

DESAFIO FINAL – ESTÁGIO COMPASS – ANÁLISE DE DADOS NO AWS

Link repositório no github com os códigos utilizados.

Para o desafio final foi proposto a realização de várias etapas de análise de dados, sua aquisição até a realização de gráficos, utilizando ferramentas do AWS.

Foi entregue um dois arquivos em csv, um sobre filmes e outro sobre séries, com dados extraídos do IMDB, era necessário escolher o tratamento de filmes ou séries e utilizar como segunda fonte de dados a API do TMDb.

Mesmo não utilizando o AWS é possível reproduzir parte do desafio na máquina, para isso recomendo a utilização do jupyter pela sua facilidade. No meu Github vou deixar o código que utilizei no jupyter, porem o que muda, em comparação do código utilizado na AWS, é a localização das pastas pra puxar o arquivo e depois salvar.

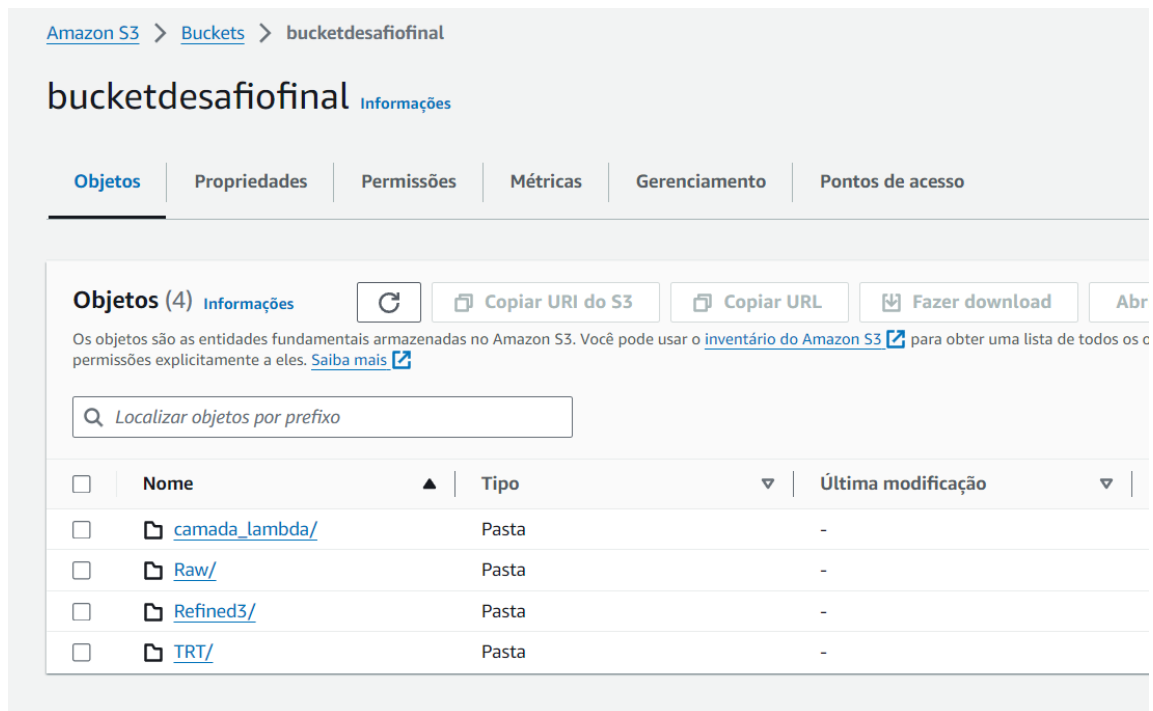
Utilizando o jupyter, inicie uma nova sessão através do terminal com o comando `jupyter notebook` e crie um novo notebook.

Caso utilize o Windows, recomendo iniciar o jupyter pelo terminal do codespace do github, utilizando a imagem do docker `docker cp <nome_do_container>:/home/jovyan ./.jupyter`, possibilitando utilizar o pyspark que será necessário no código.

É possível criar gráficos no jupyter utilizando a biblioteca Matplotlib, as instruções estão no próprio site da biblioteca.

PRIMEIRO PASSO, criação do bucket no S3. Um bucket do Amazon S3 (Simple Storage Service) é um contêiner de armazenamento na nuvem onde você pode armazenar uma quantidade ilimitada de dados de forma segura e durável. Funciona como um diretório ou pasta, mas na nuvem.

E é lá que vamos armazenar os dados.



SEGUNDO PASSO, criação da pasta RAW. RAW é a camada de dados brutos, nele salvei as planilhas entregues do IMDB e a requisição realizada na API d TMDB.

Não consigo fornecer cópia do arquivo do IMDB, porém mencionarei como ele foi utilizado.

Pelo fato do arquivo em csv do IMDB ser extenso foi necessário realizar um script em python e utilizar o docker para carregar o csv no bucket os códigos precisam serem executados na máquina.

```
```python
arquivo de nome script_filmes_series.py
as chaves do aws não são verdadeira e estão aí de exemplo

import boto3
import os
import csv
from datetime import datetime

def load_data_to_aws(filename, bucket_name, storage_layer):
 session = boto3.Session(
```

```

 aws_access_key_id='ASI5R',
 aws_secret_access_key='oazJVleg3fk5',
 aws_session_token= 'IQo8dqpsQ=='
)
Inicializar o cliente S3
s3_client = session.client('s3')

Obter a data de processamento atual
current_date = datetime.now().strftime("%Y/%m/%d")

Nome do arquivo
base_name = os.path.basename(filename)

