=====
Nouveaux clients insérés : 3
```

```
Nouvelles versions créées : 0  
Clients inchangés : 0  
Total traité : 3  
=====
```

Connexion fermée

✓ Script terminé avec succès

## 5.5 Tester le SCD Type 2 avec des changements

12

### Simuler des changements et relancer le chargement

#### Modifier des données dans la source :

Dans pgAdmin ou psql :

```
-- Modifier la ville du client 1  
UPDATE clients_source  
SET ville = 'Lyon',  
    updated_at = CURRENT_TIMESTAMP  
WHERE client_id = 1;  
  
-- Modifier le segment du client 2  
UPDATE clients_source  
SET segment = 'Platinum',  
    updated_at = CURRENT_TIMESTAMP  
WHERE client_id = 2;
```

```
-- Vérifier  
SELECT * FROM clients_source;
```

## Relancer le script de chargement :

```
{ python 03_load_scd_type2.py
```

## Nouveau résultat attendu :

```
...  
Traitement du client 1...  
    Changement détecté : ville Paris → Lyon  
→ Changement détecté  
✓ Version fermée : client_key=1  
✓ Nouvelle version créée : client_id=1, version=2, client  
  
Traitement du client 2...  
    Changement détecté : segment Silver → Platinum  
→ Changement détecté  
✓ Version fermée : client_key=2  
✓ Nouvelle version créée : client_id=2, version=2, client  
...  
=====  
RÉSUMÉ DU CHARGEMENT  
=====  
Nouveaux clients insérés : 0  
Nouvelles versions créées : 2  
Clients inchangés : 1  
Total traité : 3  
=====
```

## Vérifier l'historique dans la base :

```
-- Voir toutes les versions du client 1  
SELECT
```

```
client_key,  
client_id,  
nom,  
ville,  
date_debut,  
date_fin,  
est_courant,  
version  
FROM dim_client  
WHERE client_id = 1  
ORDER BY version;  
  
-- Résultat attendu :  
-- client_key | client_id | nom      | ville | date_debut |  
-- -----+-----+-----+-----+-----+  
--          1 |         1 | Dupont | Paris | 2025-01-01 |  
--          4 |         1 | Dupont | Lyon  | 2025-01-02 |
```

## Checkpoint Section 5

Vérifiez que vous comprenez :

- Ce qu'est un SCD Type 2 et pourquoi c'est utile
- Comment Python se connecte à PostgreSQL avec psycopg2
- La logique de détection de changements
- Comment créer une nouvelle version et fermer l'ancienne
- L'historique complet est bien conservé dans la base

## 6. Data Vault 2.0 - Modélisation Agile

### 6.1 Comprendre Data Vault

#### Qu'est-ce que Data Vault ?

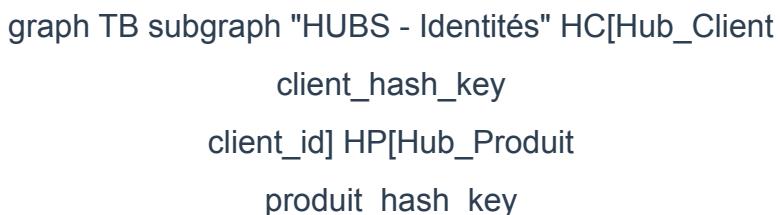
Data Vault est une méthode de modélisation qui sépare les données en 3 types de tables :

1. **HUBS** : Contiennent les identifiants uniques des entités (clients, produits, commandes)
2. **LINKS** : Représentent les relations entre les hubs (vente = client + produit)
3. **SATELLITES** : Contiennent tous les attributs descriptifs avec historique

#### Analogie simple :

- **Hub** = Une carte d'identité (juste l'ID)
- **Link** = Un certificat de mariage (relie 2 personnes)
- **Satellite** = Votre CV complet avec historique

#### Architecture Data Vault



```

produit_id] end subgraph "LINKS - Relations" LV[Link_Vente
    vente_hash_key
    client_hash_key
    produit_hash_key
vente_id] end subgraph "SATELLITES - Attributs" SC[Sat_Client
    client_hash_key
    nom, prenom, email...] SP[Sat_Produit
    produit_hash_key
    nom, prix...] SV[Sat_Vente
    vente_hash_key
montant, quantite...] end HC -->|1:N| SC HP -->|1:N| SP HC -->|1:N|
    LV HP -->|1:N| LV LV -->|1:N| SV style HC fill:#4da6ff style HP
    fill:#4da6ff style LV fill:#ff9966 style SC fill:#99cc99 style SP
    fill:#99cc99 style SV fill:#99cc99

```

## 6.2 Créer les tables Data Vault

### 13   Créer Hub\_Client

Créez un fichier `04_create_data_vault.sql` :

