Prueba técnica Ingeniero de datos - Puntored

**Duración: 24 horas a partir de la fecha de envío**

**Criterios de Evaluación**

* Precisión de las respuestas teóricas
* Claridad y eficiencia del código SQL
* Estructura y documentación del código Python

**Modo de entrega**

Cargue los archivos pertinentes al desarrollo de la prueba a un repositorio público de GitHub y comparta la URL. (15 puntos)

<https://github.com/DubanF/CI-CD-para-Datos->

# Sección 1: Preguntas Teóricas (12 puntos)

## Python (4 puntos)

1. **¿Cuál es la diferencia entre una lista y un conjunto (set) en Python? Proporcione un ejemplo (1 puntos)**

*una* ***lista*** *en Python es como una coleccion de cosas, pero ordenada, osea que guarda el orden en que metes los elementos y tambien puedes tener datos repetidos sin problema. En cambio un* ***set*** *(o conjunto) es distinto, porque* ***no guarda el orden*** *y no deja que tengas valores repetidos, osea si metes dos veces el mismo numero, él lo guarda solo una vez.*

*Tambien la lista la puedes modificar, igual que el set, pero la diferencia principal está en eso del orden y los duplicados.*

mi\_lista = [1, 2, 2, 3, 4]

print("Lista:", mi\_lista)  # muestra: [1, 2, 2, 3, 4]

mi\_conjunto = {1, 2, 2, 3, 4}

print("Set:", mi\_conjunto)  # muestra: {1, 2, 3, 4}

1. **¿Qué es un generator en Python y en qué casos es útil? Proporcione un ejemplo de implementación. (1 puntos)**

*Un* ***generator*** *en Python es una forma de crear iteradores de manera eficiente, utilizando la palabra clave yield en lugar de return. A diferencia de una función tradicional que devuelve todos los datos de una vez, un generador* ***produce los valores uno por uno*** *bajo demanda, lo que ayuda a* ***ahorrar memoria y mejorar el rendimiento****, especialmente cuando trabajamos con grandes volúmenes de datos o secuencias muy largas.*

*Esto es útil, por ejemplo, al procesar archivos grandes, leer datos en streaming o generar combinaciones sin tener que almacenarlas todas en memoria.*

*Ejemplol*

def contar\_hasta(n):

    i = 1

    while i <= n:

        yield i

        i += 1

for numero in contar\_hasta(5):

        print(numero)

*la salida será:*

*1*

*2*

*3*

*4*

*5*

*En este caso, la función contar\_hasta no devuelve una lista completa, sino que* ***va generando cada número cuando se itera sobre ella****. Eso permite trabajar con conjuntos de datos grandes de manera mucho más eficiente.*

1. **¿Qué ventajas ofrece Pandas sobre las estructuras de datos nativas de Python para el análisis de datos? (1 puntos)**

*Pandas ofrece varias vantajas frente a estructuras nativas como listas o diccionarios, sobre todo cuando se trata de manejar y analizar grandes volúmenes de datos. Una de las más importantes es que permite trabajar con datos* ***tabulares*** *(como los de Excel o bases de datos) de forma más* ***eficiente y legible****.*

*Además, tiene muchas funciones ya integradas para tareas como filtrar, agrupar, ordenar, combinar datasets o manejar valores nulos, que con estructuras nativas tomarían mucho más código y esfuerzo. Tambien, al estar basado sobre NumPy,* ***aprovecha operaciones vectorizadas****, lo que mejora el rendimiento.*

*Otra ventaja clave es que Pandas muestra los datos de forma más clara al imprimirlos, lo que hace más fácil la exploracion y el debugeo durante el analisis.*

*En resumen, Pandas* ***simplifica el trabajo con datos complejos****, reduce el código necesario y mejora tanto la productividad como el rendimiento cuando se trabaja con información estructurada.*

1. **¿Cuál es la diferencia entre apply() y map() en Pandas? Proporcione un ejemplo. (1 puntos)**

*Bueno, ambas funciones (apply() y map()) se usan para aplicar funciones a los datos en Pandas, pero no hacen exactamente lo mismo. Tienen como propósitos parecidos, pero no iguales.*