Montar o caminho de destino no S3
destination_path = f"{storage_layer}/{current_date}/{base_name}"

try:
 # Carregar o arquivo para o S3
 s3_client.upload_file(filename, bucket_name, destination_path)
 print(f"Arquivo {filename} carregado com sucesso para o S3 em
{bucket_name}/{destination_path}")
except Exception as e:
 print(f"Erro ao carregar o arquivo para o S3: {e}")

Definir o nome dos arquivos CSV
movies_filename = 'data/movies.csv'
series_filename = 'data/series.csv'

Definir o nome do bucket S3
bucket_name = 'bucketdesafiofinal'

Definir as camadas de armazenamento
storage_layer_movies = 'Raw/Local/CSV/Movies'
storage_layer_series = 'Raw/Local/CSV/Series'

Definir a origem dos dados
data_origin_movies = 'Movies'
data_origin_series = 'Series'

Definir o formato dos dados
data_format = 'CSV'

Carregar filmes para o S3
load_data_to_aws(movies_filename, bucket_name, storage_layer_movies)

```

```

Carregar séries para o S3
load_data_to_aws(series_filename, bucket_name, storage_layer_series)

...

```docker
ARG PYTHON_VERSION=3.12
FROM python:${PYTHON_VERSION}-slim as base

WORKDIR /app

RUN --mount=type=cache,target=/root/.cache/pip \
    pip install boto3

COPY . .

CMD ["python", "script_filmes_series.py"]

...

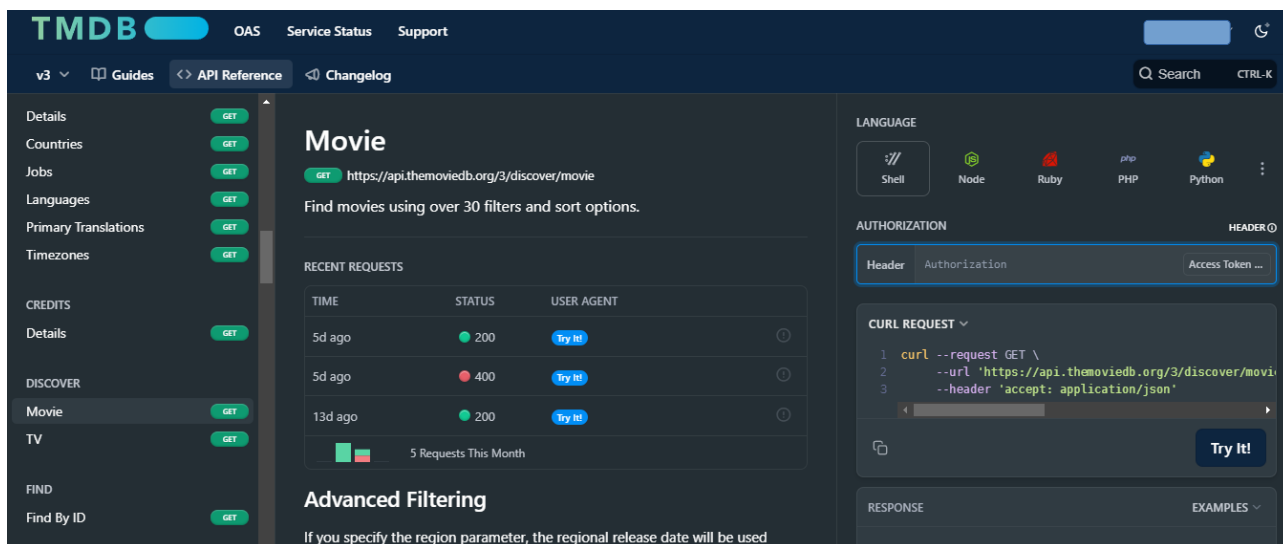
```

Em resumo, este Dockerfile cria uma imagem Docker que inclui a biblioteca boto3 e o código-fonte do aplicativo Python. Quando um contêiner é iniciado a partir dessa imagem, ele executa o script `script_filmes_series.py`. O script interage com os serviços da AWS, carregando ou processando dados relacionados a filmes e séries.

Com isso carregamos a primeira fonte de dados.

A segunda fonte de dados vem da API do TMDb, que quando utilizada de forma limitada para projetos que não visam o lucro é entregue de forma gratuita. Sendo necessário o cadastro no site <https://developer.themoviedb.org/docs/authentication-user>.

As requisições podem ser testadas no site, assim é possível analisar o que se deseja coletar.



Utilizei dois endpoints, ou seja, utilizei dois ambientes da API, a movie e o discover.

Meu grupo foi designado para analisar filmes ou séries de ficção científica e/ou fantasia, escolhi trabalhar com filmes. Então no primeiro endpoint, discover, percebi que os dados dos filmes são separados por páginas.

Assim, utilizei o for para iterar sobre a sequência de páginas e armazenar num dataframe filmes que tivessem pelo menos um dos ids de gêneros fantasia ou ficção científica, e que fossem de 2001 a 2015.

Ao receber a designação de análise de gêneros de ficção científica ou fantasia, me despertou o interesse na análise e comparação de dados com filmes voltados para adolescentes e baseados em livros em comparação com os demais, assim escolhi como norte as sagas de Harry Potter, Crepúsculo e Jogos Vorazes.

E como esses filmes foram lançados de 2001 a 2015, escolhi recolher dados da API do TMDb apenas desses anos.

Uma vez que API armazena uma variedade de dados, como, por exemplo, imagem do pôster, descrição, comentários, etc, selecionei as informações que queria sobre cada filme.

O segundo endpoint realizei a pesquisa pelo id do filme, informação que consegui no endpoint anterior e criei outro dataframe com as informações.

Depois uni os dois dataframes, e os salvei no formato JSON separados por 100 requisições por arquivo.

