Relatório - 18 - Prática: Pipelines de Dados II - Airflow (II)

Jefferson korte junior

Seção 5:

Aula 2 - Sequential Executor with SQLite:

Nessaaula, aprendemos sobre o **SQLite**, que é um banco de dados simples e leve, usado para testes e desenvolvimento. O Airflow precisa de um banco para guardar suas informações, e o SQLite é uma opção prática pra isso.

Também foi explicado o **Sequential Executor**, que executa as tarefas **uma de cada vez**, sem paralelismo. Isso quer dizer que ele não roda várias tarefas ao mesmo tempo. Por isso, ele só é usado em ambientes de teste ou para aprender.

Aula 3 - Local Executor with PostgreSQL:

O **Local Executor** já permite rodar várias tarefas ao mesmo tempo, aproveitando bem os recursos da máquina, como os núcleos do processador. Isso ajuda bastante quando o fluxo de dados é maior.

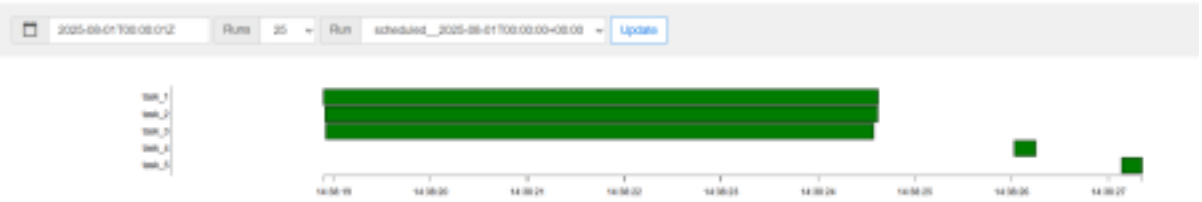
Nessa aula também vimos o **PostgreSQL**, que é um banco de dados mais completo e usado em ambientes profissionais. Ele é conhecido por ser rápido, confiável e muito bom pra trabalhar com grandes volumes de dados.

Aula 4 - Practice Executing tasks in parallel with the Local Executor

Aqui começamos a usar o **LocalExecutor** na prática. Esse comando sobe os containers necessários para rodar o Airflow utilizando o LocalExecutor, que permite a execução de tarefas em paralelo, aproveitando melhor os recursos da máquina. A estrutura do projeto já vem organizada com pastas de DAGs, arquivos de configuração e volumes mapeados.

Depois disso ele puxa e executa as imagens no docker e lista elas para ver quais containers estão em execução, ele entra no localhost e inicia a DAG mostrada. Após a DAG ser executada ele mostra a ordem em que as tasks foram executadas

Depois, foi feita uma alteração nesse parâmetro, mudando o valor para 3, o que permitiu que até três tarefas fossem executadas ao mesmo tempo.



Também foi ajustado outro parâmetro chamado **`max_active_tasks_per_dag`** para o valor **2**, limitando o número de tarefas simultâneas dentro de **uma mesma DAG**. Isso fez

com que, mesmo com o paralelismo global em 3, apenas 2 tasks rodassem ao mesmo tempo na DAG atual.

Paralellism: Número máximo de tarefas que o scheduler pode executar simultaneamente, globalmente. Na primeira imagem, temos o caso de um valor de paralellism igual a 1. Em seguida, o valor de paralellism igual a 3.

dag_concurrency: Número máximo de tarefas que podem ser executadas em paralelo por DAG.

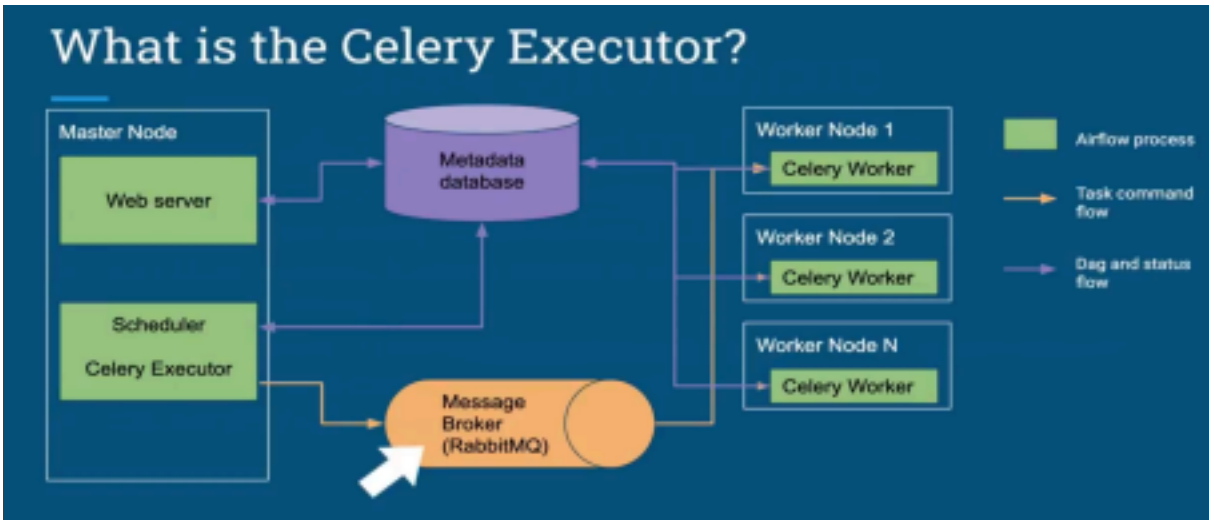
Aula 5 - Practice Ad Hoc Queries with the metadata database

