

# Memoria Técnica: Proyecto de Monitorización de la calidad del Aire

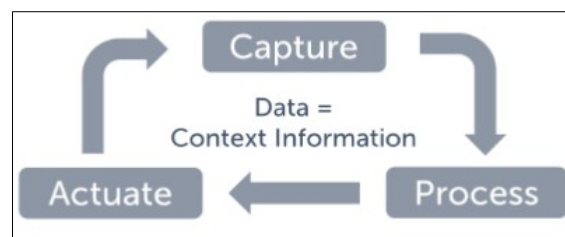
## Índice:

- 1 [Introducción](#)
- 2 [Arquitectura del Proyecto](#)
- 3 [Justificación de la Arquitectura](#)
  - 3.1 [Flujo sensor al servidor vía MQTT](#)
  - 3.2 [Creando el modelo de datos](#)
  - 3.3 [Ingestión de datos via MQTT](#)
  - 3.4 [Transformación y almacenamiento de datos](#)
  - 3.5 [Análisis y limpieza de datos](#)
  - 3.6 [Enriquecimiento de datos](#)
  - 3.7 [Creación de cuadros de comando](#)
- 4 [Más allá de Orion](#)

# 1. Introducción

El presente documento describe la arquitectura técnica propuesta para el proyecto de monitorización de la calidad del aire utilizando sensores medioambientales. Mediante un servidor MQTT, recibiremos los datos de los sensores medioambientales que se almacenarán en una base de datos **TimeScaleDB** y se procesarán posteriormente para su análisis y visualización. Además, se enriquecerán los datos con información de APIs externas y archivos de diferentes formatos como csv, parquet, json, xml, etc ... que proporcionaran datos meteorológicos, geoespaciales y de salud para un análisis más completo, intentando así responder a la pregunta de cómo afectan e impactan las condiciones medioambientales del entorno a la salud de las personas. El objetivo principal del proyecto es recopilar, procesar y visualizar datos de calidad del aire para proporcionar información útil y relevante a los usuarios finales.

El **IoT** es un campo que permite obtener datos y, a partir de ellos procesarlos y analizarlos para finalmente y actuar sobre nuestro entorno para poder mejorar la calidad de vida de las personas. Se trata de crear un ciclo infinito que comprende la captura de datos de dispositivos o sensores, su posterior procesamiento, limpieza y enriquecimiento y, finalmente, con la información que nos proveen, actuar y modificar nuestro entorno mejorando la vida de las personas.



Cada vez estamos más interconectados entre nosotros y con los objetos que nos rodean que conforman el Internet de las Cosas o **IoT**. El **IoT** pretende poder interconectar dispositivos entre sí para poder intercambiar información entre ellos y sin que haya interacción por parte del ser humano. Para que dos dispositivos puedan comunicarse es necesario establecer unos protocolos de comunicación que en el campo de **IoT** se denomina la comunicación **M2M** (*Machine-to-Machine*).

Buscamos un protocolo **IoT** que cumpla:

- **Escalable:** han de poder añadirse o retirarse dinámicamente dispositivos sin que el comportamiento global del sistema se modifique.
- **Debe mantener débil el acoplamiento entre dispositivos:** la dependencia entre los dispositivos debe ser la menor posible, y deseablemente nula.
- **Poca capacidad de procesamiento:** los dispositivos serán dispositivos embebidos, con bajo coste y escasa capacidad de cálculo.
- **Interoperabilidad:** debe funcionar con la mayor variedad de dispositivos, sistemas operativos, y lenguajes de programación.
- **Respuesta rápida:** es posible que haya un **gran número de comunicaciones simultáneas** y, en general, se requiere una respuesta rápida. Esto requiere que los mensajes transmitidos sean pequeños y, nuevamente, no requieran un gran procesamiento.

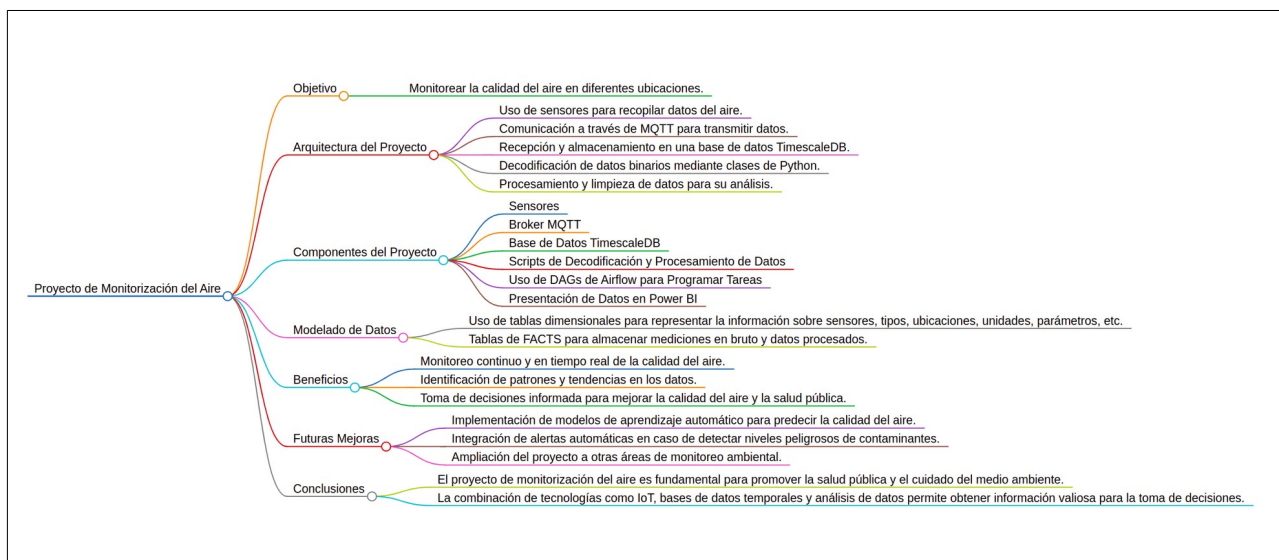
En nuestro caso tendremos múltiples sensores que enviarán multitud de datos en un período de tiempo muy corto y de manera continua. Cada una de estas tramas con las mediciones registradas, ocupan muy poco espacio. Necesitamos una solución que nos provea de un servidor central que reciba los mensajes de todos los dispositivos que están a una gran distancia, los procese y los reenvíe hacia un backend, para posteriormente poder explotarlos y visualizarlos.

## 2. Arquitectura del proyecto

El proyecto constará de los siguientes componentes principales:

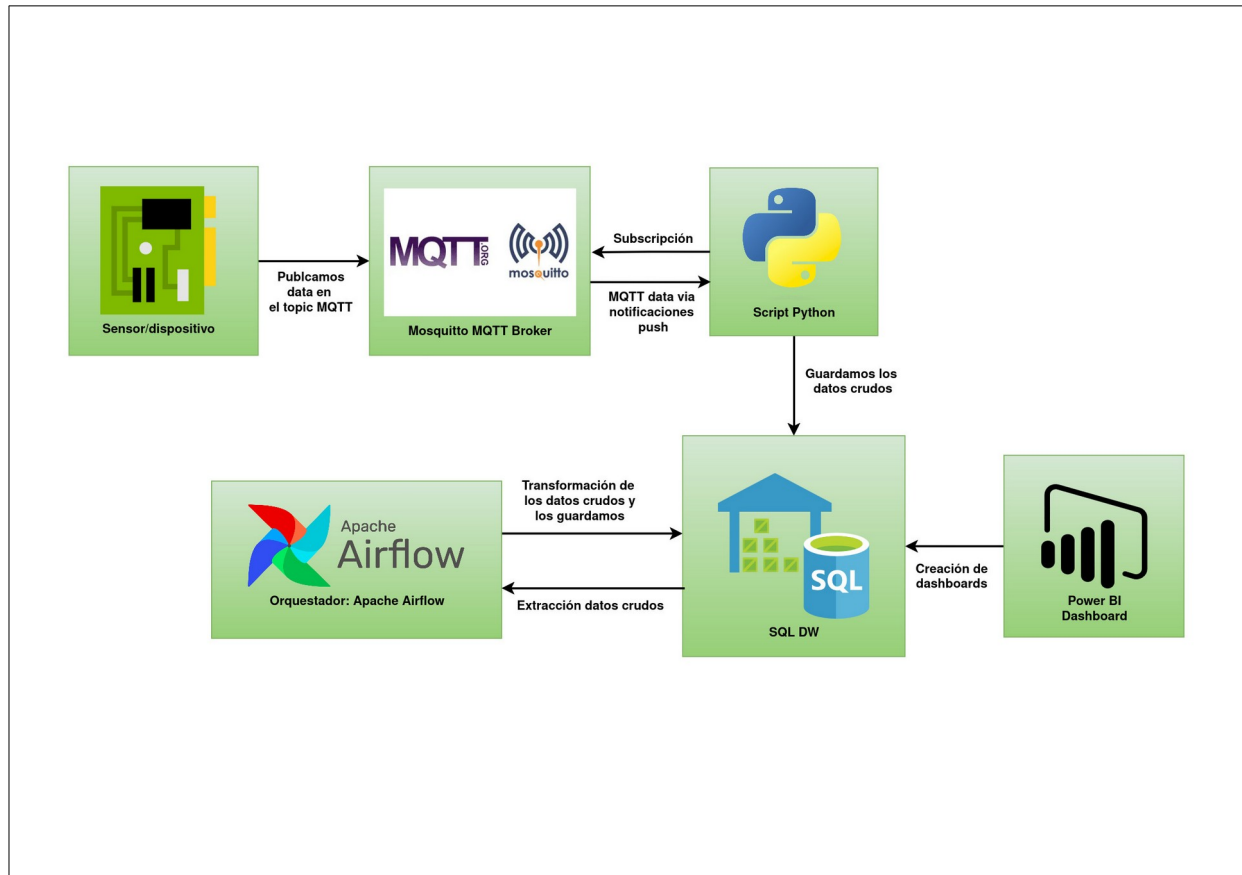
- **Sensores Medioambientales:** Dispositivos distribuidos en puntos estratégicos que recopilan datos ambientales como temperatura, humedad, calidad del aire, niveles de contaminación, etc.
- **Servidor MQTT:** Se desplegará el servidor MQTT [Mosquitto](#) en un entorno local para recibir los datos de los sensores ambientales. El servidor MQTT utilizará TLS para securizar la comunicación que se establece entre el servidor y los sensores.
- **Base de Datos TimeScaleDB:** Se utilizará la base de datos de series temporales [TimeScaleDB](#) para almacenar los datos de los sensores ambientales. Por un lado, nos permitirá tener un inventario de los sensores y, por otro lado, guardaremos los datos en crudo y los datos limpios y enriquecidos. **TimeScaleDB** es una base de datos de series temporales y está optimizada para el análisis de datos temporales.
- **Apache Airflow:** Se utilizará [Apache Airflow](#) para la orquestación y ejecución programada de los procesos de decodificación de datos en crudo obtenidos mediante el broker **MQTT** y para el enriquecimiento de los datos mediante APIs externas. **Airflow** proporciona una plataforma flexible y escalable para automatizar tareas complejas mediante **DAGs** y es posible programar flujos de trabajo complejos con ella.
- **Visualización de Datos:** Se utilizarán herramientas de visualización de datos como [Power BI](#) para crear gráficos y cuadros de mando interactivos que muestren los datos de calidad del aire y su relación con otros factores como el clima, la ubicación geográfica, niveles de contaminación y sus índices de salud pública. (índices de cáncer, de enfermedades del sistema respiratorio, etc)

### Mapa mental



### 3. Justificación de la Arquitectura

La arquitectura que planteamos consta de varias partes tal y como se puede ver en el siguiente gráfico:



Ahora pasaremos a detallar todas las piezas que componen este diagrama así como su flujo, el modelado de los datos y mostraremos algunos fragmentos de código que nos permitan entender mejor toda la arquitectura.

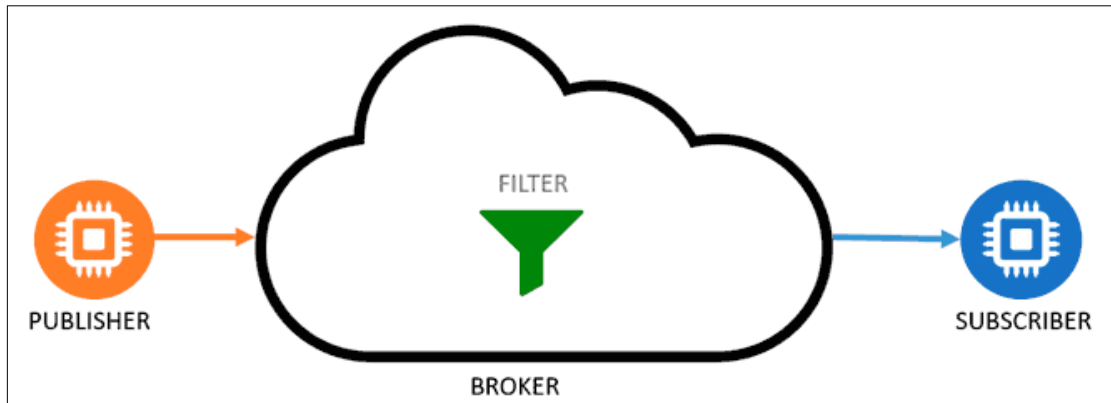
#### 3.1 Flujo sensor al servidor via MQTT

Iniciamos con el flujo continuo de datos que se establece entre los sensores y el servidor que los recibe.