Vale ressaltar, que um dataframe é uma estrutura de dados tabular bidimensional, muito comum em análise de dados e processamento de dados. E que o JSON é um formato de texto simples para representar dados estruturados e suporta uma ampla variedade de tipos de dados.

As requisições a API do TMDb foram realizadas no LAMBDA da AWS..

Qual o motivo para utilizar o Lambda?

É um serviço de computação sem servidor que permite executar código em resposta a eventos, como invocações de API's. É útil para execução de pequenos trechos de código ou scripts que respondem a eventos específicos.

Na execução do lambda é necessário criar uma camada. A camada do Lambda é um recurso que permite a inclusão de código, bibliotecas e outras dependências, que precisam ser utilizadas no código em execução.

Neste caso em específico, escolhi as bibliotecas necessárias, salvei em uma pasta que após zipada foi carregada no S3 e incluído o caminho URL nas definições do Lambda. As bibliotecas escolhidas foram, python 3.10, numpy, request, pandas e boto.

Ao subir a pasta zipada com as bibliotecas é necessário observar o a especificação no nome das pastas exigidas pela AWS.

Runtime	Path
Node.js	nodejs/node_modules
	nodejs/node14/node_modules (NODE_PATH)
	nodejs/node16/node_modules (NODE_PATH)
	nodejs/node18/node_modules (NODE_PATH)
Python	python
	python/lib/ <i>python3.x</i> /site-packages (diretórios do site)
Java	java/lib (CLASSPATH)
Ruby	ruby/gems/3.2.0 (GEM_PATH)
	ruby/lib (RUBYLIB)
Todos os runtimes	bin (PATH)
	lib (LD_LIBRARY_PATH)

Configurações de tempo de execução

Informações

Editar

Editar a configuração de gerenciamento de tempo de execução

Tempo de execução

Python 3.10

► Configuração de gerenciamento de tempo de execução

Manipulador

Informações

lambda_function.lambda_handler

Arquitetura

Informações

x86_64

Camadas

Informações

Editar

Adicionar uma camada

Ordem de mesclagem	Nome	Versão da camada	Tempos de execução compatíveis	Arquiteturas compatíveis	ARN da versão
1	testedf	1	python3.10, python3.8	x86_64	arn:aws:lambda:us-east-1:058264091199:layer:testedf:1

```
```python
```

```
import requests
import sys
import os
import csv
from datetime import datetime
import pandas as pd
import json
import boto3
import concurrent.futures
```

```
def lambda_handler(event, context):
```

```
 s3= boto3.client('s3')
```

```
 # Nome do bucket e caminho do arquivo no S3
```

```
 bucket_name = "bucketdesafiofinal"
```

```

#####
```

```
 #CRIANDO DATA FRAME DE POPULARIDADE GERAL
```

```
 # IDs dos gêneros desejados
```

```
 ids_genero = [14, 878]
```

```
 # Número máximo de páginas por ano
```

```
 max_paginas_por_ano = 500
```

```
 # Dicionário para armazenar os DataFrames de filmes por ano e por gênero
```

```
 df_por_ano_genero = {}
```

```
 # Loop pelos anos
```

```
 for ano in range(2001, 2016):
```

```
 resultados_por_ano_genero = {'Ano': [], 'Genre_ID': [], 'id': [], 'title': [], 'release_date':
[], 'popularity': [], 'vote_average': [], 'vote_count': [], 'adult':[]}
```

```
 # Loop pelos IDs de gênero
```

```
 for id_genero in ids_genero:
```

```
 # Loop pelas páginas
```

```
 for pagina in range(1, max_paginas_por_ano + 1):
```

```
 # URL base da API do TMDb para pesquisa de filmes por gênero
```

```
 base_url = "https://api.themoviedb.org/3/discover/movie"
```

```
 # Parâmetros da pesquisa
```

```
 params = {
```

```
 "api_key": "7XXXXXXXXXXda",
```

```
 "with_genres": id_genero,
```

```
 "primary_release_year": ano,
```

```
 "page": pagina
```

```
 }
```

```
 # Fazer a solicitação à API
```

```
 response = requests.get(base_url, params=params)
```

```
 # Verificar se a solicitação foi bem-sucedida
```

```
 if response.status_code == 200:
```

```
 # Converter a resposta para JSON
```

```
 data = response.json()
```

```
 # Adicionar os resultados à lista
```



```

for result in data.get('results', []):
 resultados_por_ano_genero['Ano'].append(ano)
 resultados_por_ano_genero['Genre_ID'].append(id_genero)
 resultados_por_ano_genero['id'].append(result['id'])
 resultados_por_ano_genero['title'].append(result['title'])
 resultados_por_ano_genero['release_date'].append(result['release_date'])
 resultados_por_ano_genero['popularity'].append(result['popularity'])
 resultados_por_ano_genero['vote_average'].append(result['vote_average'])
 resultados_por_ano_genero['vote_count'].append(result['vote_count'])
 resultados_por_ano_genero['adult'].append(result['adult'])

Verificar se atingiu a última página
if pagina >= data['total_pages']:
 break
else:
 print("Erro ao fazer a solicitação:", response.status_code)
 break

Criar um DataFrame com os resultados do ano atual
df_por_ano_genero[ano] = pd.DataFrame(resultados_por_ano_genero)

Juntar todos os DataFrames em um único DataFrame
df_tmdb = pd.concat(df_por_ano_genero.values(), ignore_index=True)

Remover itens duplicados com base na coluna 'id'
df_tmdb.drop_duplicates(subset='id', inplace=True)

Exibir o DataFrame final
print(df_tmdb)

