Relatório LAMIA 11 (NOVA ORDEM) Pipelines de Dados - Airflow (I)

Kaique Medeiros Lima

1 Introdução

O card 11 teve que ser refeito por causa da adição de dois novos cards ao bootcamp. O card se baseia no curso Apache Airflow: The Hands-On Guide, onde é discutido o Apache Airflow para a criação de pipelines de dados.

2 Descrição da atividade

2.1 Why Airflow

Airflow lida com problemas como:

- Substituto de cron;
- Tolerante a falhas;
- Regras de dependência;
- Código em Python;
- Manuseio de falhas de teste;
- Reporte e alerta de falhas;
- Extensível e modulável;
- Interface gráfica (UI) bonita.

2.2 What is Airflow

Definição: maneira de programaticamente autorizar, agendar e monitorar data pipelines.

2.2.1 Componentes Principais

- **Web server**: servidor Flask que roda no Gunicorn e serve para o dashboard da interface.
- **Scheduler**: daemon responsável pelo agendamento.
- **Metadata database**: banco de dados onde todos os metadados relacionados ao administrador e trabalhos são armazenados, suportando a biblioteca SQLAlchemy.

- **Executor**: define como as tarefas são executadas.
- **Worker**: processo que executa as tarefas, dependendo do executor.

2.2.2 Chaves

- **DAG**: Objeto grafo representando a data pipeline.
- **Operator**: Descreve uma tarefa única na pipeline.
- **Task**: Instância de um operador.
- **TaskInstance**: Representa a combinação de DAG, Task e um ponto no tempo.
- **Workflow**: Combinação de todos os itens acima

2.3 O que o Airflow oferece?

- Pipelines configuradas em Python, tornando-as dinâmicas;
- Representação gráfica das DAGs;
- Escalabilidade com configuração adequada;
- *Backfill*: habilidade de rodar uma DAG do passado até o presente;
- Extensível;
- Muito mais.

2.4 Como o Airflow funciona

1. O **Scheduler** lê a pasta DAG.
2. O DAG é analisado para criar a **DagRun**, baseada nos parâmetros de agendamento.
3. Uma **TaskInstance** é instanciada para cada tarefa que precisa ser executada, recebendo a flag "Agendada" no banco de metadados.
4. O **Scheduler** altera a flag para "Na fila" e envia para os executores.
5. Os executores processam as tarefas, alterando a flag para "Em andamento", e os **workers** começam a execução.
6. Ao finalizar, o estado da tarefa é registrado no banco de dados, e o **Scheduler** atualiza a **DagRun** para "Sucesso" ou "Falha". A UI é atualizada periodicamente.

2.5 Propriedades importantes das DAGs

- **dag_id**: identificador único da DAG.
- **description**: descrição da DAG.
- **start_date**: data de início da DAG.
- **schedule_interval**: frequência de execução, a partir de *start_date*.
- **default_args**: dicionário com parâmetros aplicados a operadores e tasks.
- **catchup**: define se o *scheduler* acompanhará execuções atrasadas (padrão *True*).

2.6 Operador

Determina o que é feito, representando uma tarefa singular no processo da pipeline. Operadores são independentes e podem executar simultaneamente. O mesmo resultado deve ser produzido em reexecuções. Uma tarefa é criada instanciando a classe de um operador, que define a natureza da tarefa e sua execução. Ao instanciar, a tarefa vira um nó na DAG.

2.6.1 Tipos de operadores

- **Ação**: executam uma ação (*BashOperator*, *PythonOperator*, *EmailOperator*).
- **Transferência**: movem dados entre sistemas (*PrestoToMysqlOperator*, *SftpOperator*).
- **Sensor**: aguardam eventos (*FileSensor*, etc.).

2.7 Backfill e Catchup

DagRun representa a execução de uma instância da DAG, criada manualmente ou por agendamento. Cada execução possui um *timestamp* único (*execution_date*). **Catchup** É uma funcionalidade para executar *DAG Runs* pendentes desde o último agendamento. Pode ser desativada com *catchup=False*. **Backfill** é o processo manual para executar *DAG Runs* retroativas, útil para reprocessar dados históricos.

2.8 Lidando com fuso horário no Airflow

Caso o desenvolvedor cria o objeto datetime.datetime sem passar o fuso horário, torna o programa Ciente de fuso horário, caso não especifíque o fuso horário, vira Simples. Recomenda-se usar objetos cientes em UTC.

