TP4 - Pré-stockage et Traitement des Données IoT

Module : Systèmes IoT Avancés | Durée Totale : 4 Heures 15 Minutes

Édition Expert : Pipeline, Edge Computing, TSDB & NoSQL

20 min

Le Pipeline de Données IoT

Avant de coder, il est crucial de comprendre le cycle de vie complet de la donnée, de la captation brute à l'intelligence décisionnelle.

> **Principe "Garbage In, Garbage Out" :** La qualité de l'output d'un algorithme d'IA est strictement limitée par la qualité de son input. Des données bruitées ou mal structurées mènent à des prédictions erronées.

Construire un mini système IoT complet :

Capteur simulé → Nettoyage → Traitement → Stockage → Analyse

Les 6 Étapes Fondamentales

1. **Acquisition :** Captation via capteurs (MQTT, HTTP).
2. **Nettoyage :** Détection **d'outliers** et traitement des valeurs **manquantes**.
3. **Transformation (Normalisation + Lissage):** Mise à l'échelle (Min-Max, Z-Score) pour l'homogénéité.
4. **Agrégation :** Réduction du volume via **fenêtres temporelles glissantes**.
5. **Fog Computing :** Traitement local pour réduire la latence et la bande passante.
6. **Stockage :** Persistance optimisée (TSDB, NoSQL, SQL).

Architecture de Référence : Edge vs Fog vs Cloud

Niveau	Localisation	Latence	Rôle Principal
Edge	Périphérique (Capteurs/Nodes)	Ultra-faible (<10ms)	Temps réel, Nettoyage, Sécurité à la source.
Fog	Passerelles / LAN	Moyenne (10-100ms)	Agrégation locale, Filtrage, Contrôle régional.
Cloud	Data Centers	Élevée (Dépend du réseau)	Big Data, IA Complexe, Archivage long terme.

4.1 — Génération des données IoT (Acquisition)

Objectif

Simuler un flux réel de capteur IoT

```
# --- Création du Dataset ---

def generate_data():
    temp = 22 + np.random.randn()
    humid = 45 + np.random.randn()
    #  🔹 Ajouter des valeurs aberrantes (outliers)
    if np.random.rand() < 0.1:  # 10% de chance
        temp += np.random.choice([10, -10])  # grosse variation
        humid += np.random.choice([20, -20])
    #  🔹 Ajouter des valeurs nulles (NaN)
    if np.random.rand() < 0.1:
        temp = np.nan
    if np.random.rand() < 0.1:
        humid = np.nan
    return {
        "time": pd.to_datetime(np.random.uniform((pd.Timestamp.now() - pd.DateOffset(years=1)).value,
pd.Timestamp.now().value)),
        "temp": temp,
        "humid": humid
    }
data = [generate_data() for _ in range(50)]
df = pd.DataFrame(data).set_index("time").sort_index()

print("--- Données Brutes ---")
print(df.head(5))
```

4.2 — Installation & Setup

Objectif

Préparer votre poste de travail pour le traitement de données IoT et le stockage. Nous utiliserons **Jupyter Notebook** pour le code et **Docker** pour monter les bases de données.
Prérequis logiciels : Python 3.10+, Docker Desktop (Win/Mac) ou Docker Engine (Linux), Jupyter Notebook, Pandas, InfluxDB-client, Pymongo.

Procédures d'Installation

Windows

1. **Installer Python :** Téléchargez sur python.org (Cochez "Add Python to PATH").
2. **Installer Docker Desktop :** Téléchargez depuis Docker et lancez l'application.
3. **Environnement Virtuel :**

```
mkdir tp_iot_data
cd tp_iot_data
python -m venv venv
venv\Scripts\activate
```

4. **Installer les librairies :**

```
pip install jupyter pandas numpy matplotlib scikit-learn influxdb-client pymongo
paho-mqtt
```

5. **Lancer InfluxDB (Docker) :**

```
docker run -d -p 8086:8086 -e DOCKER_INFLUXDB_INIT_MODE=setup -e
DOCKER_INFLUXDB_INIT_USERNAME=admin -e DOCKER_INFLUXDB_INIT_PASSWORD=password -e
DOCKER_INFLUXDB_INIT_ORG=emsi_iot -e DOCKER_INFLUXDB_INIT_BUCKET=sensor_data --name
influxdb-v2 influxdb:2
```

Linux (Ubuntu/Debian)

1. **Mettre à jour :** `sudo apt update && sudo apt install python3-pip python3-venv docker.io`
2. **Démarrer Docker :** `sudo systemctl start docker` (Ajoutez votre utilisateur au groupe docker si nécessaire).
3. **Environnement Virtuel :**

```
mkdir tp_iot_data
cd tp_iot_data
```

```
python3 -m venv venv
source venv/bin/activate
```

