epsi®
l'école d'ingénierie informatique

Rapport du projet MSPR TPRE501

Bloc 1 - Développement et déploiement d'une application dans le respect du cahier des charges Client, création d'un BackEnd métier permettant le nettoyage et la visualisation des données

Certification professionnelle :

Développeur en Intelligence Artificielle et Data Science RNCP 36581

Réalisé par :

DA SILVA Aimé José
GAUNET Aline
NDONGMO DJOUMESSI Wilfrid
WAMBA Forestin

2024 – 2025

Documentation du processus de collecte et nettoyage des données

Table des matières

1. Introduction

Ce document présente de manière détaillée le processus complet de collecte, traitement et nettoyage des données mis en œuvre dans notre pipeline ETL COVID-19. Il couvre l'ensemble du cycle de vie des données, depuis leur collecte jusqu'à leur stockage final dans la base de données, en passant par les différentes étapes de transformation.

L'objectif de ce pipeline est de fournir une base de données structurée et optimisée pour l'analyse des données mondiales de la pandémie COVID-19. Ces données transformées pourront ensuite être utilisées pour alimenter des tableaux de bord.

2. Méthodologie de Collecte des Données

2.1 Sources de données

Le pipeline ETL utilise comme source principale le jeu de données "COVID-19 Global Cases" accessible publiquement (sur la plateforme Kaggle). Ces données sont fournies dans un format CSV standardisé et mises à jour régulièrement.

Source principale : - Dataset : COVID-19 Global Cases - Format : CSV - Fréquence de mise à jour : Quotidienne - Couverture géographique : Mondiale - Granularité temporelle : Journalière

Les fichiers CSV sources sont placés dans le répertoire **data/raw/** du projet avant le lancement du pipeline ETL.

2.2 Justification des choix

Plusieurs facteurs ont guidé le choix de cette source de données :

1. **Qualité et structure des données** : Les données COVID-19 mondiales sont bien structurées et organisées dans un format cohérent, ce qui facilite leur traitement automatisé.

2. **Exhaustivité** : Ces données offrent une couverture mondiale complète, un élément essentiel pour une analyse épidémiologique globale.

3. **Adaptabilité au modèle d'intelligence artificielle** : Les champs disponibles (cas confirmés, décès, guérisons, etc.) correspondent directement aux besoins d'analyse prédictive et de modélisation de l'évolution de la pandémie.

2.3 Structure des données sources

Le jeu de données COVID-19 Global Cases contient les champs suivants :

Champ	Type	Description	Exemple
Date	Date (YYYY-MM-DD)	Date de l'observation	2020-03-15
Country/Region	Texte	Pays ou région concerné	France
Confirmed	Entier	Nombre total de cas confirmés	6573

Champ	Type	Description	Exemple
Deaths	Entier	Nombre total de décès	148
Recovered	Entier	Nombre total de guérisons	12
Active	Entier	Nombre de cas actifs	6413
New cases	Entier	Nouveaux cas par rapport au jour précédent	838
New deaths	Entier	Nouveaux décès par rapport au jour précédent	12
New recovered	Entier	Nouvelles guérisons par rapport au jour précédent	0
WHO Region	Texte	Région OMS correspondante	Europe

Exemple d'entrées dans le fichier source :

Extrait des données brutes:

Extrait des données brutes SQL

```
Date,Country/Region,Confirmed,Deaths,Recovered,Active,New cases,New
deaths,New recovered,WHO Region
2020-01-22,Afghanistan,0,0,0,0,0,0,0,Eastern Mediterranean
2020-01-22,Albania,0,0,0,0,0,0,0,Europe
2020-01-22,Algeria,0,0,0,0,0,0,0,Africa
...
2020-07-27,France,183079,30209,80815,72055,2551,3,342,Europe
```

Figure 1: Extrait des données brutes

Avant la visualisation des données avec power BI, ces données doivent d'abord être exploitable.

3. Procédures de Nettoyage et Transformation

Notre processus ETL applique une série de transformations aux données brutes afin de les convertir en un format exploitable et cohérent. Cette section détaille les étapes de ce processus, leurs objectifs, et les techniques utilisées.

3.1 Étape 1 : Nettoyage initial

La première étape du processus consiste à nettoyer les données brutes pour éliminer les anomalies et inconsistances qui pourraient affecter la qualité des analyses ultérieures.

Objectifs du nettoyage

- Éliminer les lignes contenant uniquement des valeurs nulles ou zéro
- Supprimer les doublons potentiels
- Corriger les formats de données incorrects

- Remplacer les valeurs négatives par zéro (pour les métriques qui ne peuvent être négatives)
- Traiter les valeurs manquantes.

Implémentation technique

Le nettoyage est implémenté dans la classe DataCleaner qui hérite de la classe abstraite

