Planejamento e Execução do Pipeline de Dados

Este documento detalha a implementação do pipeline de dados para o teste da AB InBev, cobrindo o planejamento, as decisões tomadas durante a execução e os resultados alcançados. O objetivo principal foi orquestrar as camadas Bronze, Silver e Gold no Databricks, com um plano alternativo para Dockerização que não foi viável devido a limitações da infraestrutura local.

---

1. Planejamento

1.1. Objetivo

Construir um pipeline de dados em três camadas:

- **Bronze**: Ingestão e armazenamento dos dados brutos.
- **Silver**: Limpeza e transformações intermediárias.
- **Gold**: Dados refinados prontos para consumo analítico.

1.2. Estrutura e Ferramentas

1. **Databricks**:
    - Orquestração e processamento do pipeline.
    - Manipulação de dados usando Spark.
2. **Azure Container Registry (ACR)**:
    - Planejado para armazenamento de imagens Docker.
3. **GitHub Actions**:
    - Configurado para automatizar a criação e envio de imagens Docker.
4. **Azure Resources**:
    - Criamos recursos essenciais para o projeto, como:
        - **Azure Databricks**: Para execução do pipeline.
        - **Azure Storage Account**: Para armazenar os dados das camadas Bronze, Silver e Gold.
        - **Azure Container Registry (ACR)**: Para gerenciar imagens Docker (planejado).

1.3. Configuração dos Recursos do Azure

Azure Databricks

1. Criamos um workspace no Azure Databricks.
2. Configuramos clusters com os seguintes parâmetros:
    - Tipo de nó: Standard_DS3_v2.

- Número de nós: 2 (auto-scaling habilitado).

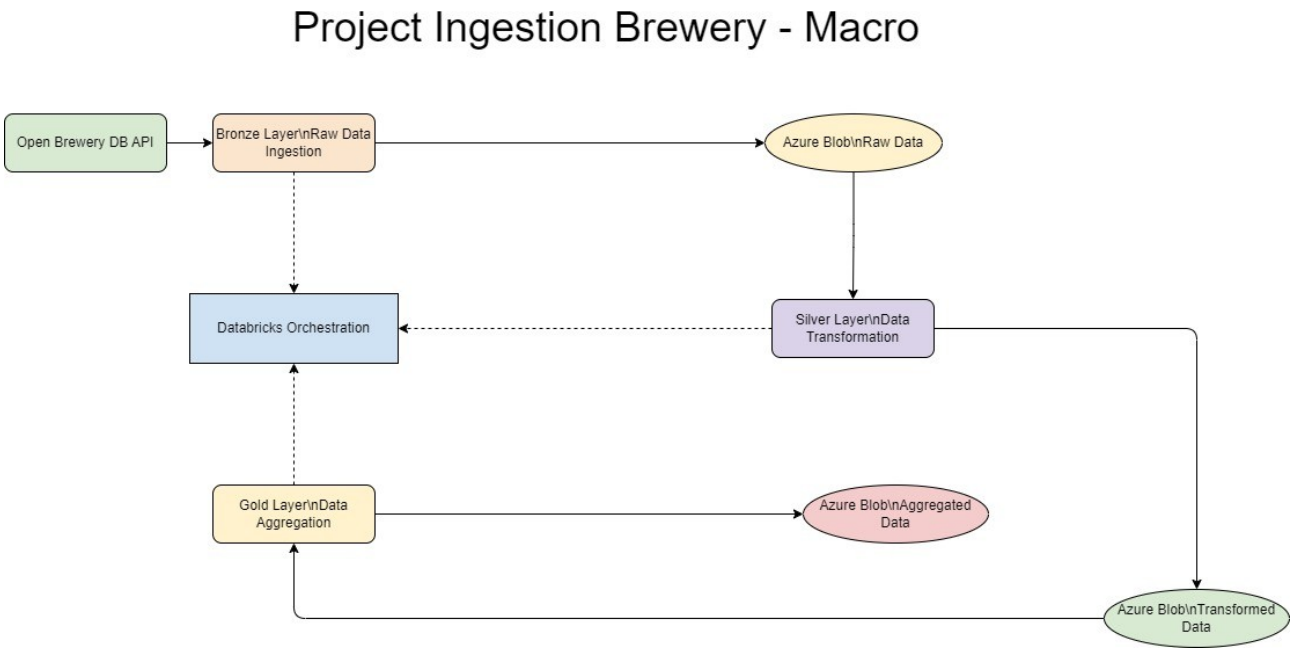
Azure Storage Account

1. Configuramos uma Storage Account para armazenar os dados do pipeline.
2. Criamos três containers correspondentes às camadas:
    - **bronze**: Para dados brutos.
    - **silver**: Para dados transformados intermediários.
    - **gold**: Para dados finais.