Nessa parte do curso foi mostrada uma função chamada **Data Profiling**, que ficava na interface web do Airflow e permitia fazer consultas rápidas no banco de dados, ver gráficos e analisar os dados direto pelo navegador.

mais essa funcionalidade foi **excluida nas versões mais novas do Airflow**, por motivos de segurança.

Aula 6 - Scale out Apache Airflow with Celery Executors and Redis

Aqui nessa aula conhecemos o **Celery Executor**, que é usado quando queremos distribuir tarefas entre várias máquinas.

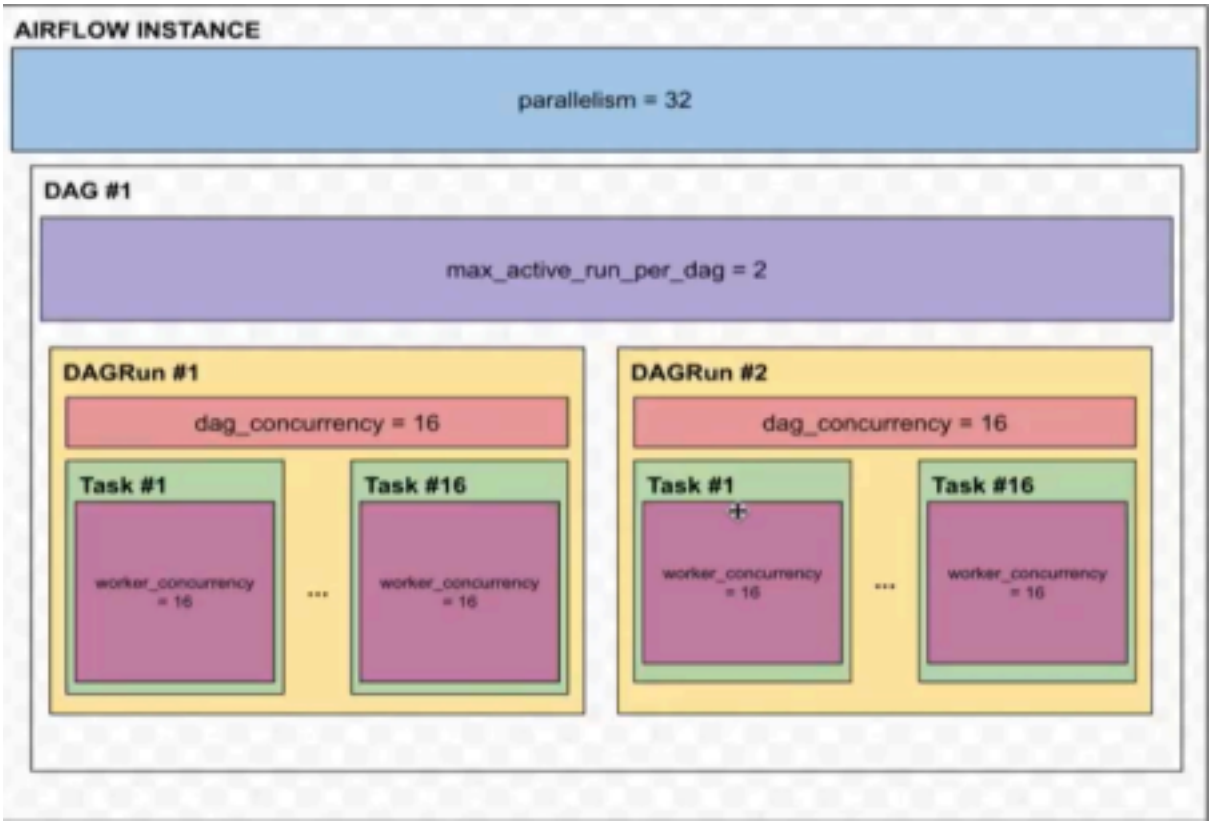


O **nó mestre** (scheduler) é quem organiza e distribui as tarefas.
message broker e quase como uma fila

Os **workers** são os que realmente rodam as tarefas, podendo estar em máquinas diferentes.