Ciente vs Simples

- Objetos **simples**: criados sem especificar fuso horário.
- Objetos **cientes**: criados especificando fuso horário.

2.8.1 Melhores práticas sobre o manuseio de fuso horário

- Sempre use objetos **cientes** de fuso horário.
- A função `datetime.datetime()` em Python fornece um objeto **simples** por padrão.
- Um objeto `datetime` sem fuso horário não está no UTC.
- Importe `airflow.timezone` para criar objetos **cientes**, ou deixe o Airflow realizar a conversão automaticamente.
- Sempre usar o fuso horário UTC é recomendado.

2.9 Como deixar as tarefas dependentes

2.9.1 depends_on_past

- Define-se no nível da task.
- Se a instância da task passada falhou, a task atual não será executada.
- Consequentemente, a task não terá um estado.
- A primeira task com `start_date` pode ser executada.

2.9.2 wait_for_downstream

- Define-se ao nível da task.
- A instância da task X vai esperar por todas as tasks imediatamente abaixo no fluxo da instância anterior à task X terminar.

2.10 Como estruturar a pasta DAG

2.10.1 O que é uma pasta DAG

- É a pasta onde as Airflow DAGs estão.
- Definida pelo parâmetro `dags_folder`.
- O caminho deve ser absoluto.
- Por padrão é: `$AIRFLOW_HOME/dags`.
- Problemas: DAGs demais, DAGs usando muitos arquivos externos.

2.10.2 Modo 1: Zip

- Criar uma pasta zip e compactar todas as DAGs e todos os seus arquivos extras.
- As DAGs devem estar no root do arquivo zip.
- O Airflow vai escanear e carregar o zip para as DAGs.
- Se dependências de módulo forem necessárias, é preciso usar `virtualenv` e `pip`.

2.10.3 Modo 2: DagBag

- Uma `DagBag` é uma coleção de DAGs, analisadas de uma árvore de pastas, que possui configurações de alto nível.
- Torna mais fácil a execução de ambientes distintos (dev/staging/prod).
- Um único sistema consegue executar múltiplos sets de configurações independentes.
- Permite a adição de novas pastas DAG por meio da criação de um script padrão na pasta `dags`.

2.11 Como lidar com falhas nas DAGs

2.11.1 Detecção nas DAGs

Ao nível da DAG:

- `dagrun_timeout`
- `sla_miss_callback`
- `on_failure_callback`
- `on_success_callback`

Ao nível das tasks:

- `email`
- `email_failure`
- `email_on_retry`
- `retries`
- `retry_delay`
- `retry_exponential_backoff`
- `max_retry_delay`
- `execution_timeout`
- `on_failure_callback`
- `on_success_callback`
- `on_retry_callback`

2.12 Como testar suas DAGs

2.12.1 Teste de unidade com PyTest

- Permite diferentes tipos de testes.
- Fácil de entender (boilerplate mínimo).
- Bonito e útil para informar falhas.
- Amplamente usado, bem documentado e extensível.

2.12.2 Testes de validação de DAGs

- Checa se a DAG é válida.
- Verifica se não existem ciclos.
- Valida os argumentos padrões.

2.12.3 Testes de definição de DAG/Pipeline

- Testa se as modificações foram intencionais.
- Checa o número total de tasks.
- Verifica a natureza das tasks.
- Analisa o fluxo acima e abaixo das dependências das tasks.

2.12.4 Testes de unidade

- Testa funções externas ou operadores personalizados.
- Verifica a lógica das funções.

2.12.5 Testes de integração

- Testa se as tasks funcionam bem entre si usando um subset de dados de produção.
- Verifica a troca de informações entre tasks, suas entradas e dependências.
- Necessita de ambientes de desenvolvimento, teste, aceitação e produção.

2.12.6 Testes cabo a rabo de Pipeline

- Testa a pipeline dos dados.
- Valida a saída, a lógica inteira e a performance.
- Também necessita de ambientes de desenvolvimento, teste, aceitação e produção.

3 Conclusão

Nesse card foi aprensentado Pipelines de Dados - Airflow. O Apache Airflow é uma ferramenta para orquestrar workflows, permitindo a criação e o gerenciamento de fluxos de tarefas de forma dinâmica e escalável. Usando DAGs, ele organiza as tarefas em pipelines, garantindo dependências e execução eficiente. Este material explica como estruturar DAGs, lidar com falhas, criar dependências entre tarefas e testar pipelines.