Azure Container Registry (ACR)

1. Criamos o registro `inbevacr`.
2. Habilitamos o usuário administrador e configuramos as credenciais para integração com o GitHub Actions.

1.4. Diagrama Macro



---

2. Motivo para Não Usar Dockerização

Originalmente, planejamos criar e orquestrar o pipeline utilizando imagens Docker. No entanto, minha máquina local não possui recursos suficientes para construir e testar imagens Docker adequadamente. Apesar disso, o workflow para Dockerização foi configurado no GitHub Actions, permitindo uma abordagem automatizada caso seja possível executar em outro ambiente.

As principais limitações encontradas foram:

- **Infraestrutura Local**: Recursos insuficientes para compilar imagens Docker.
- **Restrições de Rede**: Problemas ocasionais de conectividade para autenticação no Azure.

Como alternativa, optamos por implementar a orquestração diretamente no Databricks, o que garantiu maior confiabilidade e simplicidade na execução.

Pré-preparação de Dockerfiles: Apesar de não termos executado a dockerização completa, deixamos pré-configurados os `Dockerfiles` para as camadas Bronze e Silver. A camada Gold utilizaria o mesmo `Dockerfile` da Silver devido à similaridade de requisitos. A ideia inicial era que, após a criação dessas imagens, a orquestração seria realizada pelo **Azure Data Factory**, integrando os fluxos de execução entre as imagens.

Exemplo de Dockerfile da Bronze

```
FROM bitnami/spark:3.2.0

WORKDIR /app

# Copiando o script da camada Bronze
COPY bronze.py /app/bronze.py

# Executando o script
CMD ["spark-submit", "/app/bronze.py"]
```

Exemplo de Dockerfile da Silver

```
FROM bitnami/spark:3.2.0

WORKDIR /app

# Copiando o script da camada Silver
COPY silver.py /app/silver.py

# Executando o script
CMD ["spark-submit", "/app/silver.py"]
```

Esses arquivos foram planejados para integração futura, caso o ambiente permita a execução com Docker.

---

3. Implementação do Pipeline

3.1. Estrutura do Projeto

O projeto foi estruturado em três camadas:

- **Bronze**: Responsável por ingestão e padronização inicial dos dados.
- **Silver**: Aplica regras de limpeza e transformação.
- **Gold**: Consolida os dados para consumo analítico.

3.2. Comentários sobre os Scripts

Camada Bronze

O script da camada Bronze realiza a ingestão dos dados brutos a partir de uma API pública e os armazena em um container no Azure Blob Storage. Pontos importantes:

1. **Ingestão via API**:

- Utiliza `requests` para consumir os dados da API `https://api.openbrewerydb.org/breweries`.
- Garante que a resposta seja bem-sucedida com `response.raise_for_status()`.

2. **Estruturação de Dados**:
   - Os dados são armazenados em um formato JSON organizado, com timestamp para identificar cada ingestão.
3. **Upload para o Azure Blob Storage**:
   - O script utiliza a biblioteca `azure.storage.blob` para fazer upload do JSON diretamente para o container `bronze` no Blob Storage.

Camada Silver

O script da camada Silver processa os dados brutos da Bronze e os transforma em um formato parquet mais estruturado, aplicando partições. Pontos importantes:

1. **Leitura dos Dados da Bronze**:
   - O script identifica as pastas no container `bronze` e faz download dos arquivos JSON.
   - Converte os arquivos JSON em um DataFrame Spark.
2. **Transformações**:
   - Valida a existência de colunas importantes, como `state`, lançando exceções caso estejam ausentes.
   - Particiona os dados pela coluna `state`, o que melhora a organização e performance das consultas.
3. **Escrita na Camada Silver**:
   - Os dados transformados são gravados no container `silver` em formato parquet, com partições otimizadas.