#####
#####

Função para consultar o orçamento de um filme por ID
def consultar_dados_por_id(movie_id):
 url = f"https://api.themoviedb.org/3/movie/{movie_id}?language=en-US"
 params = {
 "api_key": "XXXXXX"
 }
 response = requests.get(url, params=params)
 if response.status_code == 200:
 data = response.json()
 return {
 'budget': data.get('budget', None),
 'imdb_id': data.get('imdb_id', None),
 'runtime': data.get('runtime', None),

```

```
 'revenue':data.get('revenue', None),
 }
 else:
 print(f"Erro ao consultar o filme com ID {movie_id}. Status code:
{response.status_code}")
 return None
```

```
Função para processar consultas em paralelo
def processar_consultas(movie_ids):
 with concurrent.futures.ThreadPoolExecutor() as executor:
 return list(executor.map(consultar_dados_por_id, movie_ids))
```

```
Lista de IDs de filmes
movie_ids = df_tmdb['id'].tolist()
```

```
Consulta dos dados em paralelo
dados_filmes = processar_consultas(movie_ids)
```

```
Criando DataFrame com os dados consultados
df_dados_filmes = pd.DataFrame(dados_filmes)
```

```
Redefinindo o índice do DataFrame df_tmdb
df_tmdb.reset_index(drop=True, inplace=True)
```

```
Redefinindo o índice do DataFrame df_dados_filmes
df_dados_filmes.reset_index(drop=True, inplace=True)
```

```
Criando DataFrame com os dados consultados
df_dados_filmes = pd.DataFrame(dados_filmes)
```

```
Combinando os DataFrames
df_tmdb = pd.concat([df_tmdb, df_dados_filmes], axis=1)
```

```
Exibindo o DataFrame com a nova coluna
print(df_tmdb)
```

```

#####
```

```
JSON GERAL
```

```
Definir o limite de registros desejado
```

```
limite_registros = 100 # Limite de registros por arquivo
```

```
Verificar o número total de registros no DataFrame
```

```
num_registros = len(df_tmdb)
```

```
Lista para armazenar os DataFrames menores
```

```
dataframes_divididos = []
```

```
Verificar se o número total de registros excede o limite
```

```
if num_registros > limite_registros:
```

```
 print("Número total de registros excede 100. Dividindo em arquivos menores.")
```

