Project.Spark.md

Documentação do Projeto: Análise de Temperatura Global

1. Visão Geral do Projeto

O projeto tem como objetivo construir um sistema para analisar e consultar dados de temperatura global. O fluxo do projeto envolve o uso do Apache Spark para processamento e análise de dados, a criação de uma API para fornecer acesso aos dados tratados.

2. Processamento e Análise com Apache Spark

2.1. Objetivo

O objetivo do uso do Apache Spark é realizar o processamento e análise de grandes volumes de dados sobre temperaturas globais, extraindo informações significativas e otimizando o armazenamento para consultas eficientes.

2.2. Importação dos Dados

Os dados foram importados para o Apache Spark a partir de um arquivo CSV. O arquivo contém registros de temperaturas globais em diversas cidades ao longo do tempo.

```
python from pyspark.sql import SparkSession

# Inicializar a SparkSession

spark = SparkSession.builder \
        .appName("Global Temperature by date Analysis") \
        .getOrCreate()

# Carregar o dataset
df = spark.read.csv('/path/to/GlobalLandTemperaturesByCity.csv',
header=True, inferSchema=True)
```

- **SparkSession**: A entrada principal para a funcionalidade do Spark. É usada para criar DataFrames e executar consultas SQL.
- `spark.read.csv`: Método para ler arquivos CSV e criar um DataFrame. O parâmetro `header=True` indica que o arquivo CSV contém um cabeçalho, e `inferSchema=True` permite que o Spark deduza automaticamente o tipo de dados de cada coluna.

2.3. Tratamento e Limpeza dos Dados

Após a importação dos dados, foi realizado um processo de limpeza para lidar com valores nulos e inconsistências.

```
from pyspark.sql.functions import col, sum

# Contar valores nulos por coluna
null_counts = [sum(col(c).isNull().cast("int")).alias(c) for c in
df.columns]
df.select(null_counts).show()

# Preencher valores nulos com valores padrão
df_cleaned = df.fillna({
    'AverageTemperatureUncertainty': 0,
    'City': 'Unknown',
    'Latitude': 'Unknown',
    'Longitude': 'Unknown'
})

# Verificar se a limpeza foi bem-sucedida
df_cleaned.select([sum(col(c).isNull().cast("int")).alias(c) for c in
df_cleaned.columns]).show()
```

- `col(c).isNull().cast("int")`: Cria uma coluna que indica se o valor é nulo (1) ou não (0). `sum()` é usado para contar o número total de valores nulos por coluna.
- `fillna()`: Preenche valores nulos com valores padrão especificados em um dicionário, onde cada chave é o nome de uma coluna e cada valor é o valor a ser usado para substituir os nulos.

2.4. Transformação dos Dados

Os dados foram transformados para análise, incluindo a conversão de tipos e a adição de novas colunas para facilitar a agregação.

```
from pyspark.sql.functions import avg, to_date, month, year

# Converter a coluna 'dt' para o formato de data
df = df.withColumn("dt", to_date(col("dt")))

# Adicionar colunas de mês e ano
df = df.withColumn("month", month(col("dt")))
df = df.withColumn("year", year(col("dt")))
```

- `to_date()`: Converte uma coluna para o tipo de dado `DateType`.
- `month()` **e** `year()`: Extraem o mês e o ano de uma coluna de data, respectivamente, e adicionam essas informações como novas colunas.

2.5. Análise e Agregação

Foi realizada uma agregação dos dados para calcular a temperatura média por país, cidade e data.

```
df_country_city_avg_temp = df_cleaned.groupBy("Country", "City", "dt").agg(
    avg("AverageTemperature").alias("AverageTemperature")
)
# Mostrar as primeiras linhas do DataFrame agregado
df_country_city_avg_temp.show(5)
```

- `groupBy()`: Agrupa os dados com base nas colunas especificadas.
- `agg()`: Executa uma agregação nas colunas agrupadas. Neste caso, calculamos a média da temperatura usando `avg()`.

2.6. Armazenamento dos Dados Tratados

Os dados tratados foram salvos em um arquivo CSV limpo.

```
df_cleaned_coalesced = df_cleaned.coalesce(1)
df_cleaned_coalesced.write.csv("/path/to/cleaned_data.csv", header=True,
mode="overwrite")
```

- `coalesce()`: Reduz o número de partições do DataFrame para uma, o que pode ajudar a otimizar o desempenho ao gravar os dados.
- `write.csv()`: Grava o DataFrame em um arquivo CSV. O parâmetro `header=True` indica que o arquivo deve conter um cabeçalho, e `mode="overwrite"` permite sobrescrever o arquivo se ele já existir.

2.7. Resumo das Funções Utilizadas

- `SparkSession`: Inicializa o contexto do Spark.
- `read.csv()`: Carrega dados de um arquivo CSV.
- `col()`: Referencia uma coluna em uma operação.
- `isNull()`: Verifica se o valor é nulo.
- `fillna()`: Preenche valores nulos com valores padrão.
- `to_date()`: Converte strings para datas.
- `month()` **e** `year()`: Extraem partes de uma data.
- `groupBy()`: Agrupa os dados por uma ou mais colunas.
- `agg()`: Executa funções de agregação.
- `coalesce()`: Reduz o número de partições do DataFrame.
- `write.csv()`: Grava o DataFrame em um arquivo CSV.