Camada Gold

O script da camada Gold realiza agregações nos dados da Silver para gerar informações consolidadas. Pontos importantes:

1. **Leitura dos Dados da Silver**:
   - Faz a leitura de todos os arquivos parquet armazenados na camada Silver.
2. **Agregações**:
   - Calcula métricas como média de latitude, máximo e mínimo de longitude, e contagem total de entradas, agrupando por `brewery_type` e `state_province`.
3. **Escrita na Camada Gold**:

- Os resultados consolidados são armazenados no container `gold`, organizados em arquivos parquet.

---

4. Configuração do Docker e GitHub Actions

Embora não tenhamos usado Docker, deixamos configurado o processo no GitHub Actions para demonstração.

Workflow do GitHub Actions:

```
name: Build and Push Docker Images

on:
  push:
    branches:
      - main

jobs:
  build-and-push:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout repository
      uses: actions/checkout@v2

    - name: Log in to Azure Container Registry
      run: echo "${{ secrets.AZURE_REGISTRY_PASSWORD }}" | docker login $
{{ secrets.AZURE_REGISTRY_NAME }}.azurecr.io -u $
{{ secrets.AZURE_REGISTRY_USERNAME }} --password-stdin

    - name: Build and push Bronze image
      run: |
        docker build -t ${{ secrets.AZURE_REGISTRY_NAME }}.azurecr.io/bronze-
layer-image:v1 -f ./bronze/Dockerfile ./bronze
        docker push ${{ secrets.AZURE_REGISTRY_NAME }}.azurecr.io/bronze-layer-
image:v1

    - name: Build and push Silver image
      run: |
        docker build -t ${{ secrets.AZURE_REGISTRY_NAME }}.azurecr.io/silver-
layer-image:v1 -f ./silver/Dockerfile ./silver
        docker push ${{ secrets.AZURE_REGISTRY_NAME }}.azurecr.io/silver-layer-
image:v1

    - name: Build and push Gold image
      run: |
        docker build -t ${{ secrets.AZURE_REGISTRY_NAME }}.azurecr.io/gold-
layer-image:v1 -f ./gold/Dockerfile ./gold
        docker push ${{ secrets.AZURE_REGISTRY_NAME }}.azurecr.io/gold-layer-
image:v1
```

---

5 Detalhamento das explicações dos scripts bronze / silver /gold

5.1 Camada Bronze

Objetivo da Camada Bronze

A camada Bronze é responsável pela ingestão dos dados brutos diretamente da API de origem e pelo armazenamento inicial no Azure Blob Storage. Nesta etapa, os dados são padronizados em formato JSON e enriquecidos com metadados, como o timestamp da ingestão.

1) Fluxo de Trabalho da Camada Bronze

Configuração do Ambiente

- Bibliotecas Utilizadas:

    - `requests`: Para consumir a API.

    - `azure.storage.blob`: Para interação com o Azure Blob Storage.

    - `logging`: Para registro de logs.

    - `datetime`: Para geração de timestamps.

    - `json`: Para manipulação do formato JSON.

- Credenciais do Azure Blob Storage:

    - As credenciais de acesso à conta do Azure são configuradas diretamente no script para propósitos de teste.

2. Consumo de Dados da API

O script consome os dados da API `https://api.openbrewerydb.org/breweries` utilizando a função `fetch_data_from_api`. Esta função realiza uma requisição HTTP GET, valida o status da resposta e retorna os dados em formato JSON.

Detalhes do Processo:

- Validação:

    - O script utiliza `response.raise_for_status()` para garantir que erros HTTP são tratados imediatamente.

3. Criação do Caminho para o Armazenamento

- Os dados são organizados em uma hierarquia de pastas baseada no timestamp de ingestão, seguindo o formato `YYYYMMDDHHMM`.

- Um arquivo JSON é criado para cada ingestão, contendo o timestamp no nome do arquivo.

4. Upload para o Azure Blob Storage

- Os dados JSON são enviados para o container configurado no Azure Blob Storage utilizando a função `upload_to_blob`.