*map() se usa principalmente cuando uno quiere transformar una sola columna (una Serie), y funciona bien para cosas sencillas como cambiar texto, aplicar una funcion rapida a cada valor, o hacer reemplazos con un diccionario. Pero solo sirve con Series, osea no se puede usar sobre todo el DataFrame.*

*En cambio, apply() es más potente, porque sí se puede usar sobre una Serie, pero también sobre el DataFrame completo. Entonces, con apply() uno puede recorrer filas completas (axis=1) y hacer cosas más complejas, digamos cuando necesito combinar info de varias columnas o aplicar una logica personalizada.*

import pandas as pd

df = pd.DataFrame({

    'nombre': ['Ana', 'Luis', 'Pedro'],

    'edad': [23, 30, 27]

})

# map() sobre una sola columna

df['nombre\_mayus'] = df['nombre'].map(str.upper

# apply() con función sobre todo el DataFrame

df['es\_mayor'] = df['edad'].apply(lambda x: x > 25)

print(df)

**Salida:**

nombre edad nombre\_mayus es\_mayor

0 Ana 23 ANA False

1 Luis 30 LUIS True

2 Pedro 27 PEDRO True

## SQL (4 puntos)

1. **Dado un esquema de base de datos con las siguientes tablas empleados (id, nombre, departamento\_id, salario, fecha\_contratación) departamentos (id, nombre)**

**Escriba una consulta para obtener el salario promedio de cada departamento, incluyendo el nombre del departamento. (1 puntos)**

SELECT d.nombre AS nombre\_departamento,

AVG(e.salario) AS salario\_promedio

FROM empleados e

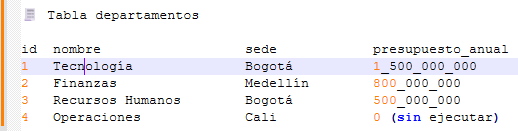
JOIN departamentos d ON e.departamento\_id = d.id

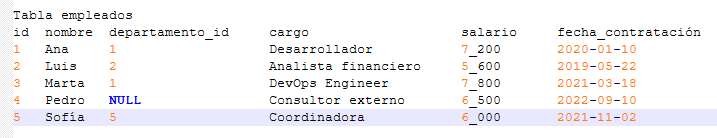
GROUP BY d.nombre;

1. **¿Cuál es la diferencia entre INNER JOIN, LEFT JOIN y FULL JOIN?. Da un ejemplo de cada uno. (1 puntos)**

*Los JOIN en SQL sirven para unir datos entre dos tablas, normalmente usando una relacion entre columnas*

*Supongamos que tenemos estas tablas:*

**

**

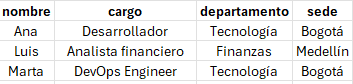
***INNER JOIN***

*Muestra* ***solo los registros que coinciden en ambas tablas****. Si hay algo en una tabla pero no tiene par en la otra, no lo muestra.*

SELECT e.nombre, d.nombre AS departamento

FROM empleados e

INNER JOIN departamentos d ON e.departamento\_id = d.id;

**

***LEFT JOIN***

*Este devulve* ***todos los registros de la tabla de la izquierda****, y lo que coincida de la derecha. Si no encuentra nada en la derecha, pues pone NULL.*

SELECT e.nombre, d.nombre AS departamento

FROM empleados e

LEFT JOIN departamentos d ON e.departamento\_id = d.id;

**

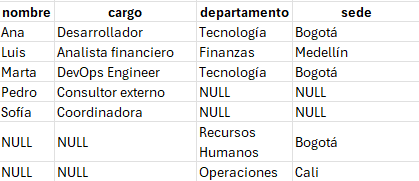
***FULL JOIN***

*Muestra* ***todo****: los que coinciden y los que no, de ambas tablas. Si no hay con quién empatar, se llena con NULL.*

SELECT e.nombre, d.nombre AS departamento

FROM empleados e

FULL OUTER JOIN departamentos d ON e.departamento\_id = d.id;

**

1. **¿Cómo optimizarías una consulta en una base de datos con millones de registros? (1 puntos)**