classe DataCleaner — Python

```python
class DataCleaner(BaseTransformer):
    def transform(self, df: pd.DataFrame) -> pd.DataFrame:
        try:
            logger.info("Début du nettoyage")

            # Supprimer les lignes où toutes les colonnes numériques sont à zéro
            numeric_columns = [
                "Confirmed",
                "Deaths",
                "Recovered",
                "Active",
                "New cases",
                "New deaths",
                "New recovered",
            ]
            df = df.drop(df[(df[numeric_columns] == 0).all(axis=1)].index)
            logger.info(
                f"Lignes avec toutes les valeurs numériques à zéro supprimées: {len(df)}
lignes restantes"
            )

            # Supprimer les lignes avec plus de 50% de valeurs manquantes
            df = df.dropna(thresh=len(df.columns) * 0.5)

            # Nettoyage par type de colonne
            for column in df.columns:
                if df[column].dtype == "object":
                    # Nettoyage des chaînes
                    df[column] = df[column].str.strip().str.title()
                elif df[column].dtype in ["int64", "float64"]:
                    # Remplacer les valeurs négatives par 0
                    df[column] = df[column].clip(lower=0)

            # Suppression des doublons
            initial_rows = len(df)
            df = self.remove_duplicates(df)
            duplicates_removed = initial_rows - len(df)
            logger.info(f"Doublons supprimés: {duplicates_removed}")

            # Gestion des valeurs manquantes restantes
            numeric_columns = df.select_dtypes(include=["int64", "float64"]).columns
            text_columns = df.select_dtypes(include=["object"]).columns

            df[numeric_columns] = df[numeric_columns].fillna(0)
            df[text_columns] = df[text_columns].fillna("Non renseigné")

            return df

        except Exception as e:
            logger.error(f"Erreur nettoyage: {str(e)}")
            raise
```

Figure 2: Extrait du code la classe Datacleaner

BaseTransformer. Voici le code principal de cette étape :

Justification des choix techniques

- **Suppression des lignes avec valeurs nulles** : Les enregistrements sans données numériques n'apportent aucune valeur à l'analyse.
- **Nettoyage des chaînes en title case** : Uniformise les noms de pays et régions pour éviter les doublons dus à des variations de casse.
- **Remplacement des valeurs négatives** : Les métriques épidémiologiques ne peuvent logiquement pas être négatives, ce qui indique une erreur de saisie.
- **Stratégie de gestion des valeurs manquantes** :
  i. Valeurs numériques : remplacement par 0.
  ii. Valeurs textuelles : remplacement par "Non renseigné" pour une identification claire.

3.2 Étape 2 : Normalisation

La deuxième étape consiste à normaliser les données en convertissant les noms de colonnes et les valeurs dans un format standardisé conforme à notre modèle de données cible.

Objectifs de la normalisation

- Renommer les colonnes selon la convention française
- Standardiser les noms de pays pour correspondre aux références géographiques
- Convertir les dates dans un format uniforme
- Assurer la cohérence des types de données entre les champs
- Préparer les données pour l'agrégation ultérieure

Implémentation technique

La normalisation est implémentée dans la classe DataNormalizer :

classe DataNormalizer Python

```python
class DataNormalizer(BaseTransformer):
    def transform(self, df: pd.DataFrame) -> pd.DataFrame:
        try:
            logger.info("Début normalisation")

            # Renommage des colonnes
            column_mapping = {
                "Country/Region": "nom_pays",
                "Confirmed": "cas_confirmes",
                "Deaths": "deces",
                "Recovered": "guerisons",
                "Active": "cas_actifs",
                "New cases": "nouveaux_cas",
                "New deaths": "nouveaux_deces",
                "New recovered": "nouvelles_guerisons",
                "Date": "date_observation",
                "WHO Region": "region_oms",
            }
            df = df.rename(columns=column_mapping)

            # Standardisation des noms de pays
            df["nom_pays"] = df["nom_pays"].str.strip()

            # Mapping spécifique des noms de pays
            pays_mapping = {
                "Us": "US",
                "Cote D'Ivoire": "Cote d'Ivoire",
                "West Bank And Gaza": "West Bank and Gaza",
                # ... autres mappings
            }
            df["nom_pays"] = df["nom_pays"].replace(pays_mapping)

            # Conversion de la date
            df["date_observation"] = pd.to_datetime(df["date_observation"])

            # Nettoyage des valeurs numériques
            numeric_columns = [
                "cas_confirmes",
                "deces",
                "guerisons",
                "cas_actifs",
                "nouveaux_cas",
                "nouveaux_deces",
                "nouvelles_guerisons",
            ]

            for col in numeric_columns:
                if col in df.columns:
                    df[col] = (
                        pd.to_numeric(df[col], errors="coerce").fillna(0).astype(int)
                    )
                else:
                    logger.warning(
                        f"Colonne {col} manquante, création avec valeurs à 0"
                    )
                    df[col] = 0

            return df

        except Exception as e:
            logger.error(f"Erreur normalisation: {str(e)}")
            raise
```

Figure 3: Extrait du code pour la classe DataNormaliser

Justification des choix techniques

- **Renommage des colonnes en français** : Facilite l'utilisation par des équipes francophones et harmonise la terminologie avec le modèle de données cible.
- **Mapping des noms de pays** : Corrige les variations orthographiques et de casse pour garantir la cohérence avec les référentiels géographiques.
- **Conversion des dates en format datetime** : Permet des opérations chronologiques et des filtres temporels précis.
- **Conversion forcée en types numériques** : Assure l'homogénéité des données pour les calculs statistiques.
- **Ajout du champ 'nom_maladie'** : Prépare les données pour l'intégration dans un modèle capable de gérer plusieurs maladies (extensibilité).

3.3 Étape 3 : Agrégation

La troisième étape agrège les données normalisées pour les préparer à l'insertion dans la base de données selon notre modèle relationnel.