4. **Installer les librairies :** `pip install jupyter pandas numpy matplotlib scikit-learn influxdb-client pymongo paho-mqtt`
5. **Lancer InfluxDB (Docker) :**

```
docker run -d -p 8086:8086 -e DOCKER_INFLUXDB_INIT_MODE=setup -e
DOCKER_INFLUXDB_INIT_USERNAME=admin -e DOCKER_INFLUXDB_INIT_PASSWORD=password -e
DOCKER_INFLUXDB_INIT_ORG=emsi_iot -e DOCKER_INFLUXDB_INIT_BUCKET=sensor_data --name
influxdb-v2 influxdb:2
```

macOS

1. **Installer Homebrew :** `/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"`
2. **Installer Docker & Python :** `brew install --cask docker python`
3. **Lancer Docker Desktop :** Ouvrez l'application depuis le dossier Applications.
4. **Environnement Virtuel :**

```
mkdir tp_iot_data
cd tp_iot_data
python3 -m venv venv
source venv/bin/activate
```

5. **Installer les librairies :** `pip install jupyter pandas numpy matplotlib scikit-learn influxdb-client pymongo paho-mqtt`
6. **Lancer InfluxDB (Docker) :**

```
docker run -d -p 8086:8086 -e DOCKER_INFLUXDB_INIT_MODE=setup -e
DOCKER_INFLUXDB_INIT_USERNAME=admin -e DOCKER_INFLUXDB_INIT_PASSWORD=password -e
DOCKER_INFLUXDB_INIT_ORG=emsi_iot -e DOCKER_INFLUXDB_INIT_BUCKET=sensor_data --name
influxdb-v2 influxdb:2
```

> Lancer InfluxDB :
> !**docker** start influxdb-v2
> Lancer MongoDB:
> !**docker** run -d -p 27017:27017 --name mongodb mongo
> !**docker** start mongodb

4.3 — Détection des Outliers

Objectif

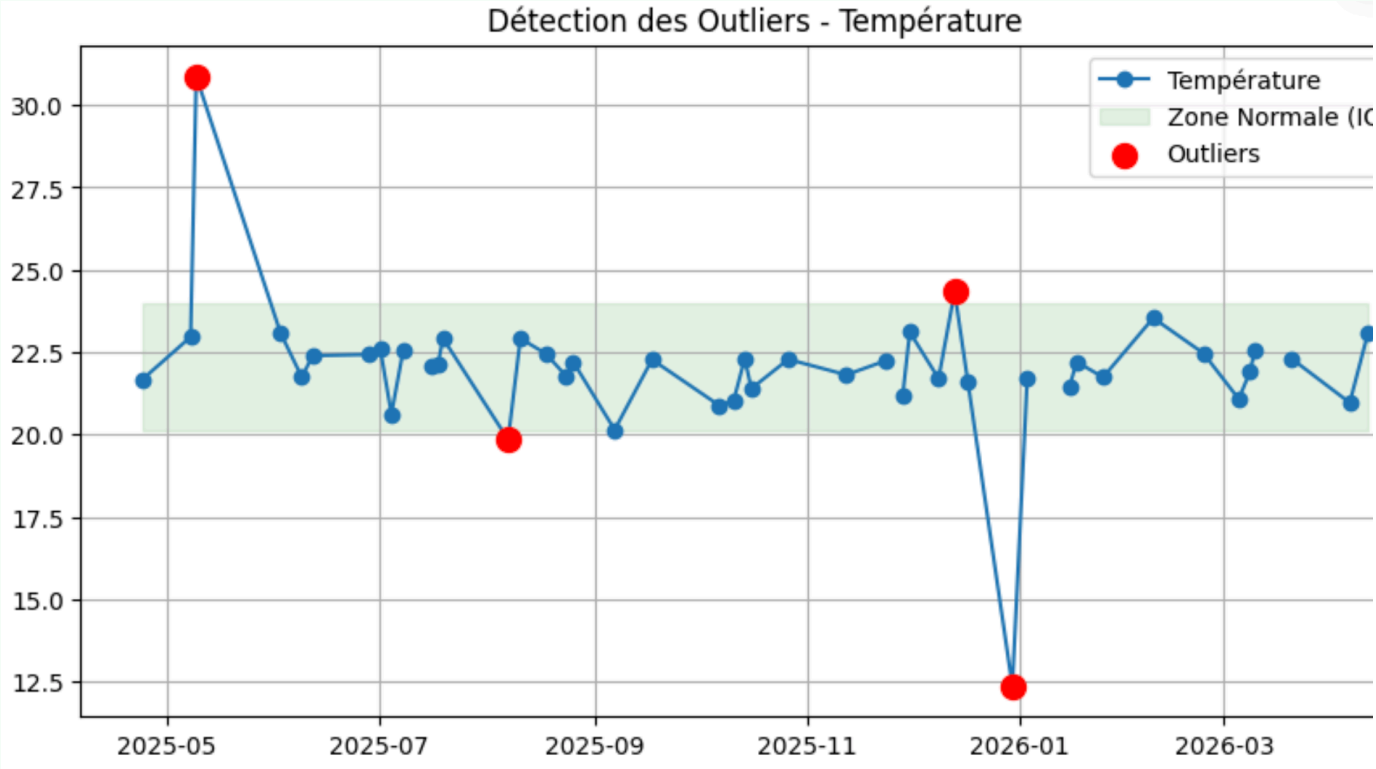
Détecter et corriger les valeurs aberrantes (Outliers).

