Para abordar este reto, propongo la siguiente solución, estructurada en pasos con implementaciones en SQL Server, Python y PySpark. También se incluye la integración con Apache Airflow para la orquestación diaria del pipeline ETL.

**1. Modelo Relacional**

El modelo relacional tiene las siguientes tablas:

* **Policy**: Información básica de las pólizas.
* **Insured**: Información del asegurado.
* **Premium**: Información de primas y límites.
* **Payments**: Información de pagos.
* **Claims**: Información de reclamos.
* **Agents**: Información de los canales de ventas.

**Diseño de las tablas en SQL:**

sql

Copiar código

CREATE TABLE Policy (

policy\_id INT PRIMARY KEY,

start\_date DATE NOT NULL,

end\_date DATE NOT NULL,

agent\_id INT,

insured\_id INT,

FOREIGN KEY (agent\_id) REFERENCES Agents(agent\_id),

FOREIGN KEY (insured\_id) REFERENCES Insured(insured\_id)

);

CREATE TABLE Insured (

insured\_id INT PRIMARY KEY,

name VARCHAR(255) NOT NULL,

email VARCHAR(255),

phone\_number VARCHAR(20)

);

CREATE TABLE Premium (

premium\_id INT PRIMARY KEY,

policy\_id INT,

premium\_amount DECIMAL(10, 2) NOT NULL,

deductible\_amount DECIMAL(10, 2),

coverage\_limit DECIMAL(10, 2),

FOREIGN KEY (policy\_id) REFERENCES Policy(policy\_id)

);

CREATE TABLE Payments (

payment\_id INT PRIMARY KEY,

policy\_id INT,

payment\_status VARCHAR(50),

payment\_date DATE,

payment\_amount DECIMAL(10, 2),

payment\_method VARCHAR(50),

FOREIGN KEY (policy\_id) REFERENCES Policy(policy\_id)

);

CREATE TABLE Claims (

claim\_id INT PRIMARY KEY,

policy\_id INT,

claim\_date DATE,

claim\_amount DECIMAL(10, 2),

claim\_status VARCHAR(50),

FOREIGN KEY (policy\_id) REFERENCES Policy(policy\_id)

);

CREATE TABLE Agents (

agent\_id INT PRIMARY KEY,

agent\_name VARCHAR(255),

channel\_type VARCHAR(50)

);

**2. Pipeline ETL con PySpark**

**a) Limpieza y Transformación de Datos**

Supongamos que tenemos un archivo insurance\_data.csv que contiene los datos. El objetivo es cargar este archivo, limpiar los datos y transformarlos.

python

Copiar código

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, to\_date, when

# Crear la sesión de Spark

spark = SparkSession.builder.appName("InsuranceETL").getOrCreate()

# Cargar el archivo CSV

data = spark.read.csv("insurance\_data.csv", header=True, inferSchema=True)

# Limpieza de datos: Manejo de valores nulos

data = data.fillna({

"premium\_amount": 0,

"deductible\_amount": 0,

"coverage\_limit": 0,

"payment\_status": "Unknown",

"claim\_status": "Pending"

})

# Conversión de fechas al formato correcto

data = data.withColumn("start\_date", to\_date(col("start\_date"), "dd-MM-yyyy"))

data = data.withColumn("end\_date", to\_date(col("end\_date"), "dd-MM-yyyy"))

# Filtrar datos inválidos

data = data.filter(col("end\_date").isNotNull())

# Mostrar los datos procesados

data.show()

**b) Carga en la Base de Datos**

Usando JDBC para cargar los datos limpios en SQL Server:

python

Copiar código

# Configuración de conexión JDBC

db\_config = {

"url": "jdbc:sqlserver://your\_server:1433;databaseName=InsuranceDB",

"user": "your\_username",

"password": "your\_password",

"driver": "com.microsoft.sqlserver.jdbc.SQLServerDriver"

}

# Cargar la tabla Policy

data.select("policy\_id", "start\_date", "end\_date", "agent\_id", "insured\_id") \

.write.format("jdbc") \

.option("url", db\_config["url"]) \

.option("dbtable", "Policy") \

.option("user", db\_config["user"]) \

.option("password", db\_config["password"]) \

.option("driver", db\_config["driver"]) \

.mode("append") \

.save()