Aula 8 - Practice Distributing your tasks with the Celery Executor

Aqui estamos usando o **Flower**, que é uma ferramenta para ver o que os workers do Celery estão fazendo. Com ele, dá pra saber o nome dos workers, se estão ligados ou desligados, quantas tarefas estão rodando, quantas já foram feitas, quantas falharam, quantas deram certo e quantas foram repetidas



A imagem ilustra como os parâmetros definidos no `airflow.cfg` influenciam a execução das tarefas dentro do Airflow. O **`parallelism`** está definido como **32**, ou seja, a instância do Airflow pode executar até 32 tarefas simultaneamente no total. O parâmetro **`max_active_runs_per_dag`** está configurado como **2**, permitindo no máximo duas execuções (DAGRuns) ativas ao mesmo tempo por DAG. Já o **`dag_concurrency`**, definido como **16**, limita o número máximo de tarefas simultâneas por execução da DAG (DAGRun). Essa mistura ajuda a usar os recursos certinho e a organizar quando várias coisas vão rodar ao mesmo tempo.

Aula 9 - Practice Adding new worker nodes with the Celery Executor

```
Marc@MacBook-Pro ~/airflow-materials/airflow-section-5 (master) $ docker exec -
it 961e45f1f05d /bin/bash
root@961e45f1f05d:/# export AIRFLOW_HOME=/usr/local/airflow/
root@961e45f1f05d:/# useradd -ms /bin/bash -d $AIRFLOW_HOME airflow
useradd: warning: the home directory already exists.
Not copying any file from skel directory into it.
root@961e45f1f05d:/# pip install "apache-airflow[celery, crypto, postgres, redi
```

Foram usados alguns comandos para entrar dentro do container. Depois de acessar, os arquivos que estavam na pasta `/local/airflow` foram exportados. E depois, foi criado um novo usuário dentro do container e foi feita a instalação do Apache Airflow, na versão 1.10.6.

Após ele inicializar o airflow dentro do container ele entra com o usuário e da um ls para listar os arquivos

```
root@961e45f1f05d:/# chown -R airflow: $AIRFLOW_HOME
root@961e45f1f05d:/# su - airflow
airflow@961e45f1f05d:~$ ls
airflow.cfg  airflow.db  dags  logs  unittests.cfg
airflow@961e45f1f05d:~$
```

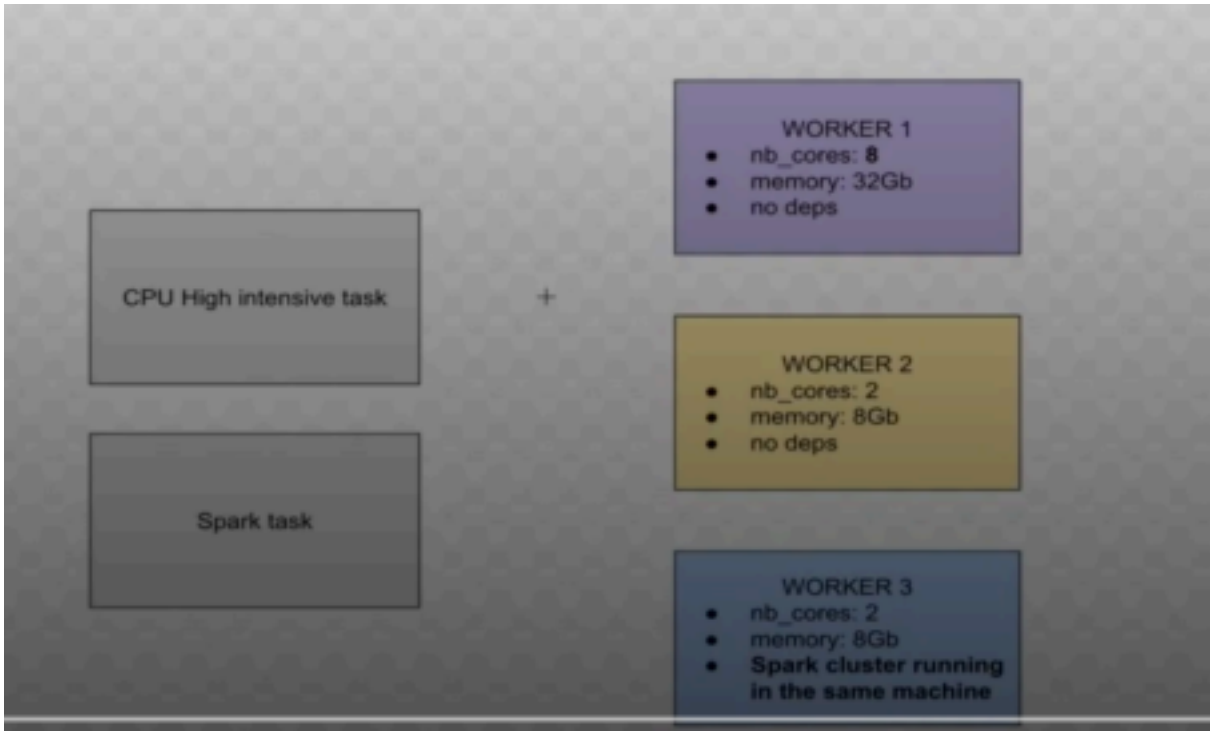
Após concluir todo esse processo, um novo worker é iniciado dentro do container. Na interface do Flower, é possível visualizar que agora há dois workers ativos.