Concepts Théoriques

- $Z - Score: (z = \frac{x-\mu}{\sigma}). Règle: (|z| > 3 \Rightarrow) Outlier.$
- $IQR(InterquartileRange): Bornes([Q1 - 1.5 \times IQR; Q3 + 1.5 \times IQR]).$

Solution Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import time
# --- 1. Méthode Z-Score (Sur Température) ---
# Note: Le Z-Score est sensible aux outliers eux-mêmes, donc on l'utilise souvent sur des données déjà nettoyées ou pour des détections simples.
mean_temp = df['temp'].mean()
std_temp = df['temp'].std()
df['Temp_ZScore'] = (df['temp'] - mean_temp) / std_temp
df['Temp_ZScore'].head(5)

# --- 2. Méthode IQR (Plus robuste) ---
Q1 = df['temp'].quantile(0.25)
Q3 = df['temp'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

print(f"\nBornes IQR Température: [{lower_bound:.2f}, {upper_bound:.2f}]")
# Identification des outliers
outliers_mask = (df['temp'] < lower_bound) | (df['temp'] > upper_bound)
print(f"Outliers détectés (IQR) :\n{df[outliers_mask]['temp']}")

# Visualisation
plt.figure(figsize=(10, 5))
plt.plot(df.index, df['temp'], 'o-', label='Température')
plt.fill_between(df.index, lower_bound, upper_bound, color='green', alpha=0.1, label='Zone Normale (IQR)')
plt.scatter(df[outliers_mask].index, df[outliers_mask]['temp'], color='red', s=100, label='Outliers', zorder=5)
plt.title('Détection des Outliers - Température')
plt.legend()
plt.grid(True)
plt.show()
```



4.4 — Gestion des données manquantes

Objectif

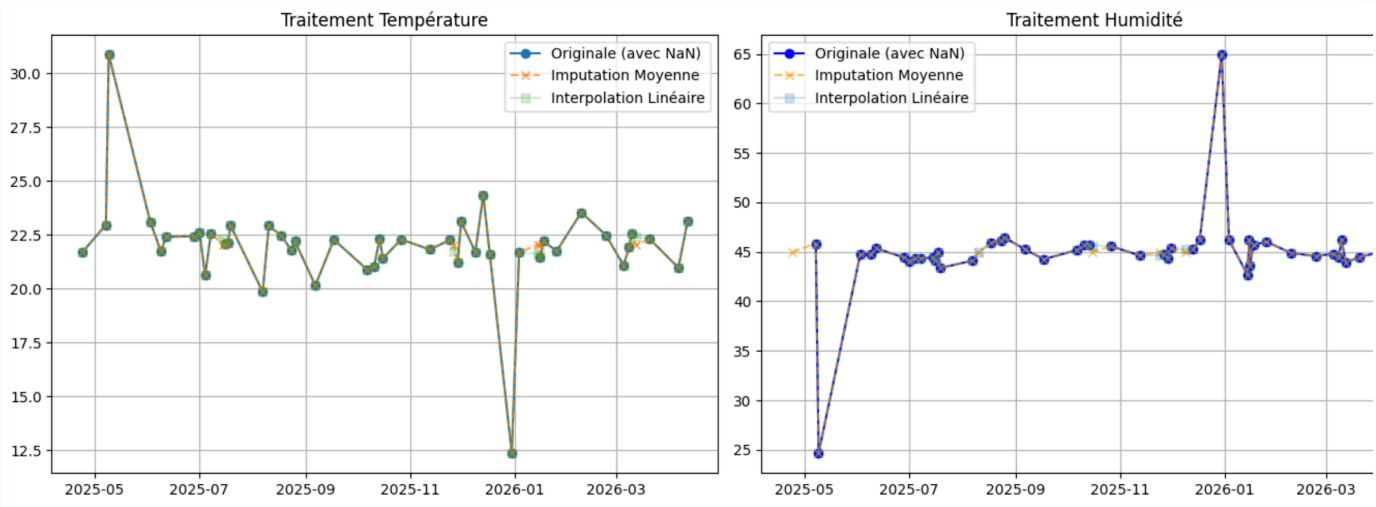
Comparez deux stratégies pour gérer les valeurs manquantes (`NaN` dans Temp, `0.0` dans Humid).

1. **Imputation par la Moyenne :** Remplacer par la moyenne globale.
2. **Interpolation Linéaire :** Estimer la valeur en fonction des points voisins.

Solution Code