**3. Reporte SQL**

El reporte de pólizas con fecha de finalización superior al 01-07-2023:

sql

Copiar código

SELECT

p.policy\_id,

p.start\_date,

p.end\_date,

i.name AS insured\_name,

a.agent\_name

FROM

Policy p

JOIN Insured i ON p.insured\_id = i.insured\_id

JOIN Agents a ON p.agent\_id = a.agent\_id

WHERE

p.end\_date > '2023-07-01';

**4. Orquestación con Apache Airflow**

**DAG en Airflow:**

python

Copiar código

from airflow import DAG

from airflow.operators.python import PythonOperator

from datetime import datetime, timedelta

# Función ETL

def run\_etl():

# Lógica ETL usando PySpark

import subprocess

subprocess.run(["spark-submit", "etl\_pipeline.py"])

default\_args = {

"owner": "airflow",

"retries": 1,

"retry\_delay": timedelta(minutes=5)

}

dag = DAG(

"insurance\_etl\_pipeline",

default\_args=default\_args,

description="Pipeline ETL para Monokera",

schedule\_interval="0 0 \* \* \*", # Ejecución diaria

start\_date=datetime(2024, 1, 1),

catchup=False

)

etl\_task = PythonOperator(

task\_id="run\_etl",

python\_callable=run\_etl,

dag=dag

)

etl\_task

**5. Entrega en GitHub**

1. **Repositorio**: Incluye los scripts SQL, PySpark, el archivo insurance\_data.csv simulado, el DAG de Airflow y documentación.
2. **Documentación**: Explicación de las decisiones, supuestos, y pasos para ejecutar el proyecto.

¿Te gustaría algún ajuste o profundización en algún punto?

4o

**Dijiste:**

Documentacion de los scripts, base de datos y DAGs

**ChatGPT dijo:**

ChatGPT

Aquí tienes una propuesta para la documentación completa del proyecto, incluyendo los scripts, la base de datos, y los DAGs en Airflow:

**Documentación del Proyecto**

**Introducción**

Este proyecto desarrolla un pipeline ETL para procesar datos de seguros y almacenarlos en una base de datos relacional, generando un reporte con políticas activas. La solución también incluye la orquestación del pipeline con Apache Airflow para actualizaciones diarias.

**Arquitectura del Proyecto**

1. **Fuente de Datos**: Archivo CSV (insurance\_data.csv) que contiene información sobre pólizas, asegurados, pagos, primas, reclamos y agentes.
2. **Pipeline ETL**:
   * Implementado en PySpark para limpiar, transformar y cargar los datos en SQL Server.
   * Gestiona valores nulos, valida formatos de datos y realiza transformaciones.
3. **Base de Datos**:
   * Modelo relacional que almacena los datos procesados.
   * Implementado en SQL Server, puede adaptarse a PostgreSQL.
4. **Reporte**:
   * Consulta SQL para listar pólizas con fecha de finalización superior a 01-07-2023.
5. **Orquestación**:
   * Implementación de un DAG en Apache Airflow para la ejecución diaria del pipeline.

**1. Scripts ETL**

**ETL en PySpark**

Archivo: etl\_pipeline.py

**Descripción de los pasos**

1. **Carga del archivo CSV**: Se utiliza PySpark para leer el archivo con soporte para esquemas inferidos y encabezados.
2. **Limpieza de datos**:
   * Relleno de valores nulos en columnas críticas.
   * Conversión de fechas al formato yyyy-MM-dd.
   * Filtrado de filas con datos inválidos.
3. **Transformaciones**:
   * Normalización de nombres de columnas.
   * Conversión de valores categóricos y numéricos según el modelo relacional.
4. **Carga en la base de datos**:
   * Conexión JDBC para insertar los datos procesados en las tablas definidas.