```
Dividir os dados em DataFrames menores com no máximo 100 registros
```

```
num_dataframes = num_registros // limite_registros
```

```
resto = num_registros % limite_registros
```

```
inicio = 0
```

```
for i in range(num_dataframes):
```

```
 fim = inicio + limite_registros
```

```
 df_temp = df_tmdb.iloc[inicio:fim]
```

```
 dataframes_divididos.append(df_temp)
```

```
 inicio = fim
```

```
Adicionar o DataFrame restante, se houver
```

```
if resto > 0:
```

```
 df_temp = df_tmdb.iloc[inicio:]
```

```
 dataframes_divididos.append(df_temp)
```

```
Escrever todos os dados em um único arquivo JSON
```

```
for i, item in enumerate(dataframes_divididos):
```

```
 corpo_obj = item.to_json(orient='records', lines=True)
```

```
 s3_chave = f'Raw/tmdb/json/2024/04/11/geral/parte_{i}.json'
```

```
 s3.put_object(Bucket=bucket_name, Key=s3_chave, Body=corpo_obj)
```

```
...
```

E assim foi finalizada a camada RAW.

Amazon S3 > Buckets > bucketdesafiofinal > Raw/

## Raw/

[Copiar URI do S3](#)

**Objetos** | Propriedades

---

**Objetos (3)** [Informações](#)

[Recarregar](#) [Copiar URI do S3](#) [Copiar URL](#) [Fazer download](#) [Abrir](#) [Excluir](#) [Ações](#) [Criar pasta](#)

[Carregar](#)

Os objetos são as entidades fundamentais armazenadas no Amazon S3. Você pode usar o [inventário do Amazon S3](#) para obter uma lista de todos os objetos em seu bucket. Para outras pessoas acessarem seus objetos, você precisará conceder permissões explicitamente a eles. [Saiba mais](#)

< 1 > ⚙

<input type="checkbox"/>	Nome ▲	Tipo ▼	Última modificação ▼	Tamanho ▼	Classe de armazenamento ▼
<input type="checkbox"/>	<a href="#">Local/</a>	Pasta	-	-	-
<input type="checkbox"/>	<a href="#">TMDB/</a>	Pasta	-	-	-
<input type="checkbox"/>	<a href="#">Unsaved/</a>	Pasta	-	-	-

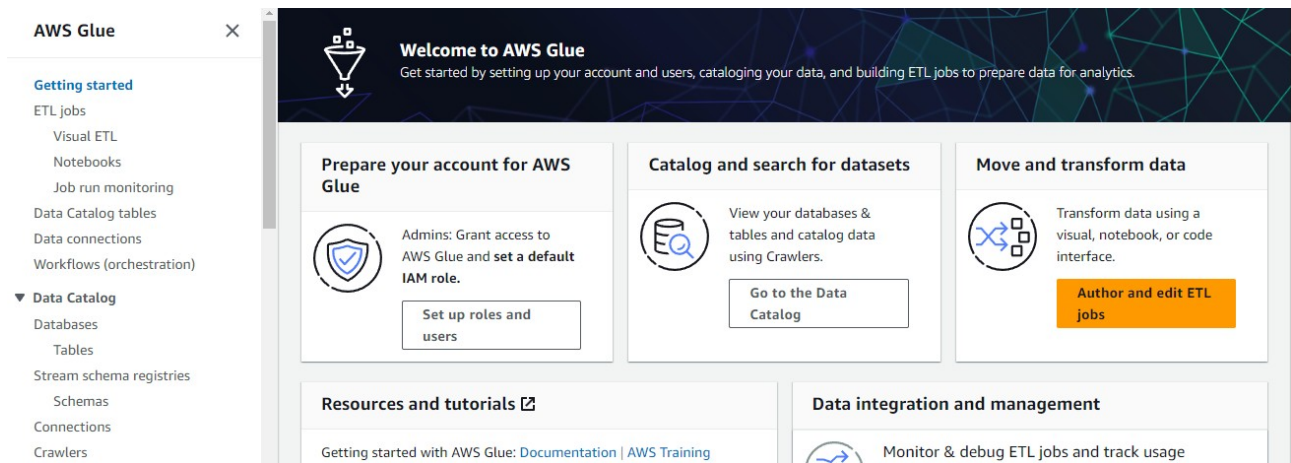
**TERCEIRO PASSO.** A segunda camada, a trusted, representa a camada confiável e se refere a uma etapa de dados confiáveis, precisos e de alta qualidade. Esta camada é fundamental para garantir a precisão e a confiabilidade das conclusões e decisões baseadas nos dados.

Os códigos foram executados no Glue da AWS e, diferente da primeira camada que foi utilizado pandas nesta, foi utilizado o pyspark.

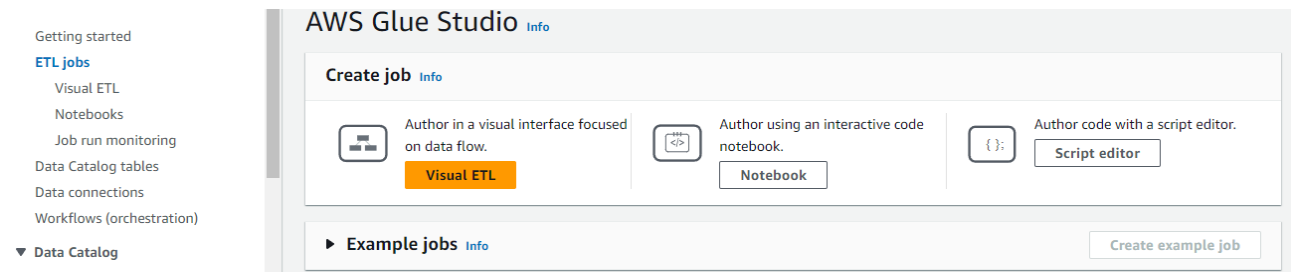
### POR QUE UTILIZAR O GLUE?

O Glue é um serviço de ETL (Extração, Transformação e Carga) totalmente gerenciado que facilita a preparação e carregamento de dados para análise.

No Glue é necessário configura o job, então ao entrar na página, aparecerá a tela a baixo, escolha o botão laranja “Author and edit ETL jobs”.



Depois, escolha a última opção “Script editor”.



As definições do meu job foram essas.

**IAM Role**  
Role assumed by the job with permission to access your data stores. Ensure that this role has permission to your Amazon S3 sources, targets, temporary directory, scripts, and any libraries used by the job.

**AWSGlueServiceRole**

**Type**  
The type of ETL job. This is set automatically based on the types of data sources you have selected.

**Spark**

**Glue version** [Info](#)  
Glue 3.0 - Supports spark 3.1, Scala 2, Python 3

**Language**  
Python 3

**Worker type**  
Set the type of predefined worker that is allowed when a job runs.

**G 1X**  
(4vCPU and 16GB RAM)

**Automatically scale the number of workers**  
☐ AWS Glue will optimize costs and resource usage by dynamically scaling the number of workers up and down throughout the job run. Requires Glue 3.0 or later.

**Requested number of workers**  
The number of workers you want AWS Glue to allocate to this job.

**2**

**Generate job insights**  
☒ AWS Glue will analyze your job runs and provide insights on how to optimize your jobs and the reasons for job failures.

**Job bookmark** [Info](#)  
Specifies how AWS Glue processes job bookmark when the job runs. It can remember previously processed data (Enable), update state information (Pause), or ignore state information (Disable).

**Disable**

**Flex execution** [Info](#)  
☐ Reduce costs by running this job on spare capacity. Ideal for non-urgent workloads that don't require fast jobs start times or consistent execution times. See recommendations, limitations and pricing in the help panel by clicking on the info link above.

**Number of retries**  
0

**Job timeout (minutes)**  
Set the execution time. The default is 2,880 minutes (48 hours) for a Glue ETL job. No job timeout is defaulted for a Glue Streaming job.

**60**

Na primeira fonte de dados do IMDB precisei realizar a busca pelos filmes das sagas, visto que sua classificação é diferente do TMDb. Após de maneira geral separei os filmes dos gêneros escolhidos, apliquei o filtro de temporariedade, tempo de duração (coloquei o mínimo de 80 minutos) e número de votos (coloquei mínimo em 30).

Exclui colunas que não utilizaria e renomeie outras colunas.