Objectifs de l'agrégation

- Joindre les données avec les tables de référence (pays, maladies)
- Remplacer les noms de pays par leurs identifiants correspondants
- Agréger les données selon les dimensions du modèle cible
- Préparer la structure finale pour le chargement

Implémentation technique

L'agrégation est implémentée dans la classe DataAggregator :

classe DataAggregator Python

```python
class DataAggregator(BaseTransformer):
    def transform(self, df: pd.DataFrame, pays_df: pd.DataFrame) -> pd.DataFrame:
        try:
            logger.info("Début agrégation")

            # Jointure pour obtenir id_pays
            df = df.merge(pays_df[["nom_pays", "id_pays"]], on="nom_pays", how="left")

            # Vérification détaillée de la jointure
            missing_pays = df[df["id_pays"].isna()]["nom_pays"].unique()
            if len(missing_pays) > 0:
                logger.error("Détails des pays manquants:")
                for pays in missing_pays:
                    logger.error(f"'{pays}' n'a pas été trouvé")
                    similaires = [
                        p
                        for p in pays_df["nom_pays"]
                        if p.lower().replace(" ", "") in pays.lower().replace(" ", "")
                        or pays.lower().replace(" ", "") in p.lower().replace(" ", "")
                    ]
                    if similaires:
                        logger.error(
                            f"Noms similaires trouvés dans la base: {similaires}"
                        )
                raise ValueError(
                    "Certains pays n'ont pas été trouvés dans la table de référence"
                )

            # Ajout de l'id_maladie pour COVID-19
            df["id_maladie"] = 1

            # Agrégation par ID
            aggregated = (
                df.groupby(["id_pays", "id_maladie", "date_observation"])
                .agg(
                    {
                        "cas_confirmes": "sum",
                        "deces": "sum",
                        "guerisons": "sum",
                        "cas_actifs": "sum",
                        "nouveaux_cas": "sum",
                        "nouveaux_deces": "sum",
                        "nouvelles_guerisons": "sum",
                    }
                )
                .reset_index()
            )

            logger.info(f"Agrégation terminée: {len(aggregated)} lignes")
            return aggregated

        except Exception as e:
            logger.error(f"Erreur agrégation: {str(e)}")
            raise
```

Figure 4: Extrait du code de la classe DataAggregetor

Justification des choix techniques

- **Jointure avec table pays** : Utilise les identifiants de pays plutôt que les noms pour la conformité avec le modèle relationnel.

- **Détection des pays manquants** : Vérification rigoureuse pour éviter les pertes de données dues à des problèmes de mapping.
- **Attribution d'ID maladie fixe** : Bien que notre cas actuel ne traite que de la COVID-19, cette approche permet d'étendre facilement le modèle à d'autres maladies.
- **Agrégation par groupes** : Assure l'unicité des enregistrements selon la clé composite (pays, maladie, date) et calcule correctement les totaux si plusieurs entrées existent pour la même combinaison.

3.4 Flux de transformation complet

Le processus complet de transformation suit un flux séquentiel où chaque étape produit les données d'entrée pour la suivante :

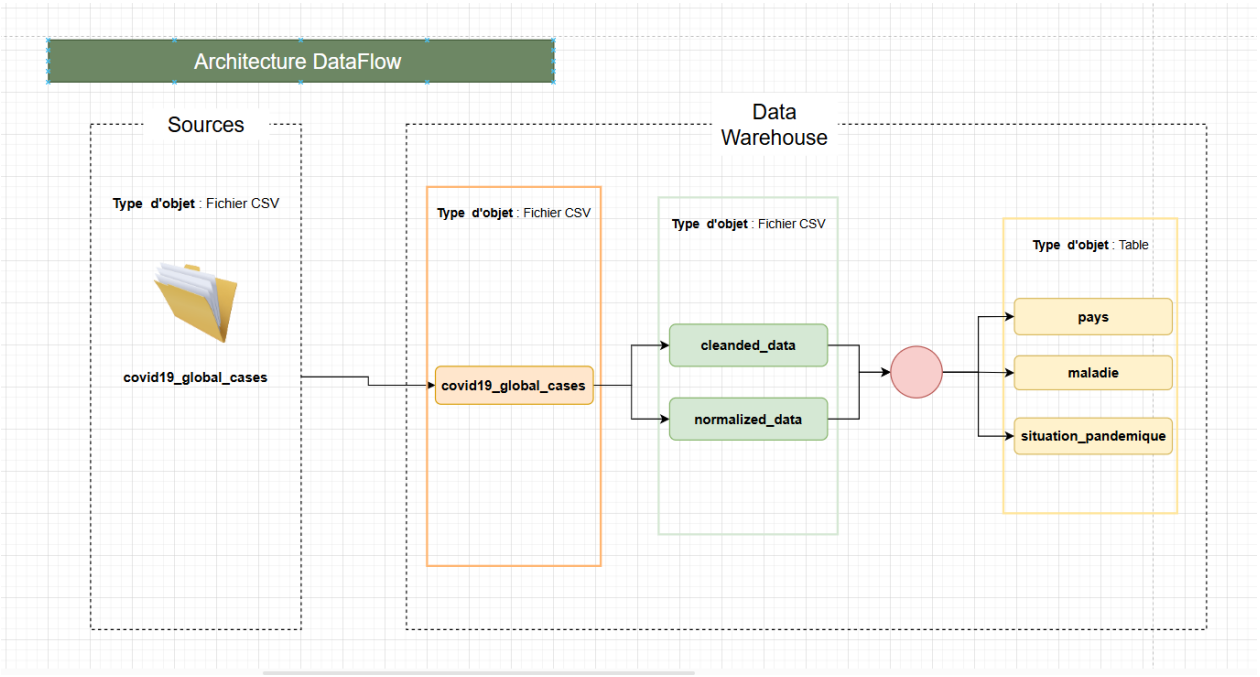

Figure 5: Diagramme du flux de transformation

Cette approche modulaire présente plusieurs avantages :

- Séparation claire des responsabilités de chaque partie du pipeline.