Em seguida, ele inicia uma DAG e mostra que os dois workers estão ativos e executando tarefas. Esse processo é chamado de escalonamento horizontal. quanto mais máquinas forem adicionadas, maior será a capacidade de processamento disponível para concluir as tasks e, consequentemente, as DAGs.

Aqui ele demonstra o que acontece ao desligar o container enquanto um worker está executando uma task ativa. isso ocorre um erro.

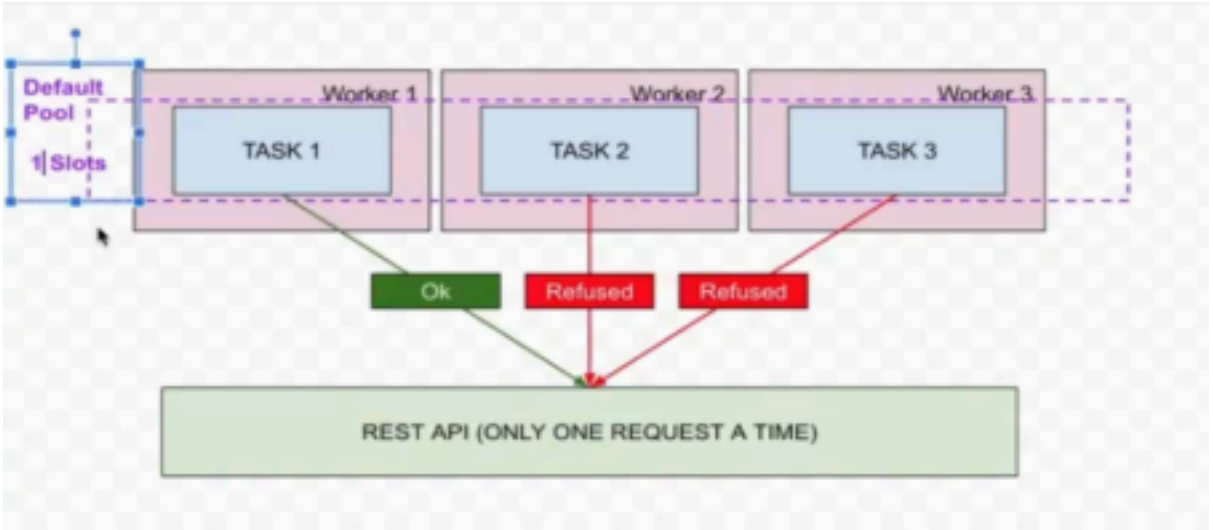
Essa task corresponde a task ativa no worker que falhou, dentro da interface do airflow é mostrado que essa task é vista como sucedida, porém ela nunca teve um resultado de saída.

Aula 10 - Practice Sending tasks to a specific worker with Queues



Aqui nesta aula é falado que algumas tasks precisam ser executadas por workers específicos. Para lidar com isso, é necessário direcionar cada task para uma fila dedicada. Dessa forma, o worker apropriado irá processar as tarefas de forma organizada.

Aula 11 - Practice Pools and priority_weights Limiting parallelism - prioritizing tasks



O diagrama acima demostra três tasks precisam acessar uma mesma API, que só permite uma chamada por vez. Isso faz com que apenas a primeira task seja executada corretamente, enquanto as demais falham por sobrecarga. Para contornar essa limitação, é

possível configurar o número de pools como 1. Com isso, as tasks passam a ser executadas de forma sequencial, respeitando a capacidade da API e garantindo que todas sejam processadas com sucesso.

Aula 12 - Kubernetes reminder

O **Kubernetes** é uma ferramenta criada pela Google em 2014 para ajudar a gerenciar containers de forma mais prática e automática. Ele cuida de tarefas como atualizações, segurança e organização dos servidores, o que acaba economizando tempo e trabalho.

A arquitetura de um **cluster Kubernetes** é composta por duas partes principais: o Control Plane e os Worker Nodes.

O Control Plane é tipo o cérebro do sistema. Ele escolhe o que fazer, como marcar tarefas ou resolver problemas

Os Worker Nodes são as máquinas que realmente executam os containers. Eles recebem as instruções do Control Plane

Aula 13 - Scaling Airflow with Kubernetes Executors:

Para contextualizar a escolha do Kubernetes Executor, é importante entender as limitações dos executores já conhecidos. O Celery Executor, por exemplo, é bastante eficiente, mas exige uma configuração mais complexa:

1 - É necessário configurar filas específicas, como o RabbitMQ,

2 - Utilizar ferramentas como o Flower para monitorar o comportamento dos workers,

3 - E gerenciar manualmente as dependências das tasks em cada worker.

