[AIRFLOW] Fábrica de DAG's

Proprietário: João Carlos Romero Monteiro
Última atualização em: ago. 13, 2025
• 2 pessoas visualizaram

Visão Geral

Esta solução é um framework robusto para a **criação dinâmica de DAGs no Apache Airflow**. Em vez de codificar cada DAG individualmente, o sistema utiliza arquivos de configuração **YAML** simples para definir os parâmetros, tarefas e a lógica de execução. Isso centraliza a configuração, melhora a manutenibilidade e reduz a duplicação de código, permitindo a criação e modificação de novos fluxos de trabalho de extração, transformação e carregamento (ETL) de forma rápida e segura.

O principal objetivo é automatizar a ingestão de dados de diversas fontes (bases de dados) para o **Google Cloud Platform (GCP)**, utilizando o **Dataproc** para o processamento de dados e o **BigQuery** como destino final.

A solução oferece os seguintes recursos:

- **Configuração centralizada**: Todos os parâmetros do DAG e das tabelas são definidos em arquivos YAML.
- **Geração dinâmica**: Um único script Python lê as configurações e cria múltiplos DAGs, um para cada arquivo YAML.
- **Reutilização de código**: Um conjunto de funções e operadores padronizados (`migracao_tabelas_bronze` , `migracao_tabelas_silver` , `utils_vm`) é reutilizado para todas as tabelas.
- **Flexibilidade de agendamento**: Cada tabela pode ter um agendamento de cron personalizado, permitindo execuções mais granulares sem a necessidade de múltiplos DAGs.
- **Gerenciamento de recursos**: O Dataproc Cluster é criado no início da execução do DAG e deletado ao final, otimizando custos e uso de recursos.
- **Segurança**: O acesso às credenciais do banco de dados é feito através do **Google Secret Manager**, com o valor sendo injetado de forma segu

Resumir

⚠️ Fábrica de DAG's só gerencia a execução de scripts Python, SQL e PySpark, ela não processada. A documentação do processo PySpark para extração de dados,

Informações

- Repositório:
  - https://github.com/trademasterbr/dados Conectar a conta do Github
- Path dos arquivos:
  - engenharia\DAG\ingestion

Estrutura e Fluxo de Execução

Cada DAG criado por este framework segue uma estrutura padronizada para processar uma ou mais tabelas. O fluxo de execução para cada DAG é o seguinte:

1. `init_task` : Ponto de partida do DAG.
2. `get_secret_manager` : Tarefa que busca as credenciais de acesso ao banco de dados no Google Secret Manager. Este valor é armazenado no XCom para ser usado pelas tarefas seguintes.
3. `create_cluster` : Cria um cluster Dataproc com a configuração especificada no arquivo YAML (tipo de VM, número de workers). O cluster é reutilizado para o processamento de todas as tabelas naquele DAG.
4. **Loop por Tabela**: Para cada tabela configurada no arquivo YAML, um conjunto de tarefas é criado:
   - `branch_table_{table_name}` : Uma tarefa condicional que decide se a tarefa de migração da tabela deve ser executada ou pulada. A decisão é baseada no `custom_cron` definido no YAML. Se a data lógica da execução do DAG (`logical_date`) corresponder ao cron, a tarefa avança para a camada Bronze; caso contrário, avança para a tarefa de "pular".
   - `migrar_tabela_bronze_{table_name}` : Tarefa PySpark que extrai dados do banco de dados de origem (SQL Server, PostgreSQL, etc.) e os salva na camada Bronze do BigQuery (dataset **bronze**).
   - `update_{prefixo}{table_name}` : Tarefa Python que processa os dados da camada Bronze e os migra para a camada Silver do BigQuery (dataset **silver**)

- `skip_table_{table_name}` : Tarefa "vazia" (`DummyOperator`) que é executada quando a tarefa Bronze é pulada pelo `BranchPythonOperator` .

sendo pulado).

5. `sync_task` : Uma tarefa de junção final que espera que todas as tarefas de `join_table_processing` de todas as tabelas sejam concluídas.
6. `delete_cluster` : Deleta o cluster Dataproc, liberando os recursos.
7. `end_task` : Ponto final do DAG.