- Possibilité de tester et de déboger chaque étape indépendamment.

- Facilité d'extension avec de nouvelles étapes de transformation.

- Traçabilité complète grâce à la journalisation à chaque étape.

Chaque transformation est accompagnée d'une validation et d'une journalisation détaillée, permettant un suivi précis du processus et facilitant le diagnostic en cas de problème.

4. Modélisation des Données

4.1 Conception du modèle

Notre modèle de données a été conçu selon les principes de la méthode Merise, en suivant les étapes classiques de modélisation : conceptuelle (MCD), logique (MLD) et physique (MPD).

Les objectifs principaux du modèle sont :

- Organiser les données de manière normalisée pour éviter la redondance

- Assurer l'intégrité référentielle entre les entités

- Faciliter les requêtes d'analyse épidémiologique

- Permettre l'extension future à d'autres maladies que la COVID-19

- Optimiser les performances pour les requêtes fréquentes

La structure comprend deux tables de référence (pays et maladie) et une table de faits (situation_pandemique) qui stocke les données temporelles de la pandémie.

4.2 Modèle Conceptuel de Données (MCD)

Le MCD définit les entités, leurs attributs et les relations entre elles à un niveau conceptuel, indépendamment des contraintes d'implémentation.

Entités principales :

- **Pays** : Représente un pays ou une région géographique

- **Maladie** : Représente une maladie épidémique (actuellement uniquement COVID-19)

- **Situation_Pandemique** : Représente l'état de la pandémie pour un pays et une maladie donnée à une date spécifique

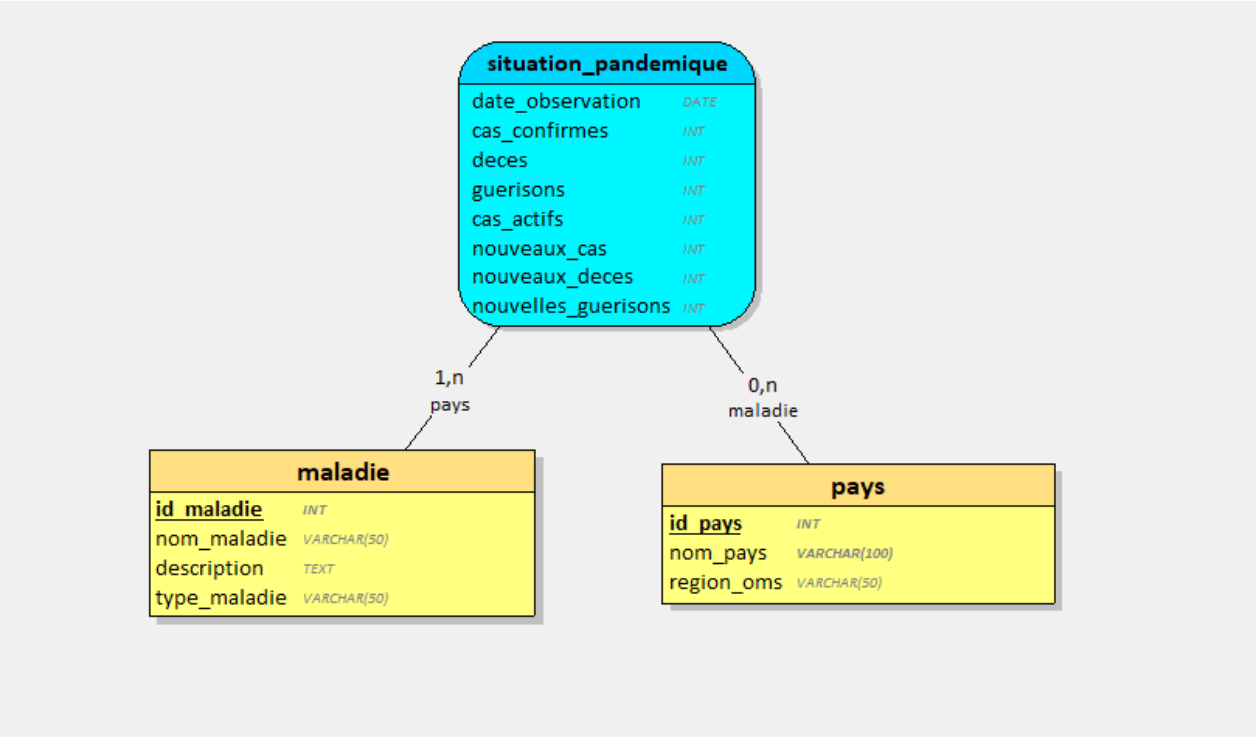

Figure 6: Diagramme du MCD

Le diagramme ci-dessus illustre les relations entre les entités et les cardinalités associées.

4.3 Modèle Logique de Données (MLD)

Le MLD traduit le modèle conceptuel en une représentation qui tient compte des contraintes du modèle relationnel.