```
# --- Préparation : Remplacer les erreurs évidentes (0.0 humidité) par NaN ---
df.loc[df['humid'] == 0.0, 'humid'] = np.nan
# --- 1. Imputation par la Moyenne ---
# Simple mais peut fausser la tendance temporelle
df['Temp_Mean'] = df['temp'].fillna(df['temp'].mean())
df['Humid_Mean'] = df['humid'].fillna(df['humid'].mean())

# --- 2. Interpolation Linéaire ---
# Préserve la dynamique temporelle
df['Temp_Interp'] = df['temp'].interpolate(method='linear')
df['Humid_Interp'] = df['humid'].interpolate(method='linear')
# --- Comparaison Visuelle ---
```

```python
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Température
axes[0].plot(df.index, df['temp'], 'o-', label='Originale (avec NaN)', alpha=1)
axes[0].plot(df.index, df['Temp_Mean'], 'x--', label='Imputation Moyenne', alpha=0.7)
axes[0].plot(df.index, df['Temp_Interp'], 's-', label='Interpolation Linéaire', alpha=0.1)
axes[0].set_title('Traitement Température')
axes[0].legend()
axes[0].grid(True)

# Humidité
axes[1].plot(df.index, df['humid'], 'o-', color='blue', label='Originale (avec NaN)', alpha=1)
axes[1].plot(df.index, df['Humid_Mean'], 'x--',color='orange', label='Imputation Moyenne', alpha=0.7)
axes[1].plot(df.index, df['Humid_Interp'], 's-', label='Interpolation Linéaire', alpha=0.1)
axes[1].set_title('Traitement Humidité')
axes[1].legend()
axes[1].grid(True)

plt.tight_layout()
plt.show()
```




```python
# Application : On choisit l'interpolation pour la suite (meilleure pour le temps)
df['Temp_Clean'] = df['Temp_Interp']
df['Humid_Clean'] = df['Humid_Interp']

print("Données après nettoyage :")
print(df[['Temp_Clean', 'Humid_Clean']])
```

4.5 - Transformation des données

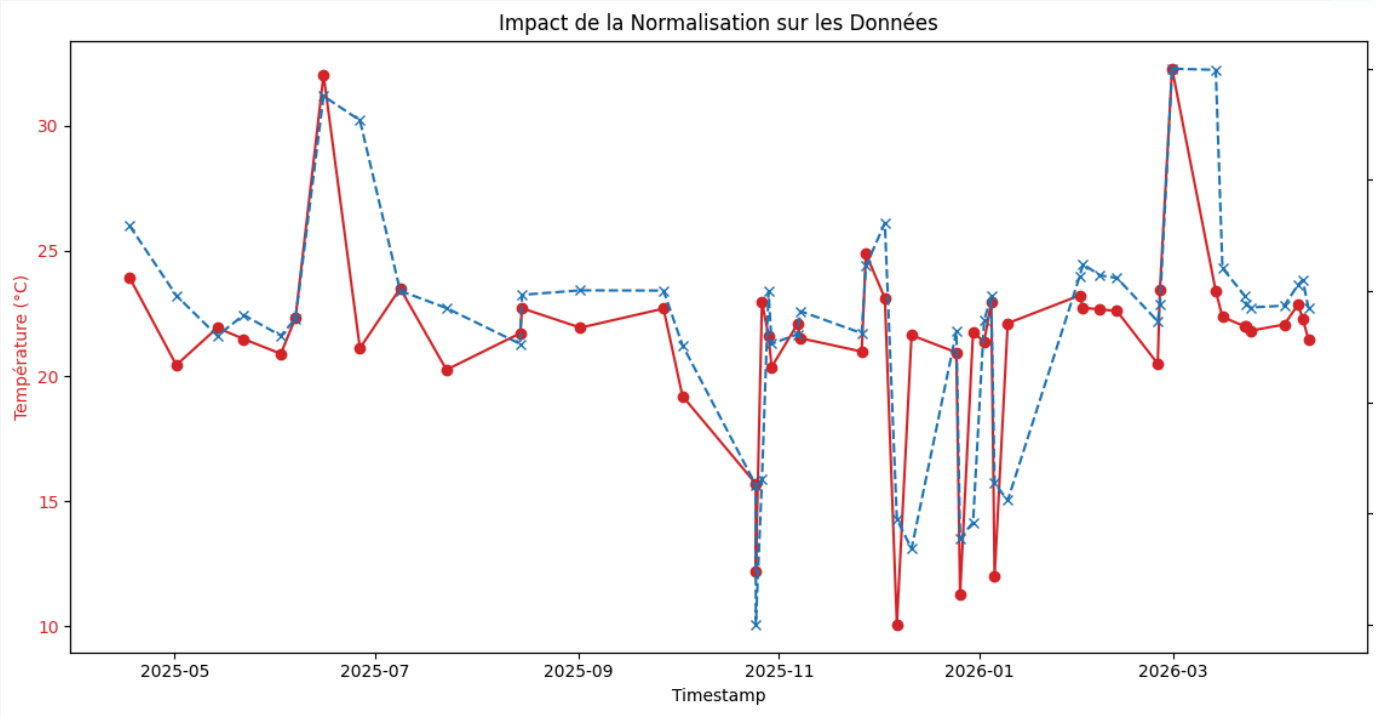
Objectif

1. $Normalisation (Min - Max)$: $Ramener dans ([0, 1])$. $Formule$: $(X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}})$.

2. $Agrégation$: $Calculer\ la\ moyenne\ mobile\ sur\ 2\ points$.

Solution Code