Existe un protocolo ligero que consume muy poco ancho de banda y que permite comunicarse a través de la publicación/suscripción para tener una comunicación bidireccional real con acuses de recibo, con una alta latencia y con poco consumo por parte de los dispositivos: el protocolo **MQTT**. **MQTT** es un protocolo **M2M** (*Machine-to-Machine*) y es el acrónimo de *Message Queue Telemetry Transport*. Está basado en un protocolo de mensajería publicación/suscripción utilizando una topología de estrella de manera que los clientes se conectan a un servidor central llamado broker. La metodología publicación/suscripción es un patrón de mensajería donde un agente, el 'Subscriber', informa al Router que quiere recibir un tipo de mensajes. Otro agente, el 'Publisher' puede publicar mensajes. El Router distribuye los mensajes a los Subscribers. En este caso, el router

**distribuye inmediatamente los mensajes a los clientes conectados.** Los mensajes se filtran por algún criterio, como el tema o el contenido del mensaje.

Se emplea un enfoque jerárquico para filtrar y dirigir los mensajes hacia los clientes correspondientes. Este proceso implica la organización de los mensajes en "**topics**", que actúan como categorías temáticas. Los clientes tienen la capacidad de publicar mensajes dentro de un **topic** específico, mientras que otros clientes pueden suscribirse a estos **topics** para recibir los mensajes pertinentes. Este intercambio de mensajes se facilita a través del intermediario (**broker**), que se encarga de distribuir los mensajes a los clientes suscritos según corresponda.



El protocolo MQTT **dispone de distintas medidas de seguridad** que podemos adoptar para proteger las comunicaciones. Esto incluye **transporte SSL/TLS y autenticación por usuario y contraseña o mediante certificado**. Sin embargo, hay que tener en cuenta que muchos de los dispositivos IoT disponen de escasa capacidad, por lo que el SLL/TLS puede suponer una carga de proceso importante.

La idea sería en una primera instancia, desplegar on-premise una solución de código libre que haga de servidor **MQTT** para recibir los datos de los sensores medioambientales. En una futura iteración y teniendo en cuenta temas de escalabilidad podría ser necesario contratar un servicio en la nube que nos provea de ese servidor **MQTT**.

De momento desplegaremos la solución on-premise y utilizaremos el servidor **MQTT** de código libre **Mosquitto**. **Mosquitto** es un broker Open Source desarrollado por la fundación Eclipse y distribuido bajo licencia EPL/EDL. Está programado en C, y es multiplataforma. Este recibirá los payloads con los datos de los sensores en formato JSON que contendrán los valores codificados.

Para su instalación realizaremos los siguientes pasos:


```
sudo apt update
sudo apt install mosquitto
sudo apt install mosquitto-clients
```

Utilizaremos **mosquitto\_passwd** para crear un archivo de contraseñas que almacenará los usuarios y sus contraseñas. Creamos el archivo e insertamos un usuario:



```
sudo mosquitto_passwd -c /etc/mosquitto/passwd myuser
```

Te pedirá que introduzcas una contraseña para el usuario **myuser**. Después abrimos el archivo de configuración de **Mosquitto**:



```
sudo nano /etc/mosquitto/mosquitto.conf
```

Añadimos las líneas siguientes para habilitar la autenticación:



```
allow_anonymous false  
use_identity_as_username true  
password_file /etc/mosquitto/passwd
```

Ahora generamos un certificado SSL/TLS que utilizaremos para securizar el broker MQTT:




```
sudo openssl req -new -x509 -days 365 -nodes  
-out /etc/mosquitto/certs/server.crt  
-keyout /etc/mosquitto/certs/server.key
```

Editamos el archivo de configuración de Mosquitto:



```
sudo nano /etc/mosquitto/mosquitto.conf
```

Añadimos el certificado y que escuche por el puerto 8883:



```
listener 8883  
require_certificate true  
certfile /etc/mosquitto/certs/server.crt  
keyfile /etc/mosquitto/certs/server.key
```



Utilizaremos **Supervisor** para monitorizar la ejecución de **Mosquitto** de manera que inicie el broker al iniciar el sistema y lo reinicie si cae el servidor. **Supervisor** es un sistema cliente/servidor que permite a sus usuarios monitorear y controlar una serie de procesos en sistemas operativos similares a UNIX. Supondremos instalado **Supervisor** en el sistema, y editamos el archivo:




```
sudo nano /etc/supervisor/conf.d/mosquitto.conf
```

y añadimos:



```
[program:mosquitto]
command=/usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf
autostart=true
autorestart=true
stderr_logfile=/var/log/mosquitto.err.log
stdout_logfile=/var/log/mosquitto.out.log
```

Añadimos la configuración de inicio automático:

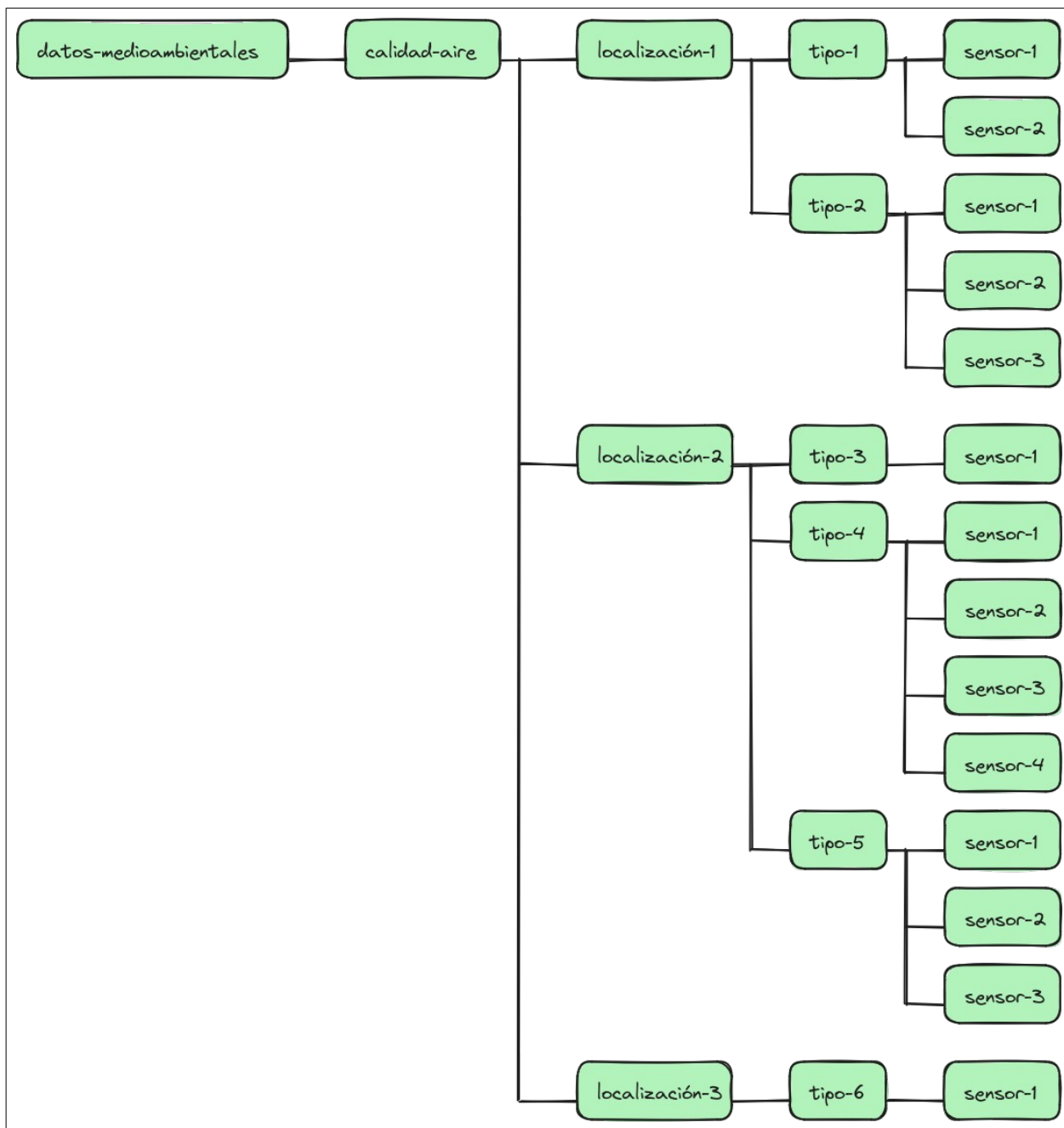


```
sudo systemctl enable mosquitto
sudo systemctl enable supervisor
stdout_logfile=/var/log/mosquitto.out.log
```

Ahora **Mosquitto** está corriendo en el puerto 8883 de manera securizada.

### 3.2 Creando el modelo de datos

Nosotros organizaremos nuestros **topics** de manera similar a una API REST. Veamos una imagen de su estructura:



Es decir tendremos una raíz común y los organizamos en una jerarquía según un tipo general **datos-medioambientales**, luego por categoría **calidad-aire** y, dentro de esta categoría, en función de la localización del sensor. Es necesario puntualizar que la localización no es la ubicación exacta del sensor, puede ser una zona de una casa o una zona de una ciudad o una ciudad dentro de una provincia. Después por el tipo del sensor y finalmente todos los sensores que son de ese tipo, que contendrá el nombre del sensor (la combinación de nombre y el tipo del sensor es única, es decir crean una primary key del sensor en la tabla correspondiente de la base de datos y de este modo la identificamos unívocamente).

Realizaremos un mapping de esta estructura en nuestra base de datos [TimeScaleDB](#), pero antes conozcamos un poco más esta base de datos.

**TimescaleDB** es una base de datos de código abierto diseñada para hacer que SQL sea escalable para datos de series temporales. Está basada en **PostgreSQL**. La ventaja de **TimescaleDB** respecto **PostgreSQL** radica en el rendimiento y la funcionalidad que dispone respecto a los datos temporales. Básicamente, podemos insertar todas nuestras lecturas para un dispositivo en particular en la misma fila, con el mismo timestamp, sin la necesidad de ningún JOIN y sin el riesgo de generar problemas en nuestra base de datos cuando está va aumentando de tamaño.

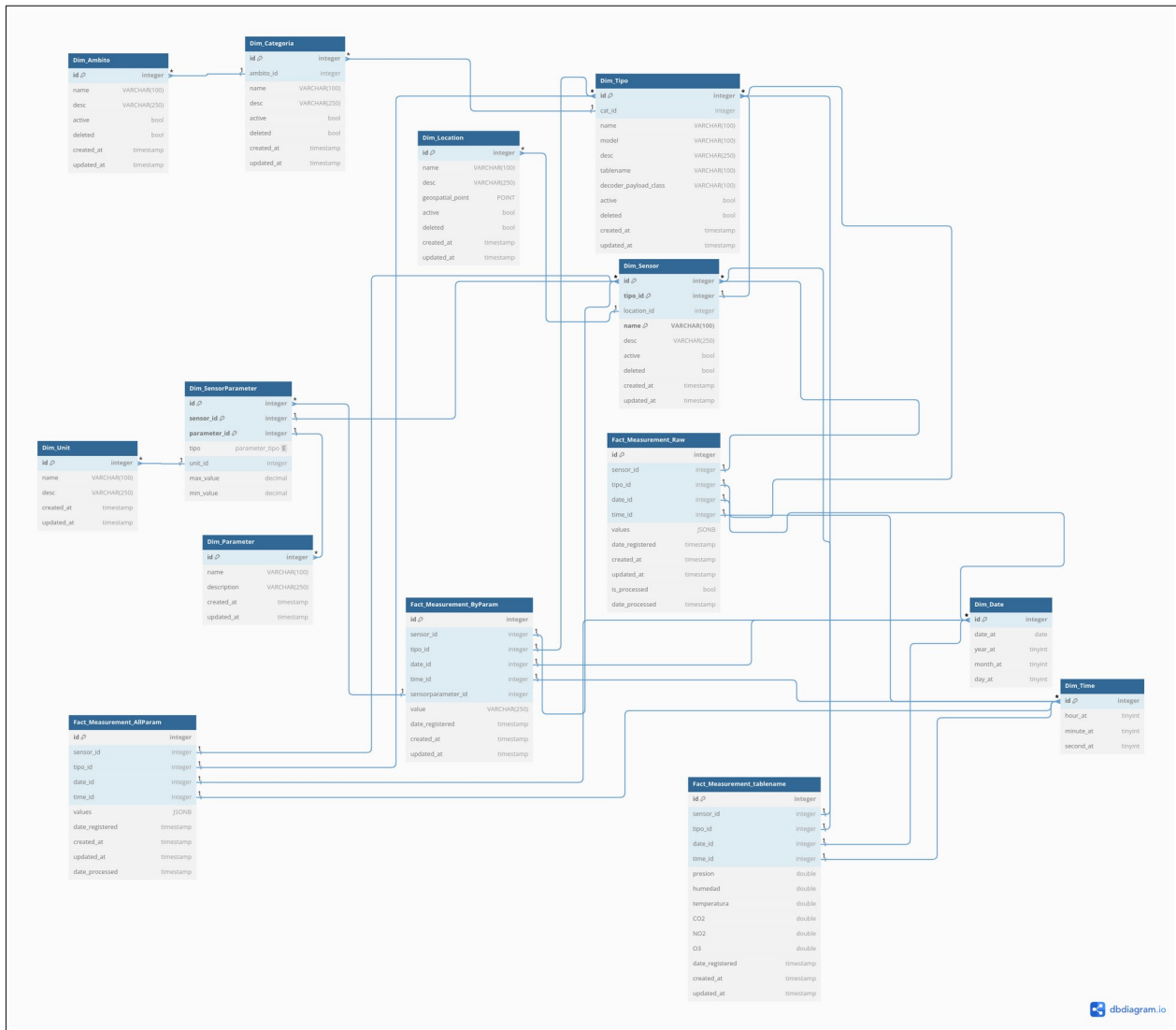
En **TimescaleDB** tenemos las *hypertables* que es similar a una tabla tradicional en una base de datos relacional, pero optimizada específicamente para manejar series temporales de datos. Las *hypertables* se dividen en "fragmentos" y se interconectan mediante un timestamp asociado a la columna para optimizarlas durante las operaciones de series temporales.