- O caminho do blob segue o padrão: `bronze/YYYYMMDDHHMM/YYYYMMDDHHMM.json`.

- O upload é realizado com a opção `overwrite=True` para garantir que o arquivo seja substituído em caso de reprocessamento.

5. Registra Logs

O script utiliza a biblioteca `logging` para registrar informações importantes durante a execução:

- Informações sobre a inicialização e finalização do upload.

- Tratamento de erros em caso de falha na requisição HTTP ou upload.

Estrutura do Código

Funções Principais

1. `fetch_data_from_api(api_url)`:

   - Realiza a requisição HTTP GET.

   - Retorna os dados em formato JSON.

2. `upload_to_blob(data, container_name, blob_name)`:

   - Conecta ao Azure Blob Storage.

   - Realiza o upload dos dados para o container especificado.

3. `main()`:

   - Orquestra as etapas do processo: consumo da API, criação do caminho e upload.

   - Adiciona timestamps para organização dos dados.

Exemplo de Estrutura no Blob Storage

Container: `abinbev`

Caminho de Armazenamento**:

bronze/

  |- 202312071230/

      |- 202312071230.json

Benefícios da Implementação

1. Automatização: O script automatiza a ingestão de dados e o armazenamento no Blob Storage.

2. Organização: Dados são armazenados com uma estrutura hierárquica baseada no timestamp.

3. Flexibilidade: A função `upload_to_blob` pode ser reutilizada para outros containers e propósitos.

4. Confiabilidade: Logs detalhados garantem que problemas sejam identificados rapidamente.

Melhorias Futuras

1. Segurança:

   - Utilizar variáveis de ambiente para armazenar as credenciais do Azure Blob Storage em vez de defini-las diretamente no código.

2. Configuração Dinâmica:

   - Permitir que o nome do container e outras configurações sejam passados como argumentos para maior flexibilidade.

5.2 Camada Silver

A camada Silver é responsável por processar os dados brutos provenientes da camada Bronze, transformando-os em um formato estruturado e particionado, pronto para análises mais detalhadas. Nesta etapa, realizamos a validação dos dados, tratamento de erros, particionamento por colunas relevantes e armazenamento em formato Parquet no Azure Blob Storage.

Fluxo de Trabalho da Camada Silver

1. Configuração do Ambiente

- Configuração do Azure Blob Storage utilizando a biblioteca `azure.storage.blob`.
- Configuração de uma sessão Spark para leitura e escrita de dados, integrando com o Azure Blob Storage.
- Credenciais e conexão com o Azure são configuradas diretamente no script.

2. Listagem das Pastas na Camada Bronze

A função `list_bronze_folders` utiliza o cliente do Azure Blob Storage para listar todas as pastas presentes na camada Bronze. Cada pasta representa um conjunto de dados a ser processado.

3. Processamento dos Dados da Bronze para a Silver

A função `process_bronze_to_silver` realiza o processamento de cada pasta da camada Bronze:
- Faz o download dos arquivos JSON da camada Bronze.
- Valida e transforma os dados em um DataFrame Spark.
- Particiona os dados pela coluna `state`.
- Armazena os dados transformados na camada Silver em formato Parquet.

Durante o processamento, as seguintes validações são realizadas:
- Verifica se os dados não estão vazios.
- Garante que a coluna `state` está presente.
- Trata erros de JSON mal formatado ou registros inválidos.

4. Estrutura de Armazenamento

Os dados processados são armazenados na camada Silver seguindo a estrutura hierárquica abaixo:

```
silver/
  |- YYYYMMDDHHMM/
      |- arquivo.parquet
```

Cada pasta na camada Silver corresponde a uma pasta processada da Bronze, garantindo organização e particionamento.

5. Registro de Logs

O script utiliza `print` para registrar logs durante a execução. Isso inclui:
- Listagem de pastas encontradas na Bronze.
- Indicação do progresso ao processar cada pasta.
- Erros encontrados durante o processamento.

Benefícios da Implementação

1. **Estruturação dos Dados**: Transforma dados brutos em um formato estruturado e pronto para análise.
2. **Validação de Qualidade**: Garante que os dados processados atendem aos requisitos mínimos de consistência.
3. **Particionamento**: Otimiza consultas futuras ao particionar os dados por colunas relevantes.
4. **Confiabilidade**: Identifica e trata erros de dados, garantindo integridade no pipeline.