**Fragmento del código**

python

Copiar código

# Limpieza de valores nulos

data = data.fillna({

"premium\_amount": 0,

"deductible\_amount": 0,

"coverage\_limit": 0,

"payment\_status": "Unknown",

"claim\_status": "Pending"

})

# Carga de datos a la tabla Policy

data.select("policy\_id", "start\_date", "end\_date", "agent\_id", "insured\_id") \

.write.format("jdbc") \

.option("url", db\_config["url"]) \

.option("dbtable", "Policy") \

.option("user", db\_config["user"]) \

.option("password", db\_config["password"]) \

.option("driver", db\_config["driver"]) \

.mode("append") \

.save()

**2. Base de Datos**

**Modelo Relacional**

El modelo se diseñó para garantizar integridad referencial y normalización. Las relaciones son las siguientes:

* **Policy** relaciona con Insured (asegurado) y Agents (agentes de venta).
* **Premium** y **Payments** están vinculados con cada póliza.
* **Claims** se asocia a una póliza para registrar reclamos.

**Scripts de Creación de Tablas**

Archivo: create\_tables.sql

**Fragmento del Código**

sql

Copiar código

CREATE TABLE Policy (

policy\_id INT PRIMARY KEY,

start\_date DATE NOT NULL,

end\_date DATE NOT NULL,

agent\_id INT,

insured\_id INT,

FOREIGN KEY (agent\_id) REFERENCES Agents(agent\_id),

FOREIGN KEY (insured\_id) REFERENCES Insured(insured\_id)

);

CREATE TABLE Insured (

insured\_id INT PRIMARY KEY,

name VARCHAR(255) NOT NULL,

email VARCHAR(255),

phone\_number VARCHAR(20)

);

**3. Reporte**

**Descripción**

El reporte muestra las pólizas cuya fecha de finalización es posterior al 01-07-2023.

**Consulta SQL**

Archivo: report.sql

sql

Copiar código

SELECT

p.policy\_id,

p.start\_date,

p.end\_date,

i.name AS insured\_name,

a.agent\_name

FROM

Policy p

JOIN Insured i ON p.insured\_id = i.insured\_id

JOIN Agents a ON p.agent\_id = a.agent\_id

WHERE

p.end\_date > '2023-07-01';

**4. DAG en Apache Airflow**

**Descripción**

El DAG orquesta la ejecución diaria del script ETL (etl\_pipeline.py), garantizando que los datos estén actualizados.

**Archivo DAG**

Archivo: insurance\_etl\_dag.py

**Estructura**

1. **Configuración del DAG**:
   * Ejecución diaria a la medianoche (0 0 \* \* \*).
   * Manejo de errores con reintentos.
2. **Operador de Python**:
   * Llama al script ETL con PySpark.

**Fragmento del Código**

python

Copiar código

from airflow import DAG

from airflow.operators.python import PythonOperator

from datetime import datetime, timedelta

# Función ETL

def run\_etl():

import subprocess

subprocess.run(["spark-submit", "etl\_pipeline.py"])

default\_args = {

"owner": "airflow",

"retries": 1,

"retry\_delay": timedelta(minutes=5)

}

dag = DAG(

"insurance\_etl\_pipeline",

default\_args=default\_args,

description="Pipeline ETL para Monokera",

schedule\_interval="0 0 \* \* \*", # Ejecución diaria

start\_date=datetime(2024, 1, 1),

catchup=False

)

etl\_task = PythonOperator(

task\_id="run\_etl",

python\_callable=run\_etl,

dag=dag

)

etl\_task

**5. Pasos para la Ejecución**

1. **Preparación del Entorno**:
   * Instalar dependencias: PySpark, SQL Server JDBC Driver, Airflow.
   * Configurar las variables de entorno para Airflow y PySpark.
2. **Base de Datos**:
   * Crear las tablas con create\_tables.sql.
3. **Pipeline ETL**:
   * Ejecutar el script etl\_pipeline.py.
4. **Orquestación**:
   * Subir el archivo insurance\_etl\_dag.py al directorio dags de Airflow.
   * Iniciar Airflow y verificar la programación del DAG.
5. **Reporte**:
   * Ejecutar report.sql en SQL Server Management Studio o cualquier cliente compatible.

**6. Recursos Adicionales**

* **Documentación Oficial**:
  + [PySpark API Reference](https://spark.apache.org/docs/latest/api/python/)
  + [Airflow Documentation](https://airflow.apache.org/docs/)
* **Bibliotecas Utilizadas**:
  + PySpark
  + Airflow
  + JDBC para SQL Server

Si necesitas adaptar algún paso o mayor detalle en la documentación, ¡avísame!

4o

Principio del formulario



Final del formulario