```

-- =====
-- HUB CLIENT
-- =====

CREATE TABLE hub_client (
    -- Hash MD5 de la clé naturelle (client_id)
    client_hash_key VARCHAR(32) PRIMARY KEY,

    -- Clé naturelle (business key)
    client_id INTEGER NOT NULL UNIQUE,

```

```
-- Métadonnées Data Vault
load_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
record_source VARCHAR(50) DEFAULT 'OLTP'
);

-- EXPLICATION :
-- client_hash_key : MD5(client_id) pour anonymiser et stocker l'identifiant
-- client_id : la vraie clé métier
-- load_date : quand cette entité est apparue la première fois
-- record_source : d'où vient cette donnée

-- =====
-- HUB PRODUIT
-- =====

CREATE TABLE hub_produit (
    produit_hash_key VARCHAR(32) PRIMARY KEY,
    produit_id INTEGER NOT NULL UNIQUE,
    load_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    record_source VARCHAR(50) DEFAULT 'OLTP'
);

-- =====
-- LINK VENTE (relation Client-Produit)
-- =====

CREATE TABLE link_vente (
    -- Hash MD5 composé de vente_id + client_id + produit_id
    vente_hash_key VARCHAR(32) PRIMARY KEY,
    -- Références aux hubs
    client_hash_key VARCHAR(32) NOT NULL,
    produit_hash_key VARCHAR(32) NOT NULL,
    -- Clé naturelle de la transaction
    vente_id INTEGER NOT NULL,
    -- Métadonnées
    load_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    record_source VARCHAR(50) DEFAULT 'OLTP',
    -- Contraintes
    FOREIGN KEY (client_hash_key) REFERENCES hub_client(client_hash_key),
    FOREIGN KEY (produit_hash_key) REFERENCES hub_produit(produit_hash_key)
);
```

```
        FOREIGN KEY (produit_hash_key) REFERENCES hub_produit
    ) ;

-- =====
-- SATELLITE CLIENT
-- =====

CREATE TABLE sat_client (
    -- Clés composites (PK)
    client_hash_key VARCHAR(32) NOT NULL,
    load_date TIMESTAMP NOT NULL,
    PRIMARY KEY (client_hash_key, load_date),

    -- Attributs descriptifs
    nom VARCHAR(100),
    prenom VARCHAR(100),
    email VARCHAR(200),
    ville VARCHAR(100),
    segment VARCHAR(50),

    -- Métadonnées
    load_end_date TIMESTAMP,    -- NULL = version actuelle
    record_source VARCHAR(50),
    hash_diff VARCHAR(32),      -- Hash de tous les attribu·

    -- Contrainte
    FOREIGN KEY (client_hash_key) REFERENCES hub_client(client_hash_key)
) ;

-- EXPLICATION :
-- Clé primaire composite : (client_hash_key, load_date)
-- Permet plusieurs versions du même client à différentes
-- dates. hash_diff : pour détecter rapidement si les données on·

-- =====
-- SATELLITE PRODUIT
-- =====

CREATE TABLE sat_produit (
    produit_hash_key VARCHAR(32) NOT NULL,
    load_date TIMESTAMP NOT NULL,
    PRIMARY KEY (produit_hash_key, load_date),

    nom_produit VARCHAR(200),
```

```

        categorie VARCHAR(100),
        prix_unitaire DECIMAL(10,2),
        cout_achat DECIMAL(10,2),

        load_end_date TIMESTAMP,
        record_source VARCHAR(50),
        hash_diff VARCHAR(32),

        FOREIGN KEY (produit_hash_key) REFERENCES hub_produit
    ) ;

-- =====
-- SATELLITE VENTE
-- =====

CREATE TABLE sat_vente (
    vente_hash_key VARCHAR(32) NOT NULL,
    load_date TIMESTAMP NOT NULL,
    PRIMARY KEY (vente_hash_key, load_date),

    date_vente TIMESTAMP,
    quantite INTEGER,
    montant_total DECIMAL(10,2),

    load_end_date TIMESTAMP,
    record_source VARCHAR(50),
    hash_diff VARCHAR(32),

    FOREIGN KEY (vente_hash_key) REFERENCES link_vente(vente_hash_key)
) ;