```
from sklearn.preprocessing import MinMaxScaler

# --- 1. Lissage (Moyenne Mobile) ---
# Window=2 signifie qu'on prend la valeur actuelle et la précédente
df['Temp_Smoothed'] = df['Temp_Clean'].rolling(window=2, min_periods=1).mean()
df['Humid_Smoothed'] = df['Humid_Clean'].rolling(window=2, min_periods=1).mean()

# --- 2. Normalisation Min-Max ---
scaler = MinMaxScaler()
# On fit et transforme sur les données lissées
df[['Temp_Norm', 'Humid_Norm']] = scaler.fit_transform(df[['Temp_Smoothed', 'Humid_Smoothed']])

# --- Visualisation Avant / Après ---
fig, ax = plt.subplots(figsize=(12, 6))

# Plot Température Brute vs Normalisée (sur axe secondaire pour l'échelle)
color = 'tab:red'
ax.set_xlabel('Timestamp')
ax.set_ylabel('Température (°C)', color=color)
ax.plot(df.index, df['Temp_Clean'], color=color, label='Température Brute', marker='o')
ax.tick_params(axis='y', labelcolor=color)

ax2 = ax.twinx()
color = 'tab:blue'
ax2.set_ylabel('Température Normalisée [0-1]', color=color)
ax2.plot(df.index, df['Temp_Norm'], color=color, linestyle='--', label='Température Normalisée', marker='x')
ax2.tick_params(axis='y', labelcolor=color)

plt.title('Impact de la Normalisation sur les Données')
fig.tight_layout()
plt.show()

print("\n--- Dataset Final Prêt pour l'IA ---")
print(df[['Temp_Clean', 'Temp_Smoothed', 'Temp_Norm']].head())
```



--- Dataset Final Prêt pour l'IA ---

time	Temp_Clean	Temp_Smoothed	Temp_Norm
2025-04-17 06:54:28.768077568	23.944445	23.944445	0.717760
2025-05-01 13:01:48.888414720	20.441340	22.192892	0.591926
2025-05-14 00:28:30.029531136	21.927550	21.184445	0.519477
2025-05-22 01:33:12.811074304	21.481335	21.704443	0.556835
2025-06-02 08:47:07.519278592	20.891911	21.186623	0.519634

4.5 – Stockage InfluxDB

Objectif

Apprendre à persister des données IoT nettoyées dans une Time Series Database (TSDB) et à effectuer des agrégations directement en base (Downsampling (sous-échantillonnage)).

Conseil de Production : En environnement réel, Pour ne pas saturer votre stockage, programmez la suppression automatique des données détaillées après **30 jours**. Gardez seulement des **résumés** (moyennes par heure ou par jour) pour vos statistiques sur **5 ans**.

Ex:

- **Données brutes :** enregistrées à haute fréquence (par exemple 1 point par seconde), elles sont très précises mais deviennent très volumineuses sur de longues périodes.

- **Données échantillonnées (ou réduites) :** en ne gardant qu'un point sur plusieurs (par exemple 1 sur 10), on diminue fortement la taille des données tout en conservant une vision globale des tendances.