*Para optimizar una consulta con millones de registros, lo primero que haría es revisar si la consulta esta usando los indices adecuados. Muchas veces se hacen joins o filtros sin tener indices en las columnas correctas, y eso hace que el motor tenga que leer demasiadas filas. Tambien revisaría el execution plan para ver si hay table scans innecesarios.*

*Otro punto importante es limitar la cantidad de datos que traemos, por ejemplo usando cláusulas WHERE bien especificas o paginación si es una consulta para una aplicación. Y si la consulta se hace muy seguido, evaluaria si se puede usar una vista materializada o cachear los resultados.*

*Tambien es recomendable evitar funciones sobre columnas en los filtros (como WHERE YEAR(fecha) = 2023) porque eso evita que se usen los indices. Mejor usar rangos (WHERE fecha BETWEEN '2023-01-01' AND '2023-12-31').*

*Además, si se usan muchos JOINs, revisar si todos son necesarios. A veces se arrastran joins innecesarios o se puede reescribir usando subconsultas más eficientes.*

*Finalmente, si la base de datos lo permite, particionar las tablas por fecha o categoría ayuda un montón, sobre todo cuando hay consultas que sólo tocan una parte del total de los datos. Puede que existan algunas otras recomendaciones.*

1. **¿Qué es la cláusula HAVING en SQL y en qué se diferencia de WHERE? (1 puntos)**

*La cláusula HAVING en SQL se usa para filtrar los resultados despues de aplicar funciones de agregación, como SUM(), AVG(), COUNT(), etc. Es decir, cuando hacemos un GROUP BY, y queremos quedarnos solo con los grupos que cumplen cierta condición, ahí usamos HAVING.*

*Por ejemplo, si queremos ver los departamentos donde el salario promedio es mayor a 3 millones, usamos HAVING AVG(salario) > 3000000.*

*La diferencia con WHERE es que WHERE filtra filas antes de hacer la agrupación, mientras que HAVING filtra los resultados agregados. No se puede usar WHERE para filtrar una función como SUM() directamente, para eso toca usar HAVING.*

## Amazon Web Services (4 puntos)

1. **¿Cuál es la diferencia entre Amazon S3, Amazon RDS y Amazon Redshift? (1 puntos)**

*Amazon S3 es un servicio de almacenamiento de objetos, osea se usa para guardar archivos como imágenes, documentos, respaldos, datasets en crudo, etc. No es una base de datos como tal, pero es super útil para almacenar información no estructurada o semi-estructurada (como JSON, CSV, Parquet, etc).*

*Amazon RDS es un servicio de base de datos relacional administrada. Puedes elegir motores como MySQL, PostgreSQL, Oracle, SQL Server, entre otros. Se usa para aplicaciones que necesitan integridad de datos, transacciones y consultas SQL tipicas. RDS te quita la carga de administrar backups, parches y alta disponibilidad.*

*Amazon Redshift, en cambio, es un data warehouse, pensado para análisis de grandes volumenes de datos. Está optimizado para hacer consultas complejas sobre muchos datos, de forma rápida, gracias a su arquitectura en columnas y procesamiento en paralelo. Se usa sobre todo para business intelligence, reportes y dashboards.*

1. **¿Cuándo usarías Amazon DynamoDB en lugar de Amazon RDS o Amazon Redshift? (1 puntos)**

*Usaría DynamoDB cuando necesito una base de datos rápida, escalable y que soporte muchas consultas al mismo tiempo, sobre todo si los datos no tienen relaciones complejas. Por ejemplo, para guardar sesiones de usuarios, historial de navegación o info de productos en un ecommerce.*

*A diferencia de RDS, que es relacional y sirve cuando necesitas hacer joins, transacciones o manejar datos más estructurados, DynamoDB es más simple y flexible. No necesitas preocuparte por el manejo de la infraestructura, y escala solo.*

*Redshift en cambio, es para análisis de datos, tipo reportes y dashboards grandes. No es buena idea usarlo para cosas en tiempo real o donde se escriban y lean datos constantemente.*