Las *hypertables* de **TimescaleDB** ofrecen varias características importantes:

- **Particionamiento Automático:** **TimescaleDB** divide automáticamente los datos en particiones de tiempo, lo que permite una gestión eficiente de grandes volúmenes de datos temporales. Esto significa que no es necesario preocuparse por la creación manual de particiones.
- **Optimización de Consultas:** Las *hypertables* están diseñadas para optimizar la velocidad de las consultas en datos temporales. Utilizan técnicas como la partición automática y la indexación especializada para acelerar la recuperación de datos.
- **Compresión de Datos:** **TimescaleDB** utiliza técnicas de compresión para reducir el tamaño de los datos almacenados en disco. Esto permite almacenar grandes cantidades de datos temporales en un espacio más reducido.
- **Funcionalidades de SQL Estándar:** A pesar de su estructura especializada para datos temporales, las *hypertables* de **TimescaleDB** se comportan como tablas SQL estándar.

Crearemos una base de datos **TimescaleDB** llamada **monitorizacion\_calidad\_aire** que por un lado contiene un inventario de los sensores que tenemos desplegados y en funcionamiento y a la vez, mapea el comportamiento de los **topics** definidos. Por otro lado, contiene la tabla de datos en crudo que después deberemos transformar y guardar en otra(s) tabla(s) de la base de datos que contendrá los datos transformados y limpios.

Pasamos a mostrar un diagrama entidad-relación (erd) que nos muestra tanto el inventario como la relación establecida en los **topics** trasladada al contexto de la base de datos así como las tablas con datos en crudo y datos limpios:



Veamos la definici3n de las tablas en detalle y de sus relaciones:

```

Table Dim_Unit {
  id integer [primary key]
  name VARCHAR(100)
  desc VARCHAR(250)
  created_at timestamp
  updated_at timestamp
}

Table Dim_Sensor {
  id integer [primary key]
  tipo_id integer
  location_id integer
  name VARCHAR(100)
  desc VARCHAR(250)
  active bool
  deleted bool
  created_at timestamp
  updated_at timestamp

  indexes {
    (tipo_id, name) [pk] // composite primary key
    (location_id)
  }
}

```

```

// Use DBML to define your database structure
// Docs: https://dbml.dbdiagram.io/docs

Table Dim_Ambito {
  id integer [primary key]
  name VARCHAR(100)
  desc VARCHAR(250)
  active bool
  deleted bool
  created_at timestamp
  updated_at timestamp
}

Table Dim_Categoria {
  id integer [primary key]
  ambito_id integer
  name VARCHAR(100)
  desc VARCHAR(250)
  active bool
  deleted bool
  created_at timestamp
  updated_at timestamp

  indexes {
    (ambito_id)
  }
}

```

```

Table Dim_Tipo {
  id integer [primary key]
  cat_id integer
  name VARCHAR(100) [unique]
  model VARCHAR(100)
  desc VARCHAR(250)
  tablename VARCHAR(100) [unique]
  decoder_payload_class VARCHAR(100)
  active bool
  deleted bool
  created_at timestamp
  updated_at timestamp

  indexes {
    (cat_id)
    (name) [unique]
    (tablename) [unique]
  }
}

Table Dim_Location {
  id integer [primary key]
  name VARCHAR(100)
  desc VARCHAR(250)
  geospatial_point POINT
  active bool
  deleted bool
  created_at timestamp
  updated_at timestamp
}

```

```

Table Dim_Parameter {
  id integer [primary key]
  name VARCHAR(100)
  description VARCHAR(250)
  created_at timestamp
  updated_at timestamp
}

enum parameter_tipo {
  integer
  decimal
  string
  jsonb
}

Table Dim_SensorParameter {
  id integer [primary key]
  sensor_id integer
  parameter_id integer
  tipo parameter_tipo
  unit_id integer
  max_value decimal
  min_value decimal

  indexes {
    (sensor_id, parameter_id) [pk] // composite
  }
  primary key
    (unit_id)
}

```

```

Table Dim_Date {
  id integer [primary key]
  date_at date
  year_at tinyint
  month_at tinyint
  day_at tinyint
}

Table Dim_Time {
  id integer [primary key]
  hour_at tinyint
  minute_at tinyint
  second_at tinyint
}

Table Fact_Measurement_Raw {
  id integer [primary key]
  sensor_id integer
  tipo_id integer
  date_id integer
  time_id integer
  values JSONB [note: "JSON con los valores
registrados del sensor codificados en binario"]
  date_registered timestamp
  created_at timestamp
  updated_at timestamp
  is_processed bool
  date_processed timestamp

  indexes {
    (sensor_id)
    (tipo_id)
    (is_processed)
    (date_id, time_id)
  }
}

```

```

Table Fact_Measurement_AllParam {
  id integer [primary key]
  sensor_id integer
  tipo_id integer
  date_id integer
  time_id integer
  values JSONB [note: "JSON con los valores
registrados del sensor decodificados"]
  date_registered timestamp
  created_at timestamp
  updated_at timestamp
  date_processed timestamp

  indexes {
    (sensor_id)
    (tipo_id)
    (date_id, time_id)
  }
}

Table Fact_Measurement_ByParam {
  id integer [primary key]
  sensor_id integer
  tipo_id integer
  date_id integer
  time_id integer
  sensorparameter_id integer
  value VARCHAR(250) [note: "Valor registrado del
sensor para el parámetro parameter_id"]
  date_registered timestamp
  created_at timestamp
  updated_at timestamp

  indexes {
    (sensor_id)
    (tipo_id)
    (date_id, time_id)
  }
}

```

```

Table Fact_Measurement_tablename {
  id integer [primary key]
  sensor_id integer
  tipo_id integer
  date_id integer
  time_id integer
  presion double
  humedad double
  temperatura double
  CO2 double
  NO2 double
  O3 double
  date_registered timestamp
  created_at timestamp
  updated_at timestamp

  indexes {
    (sensor_id)
    (tipo_id)
    (date_id, time_id)
  }
}

```

```

Ref: Dim_Ambito.id > Dim_Categoria.ambito_id // many-to-one
Ref: Dim_Categoria.id > Dim_Tipo.cat_id // many-to-one
Ref: Dim_Tipo.id > Dim_Sensor.tipo_id // many-to-one
Ref: Dim_Location.id > Dim_Sensor.location_id // many-to-one
Ref: Dim_Sensor.id > Dim_SensorParameter.sensor_id // many-to-one
Ref: Dim_Unit.id > Dim_SensorParameter.unit_id // many-to-one
Ref: Dim_Sensor.id > Fact_Measurement_Raw.sensor_id // many-to-one
Ref: Dim_Tipo.id > Fact_Measurement_Raw.tipo_id // many-to-one
Ref: Dim_Date.id > Fact_Measurement_Raw.date_id // many-to-one
Ref: Dim_Time.id > Fact_Measurement_Raw.time_id // many-to-one
Ref: Dim_Sensor.id > Fact_Measurement_AllParam.sensor_id // many-to-one
Ref: Dim_Tipo.id > Fact_Measurement_AllParam.tipo_id // many-to-one
Ref: Dim_Date.id > Fact_Measurement_AllParam.date_id // many-to-one
Ref: Dim_Time.id > Fact_Measurement_AllParam.time_id // many-to-one
Ref: Dim_Sensor.id > Fact_Measurement_ByParam.sensor_id // many-to-one
Ref: Dim_Tipo.id > Fact_Measurement_ByParam.tipo_id // many-to-one
Ref: Dim_Date.id > Fact_Measurement_ByParam.date_id // many-to-one
Ref: Dim_Time.id > Fact_Measurement_ByParam.time_id // many-to-one
Ref: Dim_SensorParameter.id > Fact_Measurement_ByParam.sensorparameter_id // many-to-one
Ref: "Dim_SensorParameter"."parameter_id" < "Dim_Parameter"."id"
Ref: "Fact_Measurement_tablename"."sensor_id" < "Dim_Sensor"."id"
Ref: "Fact_Measurement_tablename"."tipo_id" < "Dim_Tipo"."id"
Ref: "Fact_Measurement_tablename"."date_id" < "Dim_Date"."id"
Ref: "Fact_Measurement_tablename"."time_id" < "Dim_Time"."id"

```

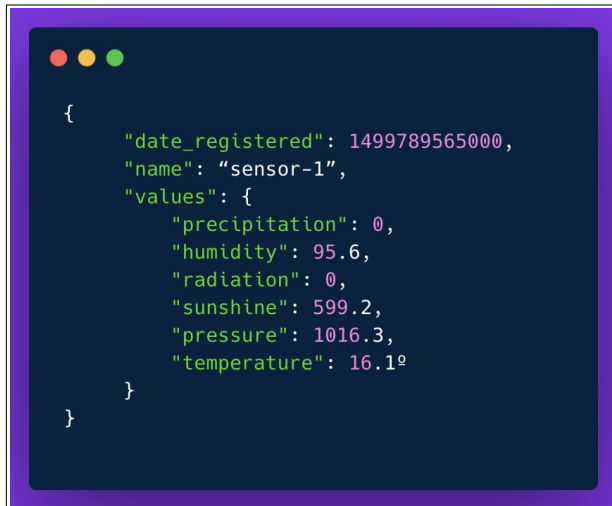
Esta estructura nos permite tener un inventario del ámbito, dentro de él que tipos de sensores medioambientales tenemos y dónde están localizados. También tenemos el tipo de sensor con el que estamos tratando y finalmente el sensor en sí y los datos recibidos por él.

El nombre del tipo de sensor (tabla **Dim\_Tipo**, campo **name**) es único y crea una llave única de manera que no puede repetirse un nombre de tipo. El **topic** contiene el tipo y a partir de ahí, identificamos unívocamente de que tipo de sensor estamos hablando.

Por otro lado, el nombre del sensor (tabla **Dim\_Sensor**, campo **name**) junto con el identificador del tipo (tabla **Dim\_Sensor**, campo **tipo\_id**) crearán una llave única de manera que no puede repetirse un nombre de sensor para cada tipo específico es decir, el nombre de un sensor no puede estar

repetido para un mismo tipo. El **topic** contiene el tipo y el nombre del sensor y a partir de ahí, identificamos unívocamente de que tipo/sensor estamos hablando.

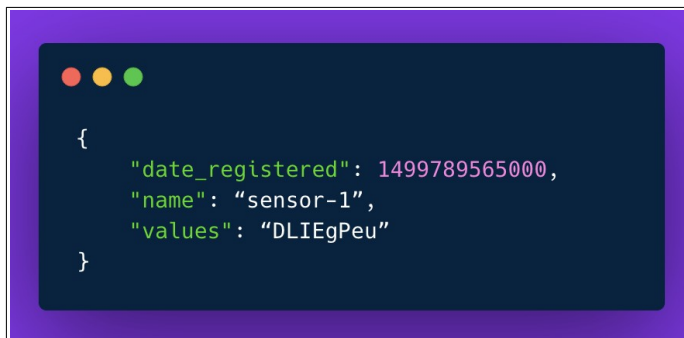
Los sensores enviarán sus datos a nuestro **broker MQTT Mosquitto** que los recibirá y mediante un script que tendremos siempre activo, recogeremos esos datos y los guardaremos en crudo en la tabla **Fact\_Measurements\_Raw**. Estos datos se corresponden con un JSON y a continuación mostramos un ejemplo de como podría ser:



```
{
  "date_registered": 1499789565000,
  "name": "sensor-1",
  "values": {
    "precipitation": 0,
    "humidity": 95.6,
    "radiation": 0,
    "sunshine": 599.2,
    "pressure": 1016.3,
    "temperature": 16.1
  }
}
```

Tenemos el *timestamp* del momento en el que se enviaron los datos desde el sensor, el id del sensor, y los valores que está midiendo. Cabe tener en cuenta que podría ser que el payload que recibiéramos estuviera codificado en binario ya que es una práctica que suele hacerse habitualmente para comprimir la longitud del mensaje. Esto quiere decir que según el tipo de sensor y su codificación interna, puede ser que tengamos que decodificar el payload que recibimos de él.

Por ejemplo podría ser que obtuviéramos un payload como el siguiente:



```
{
  "date_registered": 1499789565000,
  "name": "sensor-1",
  "values": "DLIEgPeu"
}
```

dónde **DLIEgPeu** es el mensaje codificado en binario que deberemos decodificar en el siguiente paso para conseguir un JSON similar a la imagen anterior.