SELECT 'Tables Data Vault créées avec succès' as resultat;

```

**Exécutez :**

```

{
    psql -U postgres -d retailpro_dwh -f
    04_create_data_vault.sql
}

```

## Checkpoint Section 6

Vérifiez que :

- Vous comprenez la différence entre Hub, Link et Satellite
- Toutes les tables Data Vault sont créées
- Vous comprenez pourquoi on utilise des hash keys

La section Data Vault est complétée. Passons au Lakehouse avec PySpark !

## 7. Architecture Lakehouse avec PySpark et Delta Lake

### 7.1 Comprendre l'Architecture Lakehouse

#### Les 3 couches du Lakehouse

Couche	Rôle	Exemple
<b>BRONZE</b>	Stocker les données BRUTES telles qu'elles arrivent	Copie exacte de PostgreSQL, sans modification
<b>SILVER</b>	NETTOYER et VALIDER les données	Emails validés, doublons supprimés, types corrects

**GOLD**

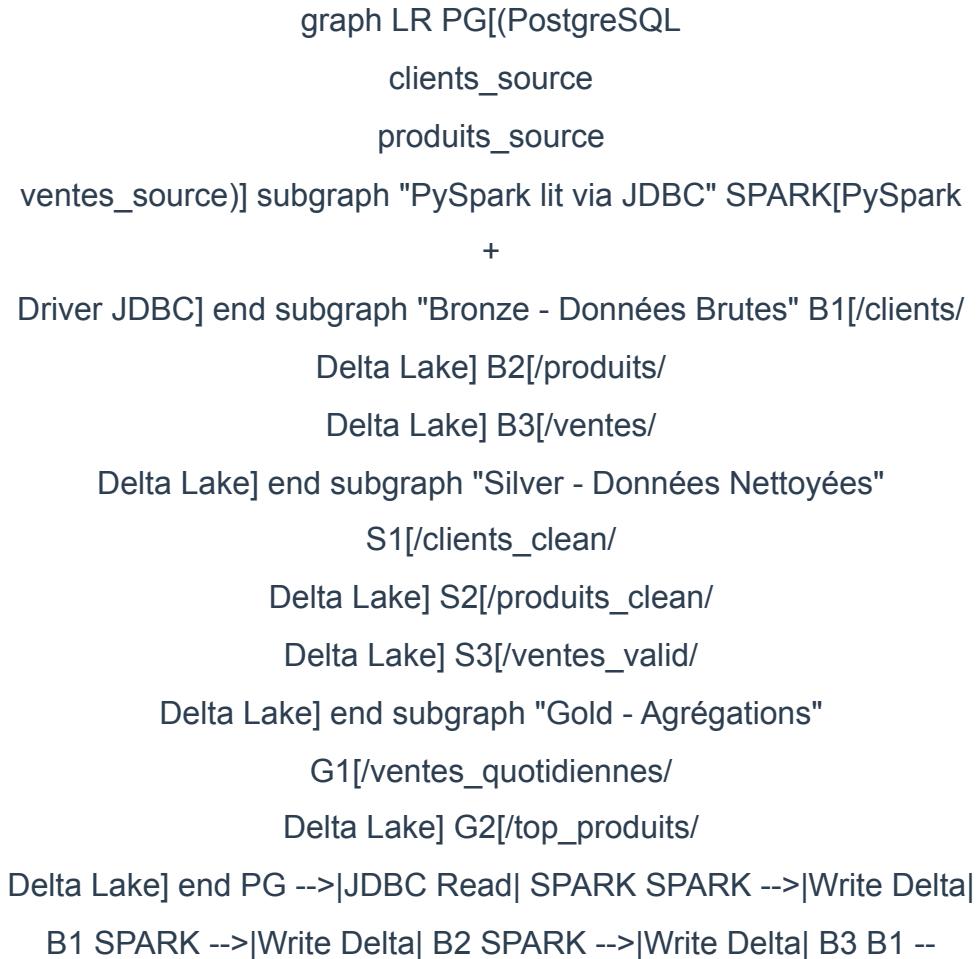
AGRÉGER pour l'analyse

Ventes par jour, Top produits,  
KPIs métier

#### Analogie de cuisine :

- **Bronze** = Ingrédients bruts du supermarché
- **Silver** = Légumes lavés, épluchés, découpés
- **Gold** = Plat préparé prêt à servir

## Pipeline Complet : PostgreSQL → Lakehouse



```
>|Transform| S1 B2 -->|Transform| S2 B3 -->|Transform| S3 S1 --  
>|Aggregate| G1 S2 -->|Aggregate| G1 S3 -->|Aggregate| G1 S2 --  
>|Aggregate| G2 style PG fill:#ff9999 style SPARK fill:#99ccff style B1  
    fill:#cd853f style B2 fill:#cd853f style B3 fill:#cd853f style S1  
    fill:#c0c0c0 style S2 fill:#c0c0c0 style S3 fill:#c0c0c0 style G1  
    fill:#ffd700 style G2 fill:#ffd700
```

## 7.2 Configuration de l'Environnement Lakehouse

### 14   Créer les répertoires pour le Lakehouse

**Windows :**