Melhorias Futuras

1. **Substituição de Logs**: Implementar uma solução de logging estruturado usando a biblioteca `logging` para maior transparência e rastreabilidade.
2. **Automatização de Configurações**: Utilizar variáveis de ambiente para gerenciar credenciais e configurações do Azure.

5.3 Camada Gold

Objetivo da Camada Gold

A camada Gold é responsável por consolidar e refinar os dados transformados na camada Silver, gerando tabelas analíticas otimizadas para consumo. Nesta camada, agregações, cálculos estatísticos e validações são realizadas para garantir a integridade e a qualidade das informações.

Fluxo de Trabalho

1. Configuração do Ambiente

- O Spark é configurado para operações distribuídas e leitura/escrita em Azure Blob Storage.

- As credenciais para o Azure Blob Storage são configuradas no script para integração com o serviço de armazenamento.

2. Leitura dos Dados da Camada Silver

- A função `list_silver_folders` lista as pastas da camada Silver disponíveis no Azure Blob Storage.

- A função `load_silver_data` carrega todos os arquivos `.parquet` da camada Silver e os combina em um DataFrame Spark, permitindo uma leitura eficiente e validação dos schemas.

3. Transformações e Agregações

- A função `create_gold_table` executa as seguintes operações no DataFrame Silver:

  - **Média da Latitude**: Calcula a média das coordenadas latitude por tipo de cervejaria (`brewery_type`) e estado (`state_province`).

  - **Extremos da Longitude**: Determina os valores máximo e mínimo das longitudes.

  - **Contagem de Entradas**: Conta o número total de registros agrupados por tipo de cervejaria e estado.

4. Armazenamento na Camada Gold

- A função `save_gold_data` salva os resultados processados em formato Parquet, garantindo organização e facilidade de acesso.

- Os arquivos são armazenados com nomes que incluem o timestamp da execução no container da camada Gold, por exemplo:

```

gold/latest_gold_table_output.parquet

```

5. Log de Execução

- Mensagens informativas sobre os arquivos processados, etapas realizadas e erros encontrados são exibidas no terminal, facilitando o monitoramento e depuração.

Benefícios da Implementação

1. **Análise Otimizada**:

   - Dados são agrupados e agregados para facilitar o consumo por ferramentas analíticas.

2. **Organização e Rastreabilidade**:

   - Estrutura de pastas e arquivos permite identificar facilmente a origem e o momento do processamento.

3. **Flexibilidade**:

   - Design modular permite adicionar novas métricas e dimensões analíticas.

4. **Escalabilidade**:

- Processamento distribuído com Spark suporta grandes volumes de dados.

Melhorias Futuras

1. Substituição de Logs:

   - Implementar a biblioteca `logging` para um registro de logs estruturado.

2. Segurança das Credenciais:

   - Substituir a configuração explícita das credenciais por variáveis de ambiente para maior segurança.

3. Notificações:

   - Implementar notificações para alertar sobre o sucesso ou falha no processamento.

---

6. Testes do script

Teste da Camada Bronze

Descrição: O teste valida a ingestão de dados brutos a partir de uma API pública. Ele verifica se:

- Os dados são recuperados corretamente da API.
- Os dados são armazenados no Azure Blob Storage na camada Bronze.

Principais Validações:

- Mock da API `https://api.openbrewerydb.org/breweries` para simular dados recebidos.
- Verificação de upload para o Azure Blob Storage utilizando mocks.

Código Principal:

- Recuperação de dados com `requests.get`.
- Upload para o Blob Storage com `BlobServiceClient`.

Teste da Camada Silver

Descrição: Esse teste verifica o processamento dos dados da camada Bronze para a Silver, incluindo:

- Transformação de arquivos JSON em um DataFrame Spark.
- Validação de colunas obrigatórias, como `state`.
- Escrita dos dados transformados no formato parquet com partição por estado.

Principais Validações:

- Os arquivos na Bronze são processados corretamente.
- A coluna `state` está presente em todos os registros.
- Arquivos parquet são particionados por estado.

Simulação de Teste:

- Mock dos blobs na camada Bronze.
- Simulação de erros em registros JSON inválidos.