### 3.3 Ingestión de datos via MQTT

Pasamos a mostrar el código del script que llamaremos **get\_data\_mqtt\_broker.py** y que realiza la inserción en la base de datos del payload recibido utilizando las librería para MQTT **paho**, la librería **psycpg2** para realizar la conexión a la base de datos y otras librerías complementarias para poder realizar el proceso:

```

import paho.mqtt.client as mqtt
import psycopg2
from datetime import datetime
from logging import getLogger, basicConfig, ERROR, WARNING, INFO, DEBUG
import json

# Configuración del cliente MQTT
mqtt_broker_host = "localhost"
mqtt_broker_port = 8883
mqtt_username = "myuser"
mqtt_password = "mypassword"

# Configuración de la base de datos TimeScaleDB
database_config = {
    "host": "localhost",
    "port": "5432",
    "database": "monitorizacion_calidad_aire",
    "user": "usuario",
    "password": "contraseña"
}

# create logging object
mqtt_logger = getLogger('MQTT')
db_logger = getLogger('DB')

class SingletonConnection:
    _instance = None

    @staticmethod
    def getInstance():
        if SingletonConnection._instance is None:
            SingletonConnection._instance = getConnection()

        return SingletonConnection._instance

class SingletonMQTTClient:
    _instance = None

    @staticmethod
    def getInstance():
        if SingletonMQTTClient._instance is None:
            SingletonMQTTClient._instance = mqtt.Client()
        return SingletonMQTTClient._instance

# Función para conectar a PostgreSQL
def getConnection():
    try:
        connection = psycopg2.connect(
            host=database_config["host"],
            port=database_config["port"],
            database=database_config["database"],
            user=database_config["user"],
            password=database_config["password"]
        )

        return connection
    except (Exception, psycopg2.Error) as error:

```

```

print("Error al conectar a PostgreSQL:", error)
db_logger.error(f'Error al conectar a PostgreSQL: {error}.')

return None

def on_connect(client, userdata, flags, rc):
    print("Conectado al broker MQTT con código de resultado:", rc)
    client.subscribe("datos-medioambientales/calidad-aire/+/+/+", 0)

def on_subscribe(client, obj, mid, granted_qos):
    """
    This runs once subscribed, and simply logs the result.
    """
    mqtt_logger.info(f'Subscribed mid: {str(mid)}, qos: {str(granted_qos)}')

def on_message(client, userdata, msg):
    try:
        payload = json.loads(msg.payload)

        # Obtener la conexión a la base de datos
        conn = SingletonConnection.getInstance()

        if conn is not None:
            topic_parts = msg.topic.split("/")
            tipo_name = topic_parts[-2]
            sensor_name = topic_parts[-1]

            # Buscar el id_tipo a partir del nombre del tipo
            id_tipo = get_tipo_id(conn, tipo_name)

            if id_tipo is not None:
                # Buscar el id_sensor a partir del id_tipo y el nombre del sensor
                id_sensor = get_sensor_id(conn, id_tipo, sensor_name)

                if id_sensor is not None:
                    # Insertar el payload en la tabla Fact_Measurement_Raw
                    insert_payload(conn, id_tipo, id_sensor, payload)

            conn.close()
        except Exception as e:
            print("Error al procesar el mensaje MQTT:", e)

# Función para buscar el id_sensor a partir del id del tipo en el topic
def get_tipo_id(conn, tipo_name):
    try:
        cur = conn.cursor()
        sql = """
        SELECT id FROM Dim_Tipo WHERE name = %s
        """
        cur.execute(sql, (tipo_name,))
        tipo_id = cur.fetchone()
        cur.close()

        return tipo_id[0] if tipo_id is not None else None
    except psycopg2.Error as e:
        print("Error al obtener el id_tipo:", e)
        return None

# Función para buscar el id_sensor a partir del id del tipo y del nombre del sensor en el topic

```



```

def get_sensor_id(conn, tipo_id, sensor_name):
    try:
        cur = conn.cursor()
        sql = """
            SELECT id
            FROM Dim_Sensor
            WHERE tipo_id = %s AND name = %s
        """
        cur.execute(sql, (tipo_id, sensor_name))
        sensor_id = cur.fetchone()
        cur.close()
        return sensor_id[0] if sensor_id is not None else None
    except psycopg2.Error as e:
        print("Error al obtener el id_sensor:", e)
        return None

def get_date_id(conn, current_date):
    try:
        cur = conn.cursor()
        year = current_date.year
        month = current_date.month
        day = current_date.day

        # Verificar si el año, mes y día ya existen en la tabla Dim_Date
        sql = """
            SELECT id
            FROM Dim_Date
            WHERE year_at = %s AND month_at = %s AND day_at = %s
        """
        cur.execute(sql, (year, month, day))
        date_id = cur.fetchone()

        # Si no existe, insertar la fecha en la tabla Dim_Date y obtener el ID resultante
        if date_id is None:
            sql = """
                INSERT INTO Dim_Date (year_at, month_at, day_at)
                VALUES (%s, %s, %s) RETURNING id
            """
            cur.execute(sql, (year, month, day))
            date_id = cur.fetchone()[0]
        else:
            date_id = date_id[0]

        return date_id
    except psycopg2.Error as e:
        print("Error al obtener o insertar la fecha en la tabla Dim_Date:", e)
        return None

def get_time_id(conn, current_date):
    try:
        cur = conn.cursor()
        hour = current_date.hour
        minute = current_date.minute
        second = current_date.second

        # Verificar si la hora, minuto y segundo ya existen en la tabla Dim_Time
        sql = """
            SELECT id
            FROM Dim_Time
        """

```

```

        WHERE hour_at = %s AND minute_at = %s AND second_at = %s
        """
    cur.execute(sql, (hour, minute, second))
    time_id = cur.fetchone()

    # Si no existe, insertar la hora en la tabla Dim_Time y obtener el ID resultante
    if time_id is None:
        sql = """
            INSERT INTO Dim_Time (hour_at, minute_at, second_at)
            VALUES (%s, %s, %s) RETURNING id
            """
        cur.execute(sql, (hour, minute, second))
        time_id = cur.fetchone()[0]
    else:
        time_id = time_id[0]

    return time_id
except psycopg2.Error as e:
    print("Error al obtener o insertar la hora en la tabla Dim_Time:", e)
    return None

# Función para insertar el payload en la tabla Fact_Measurement_Raw
def insert_payload(conn, id_tipo, id_sensor, payload):
    try:
        cur = conn.cursor()

        # Obtener la fecha y hora actuales
        date_registered = payload.get('date_registered', None)

        date_created = datetime.now()
        date_id = get_date_id(date_created);
        time_id = get_time_id(date_created);

        # Verificar si se pudieron obtener date_id y time_id
        if date_id is not None and time_id is not None:
            # Consulta SQL para insertar el payload en la tabla
            sql = """
                INSERT INTO Fact_Measurement_Raw (
                sensor_id, tipo_id, date_id, time_id, values, date_registered,
                created_at, updated_at, is_processed)
                VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)
                """
            # Insertar el payload en la tabla
            cur.execute(
                sql, (id_sensor, id_tipo, date_id, time_id, payload,
                    date_registered, current_datetime,
                    current_datetime, 0)
            )
            conn.commit()

            print("Payload insertado correctamente en la base de datos")
        else:
            print("Error al obtener date_id o time_id")

        cur.close()

        print("Payload insertado correctamente en la base de datos")
    except psycopg2.Error as e:
        print("Error al insertar el payload en la base de datos:", e)

```

```

def client_init():
    ##### Configuración del cliente MQTT #####

    # crear un cliente
    client = SingletonMQTTClient.getInstance()

    # create callbacks from functions above
    client.on_connect = on_connect
    client.on_subscribe = on_subscribe
    client.on_message = on_message

    # set logging level
    client.enable_logger(mqtt_logger)

    # set up connection
    client.username_pw_set(username=mqtt_username, password=mqtt_password)

    # Conexión al broker MQTT
    client.tls_set("/etc/mosquitto/certs/server.crt")
    client.connect(mqtt_broker_host, mqtt_broker_port)

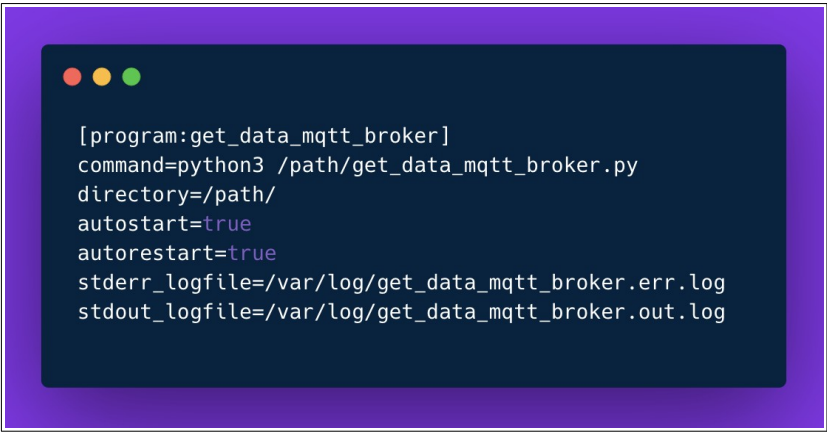
    return client

if __name__ == '__main__':
    # Inicializar el cliente MQTT
    client = client_init()

    # Mantener el cliente MQTT en funcionamiento continuo
    client.loop_forever()

```

**Supervisor** se encargará de ejecutar el script como un servicio, reiniciándolo automáticamente si se detiene. Para ello necesitamos crear un archivo de configuración para nuestro script en la carpeta `/etc/supervisor/conf.d/`. Nuestro script de Python se llama **get\_data\_mqtt\_broker.py**, entonces crearemos un archivo llamado **get\_data\_mqtt\_broker.conf** y estará ubicado en `/etc/supervisor/conf.d/get_data_mqtt_broker.conf`. Dentro de este archivo, tendremos la configuración que se seguirá para ejecutar el script **get\_data\_mqtt\_broker.py**:



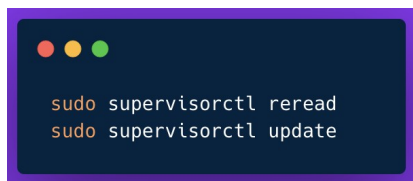
```

[program:get_data_mqtt_broker]
command=python3 /path/get_data_mqtt_broker.py
directory=/path/
autostart=true
autorestart=true
stderr_logfile=/var/log/get_data_mqtt_broker.err.log
stdout_logfile=/var/log/get_data_mqtt_broker.out.log

```

Dónde “path” es la ruta real donde se encuentra nuestro script.

Después de crear el archivo de configuración, realizaremos una recarga de Supervisor para que cargue la nueva configuración:



```
sudo supervisorctl reread
sudo supervisorctl update
```

De esta manera Supervisor cargará el nuevo archivo de configuración y comenzará a supervisar y gestionar la ejecución de nuestro script.

### 3.4 Transformación y almacenamiento de datos

Una vez guardado el payload recibido en la tabla **Fact\_Measurement\_Raw** utilizaremos el orquestador de código libre **Apache Airflow**. **Apache Airflow** es una plataforma de código abierto para desarrollar, programar y monitorear flujos de trabajo orientados por lotes. **Airflow** utiliza un framework basado en Python que es extensible y que le permite crear flujos de trabajo que se conectan con prácticamente cualquier tecnología. Dispone también de una interfaz web que permite gestionar el estado de sus flujos de trabajo.

Con los datos guardados en la tabla **Fact\_Measurement\_Raw** utilizaremos **Apache Airflow** para ejecutar tareas cada cierto intervalo que consisten en:

- Recuperar los datos de la tabla **Fact\_Measurement\_Raw** que no hayan sido procesados y,
  - Si los datos están codificados en binario, aplicarles una función que los decodifique y guardar los datos en tres tablas:
    - **Fact\_Measurement\_AllParam:** Aquí guardaremos el JSON resultante de decodificar los datos en binario.
    - **Fact\_Measurement\_ByParam:** Aquí guardaremos registro a registro cada uno de los datos de los que se compone el JSON resultante de decodificar los datos en binario.
    - **Fact\_Measurement\_<tablename>:** Cada tipo de sensor que tenemos registrado en la tabla **Dim\_Tipo** registra unos datos concretos. Para cada tipo de sensor, crearemos una tabla específica con las columnas mapeando los datos que registran esos tipos de sensores. Básicamente, a partir del tipo del sensor, **tipo\_id** podremos obtener el campo **tablename** de la tabla **Dim\_Tipo**, que es único y no puede repetirse, que nos indica a que tabla específica está vinculado este tipo de sensor.
  - Si los datos no están codificados en binario, guardarlos directamente en las tablas **Fact\_Measurement\_AllParam**, **Fact\_Measurement\_ByParam** y finalmente en la tabla **Fact\_Measurement\_<tablename>**.

Con este enfoque, a partir de los datos enviados vía MQTT de los sensores, los guardamos primero en crudo en la tabla **Fact\_Measurement\_Raw**, después los procesamos, decodificamos (si es necesario) y guardamos en tres tablas auxiliares **Fact\_Measurement\_AllParam**,

**Fact\_Measurement\_ByParam** y **Fact\_Measurement\_<tablename>**, que nos permiten tener los datos agrupados en diferentes niveles de granularidad para posteriormente poder analizarlos más fácilmente.

Si revisamos el modelo de datos, vemos que la tabla **Fact\_Measurement\_AllParam** es similar a la tabla **Fact\_Measurement\_Raw**. La diferencia entre ellas radica en que la primera contiene los datos decodificados y la segunda contiene los datos crudos tal y como los ha enviado el sensor inicialmente. Ambas tablas tienen en común que las dos contienen un campo JSONB donde se guardan todos los registros del sensor. Esto nos permite tener en un único registro de la tabla, todas las medidas que ha registrado el sensor.