```
{ mkdir C:\lakehouse mkdir C:\lakehouse\bronze mkdir  
C:\lakehouse\silver mkdir C:\lakehouse\gold
```

**macOS/Linux :**

```
{ mkdir -p ~/lakehouse/bronze mkdir -p ~/lakehouse/silver  
mkdir -p ~/lakehouse/gold
```

**Pourquoi ces dossiers ?**

Delta Lake stocke les données dans des fichiers sur le disque. Ces dossiers vont contenir vos tables au format Delta (fichiers Parquet + logs de transaction).

## 7.3 BRONZE : Ingestion depuis PostgreSQL - EXPLIQUÉ LIGNE PAR LIGNE

### 15 Script d'Ingestion Bronze

Créez un fichier `05_bronze_ingestion.py` :

```
# =====#
# Script : Ingestion Bronze depuis PostgreSQL vers Delta Lake
# Description : Lit PostgreSQL avec PySpark et écrit en Delta Lake
# =====#
# =====#
# PARTIE 1 : IMPORTS
# =====#
from pyspark.sql import SparkSession
# SparkSession : point d'entrée principal de PySpark

from delta import configure_spark_with_delta_pip
# Fonction qui configure Spark pour utiliser Delta Lake

from pyspark.sql.functions import current_timestamp, lit
# current_timestamp : fonction Spark pour la date/heure actuelle
# lit : fonction Spark pour créer une colonne avec une valeur

print("✓ Bibliothèques importées")
```

```
# =====
# PARTIE 2 : CONFIGURATION POSTGRESQL
# =====

# Informations de connexion à PostgreSQL
POSTGRES_CONFIG = {
    'url': 'jdbc:postgresql://localhost:5432/retailpro_dwh',
    # EXPLICATION :
    # - jdbc:postgresql:// = préfixe JDBC pour PostgreSQL
    # - localhost = serveur (votre machine)
    # - 5432 = port PostgreSQL
    # - retailpro_dwh = nom de la base de données

    'user': 'postgres',
    # Utilisateur PostgreSQL

    'password': 'VOTRE_MOT_DE_PASSE', # CHANGEZ ICI
    # Mot de passe PostgreSQL

    'driver': 'org.postgresql.Driver'
    # Classe Java du driver JDBC PostgreSQL
}

# Chemins des couches Lakehouse
# IMPORTANT : Adaptez selon votre système (Windows = C:/lakehouse)
BRONZE_PATH = 'C:/lakehouse/bronze' # Windows
# BRONZE_PATH = '/home/votre_user/lakehouse/bronze' # Linux

print(f"✓ Configuration définie - Bronze path: {BRONZE_PATH}")

# =====
# PARTIE 3 : CRÉATION DE LA SESSION SPARK
# =====

def creer_session_spark():
    """
    Crée et configure une session Spark avec Delta Lake et DataFrames
    Returns: SparkSession configurée
    """
    print("\nCréation de la session Spark...")
```

```
# Étape 1 : Créer le builder
builder = SparkSession.builder \
    .appName("Bronze Ingestion") \
    .master("local[*]")
# EXPLICATION :
# - builder = constructeur de session Spark
# - .appName("...") = nom de l'application (pour identifier la session)
# - .master("local[*]") = mode local, utilise tous les noyaux disponibles

# Étape 2 : Ajouter le driver JDBC PostgreSQL
builder = builder.config(
    "spark.jars",
    "C:/TP_DataWarehouse/drivers/postgresql-42.7.1.jar"
)
# EXPLICATION :
# - spark.jars = liste des fichiers .jar à charger au démarrage de la session
# - PostgreSQL driver .jar permet à Spark de se connecter à une base de données PostgreSQL
# - IMPORTANT : Utilisez le chemin ABSOLU vers votre driver JDBC

# Étape 3 : Configurer Delta Lake
builder = builder \
    .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension") \
    .config("spark.sql.catalog.spark_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")
# EXPLICATION :
# Ces 2 configs activent Delta Lake dans Spark
# - extensions = ajoute les fonctions Delta (MERGE, TRUNCATE, etc.)
# - catalog = utilise le catalogue Delta pour gérer les tables