```
df_edge = df.iloc[::10]

print("Avant:", len(df))
print("Après:", len(df_edge))
```

Tâches:

1. **Connexion :** Initialiser le client InfluxDB.
2. **Persistance :** Créer une fonction `persist_to_tsdb(df, bucket_name)`.
3. **Downsampling :** Écrire une requête Flux pour calculer la moyenne par tranche de 15 minutes.

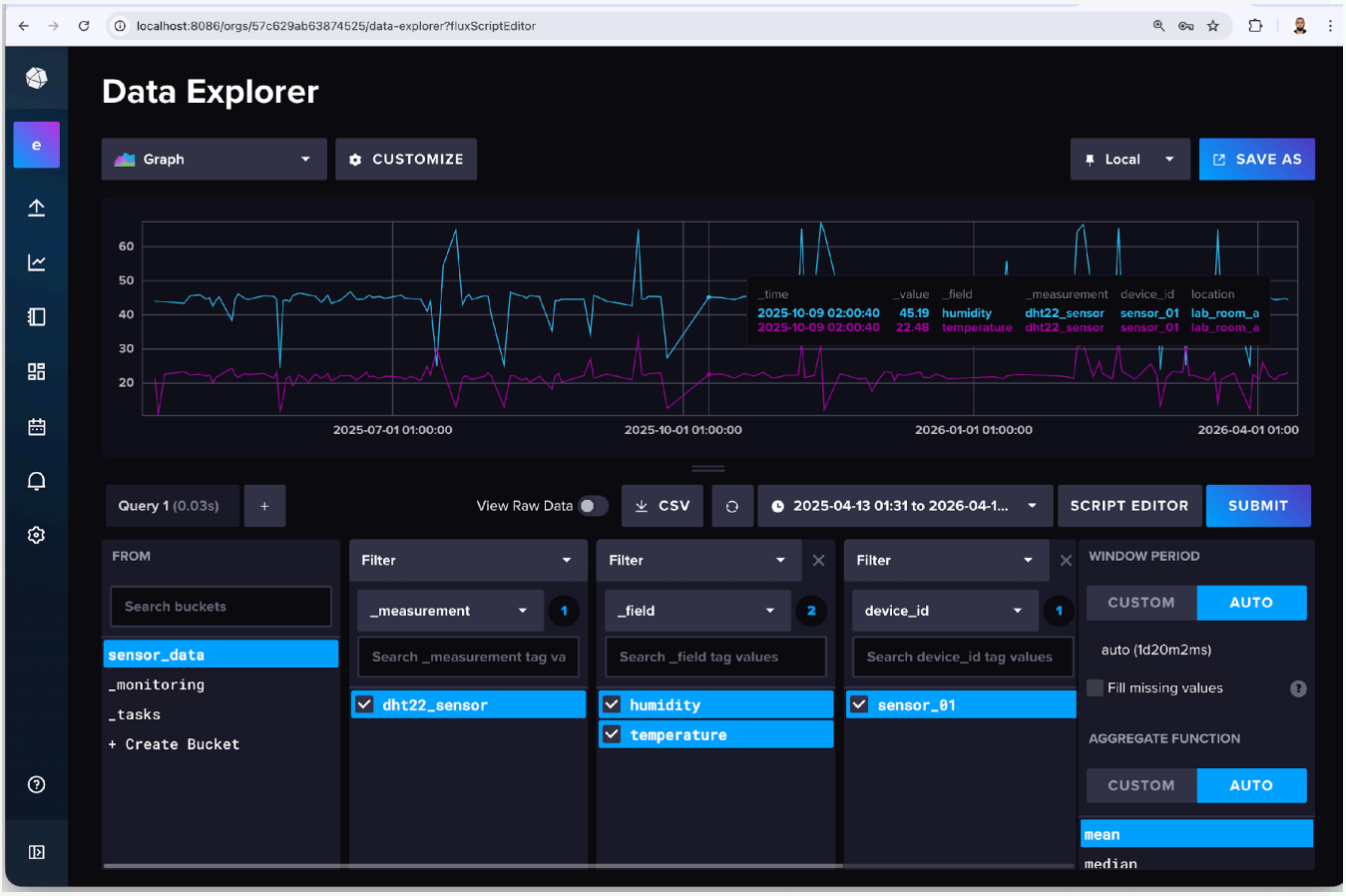
Solution Code

1. Ouvrez votre navigateur sur : http://localhost:8086
2. Connectez-vous avec votre utilisateur et mot de passe.
3. Dans le menu de gauche, allez dans : Data API Tokens.
4. Vous verrez un token appelé "Operator Token" (ou créez-en un nouveau via "Generate API Token" All Access Token").
5. Cliquez sur le token pour copier la longue chaîne de caractères (elle ressemble à vS9K...xxxxxxxx...).
6. Collez cette chaîne exacte dans votre code Python :

NE FAITES PAS : token = "password"
FAITES : token = "S7v8abc123XYZ...la_longue_cle_copiée_dans_l_interface..."