Outro problema é que, quando um novo worker é iniciado no Celery, ele já começa a usar recursos da máquina, mesmo sem estar fazendo nada. Isso pode deixar o sistema mais lento..

Já o **Kubernetes Executor** funciona diferente. Cada task é executada em um pod separado dentro do cluster Kubernetes.

Aula 14 - [Practice] Set up a 3 nodes Kubernetes Cluster with Vagrant and Rancher:

No começo da aula, foram apresentados os programas necessários para montar o ambiente de testes. Um deles é o **VirtualBox**, que serve para criar máquinas virtuais. O outro é o **Vagrant**, que ajuda a automatizar a criação e configuração dessas máquinas Também foi instalado um plugin que permite transferir arquivos entre o computador principal (host) e a máquina virtual.

Depois que tudo foi instalado, usamos o comando **`vagrant ssh master`** para entrar na máquina virtual principal pelo terminal. Com isso, conseguimos rodar comandos direto lá dentro. Um dos primeiros comandos executados foi **`kubectl get nodes`,** que serve para ver os nós ativos do cluster Kubernetes, e checar se eles estão prontos pra receber tarefas.

Em seguida, foi iniciado o **Rancher**, uma ferramenta que ajuda a gerenciar clusters Kubernetes usando uma interface gráfica. Isso foi feito com o comando:

docker run -d --restart=unless-stopped -p 80:80 -p 443:443 --name rancher rancher/rancher:v2.3.3

Aula 15 - [Practice] Installing Airflow with Rancher and the Kubernetes Executor:

Depois de configurar o **Rancher**, que serve para ver e gerenciar os clusters Kubernetes de forma visual, o próximo passo foi instalar o Apache Airflow usando o Kubernetes Executor.

Para isso, foi preciso criar um catálogo dentro do Rancher, usando o link de um repositório do GitHub que já vem preparado com tudo o que o Airflow precisa para funcionar. Esse repositório inclui o banco de dados, o scheduler e o web server. Depois que o catálogo é adicionado, todos os componentes do Airflow aparecem direto na interface gráfica do Rancher. Isso facilita bastante o acompanhamento e o gerenciamento dos workflows.

Aula 16 - [Practice] Running your DAGs with Kubernetes Executor:

Para rodar os **DAGs usando o Kubernetes Executor** no Airflow, o primeiro passo foi acessar a interface do Airflow, que está rodando dentro de um pod no cluster Kubernetes.

Pra isso, abrimos o terminal e usamos o comando **vagrant ssh master** para entrar na máquina virtual principal. Depois, foi usado um comando que serve para descobrir o nome do pod que está rodando:

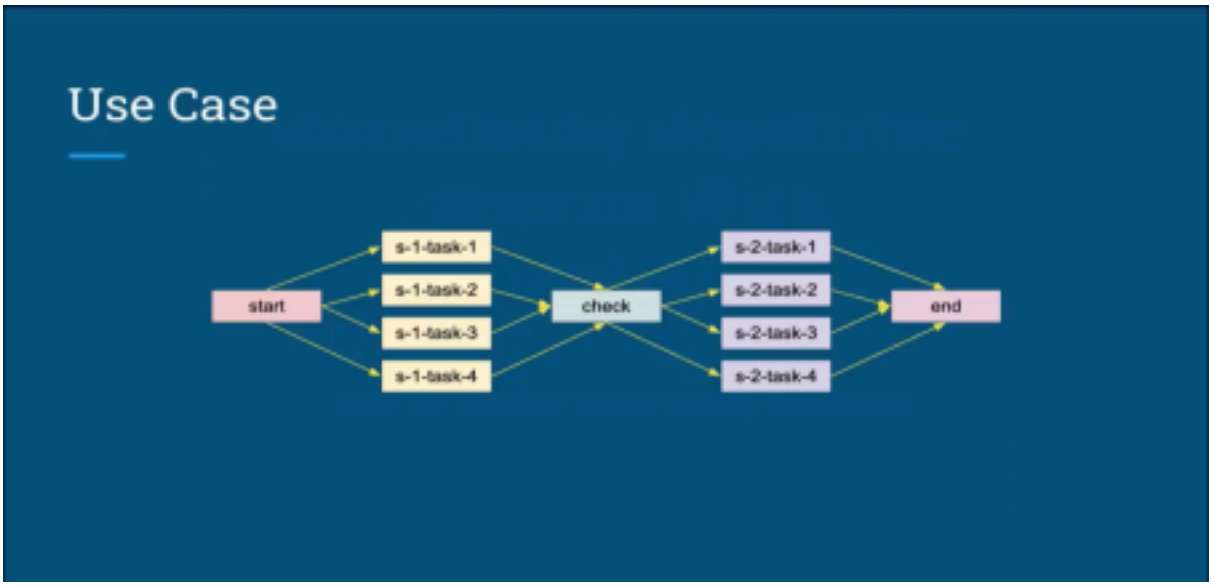
export POD_NAME=$(kubectl get pods --namespace airflow-p-xccpd -l "component=web,app=airflow-k8s" -o jsonpath="{.items[0].metadata.name}")

Depois de definir essa variável, usamos echo $POD_NAME pra mostrar o nome do pod.