# Étape 4 : Créer la session avec Delta Lake
spark = configure_spark_with_delta_pip(builder).getOrCreate()
# EXPLICATION :
# - configure_spark_with_delta_pip() = finalise la configuration du builder
# - .getOrCreate() = crée une nouvelle session OU réutilise la précédente

# Réduire les logs
spark.sparkContext.setLogLevel("WARN")
# EXPLICATION :
# Par défaut, Spark affiche BEAUCOUP de logs
# WARN = afficher seulement les warnings et erreurs

print(f"✓ Session Spark créée - Version: {spark.version}")
return spark
```

```
# =====
# PARTIE 4 : LECTURE DEPUIS POSTGRESQL
# =====

def lire_table_postgres(spark, nom_table):
    """
    Lit une table depuis PostgreSQL avec PySpark

    Args:
        spark: Session Spark
        nom_table: Nom de la table à lire (ex: 'clients_se

    Returns:
        DataFrame PySpark contenant les données de la tabl
    """
    print(f"\nLecture de la table '{nom_table}' depuis Po

    # Lire avec JDBC
    df = spark.read \
        .format("jdbc") \
        .option("url", POSTGRES_CONFIG['url']) \
        .option("dbtable", nom_table) \
        .option("user", POSTGRES_CONFIG['user']) \
        .option("password", POSTGRES_CONFIG['password']) \
        .option("driver", POSTGRES_CONFIG['driver']) \
        .load()

    # EXPLICATION DÉTAILLÉE :
    #
    # spark.read = commence une opération de lecture
    #
    # .format("jdbc") = dit à Spark "utilise JDBC pour li
    #     JDBC = Java Database Connectivity
    #     Permet à Spark (Java/Scala) de communiquer avec Po
    #
    # .option("url", ...) = adresse de la base de données
    #     Format : jdbc:postgresql://host:port/database
    #
    # .option("dbtable", nom_table) = quelle table lire
    #     Spark va exécuter : SELECT * FROM nom_table
    #
```

```

# .option("user", ...) = nom d'utilisateur PostgreSQL
#
# .option("password", ...) = mot de passe PostgreSQL
#
# .option("driver", "org.postgresql.Driver") = classe
#   Cette classe est dans le fichier .jar qu'on a ajouté
#
# .load() = EXÉCUTE la lecture
#   C'est ici que Spark se connecte réellement à PostgreSQL
#   et charge les données en mémoire
#
# RÉSULTAT :
# df = DataFrame PySpark contenant toutes les lignes de la table

nb_lignes = df.count()
# .count() = compte le nombre de lignes
# ATTENTION : C'est une ACTION, Spark va exécuter la requête

print(f"✓ {nb_lignes} lignes lues depuis '{nom_table}'")

# Afficher le schéma (structure des colonnes)
print(f"  Schéma de {nom_table}:")
df.printSchema()
# EXPLICATION :
# printSchema() affiche :
# - Le nom de chaque colonne
# - Le type de données (string, integer, decimal, etc)
# - Si la colonne peut être NULL

return df

# =====
# PARTIE 5 : AJOUT DE MÉTADONNÉES
# =====

def ajouter_metadata(df, nom_table_source):
    """
    Ajoute des colonnes de métadonnées au DataFrame

    Args:
        df: DataFrame à enrichir
        nom_table_source: Nom de la table source
    """

```

```

    Returns:
        DataFrame avec colonnes métadonnées ajoutées
    """
    # Ajouter 3 colonnes de métadonnées
    df_enrichi = df \
        .withColumn("ingestion_timestamp", current_timestamp())
        .withColumn("source_system", lit("PostgreSQL")) \
        .withColumn("source_table", lit(nom_table_source))

    # EXPLICATION DÉTAILLÉE :
    #
    # .withColumn(nom_colonne, valeur) = ajoute ou modifie une colonne
    #
    # 1. ingestion_timestamp = current_timestamp()
    #     - Ajoute une colonne avec la date/heure actuelle
    #     - Permet de savoir QUAND les données ont été ingérées
    #
    # 2. source_system = lit("PostgreSQL")
    #     - lit() crée une valeur constante
    #     - Toutes les lignes auront "PostgreSQL" dans cette colonne
    #     - Utile si vous avez plusieurs sources (PostgreSQL, MySQL, etc.)
    #
    # 3. source_table = lit(nom_table_source)
    #     - Nom de la table d'origine
    #     - Ex: "clients_source", "produits_source"
    #

    # RÉSULTAT :
    # Le DataFrame a maintenant 3 colonnes supplémentaires :
    # | ... colonnes originales ... | ingestion_timestamp | source_system | source_table |

    print(f"✓ Métadonnées ajoutées (3 colonnes)")
    return df_enrichi

# =====
# PARTIE 6 : ÉCRITURE EN DELTA LAKE
# =====

def ecrire_bronze_delta(df, nom_table):
    """
    Écrit un DataFrame en format Delta Lake dans Bronze

```