```
```pyspark
```

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
```

```

from awsglue.job import Job
from pyspark.sql.functions import col

# Inicializar o contexto do Glue
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
job.init(args['JOB_NAME'], args)

source_file = "s3://bucketdesafiofinal/Raw/Local/CSV/Movies/2024/04/12/movies.csv"

# Caminho de saída para os dados no formato Parquet
output_path_final = "s3://bucketdesafiofinal/TRT/Movies/movies"

# Criar um DynamicFrame a partir do arquivo CSV
df = glueContext.create_dynamic_frame.from_options(
    connection_type="s3",
    connection_options={"paths": [source_file]},
    format="csv",
    format_options={"withHeader": True, "separator": "|"},
    transformation_ctx="nome_transformacao"
)

# Converter DynamicFrame para DataFrame
df = df.toDF()

# Preencher valores NaN na coluna 'personagem' com '--'
#df = df.fillna('--', subset=['personagem'])

#### FAZER PESQUISA HARRY
palavras_chave_h = ['Harry Potter']

# Filtrar filmes relacionados ao Harry Potter
resultados_hp = df.filter(col('personagem').rlike('|'.join(palavras_chave_h)))

#### FAZER PESQUISA JOGOS VORAZES
palavras_chave_hg = ['Katniss Everdeen']

# Filtrar filmes relacionados aos Jogos Vorazes
resultados_hg = df.filter(col('personagem').rlike('|'.join(palavras_chave_hg)))

#### FAZER PESQUISA CREPUSCULO
palavras_chave_c = ['Bella Swan']

```

```

# Filtrar filmes relacionados a Crepúsculo
resultados_tw = df.filter(col('personagem').rlike(''.join(palavras_chave_c)))

#### JUNTAR TUDO
df_sagas = resultados_hp.union(resultados_hg).union(resultados_tw)

# Lista de IDs para excluir
ids_para_excluir = ['tt0092115', 'tt20766450', 'tt8443702']

# Filtrar DataFrame para manter apenas as linhas desejadas
df_sagas = df_sagas.filter(~col('id').isin(ids_para_excluir))

df_sagas = df_sagas.withColumn('anoLancamento',
df_sagas['anoLancamento'].cast('float'))

print(df_sagas)

#####
# FAZER DF GERAL DO CSV PELO GÊNERO
generos_procurados = ['Fantasy', 'Sci-Fi']

# Preencher valores NaN nas colunas 'genero' e 'anoLancamento'
df = df.fillna({'genero': '', 'anoLancamento': 0})

# Converter a coluna 'anoLancamento' para numérica
df = df.withColumn('anoLancamento', df['anoLancamento'].cast('int'))

# Filtrar filmes com base nos critérios especificados
df_csv_genero = df.filter((col('genero').rlike(''.join(generos_procurados))) &
(col('anoLancamento').between(2001, 2015)))

# Remover linhas duplicadas com base no ID
df_csv_genero = df_csv_genero.dropDuplicates(['id'])

# Preencher valores NaN na coluna 'tempoMinutos' com 0
df_csv_genero = df_csv_genero.fillna(0, subset=['tempoMinutos'])

# Converter a coluna 'tempoMinutos' para inteiro e remover linhas com valores menores
que 100
df_csv_genero = df_csv_genero.withColumn('tempoMinutos',
df_csv_genero['tempoMinutos'].cast('int'))
df_csv_genero = df_csv_genero.filter(df_csv_genero['tempoMinutos'] >= 80)

```



```

# Remover linhas com 'numeroVotos' menores que 30
df_csv_genero = df_csv_genero.filter(df_csv_genero['numeroVotos'] >= 30)

print(df_csv_genero)

##### JUNTAR TUDO
# Unir os DataFrames
df_final = df_sagas.union(df_csv_genero)

# Remover linhas duplicadas com base no ID
df_final = df_final.dropDuplicates(['id'])

# Remover colunas desnecessárias
colunas_para_excluir = ['genero', 'tituloOriginal', 'anoLancamento', 'generoArtista',
'personagem', 'nomeArtista', 'anoNascimento', 'anoFalecimento', 'profissao',
'titulosMaisConhecidos']
df_final = df_final.drop(*colunas_para_excluir)

# Renomear a coluna 'tituloPincipal' para 'tituloPrincipal'
df_final = df_final.withColumnRenamed('tituloPincipal', 'tituloPrincipal') \
    .withColumnRenamed('id', 'imdb_id') \
    .withColumnRenamed('notaMedia', 'notaMedia_imdb') \
    .withColumnRenamed('numeroVotos', 'numeroVotos_imdb')

print(df_final)

### SALVANDO OS DF
df_final=df_final.coalesce(1)
df_final.write.parquet(output_path_final, mode='overwrite')

job.commit()

...

```

Na segunda fonte de dados, renomeei colunas, já tinha aplicado o filtro de data de lançamento, então removi os filmes com tempo menor de 80 minutos e contagem de votos menores de 30.

```

```pyspark
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, lit
from pyspark.sql.types import IntegerType

Inicializar o SparkSession
spark = SparkSession.builder \
 .appName("Transformação TMDb") \
 .getOrCreate()

Inicializar o contexto do Glue
glueContext = GlueContext(spark.sparkContext)
spark = glueContext.spark_session
job = Job(glueContext)
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
job.init(args['JOB_NAME'], args)

source_file = "s3://bucketdesafiofinal/Raw/TMDB/json/2024/04/12/"
df_tmdb = spark.read.json(source_file)

Remover a coluna 'release_date'
df_tmdb = df_tmdb.drop("release_date")

Renomear as colunas conforme especificado
df_tmdb = df_tmdb.withColumnRenamed("Ano", "anoLancamento_tmdb") \
 .withColumnRenamed("Genre_ID", "id_genero") \
 .withColumnRenamed("id", "id_tmdb") \
 .withColumnRenamed("title", "titulo") \
 .withColumnRenamed("popularity", "popularidade") \
 .withColumnRenamed("vote_average", "votacao_media") \
 .withColumnRenamed("vote_count", "contagem_de_voto") \
 .withColumnRenamed("adult", "adulto") \
 .withColumnRenamed("budget", "orcamento") \
 .withColumnRenamed("runtime", "duracao") \
 .withColumnRenamed("revenue", "receita")

