PROJETO APLICADO DE PIPELINE DE DADOS - FASE 03

Disciplina: Pipeline de Dados - IA-CD

VISÃO GERAL DA FASE 03

Na Fase 02, vocês estruturaram o pipeline com camadas Bronze/Silver/Gold e integraram com SQLite. Agora é hora de escalar a solução, migrando para processamento distribuído com Apache Spark, evoluindo o banco de dados para trabalhar com Data Warehouses estruturados e orquestrando e automatizando o pipeline com Apache Airflow.

O que vocês farão nesta fase:

- Migrar o processamento de Pandas para Apache Spark
- Evoluir de SQLite para PostgreSQL ou MySQL
- Criar orquestração e automatização com Apache Airflow
- Implementar monitoramento básico
- Preparar pipeline para ambiente de produção

CONCEITOS IMPORTANTES

1. Por que Apache Spark?

Pandas: Funciona bem para dados pequenos (< 10 GB), mas processa tudo em memória em uma única máquina.

Spark: Processa dados de forma distribuída, dividindo o trabalho entre várias máquinas (ou cores). Ideal para:

- Datasets grandes (> 10 GB)
- Processamento que demora muito no Pandas
- Preparação para escalabilidade futura

2. Data Lake vs Data Warehouse

Data Lake (Fase 03):

- Armazena dados brutos de todas as fontes
- Formato flexível (Parquet, JSON, CSV)
- Bronze/Silver/Gold em cloud storage (S3, Azure Data Lake)
- Custo mais baixo

Data Warehouse (Fase 03):

- Armazena dados estruturados e otimizados
- PostgreSQL ou MySQL (evoluindo do SQLite)
- Focado em consultas rápidas
- Camada Gold refinada

3. Orquestração com Apache Airflow

Airflow permite agendar e monitorar o pipeline automaticamente:

- Execução diária, semanal ou em tempo real
- Visualização do fluxo de tarefas
- Alertas em caso de falhas
- Controle de dependências entre tarefas

ENTREGÁVEIS DA FASE 03

ENTREGÁVEL 1: Migração para Apache Spark

Etapa 1: Configuração do Ambiente Spark

Criem um notebook para testar Spark:

```
# 00_spark_setup.ipynb

from pyspark.sql import SparkSession
from pyspark.sql.functions import *

# Criar sessão Spark
spark = SparkSession.builder \
    .appName("Pipeline - Fase 03") \
    .config("spark.driver.memory", "4g") \
    .getOrCreate()

print(f"Spark Version: {spark.version}")
print("Spark configurado com sucesso!")

# Testar leitura
df_test = spark.read.csv('data/source/seu_dataset.csv', header=True, inferSchema=True)
print(f"Dados carregados: {df_test.count()} linhas")
df_test.printSchema()

spark.stop()
```

Etapa 2: Bronze Layer com Spark

```
# 01_spark_bronze_layer.ipynb

from pyspark.sql import SparkSession
from pyspark.sql.functions import current_timestamp, lit
from datetime import datetime

# Iniciar Spark
spark = SparkSession.builder \
    .appName("Bronze Layer") \
    .getOrCreate()

# Ler dados da fonte
df_raw = spark.read.csv('data/source/seu_dataset.csv',
                        header=True,
                        inferSchema=True)

print(f"Dados lidos: {df_raw.count()} linhas")

# Adicionar metadados
df_with_metadata = df_raw \
    .withColumn("data_ingestao", current_timestamp()) \
    .withColumn("fonte_arquivo", lit("seu_dataset.csv"))

# Salvar em Parquet (formato mais eficiente que CSV)
output_path = "data/bronze/dados_brutos.parquet"
df_with_metadata.write \
    .mode("overwrite") \
    .parquet(output_path)

print(f"Dados salvos em: {output_path}")

# Verificar
df_verificacao = spark.read.parquet(output_path)
print(f"Verificação: {df_verificacao.count()} linhas salvas")

spark.stop()
```