```
Args:
```

```
    df: DataFrame à écrire
    nom_table: Nom pour le dossier Delta (ex: 'clients')
"""

chemin_complet = f"{BRONZE_PATH}/{nom_table}"

print(f"\nÉcriture en Delta Lake : {chemin_complet}")

# Écrire en format Delta
df.write \
    .format("delta") \
    .mode("overwrite") \
    .save(chemin_complet)

# EXPLICATION DÉTAILLÉE :
#
# df.write = commence une opération d'écriture
#
# .format("delta") = utilise le format Delta Lake
#     Delta Lake est un format de stockage qui ajoute :
#         - Transactions ACID (Atomicité, Cohérence, Isolation)
#         - Versioning automatique (Time Travel)
#         - Schema enforcement (validation du schéma)
#         - Sous le capot, Delta utilise Parquet + un fichier log
#
# .mode("overwrite") = mode d'écriture
#     Options :
#         - "overwrite" : remplace toutes les données existantes
#         - "append" : ajoute aux données existantes
#         - "error" : erreur si les données existent déjà
#         - "ignore" : ne fait rien si les données existent
#
# .save(chemin_complet) = EXÉCUTE l'écriture
#     Crée un dossier avec :
#         - Fichiers .parquet (données)
#         - Dossier _delta_log/ (historique des transactions)
#
# RÉSULTAT SUR DISQUE :
# C:/lakehouse/bronze/clients/
#     ├── part-00000-xxx.snappy.parquet  (données)
#     ├── part-00001-xxx.snappy.parquet
#     └── _delta_log/
```

```
# └── 00000000000000000000.json    (log de transaction)

nb_lignes = df.count()
print(f"✓ {nb_lignes} lignes écrites en Delta Lake")
print(f"  Emplacement : {chemin_complet}")

# =====
# PARTIE 7 : FONCTION PRINCIPALE
# =====

def main():
    """
    Fonction principale qui orchestre tout le processus
    """
    print("*"*70)
    print("INGESTION BRONZE : PostgreSQL → Delta Lake")
    print("*"*70)

    try:
        # Étape 1 : Créer la session Spark
        spark = creer_session_spark()

        # Étape 2 : Liste des tables à ingérer
        tables = ['clients_source', 'produits_source', 'ventes_source']

        # Étape 3 : Traiter chaque table
        for table_source in tables:
            print(f"\n{'='*70}")
            print(f"Traitement de : {table_source}")
            print(f"{'='*70}")

            # Lire depuis PostgreSQL
            df = lire_table_postgres(spark, table_source)

            # Ajouter les métadonnées
            df_enrichi = ajouter_metadata(df, table_source)

            # Écrire en Bronze (Delta Lake)
            nom_table_bronze = table_source.replace('_source', '_bronze')
            ecrire_bronze_delta(df_enrichi, nom_table_bronze)

            print(f"\n{'='*70}")

    except Exception as e:
        print(f"Une erreur s'est produite : {e}")
```

```
    print("✓ INGESTION BRONZE TERMINÉE AVEC SUCCÈS")
    print(f"{'='*70}")

except Exception as e:
    print(f"\n✗ ERREUR : {e}")
    import traceback
    traceback.print_exc()

finally:
    # Toujours arrêter Spark proprement
    spark.stop()
    print("\n✓ Session Spark arrêtée")

# =====
# POINT D'ENTRÉE
# =====

if __name__ == "__main__":
    main()
```

## Avant d'exécuter :

1. Modifiez `VOTRE_MOT_DE_PASSE`
2. Modifiez le chemin du driver `.jar`
3. Modifiez `BRONZE_PATH` selon votre système

## Exécution :

```
{ python 05_bronze_ingestion.py }
```

## Résultat attendu :

- ✓ Bibliothèques importées
- ✓ Configuration définie - Bronze path: C:/lakehouse/bronze

Création de la session Spark...

✓ Session Spark créée - Version: 3.5.0

```
=====
Traitement de : clients_source
=====
```

Lecture de la table 'clients\_source' depuis PostgreSQL...

Schéma de clients\_source:

root

```
|-- client_id: integer (nullable = false)
|-- nom: string (nullable = true)
|-- prenom: string (nullable = true)
|-- email: string (nullable = true)
|-- ville: string (nullable = true)
|-- segment: string (nullable = true)
|-- created_at: timestamp (nullable = true)
|-- updated_at: timestamp (nullable = true)
```

✓ 3 lignes lues depuis 'clients\_source'

✓ Métadonnées ajoutées (3 colonnes)

Écriture en Delta Lake : C:/lakehouse/bronze/clients

✓ 3 lignes écrites en Delta Lake

Emplacement : C:/lakehouse/bronze/clients

[... même chose pour produits et ventes ...]