Tables du modèle :

- **pays** : Stocke les informations sur les pays et régions
- **maladie** : Stocke les informations sur les maladies épidémiques
- **situation_pandemique** : Stocke les statistiques quotidiennes par pays et maladie

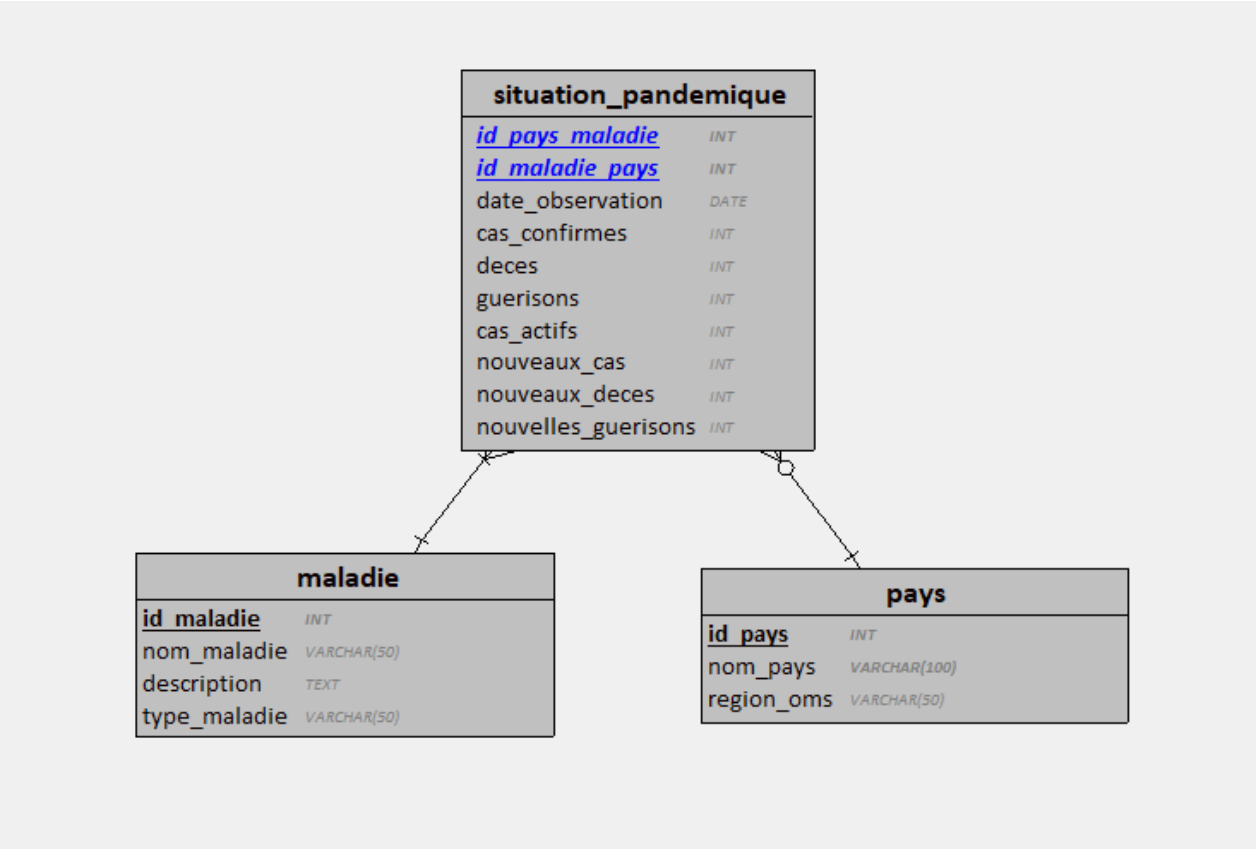

Figure 7: Diagramme du MLD

Le diagramme ci-dessus montre la structure relationnelle des tables et leurs clés primaires et étrangères.

4.4 Modèle Physique de Données (MPD)

Le MPD décrit l'implémentation technique du modèle dans PostgreSQL, avec les types de données spécifiques, contraintes et index.

```
-- Table PAYS
CREATE TABLE pays (
   id_pays SERIAL PRIMARY KEY,
   nom_pays VARCHAR(100) NOT NULL UNIQUE,
   region_oms VARCHAR(50) NOT NULL
);

-- Table MALADIE
CREATE TABLE maladie (
   id_maladie SERIAL PRIMARY KEY,
   nom_maladie VARCHAR(50) NOT NULL UNIQUE,
   type_maladie VARCHAR(50) NOT NULL,
   description TEXT
);

-- Table SITUATION_PANDEMIQUE (table de faits)
```

```
CREATE TABLE situation_pandemique (
    id_pays INTEGER,
    id_maladie INTEGER,
    date_observation DATE NOT NULL,
    cas_confirmes INTEGER DEFAULT 0,
    deces INTEGER DEFAULT 0,
    guerisons INTEGER DEFAULT 0,
    cas_actifs INTEGER DEFAULT 0,
    nouveaux_cas INTEGER DEFAULT 0,
    nouveaux_deces INTEGER DEFAULT 0,
    nouvelles_guerisons INTEGER DEFAULT 0,
    -- Clé primaire composite
    PRIMARY KEY (id_pays, id_maladie, date_observation),
    -- Clés étrangères
    FOREIGN KEY (id_pays) REFERENCES pays(id_pays),
    FOREIGN KEY (id_maladie) REFERENCES maladie(id_maladie)
);
```

4.5 Cardinalités et relations

Les relations entre les entités sont définies par les cardinalités suivantes :

1. Relation PAYS-MALADIE :

- Un PAYS peut être touché par PLUSIEURS maladies (cardinalité 0,n)
  i. Le 0 indique qu'un pays peut n'avoir aucune maladie déclarée
  ii. Le n indique qu'un pays peut avoir plusieurs maladies
- Une MALADIE peut toucher PLUSIEURS pays (cardinalité 1,n)
  i. Le 1 indique qu'une maladie doit toucher au moins un pays
  ii. Le n indique qu'une maladie peut toucher plusieurs pays

Cette modélisation permet une grande flexibilité tout en maintenant l'intégrité des données et l'efficacité des requêtes.