La tabla **Fact\_Measurement\_<tablename>** contiene los valores específicos de cada sensor desglosados en tantas columnas como medidas registra el mismo. Además tienen tipos de dato específicos. Tendremos tantas tablas de este tipo como tipos de sensores tengamos distribuidos en el sistema. Por lo tanto a partir del **id** del tipo de sensor, podemos obtener la tabla que tiene asociada y de esta manera a través de su campo **tablename**, podemos guardar allí cada medida registrada. Esto nos permite tener desglosado cada envío realizado por el sensor con cada valor medido por el sensor para cada registro de la tabla.

La tabla **Fact\_Measurement\_ByParam** contiene en cada registro una medida particular y su valor. Como los datos pueden ser heterogéneos, la tabla contiene el valor **sensorparameter\_id** que se relaciona con la tabla **Dim\_SensorParameter** que contiene para cada sensor todas las magnitudes que mide. Para cada una de ellas tenemos la unidad de medida, el tipo de dato que mide que puede ser integer, decimal, string o un jsonb, y los valores máximo y mínimo que puede registrar. Los campos que identifican al sensor (**sensor\_id**) y al parámetro que mide (**parameter\_id**) crean conjuntamente una llave única. Por lo tanto, cada medida que obtenemos, se guardará en un registro de la tabla **Fact\_Measurement\_ByParam**. Para ello a partir del **id** del sensor, recorreremos todos sus parámetros asociados y a través del nombre (campo **name**) del parámetro identificaremos el **id** y guardaremos su valor como string en el campo **value** de la tabla.

En el diagrama de entidad relación que mostramos anteriormente, vemos que aparece una tabla **Fact\_Measurement\_tablename** como ejemplo. En ella aparecen los campos específicos de *presion*, *humedad*, *temperatura*, *CO2*, *NO2* y *O3* que mapean los datos recibidos.

Veamos la estructura de ejemplo de la tabla en la imagen siguiente:

Fact_Measurement_tablename	
id	integer
sensor_id	integer
tipo_id	integer
date_id	integer
time_id	integer
presion	double
humedad	double
temperatura	double
CO2	double
NO2	double
O3	double
date_registered	timestamp
created_at	timestamp
updated_at	timestamp

Como ejemplo, imaginemos que el JSON que decodificaremos en este caso nos proporcionará unos datos similares a:



```
{
  "date_registered": 1499789565000,
  "name": "sensor-1",
  "values": {
    "presion": 0,
    "humedad": 95.6,
    "temperatura": 27,
    "CO2": 599.2,
    "NO2": 1016.3,
    "O3": 16.1
  }
}
```

A partir de estos datos, podremos mapearlos respecto a las columnas de la tabla correspondiente y guardarlos allí.

Por lo tanto, tendremos toda una familia de tablas con valores concretos de tipos de sensores concretos.

Como vemos tenemos diferentes niveles de agregación y granularidad de los datos. Esto nos permite tener un modelo versátil y facilitará la posterior implementación del análisis de los datos.

Ahora continuando con nuestro flujo, para decodificar los datos en crudo y guardarlos en las tablas correspondientes utilizaremos **pandas** y **SQLAlchemy**. Nos conectaremos a la base de datos y ejecutaremos un script que buscará los registros de la tabla **Fact\_Measurement\_Raw** que no hayan sido procesados, es decir aquellos tales que el campo booleano **is\_processed** sea **0** y los recorreremos teniendo en cuenta su vínculo con la tabla **Dim\_Tipo** para obtener el campo **decoder\_payload\_class**.

Respecto a la importancia del campo **decoder\_payload\_class**:

- Este campo contiene el nombre de la clase que realiza la transformación de los datos binarios por cada tipo de sensor. Para ello aplicaremos el patrón de diseño *Strategy* que nos permitirá tener una familia de algoritmos que estarán encapsulados en una clase separada y podremos hacer así sus objetos intercambiables. Si el valor del campo es nulo o vacío esto indicará que no es necesario aplicar ninguna transformación a los datos.

A continuación se muestra un código de ejemplo de aplicación del patrón *Strategy* en las diferentes funciones de transformación de los datos binarios de los sensores según su tipo:

```
class SensorDecoder:
    def decode(self, payload):
        raise NotImplementedError()

class TipoModelo1SensorDecoder(SensorDecoder):
    def decode(self, payload):
        # Implementa la lógica de decodificación para sensores de tipo Modelo1
        pass
```

```

class TipoModelo2SensorDecoder(SensorDecoder):
    def decode(self, payload):
        # Implementa la lógica de decodificación para sensores de tipo Modelo2
        pass

class TipoModelo3SensorDecoder(SensorDecoder):
    def decode(self, payload):
        # Implementa la lógica de decodificación para sensores de tipo Modelo3
        pass

...

class SensorDecoderContext:
    def __init__(self, decoder):
        self._decoder = decoder

    def set_decoder(self, decoder):
        self._decoder = decoder

    def get_decoder(self):
        return self._decoder;

    def decode_payload(self, payload):
        return self._decoder.decode(payload)

```

Vamos a clarificar un concepto del que haremos uso vía **Apache Airflow: DAG**. Un **DAG** (Directed Acyclic Graph) en **Apache Airflow** es un grafo dirigido acíclico que representa un conjunto de tareas y sus dependencias. En **Airflow**, un **DAG** se utiliza para definir y orquestar el flujo de trabajo de un conjunto de tareas, donde cada tarea representa una unidad de trabajo que se debe ejecutar. Las tareas están interconectadas por dependencias, lo que significa que una tarea puede depender del resultado de una o más tareas anteriores.

Creemos un **DAG** de **Apache Airflow** para que se ejecute cada 5 minutos. Configuraremos el **DAG** para que no permita que se ejecute una nueva instancia si existe una que está ejecutándose en ese momento.

De manera que el flujo será la inserción continua de datos por parte de los sensores en la base de datos y cada 5 minutos se ejecutará el DAG de Airflow que constará de varias tareas/tasks que permitirán procesar los nuevos datos transformándolos si es necesario, e insertándolos en las tablas correspondientes de la base de datos. La tarea inicial tomará los datos no procesados y los decodificará para guardarlos finalmente en la tabla **Fact\_Measurements\_AllParam**. Una vez ejecutada esta tarea se ejecutarán dos tareas justo después. Una tarea tomará los últimos datos insertados en la tabla **Fact\_Measurements\_AllParam** y los procesará para guardarlos en la tabla **Fact\_Measurements\_ByParam**. La otra tarea tomará también los datos de la tabla **Fact\_Measurements\_AllParam** buscará para cada registro vía su vínculo con la tabla **Dim\_Tipo** que tabla específica tiene asociada ese tipo de sensor (a través del campo **tablename**) y guardará los datos en esa tabla.

Definimos el script de Python **etl\_decode\_data\_to\_fact\_measurements\_allparam.py** que quedará incluido dentro de la primera tarea y nos permite guardar los datos decodificados en la tabla **Fact\_Measurements\_AllParam**:

```
from datetime import datetime, timedelta
import pandas as pd
from sqlalchemy import create_engine
from sensor_decoders import (
    SensorDecoderContext,
    TipoModelo1SensorDecoder,
    TipoModelo2SensorDecoder,
    TipoModelo3SensorDecoder,
    ...
)

# Mapeo de nombres de clases de decodificadores a las clases mismas
decoder_classes = {
    'TipoModelo1SensorDecoder': TipoModelo1SensorDecoder,
    'TipoModelo2SensorDecoder': TipoModelo2SensorDecoder,
    'TipoModelo3SensorDecoder': TipoModelo3SensorDecoder,
    # Agrega aquí el resto de las clases de decodificadores
}

# Configuración de la base de datos TimeScaleDB
database_config = {
    "host": "localhost",
    "port": "5432",
    "database": "monitorizacion_calidad_aire",
    "user": "usuario",
    "password": "contraseña"
}

# Connection URI:
# schema_identifier://username:password@host:port/db
connection_uri = f"postgresql+psycopg2://{database_config['user']}:{database_config['password']}@{database_config['host']}:{database_config['port']}/{database_config['database']}"

# Crear motor de base de datos
db_engine = create_engine(connection_uri)

# Crear una instancia del contexto de decodificación
decoder_context = SensorDecoderContext()

def extract_data():
    # Consulta SQL para obtener los datos sin procesar de los sensores con su tipo
    sql_query = """
    SELECT
        sensor_data.id,
        sensor_data.tipo_id,
        sensor_data.sensor_id,
        sensor_data.date_id,
        sensor_data.time_id,
        sensor_data.values,
        tipo.decoder_payload_class as class
    FROM Fact_Measurement_Raw AS sensor_data
    JOIN Dim_Tipo AS tipo ON tipo.id = sensor_data.tipo_id
    """
```



```

WHERE sensor_data.is_processed = 0
ORDER BY date_id, time_id
"""

# Ejecutar la consulta y cargar los resultados en un DataFrame
df = pd.read_sql(sql_query, db_engine)
return df

def apply_decoding(row):
    payload = row['values']
    if row['class']:
        decoder_class_name = row['class']

        # Comprobamos que la clase esté en el diccionario de clases
        if decoder_class_name in decoder_classes:
            decoder_class = decoder_classes[decoder_class_name]
            decoder_context.set_decoder(decoder_class)
            decoded_data = decoder_context.decode_payload(payload)
            return decoded_data
        else:
            print(f"Error: Clase de decodificador '{decoder_class_name}' no encontrada.")
    else:
        print("No se cumple la condición para decodificar el payload.")

    # Retornar el payload sin aplicar ninguna decodificación
    return payload

def transform_data(df):
    df['decoded_data_json'] = df.apply(apply_decoding, axis=1)
    return df

def save_data(df):
    # Guardar los datos procesados en la tabla Fact_Measurement_AllParam
    # utilizando SQLAlchemy
    for index, row in df.iterrows():
        conn = db_engine.connect()
        trans = conn.begin()
        try:
            conn.execute("""
            INSERT INTO Fact_Measurement_AllParam (
                sensor_id,
                tipo_id,
                date_id,
                time_id,
                date_registered,
                values,
                date_created,
                date_updated
            )
            VALUES (%s, %s, %s, %s, %s, %s, NOW(), NOW())
            """, (row['sensor_id'], row['tipo_id'], row['date_id'],
                row['time_id'], row['date_registered'], row['decoded_data_json']))

        # Actualizar el campo is_processed en la tabla Fact_Measurement_Raw
        conn.execute("""
        UPDATE Fact_Measurement_Raw
        SET is_processed = 1
        WHERE id = %s""", (row['id'],))

```

```

        trans.commit()
    except:
        trans.rollback()
        raise
    finally:
        conn.close()

# Definimos la función principal del script
def main_etl_transform():
    # Código para ejecutar el pipeline completo
    df = extract_data()
    transformed_df = transform_data(df)
    save_data(transformed_df)

if __name__ == "__main__":
    main_etl_transform()

```

Ahora crearemos el siguiente script que procesará los últimos datos de la tabla **Fact\_Measurement\_AllParam** y los guardará en la tabla **Fact\_Measurement\_ByParam**. Este script se llamará:

**save\_decoded\_data\_from\_fact\_measurements\_allparam\_to\_fact\_measurements\_byparam.py**

```

import pandas as pd
from sqlalchemy import create_engine
from datetime import datetime
import json

# Configuración de la conexión a la base de datos
database_config = {
    "host": "localhost",
    "port": "5432",
    "database": "monitorizacion_calidad_aire",
    "user": "usuario",
    "password": "contraseña"
}

# Connection URI:
# schema_identifier://username:password@host:port/db
connection_uri = f"postgresql+psycopg2://{database_config['user']}:{database_config['password']}@{database_config['host']}:{database_config['port']}/{database_config['database']}"

# Crear el motor de la base de datos
db_engine = create_engine(connection_uri)

def extract_data():
    # Obtener el último registro de Fact_Measurement_ByParam
    max_date_time_query = "
        SELECT MAX(date_id) AS max_date, MAX(time_id) AS max_time
        FROM Fact_Measurement_ByParam"

    max_date_time_result = pd.read_sql(max_date_time_query, db_engine)
    max_date = max_date_time_result['max_date'].iloc[0]
    max_time = max_date_time_result['max_time'].iloc[0]

```

```

# Obtener los datos de Fact_Measurement_AllParam que no están en
# en la tabla Fact_Measurement_ByParam
all_data_query = f"""
SELECT *
FROM Fact_Measurement_AllParam
WHERE (date_id > {max_date} OR (date_id = {max_date} AND time_id > {max_time}))
"""
all_data_df = pd.read_sql(all_data_query, db_engine)

return all_data_df

def save_data(df):
    # Iterar sobre los registros obtenidos
    for index, row in df.iterrows():
        # Leer el JSONB y convertirlo a un diccionario
        values_dict = json.loads(row['values'])

        # Iterar sobre las claves/valores del diccionario
        for key, value in values_dict.items():
            # Consulta SQL para obtener el id del parámetro desde Dim_Parameter
            parameter_id_query = f"SELECT id FROM Dim_Parameter WHERE name = '{key}'"
            parameter_id_result = db_engine.execute(parameter_id_query)
            parameter_id = parameter_id_result.scalar()

            # Si el parámetro no existe, darlo de alta en Dim_Parameter
            if parameter_id is None:
                db_engine.execute(f"INSERT INTO Dim_Parameter (name) VALUES ('{key}')"
                parameter_id_result = db_engine.execute(parameter_id_query)
                parameter_id = parameter_id_result.scalar()

            # Consulta SQL para obtener el id de Dim_SensorParameter
            sensor_parameter_query = f"""
            SELECT id
            FROM Dim_SensorParameter
            WHERE sensor_id = {row['sensor_id']}
            AND parameter_id = {parameter_id}
            """
            sensor_parameter_result = db_engine.execute(sensor_parameter_query)
            sensor_parameter_id = sensor_parameter_result.scalar()

            # Si no existe, darlo de alta en Dim_SensorParameter
            if sensor_parameter_id is None:
                db_engine.execute(f"""
                INSERT INTO Dim_SensorParameter (sensor_id, parameter_id, tipo)
                VALUES ({row['sensor_id']}, {parameter_id}, 'string')
                """)
                sensor_parameter_result = db_engine.execute(sensor_parameter_query)
                sensor_parameter_id = sensor_parameter_result.scalar()

            # Insertar en Fact_Measurement_ByParam
            db_engine.execute("""
            INSERT INTO Fact_Measurement_ByParam(
                sensor_id, tipo_id, date_id, time_id, sensorparameter_id,
                value, date_registered, created_at, updated_at)
            VALUES (%s, %s, %s, %s, %s, %s, %s, NOW(), NOW())""",
            (row['sensor_id'], row['tipo_id'], row['date_id'], row['time_id'],
            sensor_parameter_id, str(value), row['date_registered']))