Etapa 3: Silver Layer com Spark

```
# 02_spark_silver_layer.ipynb
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

spark = SparkSession.builder \
    .appName("Silver Layer") \
    .getOrCreate()

# Ler dados Bronze
df = spark.read.parquet("data/bronze/dados_brutos.parquet")
print(f"Dados Bronze: {df.count()} linhas")

# ========================================
# TRANSFORMAÇÃO 1: Remover Duplicatas
# ========================================
df_dedup = df.dropDuplicates()
duplicatas_removidas = df.count() - df_dedup.count()
print(f"Duplicatas removidas: {duplicatas_removidas}")

# ========================================
# TRANSFORMAÇÃO 2: Tratar Valores Nulos
# ========================================

# Analisar nulos
print("\nValores nulos:")
df_dedup.select([count(when(col(c).isNull(), c)).alias(c)
                 for c in df_dedup.columns]).show()

# Preencher nulos - exemplo para colunas específicas
# df_clean = df_dedup.fillna({
#     'coluna_numerica': 0,
#     'coluna_texto': 'Desconhecido'
# })

df_clean = df_dedup  # ajustar conforme necessário

# ========================================
# TRANSFORMAÇÃO 3: Padronizar Valores
# ========================================

# Exemplo: padronizar strings
# df_clean = df_clean.withColumn("nome",
#     trim(upper(col("nome")))
# )

# ========================================
# TRANSFORMAÇÃO 4: Criar Novas Colunas
# ========================================

# Exemplo: extrair ano/mês de uma data
# df_clean = df_clean \
#     .withColumn("ano", year(col("data_compra"))) \
#     .withColumn("mes", month(col("data_compra")))

# Adicionar timestamp de processamento
df_clean = df_clean.withColumn("data_processamento", current_timestamp())

# Salvar Silver
output_path = "data/silver/dados_limpos.parquet"
df_clean.write \
    .mode("overwrite") \
    .parquet(output_path)

print(f"\nDados Silver salvos: {df_clean.count()} linhas")

spark.stop()
```

```
spark.stop()
```

Etapa 4: Gold Layer com Spark

```
# 03_spark_gold_layer.ipynb

from pyspark.sql import SparkSession
from pyspark.sql.functions import *

spark = SparkSession.builder \
    .appName("Gold Layer") \
    .getOrCreate()

# Ler dados Silver
df = spark.read.parquet("data/silver/dados_limpos.parquet")

# ========================================
# AGREGAÇÃO 1: Métricas Diárias
# ========================================

# Exemplo para e-commerce
# metricas_diarias = df.groupBy("data_compra").agg(
#     count("id_pedido").alias("total_pedidos"),
#     sum("valor_total").alias("receita_total"),
#     countDistinct("id_cliente").alias("clientes_unicos"),
#     avg("valor_total").alias("ticket_medio")
# )
#
# metricas_diarias.write \
#     .mode("overwrite") \
#     .parquet("data/gold/metricas_diarias.parquet")

# ========================================
# AGREGAÇÃO 2: Análise de Clientes
# ========================================

# Exemplo
# analise_clientes = df.groupBy("id_cliente").agg(
#     max("data_compra").alias("ultima_compra"),
#     count("id_pedido").alias("frequencia"),
#     sum("valor_total").alias("valor_total_gasto")
# )
#
# analise_clientes.write \
#     .mode("overwrite") \
#     .parquet("data/gold/analise_clientes.parquet")

print("Agregações Gold criadas com sucesso")

spark.stop()
```

ENTREGÁVEL 2: Evolução do Banco de Dados

Etapa 1: Configurar PostgreSQL ou MySQL

Instalem PostgreSQL oy MySQL localmente.