Como Usar e Alterar as Configurações

A solução é configurada através de arquivos YAML. Cada arquivo YAML define um único DAG. Para criar um novo DAG, basta criar um novo arquivo YAML no diretório `cluster_process` com a seguinte estrutura:

Pontos de atenção:

- Se definir um custom_cron ele deve coincidir com o cron “global” configurado, pois esse custom cron não dispara a DAG, ele só define se naquele momento que a DAG executar a tabela será processada ou não
- A definição da variavel no yaml ‘upsert’ está relacionada ao processo de criação da tabela silver. Se upsert=true então o script da silver precisa ser um merge ou insert, pois o framework ira rodar um “append” nessa etapa. Se a consulta da silver for merge e o upsert não for marcado como true a DAG irá falhar.
- Alguns arquivos de configuração possuem a flag “user_marks”. Essa marcação indica se, no momento de redar a consulta postgres o framework irá realizar a consulta where “campo_delta” >= “yyyy-mm-dd” ou where campo_delta >= “yyyy-mm-dd”. Essa configuração é necessária para os bancos yunus!

Exemplo de Arquivo YAML (`capture_posting.yaml`)

```
1  dag_id: capture_posting_frw
2  description: 'Processa as tabelas dos projetos capture_posting da camad
3  schedule_interval: '0,20,40 2-23 * * *'
4  max_active_runs: 1
5  dagrun_timeout_hours: 2
```

```
tags:
  - capture_posting
```


```

cluster_name_prefix: cluster
vm_type: e2-highmem-2
num_workers: 5
# Para a primeira execução, se der certo.

schema_db: public
database_type: postgres
prefixo: capture_posting
database: capture-posting
secret_database_name: eng-data-postgres

tabelas_config:
  - table_name: financial_entry_detail
    column_date: 'DATE_UPDATED'
    coluna_hora: ''
    upsert: 'true'

  - table_name: financial_entry_payable
    column_date: 'DATE_UPDATED'
    coluna_hora: ''
    upsert: 'true'

  - table_name: financial_entry_exception_financial_entry_receivable
    column_date: 'date_created'
    coluna_hora: ''

  - table_name: financial_entry_contested
    column_date: 'date_updated'
    coluna_hora: ''

  - table_name: financial_entry_item
    column_date: 'date_created'
    coluna_hora: ''

  - table_name: financial_entry_exception
    column_date: 'DATE_UPDATED'
    coluna_hora: ''

```

```
49    - table_name: financial_entry
50      column_date: 'date_updated'
```


```
53
54    - table_name: financial_entry_receivable
55      column_date: 'date_updated'
56      coluna_hora: ''
57      upsert: 'true'
58
59    - table_name: financial_payable_bucket
60      column_date: 'date_updated'
61      coluna_hora: ''
62
63    - table_name: calendary
64      column_date: 'date_updated'
65      coluna_hora: ''
66      custom_cron: '0 0 1 * *'
```

Alterações no Código

Para modificar a lógica de execução ou adicionar novas funcionalidades, o arquivo principal `dag.py` pode ser alterado. As principais seções que podem ser customizadas são:

- **Lógica de agendamento (`_should_run_table`)**: A função `_should_run_table` pode ser modificada para incluir lógicas de agendamento mais complexas.
- **Parâmetros de configuração**: Variáveis globais, como tipo de cluster ou buckets, podem ser alteradas.
- **Operadores**: É possível adicionar novos operadores ou modificar os existentes (e.g., usar um `SparkKubernetesOperator` em vez do `DataprocSubmitJobOperator`).
- **Funções de utilidade**: As funções em `common/utils.py` e `common/utils_vm.py` podem ser estendidas para suportar novos tipos de banco de dados ou configurações de cluster.
- **Lógica PySpark**: Para alterar a lógica de ingestão de dados, modifique os arquivos PySpark (e.g., `migracao_tabelas_bronze.py`). A flag `use_custom_pyspark_code` no YAML permite

dados × + Adicionar categoria

Bases d / **[AIRFLOW] F...**

Seja o primeiro a adicionar uma reação