Seção 6: Improving your DAGs with advanced concepts

Aula 2 - Minimizing Repetitive Patterns with SubDAGs:

As **SubDAGs** servem para dividir uma DAG grande em partes menores, que podem ser reaproveitadas em outras situações. É como se fosse uma DAG dentro de outra DAG. A ideia é deixar os workflows mais organizados,



Se utilizarmos SubDags, ficaria assim:

Para tudo funcionar bem as **SubDags necessitam ter**:

Ter o mesmo schedule_interval e start_date que a DAG principal.

Ter default_args que respeita a DAG principal.

Usar o mesmo executor (ex: Local, Celery).

Ter um valor adequado de max_active_runs.

Aula 3 - [Practice] Grouping your tasks with SubDAGs and Deadlocks:

Esse diagrama mostra que as SubDAGs se comportam como tasks. O problema é que, se todas as SubDAGs estiverem na fila, as tasks que estão dentro delas não conseguem rodar, porque também precisam de espaço na fila.

Aula 4 - Making different paths in your DAGs with Branching:

Branching: É uma forma que permite as DAGs escolherem diferentes caminhos a serem seguidos de acordo com o resultado de uma task específica. A imagem abaixo demonstra um caso onde uma DAG verifica 3 APIs diferentes, e dependendo se elas estiverem disponíveis ou não, ele tomará a decisão automaticamente de qual escolher.

Aula 5 - [Practice] Make your first conditional task using branching:

Nessa imagem acima conseguimos ver um exemplo prático de Branching, onde apenas uma das task foram executadas.

É mostrado que a task pula as outras e executa apenas a 'ip-api', porque no código ele só deveria retornar essa task. Mudando dentro do código para retornar o 'ipinfo'. Dai sim é retornado a ipinfo

Aula 6 - Trigger rules for your tasks

No Airflow, as tarefas seguem uma ordem. só rodam se a anterior tiver dado certo. Isso evita erros no processo. No exemplo abaixo, os dados vêm de dois sites e depois são juntados. Se essa junção falhar, o sistema manda um e-mail avisando.

Aula 9 - [Practicrade] Templating your tasks

Na prática da aula 8, aprendemos sobre templates nas tasks, que permitem usar variáveis nos comandos. Um exemplo é o **`{{ ds }}`**, que mostra a data da execução. Também dá pra criar variáveis personalizadas, como a de exemplo do curso : `CASSANDRA_LOGIN` com o valor `my_login`. Depois que a DAG roda, o log completo fica salvo nesse caminho, e dá pra ver tudo que aconteceu.

Aula 10 - How to share Data Between Your Tasks with XCOM:

O **XCom** é usado no Airflow pra trocar dados entre tasks. Serve, por exemplo, pra passar o resultado de uma task pra outra dentro da mesma DAG. Pra isso, usamos `xcom_push()` pra enviar e `xcom_pull()` pra pegar os dados.

Aula 12 - TriggerDagRunOperator

O **`TriggerDagRunOperator`** serve pra uma DAG chamar outra automaticamente. Isso ajuda a dividir o processo em partes menores, deixando o código mais organizado e fácil de cuidar.

Aula 14 - Dependencies between your DAGs with the ExternalTaskSensor

O **`ExternalTaskSensor`** é usado quando uma DAG precisa esperar uma task de outra DAG terminar. Ele cria essa dependência entre DAGs pra garantir que tudo aconteça na certo.

Aula 15 - [Practice] Make your DAGs dependent with the ExternalTaskSensor:

No editor de texto do Airflow, é possível definir qual DAG terá uma dependência e qual task precisa ser executada antes. Isso é feito diretamente no código, onde se especifica que a DAG atual só deve iniciar depois que a task t2 da DAG chamada sleep_dag for concluída. Essa configuração garante que o fluxo de trabalho respeite a ordem correta de execução entre diferentes DAGs, evitando que uma DAG comece antes da hora. **Secao 7- Deploying Airflow on AWS EKS with Kubernetes Executors and Rancher Aula 2 - Quick overview of AWS EKS:**

O AWS EKS é da Amazon e serve pra rodar e arrumar os containers. Ele faz tudo sozinho e ainda dá pra usar com outros serviços da AWS.

Aula 3 - [Practice] Set up an EC2 instance for Rancher

Aprendi como criar uma instância EC2 na AWS e acessar ela de formas diferentes. Dá pra usar o navegador direto, sem precisar de outros programas. Assim, dá pra instalar e configurar tudo por ali mesmo.

Aula 4 - [Practice] Create an IAM user with permissions:

Para criar um novo usuário com permissões específicas, é necessário acessar o serviço IAM diretamente pela plataforma da AWS. É por meio desse serviço que se gerenciam os usuários, grupos e políticas de acesso aos recursos da nuvem.