```

```
Converter a coluna para inteiro e remover linhas com valores 0
df_tmdb = df_tmdb.withColumn("orcamento", df_tmdb["orcamento"].cast(IntegerType()))
df_tmdb = df_tmdb.filter(df_tmdb["orcamento"] > 0)

Converter a coluna 'orcamento' para inteiro e remover linhas com valores 0
df_tmdb = df_tmdb.withColumn("receita", df_tmdb["receita"].cast(IntegerType()))
df_tmdb = df_tmdb.filter(df_tmdb["receita"] > 0)

Remover linhas com 'contagem_de_voto' menores que 30 e tempo menos que 80
df_tmdb = df_tmdb.withColumn("contagem_de_voto",
df_tmdb["contagem_de_voto"].cast(IntegerType()))
df_tmdb = df_tmdb.filter(df_tmdb["contagem_de_voto"] >= 30)

df_tmdb = df_tmdb.withColumn("duracao", df_tmdb["duracao"].cast(IntegerType()))
df_tmdb = df_tmdb.filter(df_tmdb["duracao"] >= 80)

Remover linhas com 'imdb_id' vazias
df_tmdb = df_tmdb.filter(df_tmdb["imdb_id"].isNotNull())

Reiniciar o índice
df_tmdb = df_tmdb.withColumn("index", col("imdb_id")) # Criar uma nova coluna 'index'
com os valores de 'imdb_id'
df_tmdb = df_tmdb.drop("imdb_id") # Remover a coluna 'imdb_id'
df_tmdb = df_tmdb.withColumnRenamed("index", "imdb_id") # Renomear a coluna 'index'
para 'imdb_id'
df_tmdb = df_tmdb.orderBy("imdb_id") # Ordenar o DataFrame por imdb_id

data_extracao = '2024/04/12'
df_tmdb = df_tmdb.withColumn("data_coleta_tmdb", lit(data_extracao))

Caminho de saída para os dados no formato Parquet
output_path = "s3://bucketdesafiofinal/TRT/TMDB/Fantasy/dt=2024-04-12"

Particionar os dados e escrever em arquivos Parquet
df_tmdb.write.parquet(output_path, mode="overwrite")
```

```
Encerrar a sessão do Spark
spark.stop()
```

```
job.commit()
```

```
...
```

Salvei os resultados em uma pasta que representa a segunda camada (TRT), com arquivos no formato parquet.

O Parquet é um formato de arquivo de coluna otimizado para processamento de Big Data. Ele armazena dados em colunas em vez de linhas, o que pode proporcionar uma compactação e eficiência de leitura melhor em comparação com o JSON, especialmente para conjuntos de dados grandes.

[Amazon S3](#) > [Buckets](#) > [bucketdesafiofinal](#) > TRT/

TRT/ Copiar URI do S3

[Objetos](#) | [Propriedades](#)

**Objetos (2)** [Informações](#)

Atualizar Copiar URI do S3 Copiar URL Fazer download Abrir Excluir Ações ▼

Criar pasta Carregar

Os objetos são as entidades fundamentais armazenadas no Amazon S3. Você pode usar o [inventário do Amazon S3](#) para obter uma lista de todos os objetos em seu bucket. Para outras pessoas acessarem seus objetos, você precisará conceder permissões explicitamente a eles. [Saiba mais](#)

<input type="checkbox"/>	Nome ▲	Tipo ▼	Última modificação ▼	Tamanho ▼	Classe de armazenamento ▼
<input type="checkbox"/>	<a href="#">Movies/</a>	Pasta	-	-	-
<input type="checkbox"/>	<a href="#">TMDB/</a>	Pasta	-	-	-

**QUARTO PASSO.** A terceira e última camada, refined, é a etapa que os dados são refinados e prontos para análise. Nesta fase, as técnicas de limpeza de dados, como tratamento de valores ausentes, remoção de duplicatas e padronização de formatos, são aplicadas para garantir a qualidade e consistência dos dados.

A minha ideia é utilizar a API do TMDB para complementar o csv, assim utilizei na API apenas filmes que tinham o id do IMDB pra que acontecesse esse encontro das duas fontes de dados. Foi uma decisão ousada que limitou meus dados porém terei uma 'tabela' sem espaços vazios.

Por isso na etapa anterior eliminei as linhas no arquivo TMDB que não possuíam o ID do IMDB.

O código executado no GLUE, nesta etapa, finalmente uni as duas fontes de dados, além de outros filtros e tratamentos como mudança de nome de colunas e a escolha das colunas que iriam ser utilizada.

Criei a tabela fato e suas dimensões, fiz isso em códigos separados para os filmes em gerais e os filmes das sagas.

```
```python
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, monotonically_increasing_id

# Inicializar o SparkSession
spark = SparkSession.builder \
    .appName("Transformação TRT") \
    .getOrCreate()

# Inicializar o contexto do Glue
glueContext = GlueContext(spark.sparkContext)
spark = glueContext.spark_session
job = Job(glueContext)
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
job.init(args['JOB_NAME'], args)
```

```
source_file_movie = "s3://bucketdesafiofinal/TRT/Movies/"
source_file_tmdb = "s3://bucketdesafiofinal/TRT/TMDB/Fantasy/dt=2024-04-12/"
```

```
# Carregar dados do arquivo Parquet
df_tmdb = spark.read.parquet(source_file_tmdb)
df_movie = spark.read.parquet(source_file_movie)
```

```
# Caminho de saída para os dados no formato Parquet
output_path = "s3://bucketdesafiofinal/Refined"
```

```
# junção dataframes
```

```
df_movie = df_movie.withColumnRenamed("imdb_id", "movie_imdb_id")
df_geral = df_tmdb.join(df_movie, df_movie.movie_imdb_id == df_tmdb.imdb_id, 'inner')
```