```
=====
✓ INGESTION BRONZE TERMINÉE AVEC SUCCÈS
=====
```

✓ Session Spark arrêtée

## 7.4 Vérifier les données Bronze

16

## Script de vérification Bronze

Créez `06_verify_bronze.py` :

```
from pyspark.sql import SparkSession
from delta import configure_spark_with_delta_pip

# Créer session Spark simple
builder = SparkSession.builder.appName("Verify Bronze").master("local[*]")
spark = configure_spark_with_delta_pip(builder).getOrCreate()

# Lire une table Delta
df = spark.read.format("delta").load("C:/lakehouse/bronze/clients")

print("Données Bronze clients :")
df.show()

print("\nSchéma :")
df.printSchema()

print(f"\nNombre de lignes : {df.count() }")

spark.stop()
```

{ `python 06_verify_bronze.py`

## Checkpoint Section 7

Vérifiez que vous comprenez :

- Comment PySpark se connecte à PostgreSQL via JDBC

- Le rôle du driver .jar PostgreSQL
- Comment Spark lit les données avec .read.format("jdbc")
- Comment Delta Lake stocke les données (Parquet + logs)
- La différence entre psycopg2 (Python pur) et JDBC (pour Spark)
- Les dossiers Bronze contiennent vos données au format Delta

## 8. Projet Final - Intégration Complète

---

### 8.1 Objectif du Projet

#### Objectif

Créer un pipeline de données complet qui :

1. Lit les données depuis PostgreSQL
2. Les ingère dans la couche Bronze (Delta Lake)
3. Les nettoie dans la couche Silver
4. Crée des agrégations dans la couche Gold
5. Génère un rapport avec les insights clés

### 8.2 Pipeline Complet

17

## Créer le pipeline Silver (nettoyage)

Créez `07_silver_transformation.py` :

```
# =====
# Script : Transformation Bronze → Silver
# =====

from pyspark.sql import SparkSession
from delta import configure_spark_with_delta_pip
from pyspark.sql.functions import *

# Configuration
BRONZE_PATH = 'C:/lakehouse/bronze'
SILVER_PATH = 'C:/lakehouse/silver'

# Créer session Spark
builder = SparkSession.builder.appName("Silver Transformation")
spark = configure_spark_with_delta_pip(builder).getOrCreate()
spark.sparkContext.setLogLevel("WARN")

print("=*70)
print("TRANSFORMATION SILVER")
print("=*70)

# Lire Bronze
df_clients = spark.read.format("delta").load(f"{BRONZE_PATH}/clients")

# Nettoyage :
# 1. Supprimer les doublons
# 2. Standardiser les noms (majuscules)
# 3. Valider les emails
df_clients_clean = df_clients \
    .dropDuplicates(['client_id']) \
    .withColumn('nom', upper(col('nom'))) \
    .withColumn('prenom', initcap(col('prenom'))) \
    .filter(col('email').rlike('^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'))
```

```

# EXPLICATIONS :
# - dropDuplicates(['client_id']) : supprime les doublons
# - upper(col('nom')) : met le nom en MAJUSCULES
# - initcap(col('prenom')) : Met La Première Lettre En Majuscule
# - rlike(...) : filtre avec une regex pour valider le format

print(f"Clients Bronze : {df_clients.count()}")
print(f"Clients Silver (après nettoyage) : {df_clients_clean.count()}")

# Écrire en Silver
df_clients_clean.write.format("delta").mode("overwrite").save()

print("✓ Silver clients créé")

spark.stop()

```

## 18 Créer les agrégations Gold

Créez 08\_gold\_aggregation.py :

```

# =====
# Script : Création de la couche Gold
# =====

from pyspark.sql import SparkSession
from delta import configure_spark_with_delta_pip
from pyspark.sql.functions import *

SILVER_PATH = 'C:/lakehouse/silver'
GOLD_PATH = 'C:/lakehouse/gold'

builder = SparkSession.builder.appName("Gold Aggregation")
spark = configure_spark_with_delta_pip(builder).getOrCreate()

print("*" * 70)

```

```

print("CRÉATION COUCHE GOLD")
print("=*70)

