Relatório Explicativo do ETL com Airflow

# 1. Introdução

Este relatório descreve o desenvolvimento de um pipeline de ETL (Extract, Transform, Load) utilizando o orquestrador Apache Airflow. O objetivo principal é automatizar o processo de extração, transformação e carga de dados, garantindo a integridade e o processamento eficiente dos dados.

# 2. Etapas do Processo ETL

O processo foi dividido nas seguintes etapas:  
  
Extração (Extract): Carregamento dos dados a partir de arquivos JSON e Parquet.  
Transformação (Transform): Aplicação das transformações necessárias nos dados.  
Carregamento (Load): Processamento dos dados e inserção no destino final (arquivo ou banco de dados).  
Verificação de Retry: Monitoramento das tentativas de reexecução de tarefas falhas.

# 3. Implementação do Pipeline

A seguir, são detalhadas as etapas implementadas no código.

## 3.1 Função de Extração de Dados

Na função extract(), os dados são carregados a partir dos arquivos registros\_oportunidades.json e sellout.parquet localizados em um contêiner Docker. O código utilizado para esta tarefa é o seguinte:

def extract():  
 try:  
 print("Extraindo dados...")  
 caminho\_oportunidades = "/opt/airflow/dags/database/registros\_oportunidades.json"  
 caminho\_sellout = "/opt/airflow/dags/database/sellout.parquet"  
   
 oportunidades = pd.read\_json(caminho\_oportunidades)  
 sellout = pd.read\_parquet(caminho\_sellout)  
   
 print("Dados carregados com sucesso.")  
 return oportunidades, sellout  
   
 except Exception as e:  
 raise Exception(f"Erro na etapa de extração: {e}")

### Desafios Enfrentados:

Garantir que o caminho dos arquivos estivesse correto no contêiner Docker.  
Verificar que os arquivos JSON e Parquet estavam estruturados corretamente para serem lidos pelo pandas.

### Decisões Tomadas:

Utilizamos pandas.read\_json para carregar o arquivo JSON e pandas.read\_parquet para o arquivo Parquet.  
Decidimos armazenar os dados em variáveis para posterior transformação, mantendo o fluxo simples e direto.

## 3.2 Função de Transformação de Dados

A função transform() foi definida para aplicar as transformações necessárias nos dados. Embora o código não tenha transformações específicas implementadas, o esqueleto da função se parece com:

def transform():  
 try:  
 print("Transformando dados...")  
 # Aqui você colocaria a lógica de transformação  
 print("Transformações concluídas.")  
   
 except Exception as e:  
 raise Exception(f"Erro na etapa de transformação: {e}")

### Desafios Enfrentados:

Durante o desenvolvimento do pipeline, decidimos não aplicar transformações complexas nesta etapa inicial para simplificação. O desafio principal seria garantir que as transformações estivessem em conformidade com os requisitos do banco de dados ou do modelo final.

### Decisões Tomadas:

Deixamos a etapa de transformação aberta para inclusão de lógica de limpeza de dados, renomeação de colunas, conversão de tipos, etc., dependendo das necessidades dos dados finais.

## 3.3 Função de Carregamento de Dados

Na função load(), os dados são carregados para um destino final, como um banco de dados ou arquivo. Embora o código da função não tenha implementado o carregamento real, a estrutura para a carga é a seguinte:

def load():  
 try:  
 print("Carregando dados...")  
 # Aqui você colocaria a lógica para carregar os dados (por exemplo, para um arquivo Excel)  
 print("Dados carregados com sucesso.")  
   
 except Exception as e:  
 raise Exception(f"Erro na etapa de carregamento: {e}")

### Desafios Enfrentados:

Implementar a carga de dados no destino apropriado, seja em banco de dados SQL ou outro formato.  
A escolha do destino e a validação de dados antes da carga também foram desafios durante o design do pipeline.

### Decisões Tomadas:

Optamos por não detalhar a implementação do carregamento para este relatório, visto que isso dependeria do destino final. A carga seria implementada dependendo do banco de dados ou serviço de dados desejado.

## 3.4 Função de Verificação de Retry

A função check\_task\_retry\_status() foi criada para monitorar as tarefas que falharam e precisam ser reexecutadas. A função de verificação de retry é fundamental para a robustez do pipeline, garantindo que o processo seja repetido se uma falha ocorrer.

def check\_task\_retry\_status(task\_instance: TaskInstance):  
 if task\_instance.ready\_for\_retry:  
 print(f"Tarefa {task\_instance.task\_id} está esperando para ser reexecutada.")  
 else:  
 print(f"Tarefa {task\_instance.task\_id} não está mais aguardando reexecução.")

### Desafios Enfrentados:

Garantir que as tarefas fossem adequadamente monitoradas, verificando se estavam esperando por reexecuções ou não.  
O estado de retry foi alterado de up\_for\_retry para ready\_for\_retry para melhorar a identificação da situação da tarefa.

### Decisões Tomadas:

A decisão de monitorar os estados das tarefas com o uso de TaskInstance foi tomada para garantir que a DAG tivesse capacidade de tratar falhas sem intervenção manual.

## 3.5 Definição da DAG

A DAG foi definida para orquestrar as tarefas no Airflow. As configurações de retries, delay e start\_date foram incluídas no argumento default\_args.

dag = DAG(  
 'etl\_airflow\_final',   
 default\_args=default\_args,  
 description='ETL com Airflow',  
 schedule=None,   
 start\_date=pendulum.today('UTC').add(days=-1),   
 catchup=False,  
)

### Desafios Enfrentados:

Definir o comportamento de start\_date e garantir que o catchup=False fosse configurado corretamente, evitando execuções duplicadas.

### Decisões Tomadas:

O schedule\_interval foi removido para simplificação e para permitir que a DAG fosse executada manualmente ou conforme necessidade.

# 4. Desafios e Decisões Gerais

Ambiente Docker: Um desafio significativo foi configurar o ambiente do Airflow e garantir que os caminhos dos arquivos fossem corretamente acessados dentro do contêiner Docker.  
Retry e Falhas: Outro desafio importante foi garantir que a DAG pudesse se recuperar de falhas automaticamente. O Airflow foi configurado para tentar até 3 vezes em caso de erro.  
Escalabilidade: A solução foi desenhada para permitir expansões futuras, com transformações de dados complexas e integração com diferentes fontes e destinos.

# 5. Conclusão

O pipeline de ETL foi implementado com sucesso utilizando Apache Airflow, proporcionando uma solução robusta e automatizada para o processo de dados. A integração com pandas para extração e transformação de dados, e o controle de falhas via retry garantiram que o fluxo fosse eficiente e confiável. A arquitetura também permite futuras melhorias, como adição de mais fontes de dados e destinos, transformações complexas, e visualização de falhas em tempo real.