Criem arquivo de configuração:

```
# config/database.py

import os
from sqlalchemy import create_engine

# Configuração do banco
DB_CONFIG = {
    'host': 'localhost',  # ou URL do servidor cloud
    'port': 5432,         # 5432 para PostgreSQL, 3306 para MySQL
    'database': 'pipeline_db',
    'user': 'seu_usuario',
    'password': 'sua_senha'
}

def get_connection_string():
    """Retorna string de conexão PostgreSQL"""
    return f"postgresql://{DB_CONFIG['user']}:{DB_CONFIG['password']}@" \
           f"{DB_CONFIG['host']}:{DB_CONFIG['port']}/{DB_CONFIG['database']}"

def get_engine():
    """Retorna SQLAlchemy engine"""
    return create_engine(get_connection_string())
```

Etapa 2: Criar Schema do Banco

```
# 04_create_database_schema.ipynb

import pandas as pd
from sqlalchemy import create_engine, text
from config.database import get_engine

# Conectar ao banco
engine = get_engine()

# Script SQL para criar tabelas
sql_script = """
-- Tabela principal (fato)
CREATE TABLE IF NOT EXISTS vendas (
    id SERIAL PRIMARY KEY,
    id_cliente INTEGER,
    id_produto INTEGER,
    data_venda DATE,
    quantidade INTEGER,
    valor_unitario DECIMAL(10,2),
    valor_total DECIMAL(10,2),
    data_ingestao TIMESTAMP,
    data_processamento TIMESTAMP
);

-- Tabela de clientes (dimensão)
CREATE TABLE IF NOT EXISTS clientes (
    id_cliente INTEGER PRIMARY KEY,
    nome_cliente VARCHAR(255),
    email VARCHAR(255),
    cidade VARCHAR(100),
    estado VARCHAR(2),
    data_cadastro DATE
);

-- Tabela de produtos (dimensão)
CREATE TABLE IF NOT EXISTS produtos (
    id_produto INTEGER PRIMARY KEY,
    nome_produto VARCHAR(255),
    categoria VARCHAR(100),
    preco DECIMAL(10,2)
);

-- Tabela de métricas agregadas
CREATE TABLE IF NOT EXISTS metricas_diarias (
    data_metrica DATE PRIMARY KEY,
    total_vendas INTEGER,
    receita_total DECIMAL(10,2),
    clientes_unicos INTEGER,
    ticket_medio DECIMAL(10,2)
);

-- Criar índices para melhor performance (Opcional)
CREATE INDEX IF NOT EXISTS idx_vendas_data ON vendas(data_venda);
CREATE INDEX IF NOT EXISTS idx_vendas_cliente ON vendas(id_cliente);
CREATE INDEX IF NOT EXISTS idx_vendas_produto ON vendas(id_produto);
"""

# Executar script
with engine.connect() as conn:
    conn.execute(text(sql_script))
    conn.commit()

print("Schema do banco criado com sucesso!")
```

Etapa 3: Carregar Dados no PostgreSQL

```
# 05_load_to_postgres.ipynb

import pandas as pd
from pyspark.sql import SparkSession
from config.database import get_connection_string

# Iniciar Spark
spark = SparkSession.builder \
    .appName("Load to PostgreSQL") \
    .config("spark.jars", "postgresql-42.6.0.jar") \
    .getOrCreate()

# Ler dados Silver
df = spark.read.parquet("data/silver/dados_limpos.parquet")

# Propriedades de conexão JDBC
jdbc_url = get_connection_string().replace("postgresql://", "jdbc:postgresql://")
properties = {
    "driver": "org.postgresql.Driver",
    "user": "seu_usuario",
    "password": "sua_senha"
}

# Salvar no PostgreSQL
df.write.jdbc(
    url=jdbc_url,
    table="vendas",
    mode="overwrite",
    properties=properties
)

print("Dados carregados no PostgreSQL com sucesso!")

spark.stop()
```