5. Dictionnaire des Données

Cette section fournit une description détaillée de chaque table et attribut du modèle de données.

5.1 Table pays

Table de référence contenant les informations sur les pays et régions.

Attribut	Type	Description	Exemple	Contraintes
id_pays	SERIAL	Identifiant unique du pays (auto-incrémenté)	42	Clé primaire
nom_pays	VARCHAR(100)	Nom officiel du pays	France	NOT NULL, UNIQUE
region_oms	VARCHA	Région OMS à	Europe	NOT NULL

Attribut	Type	Description	Exemple	Contraintes
	R(50)	laquelle appartient le pays		

5.2 Table maladie

Table de référence contenant les informations sur les maladies épidémiques.

Attribut	Type	Description	Exemple	Contraintes
id_maladie	SERIAL	Identifiant unique de la maladie (auto-incrémenté)	1	Clé primaire
nom_maladie	VARCHAR(50)	Nom officiel de la maladie	COVID-19	NOT NULL, UNIQUE
type_maladie	VARCHAR(50)	Classification de la maladie	Virale	NOT NULL
description	TEXT	Description détaillée de la maladie	Maladie infectieuse causée par le SARS-CoV-2	-

5.3 Table situation_pandemique

Table de faits contenant les statistiques quotidiennes par pays et maladie.

Attribut	Type	Description	Exemple	Contraintes
id_pays	INTEGER	Référence à la table pays	42	Clé étrangère, partie de la clé primaire composite
id_maladie	INTEGER	Référence à la table maladie	1	Clé étrangère, partie de la clé primaire composite
date_observation	DATE	Date de l'observation	2020-03-15	Partie de la clé primaire composite
cas_confirmes	INTEGER	Nombre total de cas confirmés à cette date	6573	DEFAULT 0
deces	INTEGER	Nombre total de décès à cette date	148	DEFAULT 0
guerisons	INTEGER	Nombre total de guérisons à cette date	12	DEFAULT 0
cas_actifs	INTEGER	Nombre de cas actifs à cette date	6413	DEFAULT 0
nouveaux_cas	INTEGER	Nouveaux cas par	838	DEFAULT 0

Attribut	Type	Description	Exemple	Contraintes
		rapport au jour précédent		
nouveaux_deces	INTEGER	Nouveaux décès par rapport au jour précédent	12	DEFAULT 0
nouvelles_guerisons	INTEGER	Nouvelles guérisons par rapport au jour précédent	0	DEFAULT 0

6. Règles de Validation et Gestion des Exceptions

Cette section décrit les règles de validation appliquées aux données et les stratégies de gestion des exceptions dans le pipeline ETL.

6.1 Validation des données d'entrée

Avant de traiter les données, le module d'extraction effectue plusieurs validations :

a. **Vérification de l'existence du fichier** : S'assure que le fichier source existe dans le chemin spécifié
b. **Validation du format CSV** : Vérifie que le fichier est au format CSV valide
c. **Vérification des colonnes requises** : S'assure que toutes les colonnes nécessaires sont présentes

Validation des données d'entrée — Python

```python
def validate_columns(self, required_columns):
    df, profile = self.extract()
    missing_columns = set(required_columns) - set(df.columns)
    if missing_columns:
        raise ValueError(f"Colonnes manquantes: {missing_columns}")
    return df, profile
```

Figure 8: Extrait du code de la function validate_columns

6.2 Gestion des valeurs manquantes

Plusieurs stratégies sont utilisées pour gérer les valeurs manquantes selon le type de données :

a. **Valeurs numériques** :
    i. Valeurs nulles → Remplacées par 0

ii. Valeurs négatives → Remplacées par 0 (métriques épidémiologiques sont toujours positives)

b. **Chaînes de caractères** :
   i. Valeurs nulles → Remplacées par “Non renseigné”
   ii. Chaînes vides ou blanches → Nettoyées et standardisées

c. **Dates**:
   i. Formats incorrects → Tentative de parsing avec des formats alternatifs

classe DataValidator — Python

```python
import pandas as pd
from datetime import datetime


class DataValidator:
    @staticmethod
    def validate_date(date_str):
        try:
            return pd.to_datetime(date_str)
        except:
            return None

    @staticmethod
    def validate_numeric(value):
        try:
            return int(value)
        except:
            return 0

    @staticmethod
    def validate_string(value, max_length=None):
        if pd.isna(value):  # isna est utilisé pour vérifier les valeurs manquantes
            return None
        value = str(value).strip()
        return value[:max_length] if max_length else value
```

Figure 9: Extrait du code de la classe DataValidator

le choix de remplacer les valeurs manquantes par 0 ou "Non renseigné" a été effectué dans un souci de continuité du dataset. Cela permet de conserver la ligne sans supprimer l’observation. Elle Permet aussi de considerer l’information de donnée absente comme une catégorie potentiellement significative.