```
from influxdb_client import InfluxDBClient, Point, WritePrecision
from influxdb_client.client.write_api import SYNCHRONOUS
import datetime
# --- Configuration ---
# Assurez-vous que le conteneur Docker est lancé
url = "http://localhost:8086"
token =
"2FLnZ2dz5iRusqZT_fptXfwf1iFrmeBwB2iFgcTtMd0TcW5XUDwaJN0rFhE6a0VQ5cdz9GeqDZO8-i82RJzLbg=="
# Token défini dans le run Docker
org = "emsi_iot"
bucket = "sensor_data"

client = InfluxDBClient(url=url, token=token, org=org)

# --- 1. Fonction de Persistance ---
def persist_to_tsdb(df, bucket_name):
    """
    Envoie un DataFrame Pandas vers InfluxDB.
```

```
    Convertit chaque ligne en un Point InfluxDB.
    """
    write_api = client.write_api(write_options=SYNCHRONOUS)

    points = []
    for index, row in df.iterrows():
        # Création d'un point de données
        point = Point("dht22_sensor") \
            .tag("location", "lab_room_a") \
            .tag("device_id", "sensor_01") \
            .field("temperature", float(row['Temp_Clean'])) \
            .field("humidity", float(row['Humid_Clean'])) \
            .time(index, WritePrecision.NS)
        points.append(point)

    try:
        write_api.write(bucket=bucket_name, org=org, record=points)
        print(f"Succès(Données envoyées) : {len(points)} points écrits dans InfluxDB.")
    except Exception as e:
        print(f"Erreur d'écriture (Docker lancé ?) : {e}")

# Test de la fonction (utilise le df nettoyé des sections précédentes)
persist_to_tsdb(df, bucket)

# --- 2. Requête Flux : Downsampling (Agrégation sur 1 an) ---
query_api = client.query_api()

# Requête : Moyenne par jour
# bucket = base de données InfluxDB
# -1y = les données de la dernière année
query = f'''
from(bucket: "{bucket}")
  |> range(start: -1y)
  |> filter(fn: (r) => r._measurement == "dht22_sensor")
  |> filter(fn: (r) => r._field == "temperature")
  |> aggregateWindow(every: 1d, fn: mean, createEmpty: false)
  |> yield(name: "mean_daily")
'''

result = query_api.query(query, org=org)

print("\n--- Résultats du Downsampling (Moyenne journalière) ---")
count = 0
for table in result:
    for record in table.records:
      count += 1
      print(f"Time: {record.get_time()}, Température: {record.get_value()}")

print(f"\nNombre total de résultats: {count}")

client.close()
```



30 min

4.9 — Conception d’une architecture hybride MongoDB / TSDB

Objectif

Comprendre quand utiliser un **modèle Document (MongoDB)** pour les **métadonnées statiques** vs un modèle Relationnel/**TSDB** (TimescaleDB) pour les **mesures dynamiques**.

Exercice de Conception

Concevez le schéma pour stocker les informations suivantes :

- **Métadonnées (Statiques) :** ID Capteur, Type (DHT22), Localisation GPS, Date d'installation.
- **Mesures (Dynamiques) :** Timestamp, Température, Humidité.

***Question :* Pourquoi séparer ces deux types de données ?**

Solution : Approche Hybride

Réponse : Les métadonnées **changent rarement**, structure parfois flexible (**NoSQL** ideal, ex. documentaire comme MongoDB), tandis que les mesures **arrivent en continu** à haute fréquence(très nombreuses, très nombreuses) (TSDB idéal).

1. MongoDB (Métadonnées)

Avantages : Schéma flexible, pas de jointures complexes pour récupérer la config d'un capteur.

```
# Simulation d'insertion MongoDB (pymongo)
from pymongo import MongoClient
client = MongoClient("mongodb://localhost:27017/")
db = client["iot_metadata"]
collection = db["sensors"]

import pandas as pd
sensor_metadata = {
    "_id": "sensor_01",
    "type": "DHT22",
    "measurement": "dht22_sensor",  # lien avec InfluxDB
    "fields": ["temperature", "humidity"],

    "location": {
        "building": "A",
        "room": "101",
        "gps": {"lat": 48.85, "lng": 2.35}
    },

    "firmware_version": "2.1.0",

    # date d'installation (cohérente avec période de données)
    "installed_at": (pd.Timestamp.now() - pd.DateOffset(years=1)).isoformat(),

    # période des données générées
    "data_range": {
        "start": (pd.Timestamp.now() - pd.DateOffset(years=1)).isoformat(),
        "end": pd.Timestamp.now().isoformat()
    },

    # fréquence logique (même si tes données sont aléatoires)
    "sampling_note": "Données simulées aléatoires sur 1 an",

    # info agrégation (ce que tu fais avec Flux)
    "aggregation": {
        "type": "mean",
        "window": "1d"
    }
}
# collection.insert_one(sensor_metadata)
collection.update_one(
    {"_id": "sensor_01"},
    {"$set": sensor_metadata},
    upsert=True
```

```
)
print("Métadonnées insérées dans MongoDB.")
```
Métadonnées insérées dans MongoDB.