ENTREGÁVEL 3: Orquestração com Apache Airflow

Etapa 1: Instalação do Airflow (Docker)

Criem arquivo `docker-compose.yaml`:

```
version: '3.8'
services:
  postgres:
    image: postgres:13
    environment:
      POSTGRES_USER: airflow
      POSTGRES_PASSWORD: airflow
      POSTGRES_DB: airflow
    volumes:
      - postgres-db-volume:/var/lib/postgresql/data

  airflow-webserver:
    image: apache/airflow:2.7.0
    depends_on:
      - postgres
    environment:
      AIRFLOW__CORE__EXECUTOR: LocalExecutor
      AIRFLOW__DATABASE__SQL_ALCHEMY_CONN: postgresql+psycopg2://airflow:airflow@postgres/airflow
    volumes:
      - ./airflow/dags:/opt/airflow/dags
      - ./airflow/logs:/opt/airflow/logs
      - ./data:/opt/airflow/data
    ports:
      - "8080:8080"
    command: webserver

  airflow-scheduler:
    image: apache/airflow:2.7.0
    depends_on:
      - postgres
    environment:
      AIRFLOW__CORE__EXECUTOR: LocalExecutor
      AIRFLOW__DATABASE__SQL_ALCHEMY_CONN: postgresql+psycopg2://airflow:airflow@postgres/airflow
    volumes:
      - ./airflow/dags:/opt/airflow/dags
      - ./airflow/logs:/opt/airflow/logs
      - ./data:/opt/airflow/data
    command: scheduler

volumes:
  postgres-db-volume:
```

Etapa 2: Criar DAG do Pipeline

```
# airflow/dags/pipeline_etl_dag.py

from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime, timedelta
import sys
sys.path.append('/opt/airflow')

# Funções do pipeline
def extract_bronze(**context):
    """Executa camada Bronze"""
    print("Executando Bronze Layer...")
    # Importar e executar seu código Bronze
    # Aqui você pode chamar suas funções de extração
    return "Bronze concluído"

def transform_silver(**context):
    """Executa camada Silver"""
    print("Executando Silver Layer...")
    # Importar e executar seu código Silver
```

```
    return "Silver concluído"

def aggregate_gold(**context):
    """Executa camada Gold"""
    print("Executando Gold Layer...")
    # Importar e executar seu código Gold
    return "Gold concluído"

def load_database(**context):
    """Carrega dados no banco"""
    print("Carregando no banco de dados...")
    return "Load concluído"

# Configuração do DAG
default_args = {
    'owner': 'equipe-pipeline',
    'depends_on_past': False,
    'start_date': datetime(2025, 1, 1),
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG(
    'pipeline_etl_completo',
    default_args=default_args,
    description='Pipeline ETL Bronze -> Silver -> Gold',
    schedule_interval='@daily',  # Executar diariamente
    catchup=False,
    tags=['pipeline', 'etl']
)

# Definir tarefas
task_bronze = PythonOperator(
    task_id='extract_bronze',
    python_callable=extract_bronze,
    dag=dag
)

task_silver = PythonOperator(
    task_id='transform_silver',
    python_callable=transform_silver,
    dag=dag
)

task_gold = PythonOperator(
    task_id='aggregate_gold',
    python_callable=aggregate_gold,
    dag=dag
)

task_load = PythonOperator(
    task_id='load_database',
    python_callable=load_database,
    dag=dag
)

# Definir dependências (ordem de execução)
task_bronze >> task_silver >> task_gold >> task_load
```

Etapa 3: Executar Airflow

```
 # Subir os containers
docker-compose up -d

# Acessar interface web
# http://localhost:8080
# User: airflow
# Password: airflow
```