1. **¿Cuáles son las diferencias entre AWS Lambda y AWS EC2 para ejecutar cargas de trabajo? (1 puntos)**

*Lambda es serverless, o sea no tienes que preocuparte por servidores ni por infraestructura. Solo subes tu función, y se ejecuta cuando algo lo dispara (como una petición HTTP o un archivo en S3). Es buena opción para tareas pequeñas, que se ejecutan por eventos y que duran poco tiempo. Además, solo pagas por el tiempo que el código corre.*

*EC2 en cambio es una maquina virtual, como un servidor tradicional. Tienes control total del sistema operativo, puedes instalar lo que quieras y dejarlo corriendo 24/7 si es necesario. Es mejor cuando tienes una app grande, con mucho tráfico, o que necesita procesos en segundo plano todo el tiempo.*

1. **¿Cómo implementarías un mecanismo seguro para que un servicio en AWS acceda a un bucket de S3 sin usar claves de acceso en el código? (1 puntos)**

*Lo más seguro (y lo que AWS recomienda) es evitar por completo el uso de claves en el código y, en su lugar, usar* ***roles de IAM*** *que se le asignan directamente al servicio que necesita acceder a S3.*

*Primero, se debe crear un rol en IAM que tenga los permisos específicos que se requieren sobre el bucket, como por ejemplo leer archivos o escribir. Es importante no dar permisos demas,*

*Después, ese rol se* ***asocia*** *al servicio correspondiente, esto permite que el servicio obtenga* ***credenciales temporales automáticas****, sin necesidad de que el desarrollador o el equipo de devops tenga que meter claves manualmente.*

*Estas credenciales temporales se obtienen por medio de los metadatos del entorno y permiten el acceso controlado al bucket de S3. Además, se rotan solas, lo cual mejora la seguridad, y ayudan a aplicar el principio de menor priviliegio.*

# Sección 2: Prueba práctica SQL (50 puntos)

Tienes las siguientes tablas en una base de datos: **clientes**

id (INT, PRIMARY KEY)

nombre (VARCHAR)

apellido (VARCHAR)

ventas id (INT, PRIMARY KEY)

cliente\_id (INT, FOREIGN KEY a clientes.id)

producto (VARCHAR)

fecha (DATE)

monto (DECIMAL)

**Escribe una consulta para obtener los 5 clientes con mayor monto total de ventas en los últimos 6 meses. (5 puntos)**

SELECT

c.id,

c.nombre,

c.apellido,

SUM(v.monto) AS total\_ventas

FROM

clientes c

JOIN

ventas v ON c.id = v.cliente\_id

WHERE

v.fecha >= DATEADD(MONTH, -6, GETDATE())

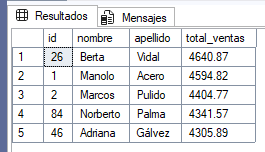
GROUP BY

c.id, c.nombre, c.apellido

ORDER BY

total\_ventas DESC

OFFSET 0 ROWS FETCH NEXT 5 ROWS ONLY;



**Escribe una consulta para calcular el ticket promedio de ventas por cliente en el último año.**

**(5 puntos)**

SELECT

c.id,

c.nombre,

c.apellido,

AVG(v.monto) AS ticket\_promedio

FROM

clientes c

JOIN

ventas v ON c.id = v.cliente\_id

WHERE

v.fecha >= DATEADD(YEAR, -1, GETDATE())

GROUP BY

c.id, c.nombre, c.apellido

ORDER BY

ticket\_promedio DESC;



**Escribe una consulta para obtener el nombre completo de los clientes y su monto total de ventas. (10 puntos)**

SELECT

CONCAT(c.nombre, ' ', c.apellido) AS nombre\_completo,

SUM(v.monto) AS total\_ventas

FROM

clientes c

JOIN

ventas v ON c.id = v.cliente\_id

GROUP BY

c.nombre, c.apellido

ORDER BY

total\_ventas DESC;