Teste da Camada Gold

Descrição: O teste verifica o processamento da camada Silver para a Gold, validando:

- Cálculo de métricas agregadas, como média de latitude, máximo e mínimo de longitude.
- Escrita dos resultados consolidados em parquet.

Principais Validações:

- Verificação de agregações corretas por `brewery_type` e `state_province`.
- Simulação de dados na Silver para validar os resultados finais.

Simulação de Teste:

- Mock dos blobs na Silver.
- Execução do notebook da Gold no Databricks

Teste de Ponta a Ponta

Descrição: Executa o pipeline completo desde a camada Bronze até a Gold, validando os resultados na última camada. Este teste garante que:

- Cada camada seja processada corretamente.
- O pipeline funcione como esperado de ponta a ponta.

Principais Validações:

- Execução sequencial dos notebooks de Bronze, Silver e Gold.
- Validação dos resultados finais no arquivo parquet da camada Gold.

Teste de Performance

Descrição: Mede o tempo de execução para cada camada do pipeline, incluindo a ingestão de dados na Bronze e os processamentos nas Silver e Gold.

Métricas Coletadas:

- Tempo de ingestão na Bronze.
- Tempo de processamento nas camadas Silver e Gold.
- Resultados apresentados em segundos para cada camada.

Simulação de Teste:

- Geração de um conjunto de dados fictício com 1 milhão de registros.
- Execução dos notebooks de cada camada e registro do tempo de execução.

---

7 Orquestração no databricks - Implementação de Tratamento de Erros e Monitoramento da Qualidade de Dados no Pipeline

Para melhorar a robustez e a confiabilidade do pipeline de dados, implementamos os seguintes mecanismos:

7.1. Mecanismo de Retentativas para Tratamento de Erros

Um mecanismo de retentativas garante que falhas transitórias durante a execução de qualquer camada no pipeline não interrompam o processo completamente. Essa funcionalidade está encapsulada na função `run_with_retries`, que tenta reexecutar um notebook com falha até um número especificado de vezes.

Características do Mecanismo de Retentativas:

- **Número Configurável de Retentativas:** A função permite configurar um número máximo de tentativas (padrão é 3).
- **Atraso Entre as Tentativas:** Um atraso é incorporado entre as tentativas para permitir que problemas transitórios (por exemplo, conectividade) sejam resolvidos.
- **Logging:** Logs informativos para execuções bem-sucedidas, logs de aviso para tentativas falhas e logs de erro para falhas fatais.

Exemplo: Se o notebook da camada Silver encontrar uma falha, o mecanismo tentará reexecutá-lo até três vezes antes de levantar um erro.

7.2. Monitoramento da Qualidade dos Dados

Para garantir a precisão e a integridade dos dados processados em cada camada, implementamos verificacoes de validação de qualidade dos dados após o processamento de cada camada.

Características da Validação de Dados:

- **Verificações Específicas por Camada:** Cada camada pode ter uma lógica de validação personalizada, como verificar a presença de colunas críticas, checar datasets não vazios e garantir a consistência do schema.
- **Logging:** Logs informativos capturam o status do processo de validação, destacando verificações bem-sucedidas ou potenciais problemas.

Exemplo de Validação:

- Na camada Silver, a lógica de validação garante que a coluna `state_province` esteja presente e que o dataset contenha registros.
- Na camada Gold, agregados ou métricas-chave podem ser verificados para garantir consistência.

Visão Geral do Workflow

A execução do pipeline segue um workflow estruturado:

1. **Processamento da Camada Bronze:**
    - O notebook da camada Bronze é executado utilizando o mecanismo de retentativas.
    - A validação da qualidade dos dados garante que a saída dessa camada atenda às expectativas.
2. **Processamento da Camada Silver:**
    - O notebook da camada Silver é executado com retentativas.
    - As validações garantem que as transformações dos dados e o schema estejam alinhados com o esperado.
3. **Processamento da Camada Gold:**
    - O notebook da camada Gold é executado com retentativas.
    - Agregações e datasets refinados são validados para garantir a correção.