ENTREGÁVEL 4: Monitoramento Básico

Criem um notebook para monitorar o pipeline:

```
# 07_monitoring.ipynb

import pandas as pd
from datetime import datetime
import matplotlib.pyplot as plt

# ========================================
# MONITORAMENTO DE EXECUÇÃO
# ========================================

# Simular log de execuções
execucoes = {
    'data_execucao': [],
    'camada': [],
    'status': [],
    'tempo_segundos': [],
    'registros_processados': []
}

# Adicionar logs (vocês devem adaptar para seus dados reais)
# execucoes['data_execucao'].append(datetime.now())
# execucoes['camada'].append('bronze')
# execucoes['status'].append('sucesso')
# ...

df_log = pd.DataFrame(execucoes)

# Análises
print("="*50)
print("MONITORAMENTO DO PIPELINE")
print("="*50)

# Taxa de sucesso
if len(df_log) > 0:
    taxa_sucesso = (df_log['status'] == 'sucesso').sum() / len(df_log) * 100
    print(f"\nTaxa de Sucesso: {taxa_sucesso:.2f}%")

    # Tempo médio por camada
    print("\nTempo Médio por Camada:")
    print(df_log.groupby('camada')['tempo_segundos'].mean())

    # Visualização
    plt.figure(figsize=(12, 4))

    plt.subplot(1, 2, 1)
    df_log['status'].value_counts().plot(kind='bar')
    plt.title('Status das Execuções')
    plt.ylabel('Quantidade')

    plt.subplot(1, 2, 2)
    df_log.groupby('camada')['tempo_segundos'].mean().plot(kind='bar')
    plt.title('Tempo Médio por Camada')
    plt.ylabel('Segundos')

    plt.tight_layout()
    plt.savefig('pipeline_monitoring.png')
    print("\nGráficos salvos em: pipeline_monitoring.png")
```

DOCUMENTAÇÃO

Atualizem a documentação com os avanços do projeto:

```
# Pipeline de Dados - Fase 03

## Arquitetura

### Processamento
- Apache Spark para processamento distribuído
- Parquet como formato de armazenamento
- Camadas Bronze/Silver/Gold

### Armazenamento
- Data Lake: AWS S3 / Azure Blob Storage
- Data Warehouse: PostgreSQL
- Formato: Parquet (mais eficiente que CSV)

### Orquestração
- Apache Airflow para automação
- Execução: Diária
- Monitoramento via Airflow UI

## Estrutura de Dados

### Data Lake (Cloud)
- Bronze: `s3://bucket/bronze/` ou `azure://container/bronze/`
- Silver: `s3://bucket/silver/` ou `azure://container/silver/`
- Gold: `s3://bucket/gold/` ou `azure://container/gold/`

### Data Warehouse (PostgreSQL)
- Tabela: `vendas` (fato principal)
- Tabela: `clientes` (dimensão)
- Tabela: `produtos` (dimensão)
- Tabela: `metricas_diarias` (agregações)

## Como Executar

### 1. Processar com Spark
```bash
# Executar notebooks na ordem
01_spark_bronze_layer.ipynb
02_spark_silver_layer.ipynb
03_spark_gold_layer.ipynb
```

2. Carregar no Banco

```
04_create_database_schema.ipynb
05_load_to_postgres.ipynb
```

3. Executar Airflow

```
docker-compose up -d
# Acessar: http://localhost:8080
```

Melhorias em Relação à Fase 02

- Processamento 10x mais rápido com Spark
- Banco de dados robusto (PostgreSQL)
- Orquestração e automação completa com Airflow
- Preparado para escalabilidade

Próximos Passos (Fase 04)

- Refatoração do código (funções e classes)
- Dashboards (Opcional)
- Documentação final
```

- Apresentação do Pipeline de Dados

```

---

## DATA DE ENTREGA

**Prazo:** 17/11/2025

**Formato:**

- Arquivo ZIP com código completo (se preferir) - Abrirei uma atividade de entrega na D2L.
- Todos os notebooks executáveis
- Airflow rodando (printscreen da DAG no Airflow)

---

Bom trabalho na Fase 03!
```