**Escribe una consulta para obtener el ingreso promedio de ventas por mes. (10 puntos)**

SELECT

FORMAT(v.fecha, 'yyyy-MM') AS mes,

AVG(mensual.total\_mes) AS ingreso\_promedio

FROM (

SELECT

YEAR(fecha) AS anio,

MONTH(fecha) AS mes,

SUM(monto) AS total\_mes

FROM ventas

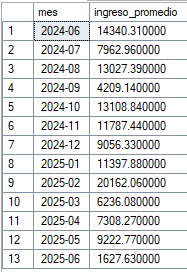
GROUP BY YEAR(fecha), MONTH(fecha)

) AS mensual

JOIN ventas v ON YEAR(v.fecha) = mensual.anio AND MONTH(v.fecha) = mensual.mes

GROUP BY FORMAT(v.fecha, 'yyyy-MM')

ORDER BY mes;



**Escribe una consulta para calcular el ranking de clientes por ventas en el último año. (10 puntos)**

SELECT

RANK() OVER (ORDER BY SUM(v.monto) DESC) AS ranking,

c.id,

c.nombre,

c.apellido,

SUM(v.monto) AS total\_ventas

FROM

clientes c

JOIN

ventas v ON c.id = v.cliente\_id

WHERE

v.fecha >= DATEADD(YEAR, -1, GETDATE())

GROUP BY

c.id, c.nombre, c.apellido

ORDER BY

ranking;

****

**Escribe una consulta para calcular el total de ventas por cliente y luego selecciona solo los clientes cuyo total de ventas es superior al promedio general. (10 puntos)**

WITH total\_ventas\_por\_cliente AS (

SELECT

cliente\_id,

SUM(monto) AS total\_cliente

FROM ventas

GROUP BY cliente\_id

),

promedio\_general AS (

SELECT

AVG(total\_cliente) AS promedio

FROM total\_ventas\_por\_cliente

)

SELECT

c.id,

c.nombre,

c.apellido,

tv.total\_cliente

FROM

total\_ventas\_por\_cliente tv

JOIN

clientes c ON c.id = tv.cliente\_id

CROSS JOIN

promedio\_general pg

WHERE

tv.total\_cliente > pg.promedio

ORDER BY

tv.total\_cliente DESC;



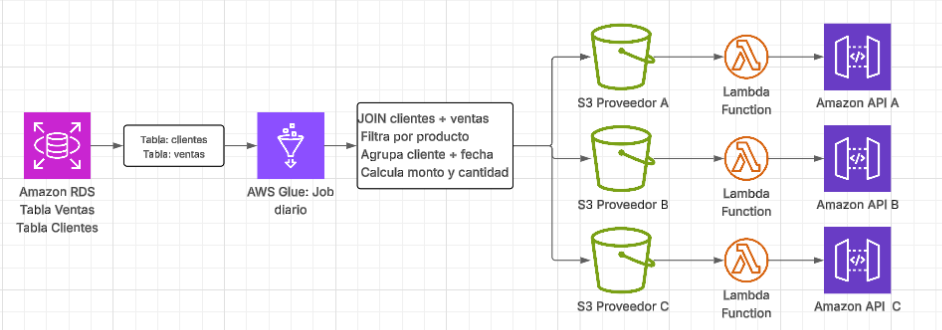
# Sección 3: Prueba práctica python y AWS (50 puntos)

Una empresa requiere disponibilizar a través de un API la información de ventas de sus clientes a 3 diferentes proveedores donde solo se encuentre la información de los productos asociados a cada proveedor. La información que necesitan los proveedores es la siguiente:

* cantidad de transacciones por cliente por dia
* monto total ($) por cliente por día

Nota: Para el ejercicio suponga que la información se encuentra en una base de datos alojada en un RDS de AWS que tiene la estructura de datos expuesta en la sección 2 de la prueba y que existen solamente 3 productos (1 por cada proveedor).