Logging e Transparência

Utilizando a biblioteca `logging` do Python, o pipeline captura logs detalhados para cada etapa do processo:

- **Logs INFO:** Documentam execuções e validações bem-sucedidas.
- **Logs WARNING:** Destacam tentativas falhas durante as retentativas.
- **Logs ERROR:** Registram problemas fatais após todas as tentativas falharem.

Benefícios

1. **Resiliência:** Retentativas de falhas transitórias garantem que o pipeline permaneça operacional mesmo sob interrupções temporárias.
2. **Integridade dos Dados:** Verificações de qualidade impedem que dados inválidos ou incompletos avancem para as camadas subsequentes.

3. **Visibilidade:** Logs abrangentes oferecem um rastro claro de execução, facilitando a depuração e o monitoramento.

Essa implementação garante que o pipeline seja robusto, confiável e capaz de lidar com desafios operacionais do mundo real de forma eficaz.

---

8 Configuração de Jobs e Monitoração no Databricks

8.1 Configuração do Job para Execução do Pipeline

Para automatizar a execução do pipeline, configuramos um Job no Databricks com as seguintes características:

- **Nome do Job**: `Pipeline_ABInBev`
- **Frequência**: Execução agendada a cada 5 minutos.
- **Ordem de Execução**:
    1. Notebook da camada Bronze.
    2. Notebook da camada Silver.
    3. Notebook da camada Gold.

8.2 Monitoração e Notificações

Para garantir a confiabilidade do pipeline, configuramos a monitoração dos Jobs e notificamos o engenheiro responsável em caso de erros.

- **Monitoramento via Interface do Databricks**: Utilizamos a interface de Jobs para monitorar o status de cada execução. Logs detalhados estão disponíveis para diagnosticar problemas.
- **Notificações por E-mail**: Configuramos alertas para que o engenheiro responsável receba e-mails em caso de falha na execução.

Exemplo de Configuração de Notificação no Job

No Databricks:

1. Acesse a página do Job configurado.
2. Adicione um **Email Notification** na seção de "Advanced Options".
3. Informe o e-mail do engenheiro responsável: `re046620@qintess.com`.
4. Configure para notificar somente em caso de falha.

8.3 Benefícios

A configuração do Job e a monitoração no Databricks garantem:

1. **Automatização**: Execução periódica sem necessidade de intervenção manual.
2. **Confiabilidade**: Logs detalhados e notificações permitem uma rápida identificação e correção de erros.

3. **Eficiência**: O intervalo de 5 minutos entre as camadas permite maior resiliência e organização.

---

9. Configurações do Azure Cloud

Este documento descreve como configurar e utilizar os serviços na nuvem utilizados no pipeline de dados, incluindo Azure Blob Storage e Databricks. As instruções visam garantir que todos os recursos sejam configurados com segurança e organizados para facilitar a execução do pipeline e a manutenção dos dados.

Configuração do Azure Blob Storage

Estrutura de Containers

1. Criamos um Storage Account no Azure.

2. No Blob Storage, crie os seguintes containers para organização das camadas:

- bronze: Para armazenar os dados brutos.

- silver: Para dados transformados e limpos.

- gold: Para dados consolidados e refinados.

Configuração de Credenciais

Utilizamos variáveis de ambiente para armazenar as credenciais do Azure:

`AZURE_STORAGE_ACCOUNT`: Nome da conta de armazenamento.

`AZURE_STORAGE_KEY`: Chave de acesso.

Essas variáveis podem ser configuradas em sistemas locais ou no Databricks para evitar hardcoding.

Atribuição de Permissões

- Configuramos o princípio de menor privilégio:

- Permitimos que a conta de serviço acesse apenas os containers necessários.

Configuração do Databricks

Criação do Workspace

1. Acessamos o portal do Azure e crie um workspace do Databricks.

2. Configuramos o cluster para execução dos notebooks:

- Tipo de Instância: Standard_DS3_v2 (ou outra de acordo com o tamanho do workload).

- Auto-scaling: Habilitado para gerenciar picos de carga automaticamente.

- Políticas de Terminação: Configure para encerrar o cluster após 30 minutos de inatividade.