6.3 Normalisation des noms de pays

Afin d’assurer une cohérence des noms de pays dans la base données nous avons de normaliser les nom des pays selon un mapping défini pour assurer la cohérence avec la table de référence :

1. **Standardisation de casse** : Conversion en Title Case (première lettre en majuscule)
2. **Suppression des espaces superflus** : Nettoyage des espaces en début et fin
3. **Mapping spécifique** : Table de correspondance pour les cas particuliers

4. **Détection des similitudes** : En cas d'échec de jointure, recherche de noms similaires

```
# Mapping spécifique des noms de pays
pays_mapping = {
    'Us': 'US',
    "Cote D'Ivoire": "Cote d'Ivoire",
    'West Bank And Gaza': 'West Bank and Gaza',
    'Antigua And Barbuda': 'Antigua and Barbuda',
    'Bosnia And Herzegovina': 'Bosnia and Herzegovina',
    'Saint Kitts And Nevis': 'Saint Kitts and Nevis',
    'Saint Vincent And The Grenadines': 'Saint Vincent and the Grenadines',
    'Sao Tome And Principe': 'Sao Tome and Principe',
    'Trinidad And Tobago': 'Trinidad and Tobago'
}
```

7. Scripts de Création de la Base de Données

Cette section présente les scripts SQL utilisés pour créer et configurer la base de données PostgreSQL.

7.1 Création des tables

Le script suivant crée les tables du modèle de données :

```
-- Table PAYS
CREATE TABLE pays (
    id_pays SERIAL PRIMARY KEY,
    nom_pays VARCHAR(100) NOT NULL UNIQUE,
    region_oms VARCHAR(50) NOT NULL
);

-- Table MALADIE
CREATE TABLE maladie (
    id_maladie SERIAL PRIMARY KEY,
    nom_maladie VARCHAR(50) NOT NULL UNIQUE,
    type_maladie VARCHAR(50) NOT NULL,
    description TEXT
);

-- Table SITUATION_PANDEMIQUE (table de faits)
CREATE TABLE situation_pandemique (
    id_pays INTEGER,
    id_maladie INTEGER,
    date_observation DATE NOT NULL,
    cas_confirmes INTEGER DEFAULT 0,
    deces INTEGER DEFAULT 0,
    guerisons INTEGER DEFAULT 0,
    cas_actifs INTEGER DEFAULT 0,
    nouveaux_cas INTEGER DEFAULT 0,
    nouveaux_deces INTEGER DEFAULT 0,
    nouvelles_guerisons INTEGER DEFAULT 0,
    -- Clé primaire composite
    PRIMARY KEY (id_pays, id_maladie, date_observation),
```

```
    -- Clés étrangères
    FOREIGN KEY (id_pays) REFERENCES pays(id_pays),
    FOREIGN KEY (id_maladie) REFERENCES maladie(id_maladie)
);
```

7.2 Justification des choix techniques

Plusieurs choix techniques ont été faits pour optimiser la base de données :

a. **Utilisation de SERIAL** : Type auto-incrémenté de PostgreSQL qui crée automatiquement une séquence, garantissant l'unicité des identifiants et simplifiant l'insertion.

b. **Valeurs par défaut** : Les compteurs statistiques ont une valeur par défaut de 0, ce qui simplifie les requêtes et évite les valeurs NULL qui compliqueraient les calculs.

c. **Contrainte UNIQUE** : Appliquée aux noms de pays et de maladies pour éviter les doublons et maintenir l'intégrité des données.

d. **Index stratégiques** : Créés sur les colonnes fréquemment utilisées dans les clauses WHERE et les jointures pour accélérer les requêtes.

e. **Contraintes NOT NULL** : Empêchent l'insertion de valeurs NULL dans les champs critiques, garantissant la qualité des données.

f. **Clés étrangères** : Assurent l'intégrité référentielle entre les tables, empêchant la suppression de données référencées.

g. **Clé primaire composite** : La table situation_pandemique utilise une clé primaire composée de (id_pays, id_maladie, date_observation), définissant de manière unique chaque observation.

8. Exemples et Cas d'Utilisation

Cette section illustre comment utiliser les données transformées pour répondre à des questions analytiques concrètes.

8.1 Cas d'utilisation typique

Voici le flux complet d'un cas d'utilisation typique :

a. **Chargement des données sources** :
   i. Téléchargement des données COVID-19 mondiales
   ii. Placement du fichier CSV dans le répertoire **data/raw/**

b. **Exécution du pipeline ETL** :
   i. Lancement de la commande python run_etl.py
   ii. Traitement et chargement automatique des données

c. **Analyse des données** :
   i. Connexion à la base PostgreSQL
   ii. Exécution de requêtes analytiques
   iii. Visualisation via PowerBI

8.2 Exemples de requêtes analytiques

Voici quelques exemples de requêtes SQL qui peuvent être exécutées sur la base de données transformée :

a. Évolution des cas par pays :

```sql
SELECT p.nom_pays, s.date_observation, s.cas_confirmes, s.nouveaux_cas
FROM situation_pandemique s
JOIN pays p ON s.id_pays = p.id_pays
WHERE p.nom_pays IN ('France', 'Germany', 'Italy')
ORDER BY p.nom_pays, s.date_observation;
```

b. Taux de létalité par région OMS :

```sql
SELECT p.region_oms,
   SUM(s.deces) as total_deces,
   SUM(s.cas_confirmes) as total_cas,
   ROUND(SUM(s.deces) * 100.0 / NULLIF(SUM(s.cas_confirmes), 0), 2) as taux_letalite
FROM situation_pandemique s
JOIN pays p ON s.id_pays = p.id_pays
GROUP BY p.region_oms
ORDER BY taux_letalite DESC;
```

c. Top 10 pays avec le plus de nouveaux cas sur les 7 derniers jours :

```sql
WITH derniere_date AS (
   SELECT MAX(date_observation) as date_max
   FROM situation_pandemique
)
SELECT p.nom_pays, SUM(s.nouveaux_cas) as total_nouveaux_cas
FROM situation_pandemique s
JOIN pays p ON s.id_pays = p.id_pays
JOIN derniere_date d ON s.date_observation BETWEEN (d.date_max - INTERVAL '7 days') AND
d.date_max
GROUP BY p.nom_pays
ORDER BY total_nouveaux_cas DESC
LIMIT 10;
```

d. Courbe de guérison vs. nouveaux cas :

```sql
SELECT
   date_observation,
   SUM(nouveaux_cas) as total_nouveaux_cas,
   SUM(nouvelles_guerisons) as total_nouvelles_guerisons
FROM situation_pandemique
GROUP BY date_observation
ORDER BY date_observation;
```