Aula 5 - [Practice] Create an ECR repository:

Agora vamos utilizar o ECR um serviço da AWS que permite criar repositórios para armazenar e gerenciar imagens de containers Docker.

Depois que criei a instância EC2 e instalei a AWS CLI, também configurei as credenciais do usuário IAM. Com isso criei uma imagem Docker chamada airflow usando o Dockerfile.

Esse comando diz ao Docker para construir uma imagem chamada **`airflow`** usando

o arquivo **`Dockerfile`**

Depois manda essa imagem pro repositório da AWS (ECR) usando o comando **docker push**.Quando tudo é enviado, ela fica disponível no repositório pra usar nos
projetos.

docker push 953076219999.dkr.ecr.eu-west-3.amazonaws.com/airflow:v1.0

Aula 6 - [Practice] Create an EKS cluster with Rancher:

Diferentemente do que foi feito na seção anterior, Dessa vez em vez de importar um cluster, se criai um novo direto no Amazon EKS usando as credenciais do usuário IAM. Depois de configurar, é só esperar alguns minutos até o cluster ficar feito.

Quando ele aparece no Rancher, dá pra ver tudo pela interface. possível ver todos os pods que estão rodando e seus nomes e a quanto tempo tambem.

Aula 8 - [Practice] Deploy Nginx Ingress with Catalogs (Helm):

Instalei o **Nginx** usando a interface do Rancher depois tem que ativar o catálogo Helm, depois os procurar pela instalação e instalar e após é só fazer algumas configurações e pronto.

Aula 9 - [Practice] Deploy and run Airflow with the Kubernetes Executor on EKS:

Nessa aula adicionei um catálogo chamado **`airflow eks`** no Rancher. Depois disso, o Nginx e o Airflow apareceram na interface. Aí foi só esperar o banco de dados, o scheduler e o webserver do Airflowiniciar. Depois consegui acessar Airflow pela porta 80 do Nginx.

Aula 10 - [Practice] Cleaning your AWS services:

Nessa aula foi aprendido como desligar tudo o que foi usado na AWS, pra não ficar gerando custo sem precisar. A AWS cobra por tempo. Também vi como usar o Cost Explorer e o Billing pra conferir se ainda tem algo ativo.

8. Monitoring Apache Airflow

Aula 2 - How the logging system works in Airflow:

Os logs do Airflow usam a mesma base do sistema de logs do Python. Dá pra mudar o formato e escolher o que mostrar e pra onde mandar. Isso ajuda a deixar os registros certos.

Aula 3 - [Practice] Setting up custom logging:

Aqui aprendi como mudar o jeito que os logs do Airflow funcionam. Algumas coisas dá pra ajustar direto no arquivo **`airflow.cfg`**, como o formato do log. Mas se quiser algo mais avançado, dá pra criar um arquivo Python só pra isso,

Aula 4 - [Practice] Storing your logs in AWS S3:

Nessa aula, vi como mandar os logs do Airflow pra um bucket S3 da AWS. Primeiro tem que criar o bucket e configurar um usuário IAM com permissão só pra ler e escrever nele. Depois ajustei as configurações no airflow.cfg, ativando o **remote_logging** e informando a conexão.

remote_logging = True: ativa o registro de logs em um destino remoto.

Com essas configurações aplicadas, basta iniciar a execução da DAG e no **Graph View**, ao acessar os detalhes de uma task da de se o log foi armazenado com sucesso no bucket S3.

Aula 5 - Elasticsearch Reminder:

O Elasticsearch é uma ferramenta pra guarda e organiza grandes volumes de dados. Ele é bom pra armazenar logs e ajuda também a criação de gráficos e análises, mostrando o que está acontecendo no sistema.

Aula 6 - [Practice] Configuring Airflow with Elasticsearch:

Nessa aula vai ser feita a configuração do Airflow juntamente do Elasticsearch, a arquitetura a ser utilizada será essa:

Aqui eu aprendi a configurar o Airflow pra mandar os logs pro Elasticsearch. Fiz algumas mudanças no arquivo airflow.cfg e deixei os logs no formato JSON, que é mais fácil de ler e pesquisar.

Depois disso, iniciei o contêiner do Airflow e entrei nele pelo terminal pra ajustar algumas configurações. Quando ajustei tudo e só rodei uma DAG e consegue ver os logs pela interface do Kibana.

Aula 7 - [Practice] Monitoring your DAG's with Elasticsearch

Primeiro é limando o histórico das tasks no Airflow e também apagado os arquivos de log e o index criado no Kibana. Com tudo isso zerado é so rodar a DAG chamada data_dag pra gerar novos dados.

Depois disso se cria um novo index pattern e filtrei os logs por hora. Dai da pra montar um dashboard no Kibana com gráficos que mostram, por exemplo, quantas tasks falharam. Aqui está mostrando a quantidade de falhas dentro de uma determinada DAG.