```
## Vérification
print(collection.find_one({"_id": "sensor_01"}))
```
{'_id': 'sensor_01', 'type': 'DHT22', 'measurement': 'dht22_sensor', 'fields': ['temperature', 'humidity'], 'location': {'building': 'A', 'room': '101', 'gps': {'lat': 48.85, 'lng': 2.35}}, 'firmware_version': '2.1.0', 'installed_at': '2025-04-14T00:13:06.234737', 'data_range': {'start': '2025-04-14T00:13:06.235002', 'end': '2026-04-14T00:13:06.235138'}, 'sampling_note': 'Données simulées aléatoires sur 1 an', 'aggregation': {'type': 'mean', 'window': '1d'}}

2. TimescaleDB (Hypertables)

Avantages : Puissance SQL pour les jointures avec les données métier, partitionnement automatique du temps.

- **Lancer PostgreSQL + TimescaleDB**

```batch----------

```
!docker run -d -p 5432:5432 \
-e POSTGRES_PASSWORD=postgres \
--name timescaledb \
timescale/timescaledb:latest-pg14

pip install psycopg2-binary
```

```python----------

```
import psycopg2

conn = psycopg2.connect(
    host="localhost",
    database="postgres",
    user="postgres",
    password="postgres"
)

cursor = conn.cursor()

# Création table
cursor.execute("""
CREATE TABLE IF NOT EXISTS sensor_data (
    time TIMESTAMPTZ NOT NULL,
    sensor_id TEXT NOT NULL,
    temperature DOUBLE PRECISION,
    humidity DOUBLE PRECISION
);
""")

# Hypertable
```

```
cursor.execute("""
SELECT create_hypertable('sensor_data', 'time', if_not_exists => TRUE);
""")

# Insertion
cursor.execute("""
INSERT INTO sensor_data (time, sensor_id, temperature, humidity)
VALUES (NOW(), 'sensor_01', 22.5, 45.2);
""")

print("✅ Données insérées")

cursor.execute("""SELECT time, temperature
FROM sensor_data
WHERE temperature > 30; """)
rows = cursor.fetchall()
print("\n--- Résultats de la requête ---")

for row in rows:
    print(f"Time: {row[0]}, Temperature: {row[1]}")

conn.commit()
cursor.close()
conn.close()
```

Exercice : Choix Technologique (Smart Factory)

Contexte : 10 000 capteurs de vibration. Haute disponibilité requise. Rétention 5 ans.
Mission : Argumentez le choix entre InfluxDB, Cassandra et TimescaleDB.

Argumentation

Critère	InfluxDB	Cassandra	TimescaleDB
Scalabilité	Verticale (Cluster limité)	**Horizontale Illimitée**	Verticale (Postgres)
Disponibilité	Moyenne	**Haute (No SPOF)**	Haute (Replication)
Requêtes SQL	Non (Flux lang)	Non (CQL)	**Oui (Standard SQL)**

Verdict : Pour 10 000 capteurs avec rétention longue et haute disponibilité, **Cassandra** est souvent privilégié pour l'ingestion massive. Si l'équipe BI a besoin de SQL, **TimescaleDB** est le compromis idéal.

Tableau Récapitulatif : Choix de la Base de Données IoT

Technologie	Vitesse Écriture	Requêtes Temporelles	Scalabilité	Cas d'Usage Idéal	
InfluxDB	TSDB	Optimisée	Native (Flux)	Verticale	Monitoring temps réel, Alertes.
Cassandra	NoSQL (Wide Column)	Très Haute	Faible (CQL)	Horizontale Illimitée	Big Data, Logs massifs, Télécoms.
MongoDB	NoSQL (Document)	Moyenne	Flexible (JSON)	Horizontale	Métadonnées capteurs, Configs.
TimescaleDB	SQL (PostgreSQL)	Haute	SQL Standard	Verticale	Analytique complexe, Jointures métier.

Checklist de Livraison

- **Code :** Notebook fonctionnel (Kernel Restart & Run All).
- **Nettoyage :** Implémentation IQR et Interpolation.
- **Stockage :** Fonction `persist_to_tsdb` testée (ou log simulée).
- **Requêtes :** Script Flux pour le downsampling.
- **Architecture :** Schéma MongoDB vs TimescaleDB défini.

Conclusion

Ce TP a démontré que le choix de l'architecture de stockage est aussi critique que le nettoyage des données. L'Edge Computing réduit la latence, les TSDB gèrent la vélocité, et le NoSQL gère la variété. Une solution IoT robuste combine souvent ces technologies.