1. **Diseñe un pipeline (diagrama + descripción) para extraer los datos de la base y disponibilizar la información que requiere cada proveedor en batch diario usando los servicios de AWS. (15 puntos)**

****

**Descripción paso por paso**

1. *Fuente de datos*

*Los datos estan almacenados en Amazon RDS, con dos tablas:*

* *clientes(id, nombre, apellido)*
* *ventas(id, cliente\_id, producto, fecha, monto)*

*La tabla ventas contiene los registros de compras y esta relacionada con clientes por cliente\_id.*

### ***Proceso ETL diario con AWS Glue***

*Se crea un* ***Glue Job*** *que corre todos los dias (con un trigger o evento programado).*

* *El Glue job:*
  + *Se conecta a la base en RDS*
  + *Hace JOIN entre ventas y clientes*
  + *Filtra por producto (uno por proveedor)*
  + *Agrupa por cliente y fecha*
  + *Calcula:*
    - *cantidad de transacciones (COUNT(\*))*
    - *monto total (SUM(monto))*

*ejemplo del query:*

SELECT

v.cliente\_id,

CONCAT(c.nombre, ' ', c.apellido) AS nombre\_completo,

v.fecha,

COUNT(\*) AS cantidad\_transacciones,

SUM(v.monto) AS monto\_total

FROM ventas v

JOIN clientes c ON v.cliente\_id = c.id

WHERE v.producto = 'PROD\_A'

GROUP BY v.cliente\_id, c.nombre, c.apellido, v.fecha;

1. *Salida de datos en Amazon S3*

*Cada proveedor recibe un archivo en su carpeta, por fecha*

*s3://empresa/proveedorA/2025-06-13/ventas.csv*

*s3://empresa/proveedorB/2025-06-13/ventas.csv*

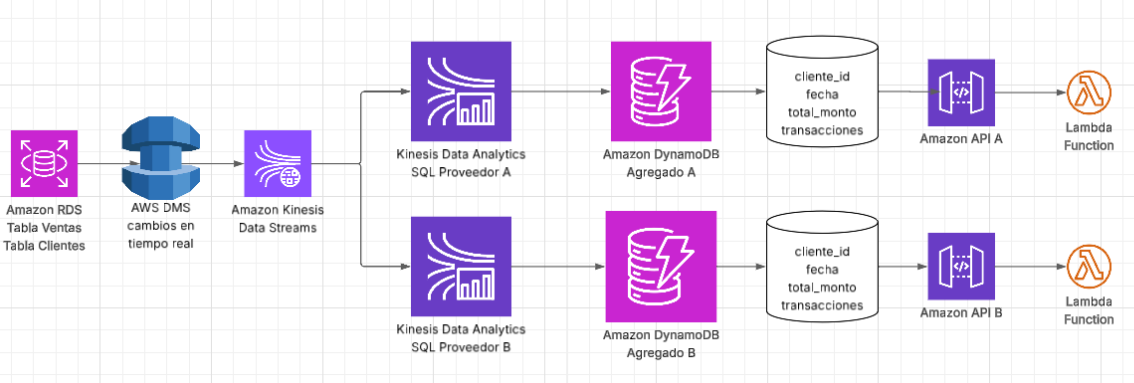
*s3://empresa/proveedorC/2025-06-13/ventas.csv*

1. *API para exponer la data*

*Se crea una* ***Lambda por proveedor****, que lee el archivo del dia desde S3 y lo expone a travez de* ***API Gateway****.*

* *La lambda toma como parámetro la fecha*
* *Busca el archivo correspondiente*
* *Retorna los datos como JSON*

1. **Diseñe un pipeline (diagrama + descripción) para extraer los datos de la base y disponibilizar la información que requiere cada proveedor en tiempo real usando los servicios de AWS. (15 puntos)**

****

**Descripcion paso a paso**

* 1. *Amazon RDS*
* *Contiene la fuente de datos (clientes, ventas)*
* *Se habilita CDC (Change Data Capture) mediante AWS DMS*
  1. *AWS DMS (CDC)*
* *Detecta inserciones en tiempo real en ventas*
* *Envia los cambios a un stream (Kinesis)*
  1. *Amazon Kinesis Data Streams*
* *Recibe cada evento de venta en tiempo real*
* *Entrega los datos a distintos consumidores (1 por producto/proveedor)*
  1. Amazon Kinesis Data Analytics (SQL)
* Se crea una aplicación SQL para cada proveedor:
  + Filtra por producto
  + Agrupa por cliente\_id, fecha
  + Calcula:
    - COUNT(\*) → transacciones
    - *SUM(monto) → total*