Aula 8 - Introduction to metrics:

Nessa aula aprendemos sobre StatsD que serve pra ajudar no envio de algumas metricas de forma leve. Ele pega os dados e organiza todos antes de mandar, e o legal que isso ajuda a nao sobrecarregar o sistema.
A TIG Stack é um jeito forte e completo de cuidar e ver como os sistemas estão funcionando. O Telegraf atua como agente de coleta, capturando métricas e logs de diversas fontes em tempo real. Esses dados são então armazenados de forma eficiente no InfluxDB, um banco de dados especializado em séries temporais.

Aula 9 - [Practice] Monitoring Airflow with TIG stack

Pra usar a **TIG Stack**, o primeiro passo é configurar o Telegraf e editar o arquivo **telegraf.conf** e tambem tem que definir os plugins certos pra ele mandar os dados pro InfluxDB. Depois, dentro da interface, criar uma conexão com o InfluxDB.

9. Security in Apache Airflow:

Aula 2 - [Practice] Encrypting sensitive data with Fernet

Começa se criando uma conexão pelo Airflow pra acessar um banco externo. Quando olhamos os dados salvos no terminal, podemos ver que não estavam criptografados, oque dependendo pode ser perigoso.

Então tem que ativar o modo seguro no airflow.cfg e coloca se **secure_mode = True**, e depois configura a criptografia com o Fernet. Gerai uma chave com o comando e se coloca ela no fernet_key e, com isso, o Airflow começa a proteger senhas e tokens automaticamente.

Essa configuração melhora muito a segurança dos dados que o Airflow usa. **Aula 3 -**

[Practice] Rotating the Fernet Key:

Para maior segurança é explicado que a melhor maneira de usar o Fernet é alterar de tempos em tempos a sua chave, e vai ser ensinado a fazer isso sem que se perca acesso às informações encriptadas. Dentro do container do webserver ele executa o comando que gera uma Fernet Key:

E ele copia essa nova chave para o parâmetro ‘fernet_key’ antes da chave já existente e separa ambas as chaves por uma vírgula. E com apenas um comando dentro do terminal já é possível que a rotação seja feita:

A rotação está pronta e a criptografia já mudou. É possível apagar a chave antiga do airflow.cfg. E como recomendado para maior segurança não se deve deixar no airflow.cfg, o mais adequado é deixar em uma variável de ambiente dentro do docker compose. **Aula 4 - [Practice] Hiding variables:**

No Airflow, dá pra criar variáveis personalizadas que podem ser usadas nas DAGs. Isso é bom pra deixar o código melhor. É só ir em “Admin” -> “Variables” e cadastrar o nome e valor da variável.

O problema é que essas variáveis não ficam escondidas por padrão. Qualquer pessoa que tiver acesso à interface pode ver. Pra resolver precisamos usar nomes com palavras como password, secret, token ou key. O Airflow entende que esses valores são sensíveis e aplica

proteção extra..
Como a chave tem ‘password’ escrito nela o valor dela é automaticamente encriptado. Lembrando que dentro do banco de dados ambos estão encriptados com o Fernet, porém na interface só um aparece.

Aula 5 - [Practice] Password authentication and filter by owner:

Na etapa de configuração de autenticação por senha no Airflow, o primeiro passo é alterar o parâmetro **authenticate** no **arquivo airflow.cfg**. Por padrão, esse parâmetro pode estar desativado, permitindo acesso irrestrito à interface web. Ao definir **authenticate = True**, o Airflow passa a exigir autenticação dos usuários antes de permitir o acesso à interface, adicionando uma camada essencial de segurança.

Aula 6 - [Practice] RBAC UI:

Para organizar os usuários por categorias e definir permissões específicas dentro do Airflow, é necessário ativar o controle baseado em funções (RBAC – Role-Based Access
Control). O primeiro passo é alterar o parâmetro **`rbac`** no arquivo **`airflow.cfg`,** definindo-o como **`True`**. Essa configuração habilita uma interface de administração mais avançada, permitindo a criação de grupos de usuários com diferentes níveis de acesso e controle sobre as funcionalidades da plataforma.

Agora tem uma aba para segurança que pode ser listado todos os usuários e suas ‘roles’, podendo alterar em tudo. Tem uma aba para o próprio usuário no canto superior direito e entre outras mudanças dando acesso total ao administrador.

Insight visual original: Vou liberar o PDF do insight visual original no Git - Hub para melhor visualização.

,

Conclusão: Esse card me ajudou a entender melhor como o Airflow funciona. Aprendi sobre os tipos de executores e como configurar o paralelismo das tasks. Também usei coisas como filas, pools, SubDAGs, XCom e branching pra organizar melhor os fluxos.

Outra parte importante foi aprender a fazer o deploy do Airflow na nuvem com Docker, Rancher e AWS. E ainda vi como monitorar e proteger as DAGs usando S3 e Fernet.

Referências:

Card 18 - Bootcamp (Lamia)