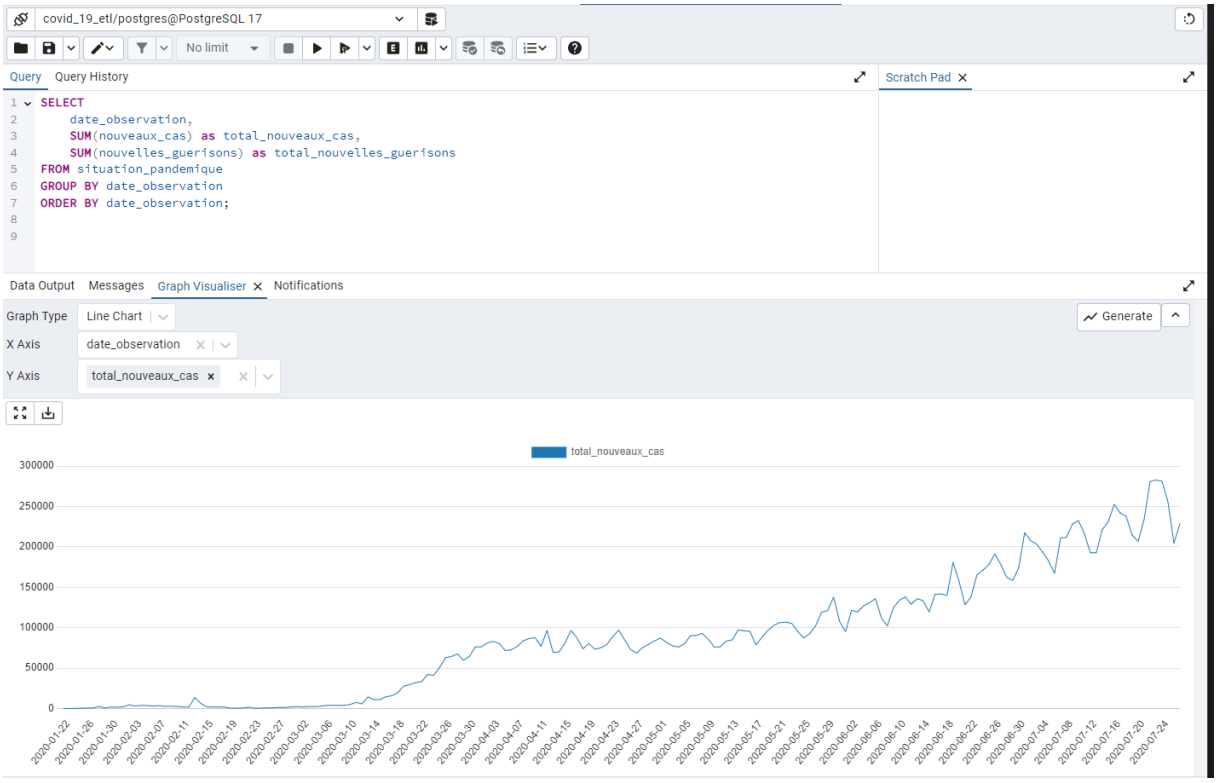

Figure 10: Courbe de guérison vs. nouveaux cas

Ces requêtes illustrent la puissance analytique offerte par notre modèle de données et montrent comment il peut être utilisé pour extraire des insights épidémiologiques significatifs.

9. Conclusion

Ce projet a permis de mettre en place un pipeline ETL robuste pour le traitement des données COVID-19 mondiales. L'architecture modulaire et extensible qui a été développée offre une base solide pour l'analyse épidémiologique.

10 Annexes

10.1 Glossaire

Terme	Définition
ETL	Extract, Transform, Load - Processus d'extraction, transformation et chargement de données
CSV	Comma-Separated Values - Format de fichier texte où les valeurs sont séparées par des virgules
PostgreSQL	Système de gestion de base de données relationnelle open-source
ORM	Object-Relational Mapping - Technique de programmation pour convertir des données entre systèmes incompatibles

Terme	Définition
Pandas	Bibliothèque Python pour la manipulation et l'analyse de données
SQLAlchemy	Bibliothèque Python SQL toolkit et ORM
Airflow	Plateforme d'orchestration de workflows de traitement de données
Kafka	Plateforme de streaming distribuée pour le traitement des flux de données en temps réel

10.2 Références

- Documentation Python : https://docs.python.org/3/
- Documentation PostgreSQL : https://www.postgresql.org/docs/
- Documentation Pandas : https://pandas.pydata.org/docs/
- Documentation SQLAlchemy : https://docs.sqlalchemy.org/
- Tutoriel ETL avec Python : *https://www.youtube.com/watch?v=dfouoh9QdUw&t=519s*