Criação da API

3.1. Objetivo

A API foi criada para fornecer acesso aos dados de temperatura tratados. Ela expõe endpoints para consultar dados com base na cidade e na data.

3.2. Implementação da API

A API foi implementada em Python usando o Flask e expõe um endpoint para consultar as temperaturas armazenadas no Azure.

```
# Configurar a API com Flask
from flask import Flask, request, jsonify
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg, to_date, col
```

```
app = Flask(__name__)

# Inicializar a SparkSession
spark = SparkSession.builder \
    .appName("Global Temperature Analysis by date by city API") \
    .getOrCreate()

# Carregar o dataset limpo
df = spark.read.csv('cleanedUnico_bytemperature_bydatabycity.csv',
header=True, inferSchema=True)

# Certificar-se de que a coluna 'dt' está no formato de data
df = df.withColumn("dt", to_date(col("dt")))

@app.route('/get_countrys_temperature', methods=['GET'])
def get_temperature_by_country():
    country = request.args.get('country')
    df_filtered = df.filter(df['Country'] == country)
    avg_temp =
df_filtered.agg(avg("AverageTemperature").alias("AverageTemperature")).colle
ct()

    if avg_temp:
        temperature = avg_temp[0]['AverageTemperature']
        return jsonify({'country': country, 'average_temperature':
temperature})
    else:
        return jsonify({'error': 'Country not found'}), 404


@app.route('/get_temperature_by_date_country', methods=['GET'])
def get_temperature_by_date_country():
    country = request.args.get('country')
    date = request.args.get('date')
    df_filtered = df.filter((df['Country'] == country) & (df['dt'] == date))
    avg_temp =
df_filtered.agg(avg("AverageTemperature").alias("AverageTemperature")).colle
ct()

    if avg_temp:
        temperature = avg_temp[0]['AverageTemperature']
        return jsonify({'country': country, 'date': date,
'average_temperature': temperature})
    else:
        return jsonify({'error': 'Temperature data not found for the given
```

```
country and date'}), 404


@app.route('/get_temperature_by_city_date', methods=['GET'])
def get_temperature_by_city_date():
    city = request.args.get('city')
    date = request.args.get('date')
    df_filtered = df.filter((df['City'] == city) & (df['dt'] == date))
    avg_temp =
df_filtered.agg(avg("AverageTemperature").alias("AverageTemperature")).colle
ct()

    if avg_temp:
        temperature = avg_temp[0]['AverageTemperature']
        return jsonify({'city': city, 'date': date, 'average_temperature':
temperature})
    else:
        return jsonify({'error': 'Temperature data not found for the given
city and date'}), 404

@app.route('/get_temperature_by_country_city', methods=['GET'])
def get_temperature_by_country_city():
    country = request.args.get('country')
    city = request.args.get('city')
    df_filtered = df.filter((df['Country'] == country) & (df['City'] ==
city))
    avg_temp =
df_filtered.agg(avg("AverageTemperature").alias("AverageTemperature")).colle
ct()

    if avg_temp:
        temperature = avg_temp[0]['AverageTemperature']
        return jsonify({'country': country, 'city': city,
'average_temperature': temperature})
    else:
        return jsonify({'error': 'Temperature data not found for the given
country and city'}), 404


@app.route('/get_temperature_by_country_date_range', methods=['GET'])
def get_temperature_by_country_date_range():
    country = request.args.get('country')
    start_date = request.args.get('start_date')
    end_date = request.args.get('end_date')

    df_filtered = df.filter(
```

```
        (df['Country'] == country) &
        (df['dt'] >= start_date) &
        (df['dt'] <= end_date)
    )

    df_filtered = df_filtered.select('dt',
'AverageTemperature').orderBy('dt')
    results = df_filtered.collect()

    if results:
        temperatures = [{'date': row['dt'].strftime('%Y-%m-%d'),
'temperature': row['AverageTemperature']} for row in results]
        return jsonify({'country': country, 'temperature_data':
temperatures})
    else:
        return jsonify({'error': 'Temperature data not found for the given
country and date range'}), 404


if __name__ == "__main__":
    app.run(port=5003)
```

4. Armazenamento dos Dados no Firebase

4.1. Objetivo

Os dados tratados foram armazenados no Firebase Realtime Database para acesso rápido e eficiente pela API e pelo aplicativo Android.

4.2. Transformação para JSON

Os dados foram transformados em JSON para serem compatíveis com o formato aceito pelo Firebase Realtime Database.

```
import pandas as pd

# Ler o arquivo CSV limpo
df_cleaned = pd.read_csv('/path/to/cleaned_data.csv')

# Converter para JSON
df_cleaned.to_json('/path/to/cleaned_data.json', orient='records')
```

4.3. Upload para o Firebase

Os dados JSON foram carregados para o Firebase Realtime Database usando o console do Firebase.

5. Desenvolvimento do Front-End

5.1. Objetivo

Desenvolver uma interface FrontEnd para permitir que os usuários consultem a temperatura em uma cidade e data específicas.

6. Diagrama da Arquitetura

Aqui está um diagrama simplificado da arquitetura do projeto:

```
```