### ***Amazon DynamoDB***

* *Se almacena la agregación por cliente y día*
* *Datos están listos para ser consultados por API*
  1. *Amazon API Gateway + Lambda*
* *Cada proveedor tiene su endpoint exclusivo (/ventas?cliente\_id=X&fecha=Y)*
* *Lambda consulta DynamoDB y responde en tiempo real*

1. **Script en python para extraer la información de la base de datos en el formato que requiere el proveedor. (10 puntos)**

import psycopg2

import pandas as pd

conn = psycopg2.connect(

host='tu-host.rds.amazonaws.com',

dbname='mi\_basedatos',

user='admin',

password='tu\_contraseña',

port=5432

)

query = """

SELECT

v.cliente\_id,

CONCAT(c.nombre, ' ', c.apellido) AS nombre\_completo,

v.fecha,

COUNT(\*) AS cantidad\_transacciones,

SUM(v.monto) AS monto\_total

FROM ventas v

JOIN clientes c ON v.cliente\_id = c.id

WHERE v.producto = 'PROVEEDOR\_A'

GROUP BY v.cliente\_id, c.nombre, c.apellido, v.fecha

ORDER BY v.fecha, v.cliente\_id;

"""

df = pd.read\_sql(query, conn)

df.to\_csv('ventas\_proveedor\_A.csv', index=False)

conn.close()

* *que hace la consulta:*

*Se conecta a una base de datos, extrae las ventas de un producto específico (por proveedor), agrupa por cliente y fecha, calcula cuántas ventas hubo y cuánto se vendió en total, y guarda eso en un archivo CSV.*

* *Salida esperada:*

**

1. **Script en python para disponibilizar la información en el API en formato JSON, Implementar logs y manejo de errores (10 puntos)**

from flask import Flask, request, jsonify

import pandas as pd

import logging

# Configurar logging

logging.basicConfig(

filename='api\_ventas.log',

level=logging.INFO,

format='%(asctime)s %(levelname)s: %(message)s'

)

app = Flask(\_\_name\_\_)

# Cargar el archivo procesado (CSV) con las ventas por proveedor

try:

df = pd.read\_csv('ventas\_proveedor\_A.csv')

logging.info("Archivo CSV cargado correctamente.")

except Exception as e:

logging.error(f"Error al cargar CSV: {e}")

df = pd.DataFrame() # DataFrame vacío como fallback

@app.route('/ventas', methods=['GET'])

def get\_ventas():

try:

cliente\_id = request.args.get('cliente\_id', type=int)

fecha = request.args.get('fecha') # formato: YYYY-MM-DD

if cliente\_id is None or fecha is None:

logging.warning("Solicitud inválida: parámetros faltantes.")

return jsonify({'error': 'Parámetros requeridos: cliente\_id y fecha'}), 400

resultado = df[

(df['cliente\_id'] == cliente\_id) & (df['fecha'] == fecha)

]

if resultado.empty:

logging.info(f"No se encontraron datos para cliente {cliente\_id} en fecha {fecha}.")

return jsonify({'mensaje': 'No hay datos para el cliente en esa fecha.'}), 404

data = resultado.to\_dict(orient='records')

logging.info(f"Datos entregados para cliente {cliente\_id} en fecha {fecha}.")

return jsonify(data)

except Exception as e:

logging.error(f"Error en la API: {e}")

return jsonify({'error': 'Error interno del servidor'}), 500

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=5000)

* *Que hace la consulta:*

*Levanta un API con Flask.*

*Lee el archivo ventas\_proveedor\_A.csv.*

*Responde en GET /ventas?cliente\_id=1&fecha=2025-06-13.*

*Devuelve los datos en formato JSON.*

*Implementa:*

*✅ Logs (api\_ventas.log)*

*✅ Manejo de errores y respuestas claras.*

* *Ejemplo de una respuesta en JSON:*

*[*

*{*

*"cliente\_id": 1,*

*"nombre\_completo": "Juan Pérez",*

*"fecha": "2025-06-13",*

*"cantidad\_transacciones": 3,*

*"monto\_total": 125000.0*

*}*

*]*

# Sección 4: CI/CD para Datos (35 puntos)

**Contexto: Estás trabajando en un repositorio de ingeniería de datos que contiene scripts Python de ETL y tests automatizados. Debes asegurar que todo cambio pase por validaciones automáticas antes de desplegarse a producción.**