# Lire Silver
df_ventes = spark.read.format("delta").load(f"{SILVER_PATH}/ventes")

# Agrégation : Ventes par jour
ventes_quotidiennes = df_ventes \
    .withColumn('date', to_date(col('date_vente'))) \
    .groupBy('date') \
    .agg(
        count('*').alias('nb_ventes'),
        sum('montant_total').alias('ca_total'),
        avg('montant_total').alias('panier_moyen')
    ) \
    .orderBy('date')

# EXPLICATIONS :
# - to_date(col('date_vente')) : convertit timestamp en date
# - groupBy('date') : regroupe par jour
# - agg(...) : applique des agrégations
#   * count('*') : compte les ventes
#   * sum('montant_total') : somme des montants
#   * avg('montant_total') : moyenne des montants
# - orderBy('date') : trie par date

ventes_quotidiennes.show()

# Écrire en Gold
ventes_quotidiennes.write.format("delta").mode("overwrite").save(f"{GOLD_PATH}/ventes_quotidiennes")

print("✓ Gold ventes_quotidiennes crééé")

spark.stop()

```

## 8.3 Générer un Rapport Final

19

## Script de génération de rapport

Créez `09_generer_rapport.py` :

```
# =====
# Script : Génération de rapport final
# =====

from pyspark.sql import SparkSession
from delta import configure_spark_with_delta_pip
from datetime import datetime

GOLD_PATH = 'C:/lakehouse/gold'

builder = SparkSession.builder.appName("Rapport Final").master("local[*]")
spark = configure_spark_with_delta_pip(builder).getOrCreate()

print("\n" + "="*70)
print("RAPPORT DATA WAREHOUSE - " + datetime.now().strftime("%d-%m-%Y"))
print("=*70 + "\n")

# Lire Gold
df_ventes_quot = spark.read.format("delta").load(f"{GOLD_PATH}/quot")

# Statistiques globales
stats = df_ventes_quot.agg(
    sum('nb_ventes').alias('total_ventes'),
    sum('ca_total').alias('ca_global'),
    avg('panier_moyen').alias('panier_moyen_global')
).collect()[0]

print("STATISTIQUES GLOBALES")
print("-"*70)
print(f"Total des ventes : {stats['total_ventes']} ")
print(f"Chiffre d'affaires : {stats['ca_global']:.2f} €")
print(f"Panier moyen : {stats['panier_moyen_global']:.2f}")

print("\n" + "="*70)
```

```
print("TOP 5 MEILLEURS JOURS")
print("=*70)
df_ventes_quot.orderBy(col('ca_total').desc()).show(5)

print("\n✓ Rapport généré avec succès")

spark.stop()

{
    python 09_générer_rapport.py
```

## 8.4 Livrables du Projet

### Ce que vous devez rendre

#### 1. Code Source

- Tous les scripts Python (.py)
- Tous les scripts SQL (.sql)

#### 2. Documentation

- README expliquant comment exécuter les scripts
- Diagrammes d'architecture
- Captures d'écran des résultats

#### 3. Rapport d'Analyse

- Insights tirés des données Gold
- Recommandations métier

## 8.5 Grille d'Évaluation

Critère	Points	Description
<b>Installation et Configuration</b>	10	PostgreSQL, Python, PySpark correctement installés
<b>SCD Type 2</b>	20	Dimension client avec historique fonctionnel
<b>Data Vault</b>	15	Hubs, Links, Satellites correctement créés
<b>Lakehouse Bronze</b>	15	Ingestion PostgreSQL vers Delta Lake
<b>Lakehouse Silver</b>	15	Transformations et nettoyage des données
<b>Lakehouse Gold</b>	15	Agrégations et tables analytiques
<b>Documentation</b>	10	Clarté, complétude, diagrammes
<b>TOTAL</b>	<b>100</b>	

## 8.6 Ressources Complémentaires

### Pour aller plus loin

- Documentation PySpark : [spark.apache.org/docs](http://spark.apache.org/docs)
- Documentation Delta Lake : [docs.delta.io](http://docs.delta.io)

- **Documentation PostgreSQL** : [postgresql.org/docs](https://postgresql.org/docs)
- **Data Vault 2.0** : [datavaultalliance.com](https://datavaultalliance.com)

## 8.7 Conclusion

Vous avez maintenant construit un Data Warehouse moderne complet avec :

- PostgreSQL comme source de données transactionnelles
- PySpark comme moteur de transformation
- Delta Lake pour le stockage ACID
- Architecture Lakehouse en 3 couches (Bronze, Silver, Gold)
- Modélisation SCD Type 2 pour l'historisation
- Data Vault 2.0 pour l'agilité

**Ces compétences sont très recherchées en entreprise !**