```

```
# Definimos la función principal del script
def main_data_to_fact_measurements_byparam():
    df = extract_data()
    save_data(df)

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

Ahora crearemos el siguiente script que procesará los últimos datos de la tabla **Fact\_Measurement\_AllParam** y los guardará en cada tabla definida por tipo de sensor y que podemos encontrar en el campo **tablename** de la tabla **Dim\_Tipo**. Este script se llamará:

**save\_decoded\_data\_from\_fact\_measurements\_allparam\_to\_sensor\_type\_table.py**

```
import pandas as pd
from sqlalchemy import create_engine
from datetime import datetime
import json

# Configuración de la conexión a la base de datos
database_config = {
    "host": "localhost",
    "port": "5432",
    "database": "monitorizacion_calidad_aire",
    "user": "usuario",
    "password": "contraseña"
}

# Connection URI:
# schema_identifier://username:password@host:port/db
connection_uri = f"postgresql+psycopg2://{database_config['user']}:{database_config['password']}@{database_config['host']}:{database_config['port']}/{database_config['database']}"

# Crear el motor de la base de datos
db_engine = create_engine(connection_uri)

def extract_data():
    # Consulta SQL para obtener los nombres de tabla únicos de Dim_Tipo
    distinct_tables_query = "SELECT DISTINCT tablename FROM Dim_Tipo"
    distinct_tables_result = pd.read_sql(distinct_tables_query, db_engine)

    # Lista para almacenar los DataFrames de cada tabla
    dataframes_list = []

    # Para cada tabla obtenida
    for table_name in distinct_tables_result['tablename']:
        # Consulta SQL para obtener el maximo de date_id y time_id de la tabla
        max_date_time_query = f"SELECT MAX(date_id) AS max_date, MAX(time_id) AS max_time FROM {table_name}"

        max_date_time_result = pd.read_sql(max_date_time_query, db_engine)
        max_date = max_date_time_result['max_date'].iloc[0]
```

```

max_time = max_date_time_result['max_time'].iloc[0]

# Consulta SQL para obtener los datos de Fact_Measurement_AllParam que
# no están en la tabla

all_data_query = f"""
SELECT *
FROM Fact_Measurement_AllParam
WHERE (date_id > {max_date} OR (date_id = {max_date} AND time_id > {max_time}))
"""

dataframes_list.append(pd.read_sql(all_data_query, db_engine))

return dataframes_list

def save_data(dataframes_list):
    conn = db_engine.connect()
    trans = conn.begin()
    try:
        for df in dataframes_list:
            for index, row in df.iterrows():
                # Leer el JSONB y convertirlo a diccionario
                values_dict = json.loads(row['values'])

                # Iterar sobre las claves/valores del diccionario
                columns_values = []
                for key, value in values_dict.items():
                    # Consulta SQL para obtener el id de Dim_SensorParameter y el tipo
                    # asociado a la key
                    sensor_parameter_query = f"""
                    SELECT id, tipo
                    FROM Dim_SensorParameter
                    WHERE sensor_id = {row['sensor_id']}
                    AND parameter_id = (SELECT id FROM Dim_Parameter WHERE name = '{key}')
                    """
                    sensor_parameter_result = conn.execute(sensor_parameter_query)
                    sensor_parameter_id, tipo = sensor_parameter_result.fetchone()

                    # Realizar la conversión del valor según el tipo obtenido
                    if tipo == 'integer':
                        converted_value = int(value)
                    elif tipo == 'decimal':
                        converted_value = float(value)
                    elif tipo == 'string':
                        converted_value = str(value)
                    elif tipo == 'jsonb':
                        converted_value = json.dumps(value) # Convertir a cadena JSON

                    columns_values.append((key, converted_value))

                # Insertar los datos en la tabla correspondiente
                table_name = row['tablename']

                # Insertar en la tabla definida por tablename
                columns_str = ', '.join([f"{col[0]}" for col in columns_values])
                values_str = ', '.join(
                    [f"{col[1]}" if isinstance(col[1], str) else str(col[1]) for col in columns_values]
                )

                conn.execute(f"INSERT INTO {table_name} (

```

```

        sensor_id, tipo_id, date_id, time_id, {key}, date_registered,
        created_at, updated_at)
        VALUES (%s, %s, %s, %s, %s, %s, NOW(), NOW())""",
        (row['sensor_id'], row['tipo_id'], row['date_id'], row['time_id'],
        converted_value, row['date_registered']))

    trans.commit()
except:
    trans.rollback()
    raise
finally:
    conn.close()

# Definimos la función principal del script
def main_data_to_sensor_type_table():
    dataframes_list = extract_data()
    save_data(dataframes_list)

if __name__ == "__main__":
    main_data_to_sensor_type_table()

```

Asumimos que los scripts anteriores son accesibles desde nuestra instancia de Apache Airflow.

Mostramos el código para poder ejecutar los scripts anteriores:

```

from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta

# Importar las funciones definidas en los scripts de Python
from etl_decode_data_to_fact_measurements_allparam import main_etl_transform

from save_decoded_data_from_fact_measurements_allparam_to_fact_measurements_byparam
import main_data_to_fact_measurements_byparam

from save_decoded_data_from_fact_measurements_allparam_to_sensor_type_table import
main_data_to_sensor_type_table

# Definir los argumentos predeterminados del DAG
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2024, 3, 26),
    'retries': 1,
    'retry_delay': timedelta(minutes=1)
}

# Definir el DAG
dag = DAG(
    'sensor_data_processing',
    default_args=default_args,
    description='DAG para procesar datos de sensores',
    schedule_interval=timedelta(minutes=5),
    max_active_runs=1 # Permitir solo una instancia activa

```

```

)

# Definir las tareas del DAG
# Task para ejecutar el primer script
etl_task = PythonOperator(
    task_id='run_etl_decode_data',
    python_callable=main_etl_transform,
    dag=dag
)

# Task para ejecutar el segundo script
save_byparam_task = PythonOperator(
    task_id='run_save_to_fact_measurements_byparam',
    python_callable=main_data_to_fact_measurements_byparam,
    dag=dag
)

# Task para ejecutar el tercer script
save_sensor_table_task = PythonOperator(
    task_id='run_save_to_sensor_type_table',
    python_callable=main_data_to_sensor_type_table,
    dag=dag
)

# Establecer las dependencias entre las tareas
etl_task >> [save_byparam_task, save_sensor_table_task]

```

### 3.5 Análisis y limpieza de datos

Ahora realizaremos una limpieza de los datos. Debemos abordar un tema que afecta a la mayoría de los conjuntos de datos, los **valores faltantes**. Dado que los datos de IoT a menudo se recopilan mediante sensores que funcionan con baterías y en ubicaciones remotas, es posible que se produzcan fallos en la transmisión o lectura de los datos. Podríamos tener:

- Interrupciones en el servicio de red (Problemas de conectividad).
- La batería podría agotarse (Sin batería).
- Otros factores fuera de nuestro control.

Esto significa que es posible que no dispongamos de datos durante algunas horas o días, o que algunas medidas estén incompletas y nos falten datos de una o varias magnitudes físicas que mide el sensor.

Debemos preguntarnos como abordar este tema de datos faltantes ya que tenemos diversas estrategias. Respecto a los datos faltantes podemos procesarlos:

- Directamente durante la recopilación de los datos), lo que a menudo es necesario si aplicamos un algoritmo en tiempo real y así poder ahorrarnos espacio en disco.
- Almacenando las medidas de todos modos y limpiando los datos durante la parte de análisis.

Vamos a estudiar como tratar los datos faltantes. Para limpiar estos datos faltantes, podemos eliminar los registros con valores nulos pero esta estrategia nos hace perder valores que pueden ser útiles.

Otro abordaje del problema sería informar los valores faltantes con valores que tengan sentido en el contexto en el que estamos y, de esta manera, no perderíamos información.

Nosotros utilizaremos la librería de código abierto **pandas** que permite desde la manipulación hasta el análisis de los datos.

Dependiendo de la cantidad de datos que faltan, y en qué columna aparecen, tenemos diferentes métodos a nuestra disposición para tratar con ellos:

- Si solo faltan unas pocas observaciones, podemos rellenarlas con el valor medio de la serie. La librería **pandas** nos ofrece varias alternativas como *forward-fill* o *backward-fill*, que nos permite rellenar los datos faltantes con el primero de la serie o con el último de ellos.
- También podemos eliminar los registros con valores faltantes, pero esta acción reducirá la cantidad de datos disponibles para futuros análisis y para modelos de machine learning, por lo tanto podría jugar en nuestra contra.

Para detectar valores faltantes podemos primero utilizar **Pandas** para cargar los datos en un *dataframe* que llamaremos **df**. Una vez tenemos el *dataframe* podemos realizar acciones con él que nos permitan analizar cómo son los datos.

En nuestro caso, el método **df.info()** nos permite obtener información muy útil del *dataset* y nos permitirá saber cuantos registros por columna son no nulos sobre el total de registros y, por lo tanto, el número de valores faltantes por columna del dataframe.

Una vez detectados valores faltantes en el *dataframe*, podemos optar por eliminarlos usando el método **df.dropna()** pero como hemos dicho, esto elimina todo el registro y si por ejemplo en un registro tenemos 10 columnas de las cuáles dos o tres tienen valores nulos pero las demás tienen valores informados, al utilizar **dropna()** eliminaremos estas filas y por lo tanto, también eliminaremos valores que pueden ser valiosos para nuestro análisis.

Una manera simple de evitar esto es “rellenar” estos valores faltantes o nulos. Debemos informar esos datos faltantes con información que sea relevante para el contexto en el que nos encontramos. Para ello contamos con los métodos **df.ffill()** y **df.bfill()**:

- **df.ffill()**: Si utilizamos *forward-fill* los valores que usaremos para rellenar, no nulos, se propagarán hacia adelante, por lo que si, por ejemplo, tenemos valores nulos en la fila 2 y no nulos en la fila 1, los valores de la fila 1 se copiarán a la fila 2.
- **df.bfill()**: Si utilizamos *backward-fill* los valores que usaremos para rellenar, no nulos, se propagarán hacia atrás, por lo que si, por ejemplo, tenemos valores nulos en la fila 1 y no nulos en la fila 2, los valores de la fila 2 se copiarán a la fila 1.

Con esto podemos evitar los valores faltantes.

Ahora bien, ¿cómo podemos detectar si durante un período de tiempo no se han recibido datos? Una manera de hacerlo puede ser usando el método **.isna()**, que devuelve True si el contenido es NaN.



De manera que podemos hacer:

```
df.isna().sum()
```

que retornará la suma total de valores faltantes y nos dará una pista de cuántos hay pero realmente la función que puede ayudarnos a detectar estas interrupciones es **resample()**. Esta función es similar a hacer un *groupby* pero agrupando intervalos temporales. Nos permite cambiar el intervalo de tiempo como por ejemplo, si tenemos el *dataframe* desglosado mes a mes, podemos agregar esos datos mensuales en datos anuales o, también podemos tomar datos tomados cada hora y transformarlos en datos minuto a minuto.

En nuestro caso, podemos realizar un resample y agrupar los datos cada cierto período de tiempo para poder detectar posibles interrupciones. Por ejemplo si recibimos datos cada 5 minutos, podemos hacer el resample:

```
df_resampled = df.resample("5min").asfreq
```

Ahora realizando

```
df_resampled.isna().sum()
```

podemos obtener el total de valores faltantes.

Tras realizar este estudio y ver si existen datos faltantes podemos imputarlos con diferentes agregaciones según el contexto y el tipo de medida que estamos registrando con el sensor. Por ejemplo si el sensor mide la temperatura y los segundos de sol que se han producido, podemos realizar un resample de los datos y definir:

```
df_5min = df.resample("5min").agg({"temperatura": "max", "suntime": "sum"})
```

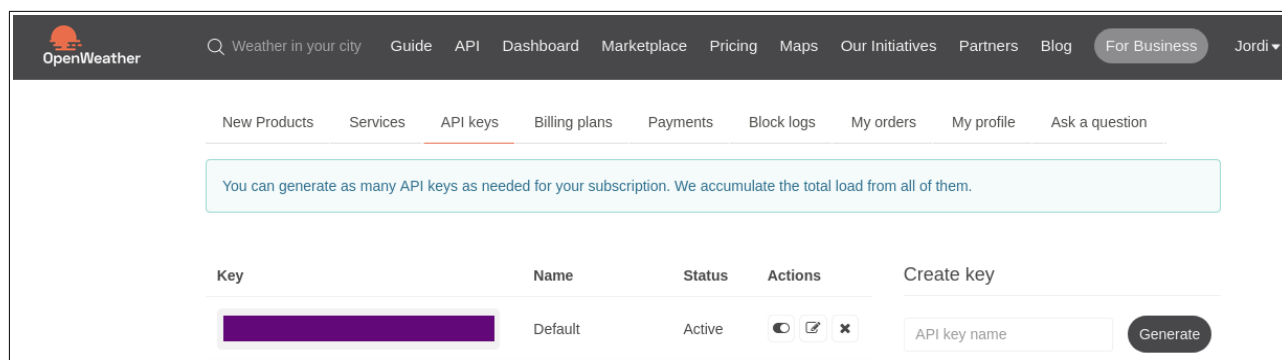
dado que se generan agrupaciones cada 5 minutos, tomamos en el caso de la temperatura el valor máximo en ese período (también podríamos haberlo imputado con la media) y para el caso de *suntime*, tomamos la suma.

De todos modos, esto tal vez ya está en el terreno de los analistas de datos.

### 3.6 Enriquecimiento de datos

Con los datos de los sensores guardados podemos extraer una serie de conclusiones tras analizarlos. No obstante es interesante enriquecer estos datos a través de fuentes de datos abiertas y APIs que nos provean de datos relevantes para cruzar esos datos con los obtenidos. Debemos pues investigar qué APIs hay disponibles y que portales de datos abiertos existen que puedan proporcionarnos datos de tipo meteorológico, de niveles de contaminación, índices de salud pública en la zona, incidencia de cáncer, índice de enfermedades respiratorias, etc.

Para obtener datos vía APIs podemos utilizar la librería **requests** de Python que nos permite conectarnos y obtener los datos que necesitamos. Como ejemplo nos conectaremos a la API pública **OpenWeatherMap** y obtendremos datos meteorológicos de unas poblaciones concretas. Para ello necesitaremos darnos de alta en <https://openweathermap.org/> e iremos al apartado de la gestión de las API Keys [https://home.openweathermap.org/api\\_keys](https://home.openweathermap.org/api_keys) y daremos de alta una de ellas.



Ahora mostraremos un script de Python que permite conectarte a la API y obtener datos para unas ciudades en concreto. Tenemos un fichero de configuración yaml que contiene la lista de las ciudades a considerar, las urls, *api keys*, un correo electrónico válido para poder acceder a la API de **Nominatim** y el *path* dónde guardar el json resultante en un archivo de tipo parquet.

Aquí tenemos el fichero de configuración:

```
openweathermap:
  - city: Girona
    country: Spain
  - city: Vilablareix
    country: Spain
  - city: Riudellots
    country: Spain
  - city: Caldes de Malavella
    country: Spain
  - city: Sils
    country: Spain
  - city: Barcelona
```

```

    country: Spain

user_agent: 'correo@test.org'

openweathermap:
    api_key: 'API_KEY' # API Key de OpenWeatherMap
    base_url: 'http://api.openweathermap.org/data/2.5/weather' # base_url
    url_icono: 'https://openweathermap.org/img/wn/'

meteodata:
    path: './data/'
    filename: 'meteo_data_{timestamp}.parquet'

```

Ahora mostramos el script de Python:

```

import requests
import pandas as pd
from geopy.geocoders import Nominatim
from geopy.extra.rate_limiter import RateLimiter
import datetime
import geopandas as gpd
import contextily as ctx
from skimage import io
import matplotlib.pyplot as plt
from matplotlib.offsetbox import AnnotationBbox, OffsetImage
import time
import yaml

# Archivo de configuración
CONFIG_YAML_FILE = './config.yaml'

def get_yaml(path):
    """
    Helper function to get yaml file contents.
    """
    with open(path) as yaml_file:
        data = yaml.safe_load(yaml_file)

    return data

def obtener_coordenadas(my_geocode, my_city, my_country):
    response = my_geocode(query={"city": my_city, "country": my_country})

    return {
        "latitude": response.latitude,
        "longitude": response.longitude
    }

def get_dataframe_geocodification(df):
    locator = Nominatim(user_agent=config['user_agent'])
    geocode = RateLimiter(locator.geocode, min_delay_seconds=1)

    df_coordenadas = df.apply(
        lambda x: obtener_coordenadas(geocode, x.city, x.country),
        axis=1)

    df = pd.concat([df, pd.json_normalize(df_coordenadas)], axis=1)

```

```

    return df

# Obtenemos los datos meteorologicos por ciudad
def obtener_datos_meteorologicos(row):
    my_url = f"{config['openweathermap']['base_url']}?
               lat={row.latitude}&lon={row.longitude}&
               appid={config['openweathermap']['api_key']}"

    my_response = requests.get(my_url)
    my_response_json = my_response.json()

    # my_response_json contiene una lista de diccionarios.
    # Comprobamos si el valor de la llave "cod" es igual a "404",
    # ya que eso significa que no hemos encontrado la ciudad.

    if my_response_json["cod"] != "404":
        my_response_json = my_response.json()
        sunset_utc = datetime.datetime.fromtimestamp(
            my_response_json["sys"]["sunset"])

        return {
            "temperatura": my_response_json["main"]["temp"] - 273.15,
            "presion": my_response_json["main"]["pressure"],
            "humedad": my_response_json["main"]["humidity"],
            "nubes": my_response_json["clouds"]["all"],
            "viento_speed": my_response_json["wind"]["speed"],
            "viento_deg": my_response_json["wind"]["deg"],
            "descripcion": my_response_json["weather"][0]["description"],
            "icono": my_response_json["weather"][0]["icon"],
            "sunset_utc": sunset_utc,
            "sunset_local": sunset_utc +
                datetime.timedelta(seconds=my_response_json["timezone"])
        }
    else:
        return {
            "error": "Ciudad no encontrada"
        }

def get_dataframe_meteo_by_city(df):
    df_meteo = df.apply(lambda x: obtener_datos_meteorologicos(x), axis=1)
    df = pd.concat([df, pd.json_normalize(df_meteo)], axis=1)

    return df

def save_dataframe_to_parquet(config, df):
    # filename = f'meteo_data_{time.strftime("%Y%m%d_%H%M%S")}'
    # filename_path = f'{config["meteodata"]["path"]}{filename}.parquet'

    filename_prefix = config['meteodata']['filename']
    filename_path = f'{config["meteodata"]["path"]}'
                    {filename_prefix.format(
                        timestamp=time.strftime('%Y%m%d_%H%M%S'))}"

    df.to_parquet(filename_path, engine='fastparquet')

# Dibujamos el mapa
def get_geodataframe(df):
    geo_df = gpd.GeoDataFrame(
        df,
        geometry=gpd.points_from_xy(df.longitude, df.latitude),

```

```

        crs=4326)
    return geo_df

def add_icon(ax, row):
    img = io.imread(
        f"{config['openweathermap']['url_icono']}{row.icono}@2x.png"
    )

    img_offset = OffsetImage(img, zoom=.4, alpha=1, )
    ab = AnnotationBbox(
        img_offset,
        [row.geometry.x+150000, row.geometry.y-110000],
        frameon=False)

    ax.add_artist(ab)

def draw_map(geo_df):
    # Dibujamos la localización de la ciudad
    ax = geo_df.to_crs(epsg=3857).plot(figsize=(12,8), color="black")

    # Añadimos el icono
    geo_df.to_crs(epsg=3857).apply(lambda row: add_icon(ax, row), axis=1)

    # Nombre de la ciudad
    geo_df.to_crs(epsg=3857).apply(
        lambda x: ax.annotate(
            text=f"{x.city} ",
            fontsize=10,
            color="black",
            xy=x.geometry.centroid.coords[0],
            ha='right'),

        axis=1)

    # Temperatura registrada
    geo_df.to_crs(epsg=3857).apply(
        lambda x: ax.annotate(
            text=f" {round(x.temperatura)}°",
            fontsize=12,
            color="black",
            xy=x.geometry.centroid.coords[0],
            ha='left'),

        axis=1)

    # Márgenes del mapa
    xmin, ymin, xmax, ymax = geo_df.to_crs(epsg=3857).total_bounds
    margin_y = .2
    margin_x = .2
    y_margin = (ymax - ymin) * margin_y
    x_margin = (xmax - xmin) * margin_x

    ax.set_xlim(xmin - x_margin, xmax + x_margin)
    ax.set_ylim(ymin - y_margin, ymax + y_margin)

    # Añadimos el mapa base
    ctx.add_basemap(ax, source=ctx.providers.CartoDB.Positron)

    ax.set_axis_off()

    plt.show()

```

```

if __name__ == "__main__":
    # Obtenemos nuestro archivo de configuración

    try:
        config = get_yaml(CONFIG_YAML_FILE)
    except Exception as e:
        print(f'No existe el archivo de configuración {CONFIG_YAML_FILE}')
        exit(-1)

    # Cargamos el dataframe con las ciudades y países
    df = pd.DataFrame(config['cities'], columns=["city", "country"])

    # Necesitaremos realizar una geocodificación para cada ciudad, ya que el
    # endpoint de la API de OpenWeatherMap necesita coordenadas geográficas
    # de latitud y longitud. Para ello, utilizaremos el geocodificador
    # Nominatim proporcionado por OpenStreetMap.
    df = get_dataframe_geocodification(df)

    # Obtenemos los datos meteorológicos por ciudad
    df = get_dataframe_meteo_by_city(df)

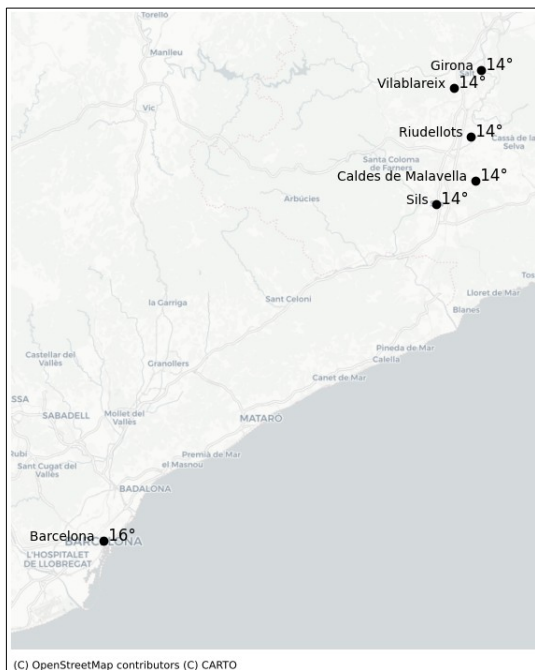
    # Guardamos el dataframe resultante en un archivo de tipo parquet
    save_dataframe_to_parquet(config, df)

    # Utilizaremos las librerías Geopandas, contextily y Matplotlib para
    # mostrar en un mapa los resultados. Para ello convertiremos el DataFrame
    # de pandas en un GeoDataFrame y crearemos una columna con coordenadas.
    # Se establecerá el CRS en 4326, que define el sistema de coordenadas como
    # WGS84 – World Geodetic System 1984, que utiliza latitud y longitud en
    # grados (unidad).
    geo_df = get_geodataframe(df)

    # Dibujamos el mapa
    draw_map(geo_df)

```

Obtenemos un mapa similar al siguiente:



Y también obtenemos el siguiente archivo parquet:

city	country	latitude	longitude	temperatura	presion	humedad	nubes	viento_speed	viento_deg	descripcion	icono	sunset_utc	sunset_local
Girona	Spain	41.9793006	2.8199439	18.220000000000027	996	47	20	8.23	200	few clouds	02d	2024-03-30T19:11:37.000Z	2024-03-30T20:11:37.000Z
Vilablareix	Spain	41.9569396	2.7728439	18	996	47	20	8.23	200	few clouds	02d	2024-03-30T19:11:48.000Z	2024-03-30T20:11:48.000Z
Riudellots	Spain	41.8954981	2.8022296	17.629999999999995	996	47	20	8.23	200	few clouds	02d	2024-03-30T19:11:38.000Z	2024-03-30T20:11:38.000Z
Caldes de Malavella	Spain	41.8398724	2.8099858	17.379999999999995	996	47	20	8.23	200	few clouds	02d	2024-03-30T19:11:35.000Z	2024-03-30T20:11:35.000Z
Sils	Spain	41.8098347	2.743055	17.640000000000043	996	47	20	8.23	200	few clouds	02d	2024-03-30T19:11:50.000Z	2024-03-30T20:11:50.000Z
Barcelona	Spain	41.3828939	2.1774322	16.640000000000043	999	59	20	6.69	220	few clouds	02d	2024-03-30T19:13:51.000Z	2024-03-30T20:13:51.000Z

Los portales de datos abiertos, observatorios, suelen tener disponibles *datasets* en diversos formatos tales como csv, json, xml o parquet que podemos utilizar para nuestros análisis. En nuestro caso hemos hallado:

- <https://datos.gob.es>: Es el portal nacional de datos abiertos de España, administrado por la Secretaría de Estado de Digitalización e Inteligencia Artificial. Su objetivo es proporcionar acceso gratuito a conjuntos de datos públicos de diferentes organismos y entidades del gobierno español. Los datos disponibles abarcan una amplia gama de áreas, como transporte, medio ambiente, economía, salud, educación, entre otros. El propósito principal de este portal es fomentar la transparencia, la participación ciudadana y la innovación mediante el uso y reutilización de datos abiertos.

En particular hemos hallado el siguiente conjunto abierto de datos:

[https://datos.gob.es/es/catalogo/ea0010587-cataluna-defunciones-por-provincia-de-residencia-causas-lista-reducida-sexo-y-edad-ecm-identificador-api-tpx-sociedad\\_2589-salud\\_2590-edcm\\_2591-a2022\\_9034-l0-02010-px](https://datos.gob.es/es/catalogo/ea0010587-cataluna-defunciones-por-provincia-de-residencia-causas-lista-reducida-sexo-y-edad-ecm-identificador-api-tpx-sociedad_2589-salud_2590-edcm_2591-a2022_9034-l0-02010-px)

Tabla de INEbase Cataluña. Defunciones por provincia de residencia, causas (lista reducida), sexo y edad. Nacional. Estadística de Defunciones según la Causa de Muerte.

Nos puede resultar útil para discernir las defunciones por causas respiratorias y por diversos tipos de cáncer.

Mediante un sencillo script podemos ver que datos alberga:

```
import pandas as pd

pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
pd.set_option('display.max_colwidth', None)

df = pd.read_json('./data/catalunya_defunciones_por_provincia.json')
df.head()
df.info()
```

obtenemos:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30600 entries, 0 to 30599
Data columns (total 3 columns):
#   Column      Non-Null Count  Dtype
---  -
0   Nombre      30600 non-null  object
1   MetaData    30600 non-null  object
2   Data        30600 non-null  object
dtypes: object(3)
memory usage: 717.3+ KB
```

	Nombre	MetaData	Data
0	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, Todas las edades	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'Todas las edades', 'Codigo: 'todaslasedades')]", [{"Valor": 33415.0}]]	
1	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, Menores de 1 año	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'Menores de 1 año', 'Codigo: 'menoresde1ano')]", [{"Valor": 74.0}]]	
2	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, De 1 a 14 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 1 a 14 años', 'Codigo: 'de1a14anos')]", [{"Valor": 51.0}]]	
3	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, De 15 a 29 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 15 a 29 años', 'Codigo: 'de15a29anos')]", [{"Valor": 155.0}]]	
4	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, De 30 a 39 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 30 a 39 años', 'Codigo: 'de30a39anos')]", [{"Valor": 217.0}]]	
5	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, De 40 a 44 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 40 a 44 años', 'Codigo: 'de40a44anos')]", [{"Valor": 241.0}]]	
6	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, De 45 a 49 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 45 a 49 años', 'Codigo: 'de45a49anos')]", [{"Valor": 392.0}]]	
7	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, De 50 a 54 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 50 a 54 años', 'Codigo: 'de50a54anos')]", [{"Valor": 633.0}]]	
8	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, De 55 a 59 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 55 a 59 años', 'Codigo: 'de55a59anos')]", [{"Valor": 1077.0}]]	
9	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, De 60 a 64 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 60 a 64 años', 'Codigo: 'de60a64anos')]", [{"Valor": 1488.0}]]	
10	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, De 65 a 69 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 65 a 69 años', 'Codigo: 'de65a69anos')]", [{"Valor": 1802.0}]]	
11	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, De 70 a 74 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 70 a 74 años', 'Codigo: 'de70a74anos')]", [{"Valor": 2618.0}]]	
12	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, De 75 a 79 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 75 a 79 años', 'Codigo: 'de75a79anos')]", [{"Valor": 3669.0}]]	
13	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, De 80 a 84 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 80 a 84 años', 'Codigo: 'de80a84anos')]", [{"Valor": 4283.0}]]	
14	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, De 85 a 89 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 85 a 89 años', 'Codigo: 'de85a89anos')]", [{"Valor": 5677.0}]]	
15	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, De 90 a 94 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 90 a 94 años', 'Codigo: 'de90a94anos')]", [{"Valor": 6478.0}]]	
16	Cataluña, 001-102 I-XXII.Todas las causas, Ambos sexos, De 95 años y más	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Ambos sexos', 'Codigo: 'ambossexos', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 95 años y más', 'Codigo: 'de95anonymas')]", [{"Valor": 3560.0}]]	
17	Cataluña, 001-102 I-XXII.Todas las causas, Hombres, Todas las edades	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Hombres', 'Codigo: 'hombres', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'Todas las edades', 'Codigo: 'todaslasedades')]", [{"Valor": 16714.0}]]	
18	Cataluña, 001-102 I-XXII.Todas las causas, Hombres, Menores de 1 año	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Hombres', 'Codigo: 'hombres', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'Menores de 1 año', 'Codigo: 'menoresde1ano')]", [{"Valor": 43.0}]]	
19	Cataluña, 001-102 I-XXII.Todas las causas, Hombres, De 1 a 14 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Hombres', 'Codigo: 'hombres', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 1 a 14 años', 'Codigo: 'de1a14anos')]", [{"Valor": 29.0}]]	
20	Cataluña, 001-102 I-XXII.Todas las causas, Hombres, De 15 a 29 años	[{"Variable": "(Nombre: 'provincia de residencia', 'Codigo: 'provinciaderesidencia', 'Nombre: 'Cataluña', 'Codigo: 'cataluña', 'Variable: '(Nombre: 'causas (lista reducida)', 'Codigo: 'causalistareducida', 'Nombre: '001-102 I-XXII.Todas las causas', 'Codigo: '001102ixiitodaslascausas', 'Variable: '(Nombre: 'sexo', 'Codigo: 'sexo', 'Nombre: 'Hombres', 'Codigo: 'hombres', 'Variable: '(Nombre: 'edad', 'Codigo: 'edad', 'Nombre: 'De 15 a 29 años', 'Codigo: 'de15a29anos')]", [{"Valor": 114.0}]]	

- <https://canalsalut.gencat.cat>: Se trata del portal de datos abiertos de salud de la Generalitat de Catalunya, la administración autonómica de Cataluña. Su objetivo principal es ofrecer acceso a datos relacionados con la salud y el sistema sanitario de Cataluña, incluyendo estadísticas, indicadores epidemiológicos, informes y otros recursos relacionados con la salud pública. Este portal tiene como finalidad proporcionar información transparente y accesible sobre la situación sanitaria de la región, así como promover la investigación y el desarrollo de soluciones innovadoras en el ámbito de la salud.

En este caso hemos hallado un csv con datos medioambientales muy interesante que está ubicado en



[https://analisi.transparenciacatalunya.cat/Medi-Ambient/Qualitat-de-l-aire-als-punts-de-mesurament-autom-t/tasf-thgu/about\\_data](https://analisi.transparenciacatalunya.cat/Medi-Ambient/Qualitat-de-l-aire-als-punts-de-mesurament-autom-t/tasf-thgu/about_data)

Contiene valores de alto interés tales como los contaminantes medidos diariamente. A parte de la concentración de los diferentes contaminantes también contiene las coordenadas donde está ubicado el punto de medida, el nombre y tipo de estación, las unidades y la fecha.

Aquí mostramos unas capturas del *dataframe*:

```
df.describe()
```

	codi_eoi	magnitud	codi_line	codi_comarca	h01	h02	h03	h04	h05	h06	h07	h08	h09	h10	h11	h12	h13	h14	h15
count	210.0	210.000000	210.0	210.0	210.000000	210.000000	210.000000	203.000000	203.000000	203.000000	203.000000	203.000000	203.000000	203.000000	203.000000	196.000000	196.000000	203.000000	203.000000
mean	17079003.0	7.571429	17079.0	20.0	11.839048	10.093333	8.933333	7.484729	7.084729	8.238916	11.014286	16.208867	20.410345	19.266995	12.849754	8.980102	7.672449	7.279592	6.833005
std	0.0	3.252845	0.0	0.0	12.938966	11.148023	10.891761	7.234462	6.717692	7.967733	12.011931	20.476618	27.817735	25.800085	15.169434	9.366468	7.597365	6.860402	6.559797
min	17079003.0	1.000000	17079.0	20.0	0.200000	0.200000	0.200000	0.200000	0.200000	0.200000	0.200000	0.200000	0.200000	0.200000	0.200000	0.200000	0.200000	0.200000	0.200000
25%	17079003.0	6.000000	17079.0	20.0	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	2.000000	2.000000	1.000000	1.000000	1.000000	1.000000	1.000000
50%	17079003.0	8.000000	17079.0	20.0	9.000000	7.000000	6.000000	5.000000	5.000000	6.000000	7.000000	9.000000	12.000000	10.000000	8.000000	6.000000	5.000000	5.000000	4.000000
75%	17079003.0	10.000000	17079.0	20.0	18.000000	16.000000	13.000000	12.000000	12.000000	13.000000	16.500000	23.500000	25.000000	24.000000	19.000000	14.000000	12.000000	12.000000	11.000000
max	17079003.0	12.000000	17079.0	20.0	107.000000	101.000000	108.000000	32.000000	34.000000	34.000000	62.000000	118.000000	176.000000	147.000000	91.000000	48.000000	46.000000	34.000000	32.000000

```
df.info(verbose=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 210 entries, 0 to 209
Data columns (total 40 columns):
#   Column              Non-Null Count  Dtype
---  ---
0   codi_eoi             210 non-null   int64
1   nom_estacio         210 non-null   object
2   data                210 non-null   object
3   magnitud            210 non-null   int64
4   contaminant         210 non-null   object
5   unitats             210 non-null   object
6   tipus_estacio       210 non-null   object
7   area_urbana         210 non-null   object
8   codi_line           210 non-null   int64
9   municipi            210 non-null   object
10  codi_comarca         210 non-null   int64
11  nom_comarca         210 non-null   object
12  h01                 210 non-null   float64
13  h02                 210 non-null   float64
14  h03                 210 non-null   float64
15  h04                 203 non-null   float64
16  h05                 203 non-null   float64
17  h06                 203 non-null   float64
18  h07                 203 non-null   float64
19  h08                 203 non-null   float64
...
38  longitud            210 non-null   float64
39  geocoded_column     0 non-null     float64
dtypes: float64(27), int64(5), object(8)
memory usage: 65.8+ KB
```

Podemos combinar estos *datasets* a través de **Pandas** para poder cruzar y analizar los datos.

### 3.7 Creación de cuadros de comando

Finalmente tras la recolección, limpieza, análisis y enriquecimiento de los datos podemos generar cuadros de comando que nos permitan visualizarlos, interpretarlos y tomar decisiones informadas en base a la información que nos proveen.

En nuestro caso, la idea sería generarlos utilizando **Power BI**.

## 4. Más allá de Orion.

Como propósito de mejora, tendríamos:

- i. **Migración del servidor MQTT a la nube:** Este paso permitiría mejorar la escalabilidad y el dimensionamiento del sistema. Al trasladar el servidor MQTT a la nube, se obtendrían ventajas significativas en términos de flexibilidad y capacidad de gestión. Específicamente, en el caso de utilizar sensores con tecnología LoRaWAN (Long Range Area Wide Area Network), se podría aprovechar el servicio en la nube ofrecido por *The Things Industries*. La plataforma *The Things Industries* proporciona una infraestructura robusta y altamente escalable para la gestión y conectividad de dispositivos IoT basados en la tecnología LoRaWAN. Con ello, se lograría una mayor eficiencia en la gestión de los datos generados por los sensores, así como una optimización en la administración de los recursos computacionales necesarios para el procesamiento y almacenamiento de la información recopilada.
- ii. **Snowflake:** Utilizar **Snowflake** como *data warehouse* junto con **dbt** (*Data Build Tool*) para el procesamiento y limpieza de los datos.
- iii. **Implementación de técnicas de machine learning:** Otra área de mejora consiste en la aplicación de técnicas de machine learning con el fin de predecir posibles enfermedades respiratorias y cánceres asociados a las condiciones medioambientales registradas. Mediante el análisis avanzado de los datos recopilados, se podrían desarrollar modelos predictivos capaces de identificar patrones y correlaciones entre la calidad del aire, la exposición a contaminantes y la incidencia de enfermedades respiratorias y cánceres específicos. Esto permitiría no solo una mejor comprensión de los riesgos para la salud asociados al medio ambiente, sino también la implementación de estrategias preventivas y políticas de salud pública más efectivas y personalizadas.
- iv. **Desarrollo de aplicaciones móviles para la salud:** Para involucrar a la comunidad y promover la conciencia sobre los riesgos ambientales para la salud, se podría desarrollar una aplicación móvil que muestre información en tiempo real sobre la calidad del aire, los niveles de contaminación, así como incluir alertas de niveles de calidad del aire, consejos de salud personalizados según la ubicación y la exposición a contaminantes, así como la capacidad de informar sobre eventos ambientales y de salud pública. La participación de los ciudadanos en la vigilancia ambiental podría contribuir a la generación de datos en tiempo real y a la identificación de áreas problemáticas que requieran atención inmediata.
- v. **Polars:** Utilizar la librería **Polars** que está diseñada para el procesamiento de grandes conjuntos de datos en memoria, obteniendo así un acceso rápido y eficiente. Está inspirada en **Pandas**, no obstante, **Polars** va más allá y proporciona un rendimiento superior, así como capacidades de procesamiento distribuido que le permite dividir el trabajo entre varios nodos para acelerar el procesamiento. **Pandas** carece de esta característica. A diferencia de **Pandas**, que trabaja mejor con datos homogéneos, **Polars** ofrece un sólido soporte para tipos de datos heterogéneos, lo que significa que puede manejar conjuntos de datos con diferentes tipos de datos en las columnas.