**Configura un pipeline de CI/CD simple usando GitHub Actions (u otra herramienta como GitLab CI, Jenkins o Azure DevOps) que haga lo siguiente:**

* **Instale las dependencias necesarias.**
* **Ejecute pruebas unitarias y de integración (por ejemplo, tests de pytest).**
* **Valide la sintaxis de los scripts.**
* **Genere documentación automáticamente.**

**Entregables esperados:**

* **Un archivo .github/workflows/ci.yml (o equivalente).**
* **Un pequeño README que explique cómo correrlo y qué hace cada paso.**
* **Captura de pantalla (o logs) que demuestren que el pipeline se ejecuta correctamente.**

***Link del repo:***

[***https://github.com/DubanF/CI-CD-para-Datos-***](https://github.com/DubanF/CI-CD-para-Datos-)

*el readme se encuentra dentro del repositorio.*

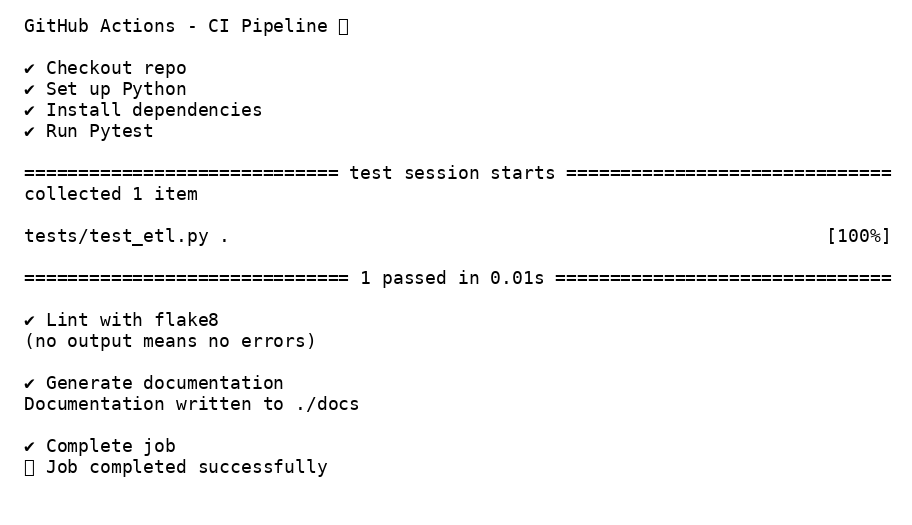
*Este pipeline implementado con* ***GitHub Actions*** *automatiza la validación de cambios en un proyecto de ingeniería de datos que contiene scripts ETL y pruebas automatizadas. Su objetivo es garantizar que todo nuevo código cumpla con los estándares de calidad antes de integrarse a la rama principal.*

*Cada vez que se hace un push o un pull request a la rama main, el pipeline ejecuta automáticamente los siguientes pasos:*

1. ***Instala las dependencias del proyecto*** *desde requirements.txt.*
2. ***Ejecuta pruebas unitarias*** *usando pytest, para verificar que las funciones ETL se comporten correctamente.*
3. ***Valida la sintaxis del código*** *con flake8, asegurando cumplimiento con las buenas prácticas de Python.*
4. ***Genera documentación automática*** *con pdoc, a partir del contenido de la carpeta src/.*

*Este proceso asegura una entrega continua controlada, ayudando a prevenir errores antes de llegar a producción.*

***Captura de logs***



**Preguntas rápidas:**

* Windows o Linux?

**Windows**

* MySQL o PostgreSQL?

**MySQL**

* Batch processing o streaming?

**Batch Processing**

* ETL o ELT?

**ETL**

* Parquet o CSV?

**CSV**

* Spark o Pandas?

**Spark**