Integração com Azure Blob Storage

Configuramos as credenciais de acesso:

- Configuramos `spark.conf` no Databricks para autenticação:

```python
spark.conf.set("fs.azure.account.key.<STORAGE_ACCOUNT_NAME>.blob.core.windows.net", "<STORAGE_ACCOUNT_KEY>")
```

- Substituir `<STORAGE_ACCOUNT_NAME>` e `<STORAGE_ACCOUNT_KEY>` pelas credenciais da sua conta de armazenamento.

Monitoramento e Auditoria

1. Azure Monitor:

   - Configuramos métricas e logs no Azure Monitor para rastrear o uso e identificar falhas no Blob Storage.

2. Databricks Jobs UI:

   - Usamos a interface do Databricks para monitorar a execução de jobs, incluindo logs detalhados de cada etapa.

3. Auditoria:

   - Habilitamos logs de auditoria no Azure para rastrear acessos e operações realizadas nos containers.

Segurança

1. Backup e Recuperação:

   - Configuramos backups automáticos no Azure Blob Storage para garantir a recuperação de dados em caso de falhas.

2. Logs Críticos:

   - Registramos os logs de erros e eventos críticos em um local seguro e centralizado.

3. Criptografia:

   - Certificamo-nos de que todos os dados sejam criptografados em trânsito e em repouso, utilizando as opções nativas do Azure.

Automatização

1. Provisionamento de Recursos:

   - Utilizamos o Azure CLI para criar recursos automaticamente. Exemplo de comando para criar um container:

     ```bash
     az storage container create --account-name <STORAGE_ACCOUNT_NAME> --name bronze
     ```

2. Execução Automatizada de Jobs:

   - Configuramos jobs no Databricks para rodar em intervalos regulares:

     - Use a interface do Databricks para agendar a execução.

- Configuramos notificações por e-mail em caso de falhas.

Melhorias Futuras

1. Integração com CI/CD:

- Usar o GitHub Actions ou Azure DevOps para automatizar o deploy de alterações no pipeline.

2.Otimização de Custo:

- Analisar o uso do cluster no Databricks e ajuste a capacidade conforme a demanda.

3. Documentação Dinâmica:

- Gerar relatórios automáticos para documentar execuções e resultados do pipeline.

---

10. Resultados e Conclusão

10.1. Resultados

1. **Pipeline Orquestrado:**
    - Todas as camadas foram processadas com sucesso no Databricks.
    - Intervalos adicionados entre os processamentos para maior resiliência.
2. **Dockerização Planejada:**
    - Configuração concluída para o GitHub Actions e Azure Container Registry, mas não executada devido às limitações locais.
    - Os `Dockerfiles` para as camadas Bronze e Silver foram deixados prontos, com a camada Gold utilizando o mesmo `Dockerfile` da Silver.
    - A ideia inicial era integrar as imagens criadas ao **Azure Data Factory** para realizar a orquestração.
3. **Validação e Análise Final:**

Para garantir que os dados processados na camada Gold estavam corretos, realizamos uma análise final utilizando um script de validação. O script recupera os dados da camada Gold no Azure Blob Storage, realiza algumas agregações e gera gráficos para visualização.

- **Agregações Realizadas:**
    - Contagem de entradas por tipo de cervejaria (`brewery_type`).
    - Contagem de cervejarias por estado ou província (`state_province`).
    - Análise específica para o estado do Colorado.
- **Gráficos Gerados:**
    - Contagem de tipos de cervejarias.
    - Contagem de cervejarias por estado ou província.

- **Exemplo de Código:**

```
# Exemplo de agregação
type_count_df = gold_df.groupBy('brewery_type').count()
state_count_df = gold_df.groupBy('state_province').count()

# Gerar gráficos com matplotlib
plt.figure(figsize=(10, 6))
plt.bar(type_count_pd['brewery_type'], type_count_pd['count'])
plt.xlabel('Brewery Type')
plt.ylabel('Count')
plt.title('Count of Brewery Types')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Esses passos finais garantiram a consistência e a qualidade dos dados gerados no pipeline.

10.2. Conclusão

A decisão de realizar a orquestração diretamente no Databricks garantiu a entrega do pipeline funcional dentro do prazo e com alta confiabilidade. A documentação detalha as soluções implementadas e as alternativas planejadas, demonstrando habilidade em superar desafios técnicos e flexibilidade para adaptar-se às condições disponíveis.