```
# Removendo linhas duplicadas com base na coluna imdb_id para garantir que seja uma
chave primária
```

```
df_geral = df_geral.dropDuplicates(["imdb_id"])
# Criando uma view temporária a partir do DataFrame geral
df_geral.createOrReplaceTempView('df_geral_view')
```

```
# Visualizar o esquema do DataFrame df_geral
df_geral.printSchema()
```

```
# Definir as consultas para as tabelas
```

```
df_data_query = """
    SELECT anoLancamento_tmdb AS Ano_Lancamento
    FROM df_geral_view
    """
```

```
df_titulo_query = """
    SELECT tituloPrincipal AS Titulo,
           imdb_id as titulo_imdb_id
    FROM df_geral_view
    """
```

```
df_adulto_query = """
    SELECT adulto AS Adulto,
           imdb_id as adulto_imdb_id
    FROM df_geral_view
    """
```

Criar DataFrames usando as consultas

```
df_data = spark.sql(df_data_query)
df_titulo = spark.sql(df_titulo_query)
df_adulto = spark.sql(df_adulto_query)
```

```
df_data.printSchema()
df_titulo.printSchema()
df_adulto.printSchema()
```

Adicionar uma nova coluna de ID em cada DataFrame

```
df_data = df_data.withColumn("id_data", monotonically_increasing_id()+1)
df_titulo = df_titulo.withColumn("id_titulo", monotonically_increasing_id()+1)
df_adulto = df_adulto.withColumn("id_adulto", monotonically_increasing_id()+1)
```

```
df_fato_query = """
    SELECT DISTINCT
           df_geral_view.imdb_id as fato_imdb_id,
           orcamento,
           receita,
           tempoMinutos,
           id_genero,
           popularidade,
           contagem_de_voto,
           votacao_media,
           data_coleta_tmdb,
           anoLancamento_tmdb,
           adulto
    FROM df_geral_view
    """
```

```
df_fato = spark.sql(df_fato_query)
```

```
df_fato.printSchema()
```

Converter id_adulto para string

```
df_adulto = df_adulto.withColumn("id_adulto", col("id_adulto").cast("string"))
df_data = df_data.withColumn("id_data", col("id_data").cast("string"))
```

```
df_titulo = df_titulo.withColumn("id_titulo", col("id_titulo").cast("string"))
```

```
df_fato = df_fato.join(df_data.select('id_data', 'Ano_Lancamento'),  
                        df_fato.anoLancamento_tmdb==df_data.Ano_Lancamento, 'inner')  
#df_fato = df_fato.drop('anoLancamento_tmdb', 'Ano_Lancamento')
```

```
df_fato = df_fato.join(df_titulo.select('id_titulo', 'titulo_imdb_id'),  
                        df_fato.fato_imdb_id==df_titulo.titulo_imdb_id, 'inner')  
#df_fato = df_fato.drop('titulo_imdb_id')
```

```
df_fato = df_fato.join(df_adulto.select('id_adulto', 'adulto_imdb_id'),  
                        df_fato.fato_imdb_id==df_adulto.adulto_imdb_id, 'inner')  
#df_fato = df_fato.drop('adulto', 'Adulto')
```

```
df_temp = df_fato.toPandas()  
drop_colunas = ['anoLancamento_tmdb', 'Ano_Lancamento', 'titulo_imdb_id', 'adulto',  
                'adulto_imdb_id']  
df_temp = df_temp.drop(drop_colunas, axis=1)  
df_fato = spark.createDataFrame(df_temp)
```

```
df_fato.orderBy('fato_imdb_id').show  
df_fato.printSchema()
```

```
# Escrever DataFrames como tabelas no AWS Glue
```

```
def salvar_tabela(df, tabela_nome):  
    df.write.format("parquet") \  
        .mode("overwrite") \  
        .save(output_path + "/" + tabela_nome)
```

```
# Salvando as tabelas
```

```
salvar_tabela(df_fato, "tabela_fato")  
salvar_tabela(df_data, "tabela_data")  
salvar_tabela(df_titulo, "tabela_titulo")  
salvar_tabela(df_adulto, "tabela_adulto")
```



```
# Encerrar a sessão do Spark
```

```
spark.stop()
```

```
job.commit()
```

```
...
```

```
```python
```

```
import sys
```

```
from awsglue.transforms import *
```

```
from awsglue.utils import getResolvedOptions
```

```
from pyspark.context import SparkContext
```

```
from awsglue.context import GlueContext
```

```
from awsglue.job import Job
```

```
from pyspark.sql import SparkSession
```

```
from pyspark.sql.functions import col, row_number, concat, lit
```

```
from pyspark.sql.window import Window
```

```
Inicializar o SparkSession
```

```
spark = SparkSession.builder \
```

```
 .appName("Transformação TRT") \
```

```
 .getOrCreate()
```

```
Inicializar o contexto do Glue
```

```
glueContext = GlueContext(spark.sparkContext)
```

```
spark = glueContext.spark_session
```

```
job = Job(glueContext)
```

```
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
```

```
job.init(args['JOB_NAME'], args)
```

```
source_file_movie = "s3://bucketdesafiofinal/TRT/Movies/"
```

```
source_file_tmdb = "s3://bucketdesafiofinal/TRT/TMDB/Fantasy/dt=2024-04-12/"
```

```
Carregar dados do arquivo Parquet
```

```
df_tmdb = spark.read.parquet(source_file_tmdb)
```

```
df_movie = spark.read.parquet(source_file_movie)
```

```
Caminho de saída para os dados no formato Parquet
```

```
output_path = "s3://bucketdesafiofinal/Refined2"
```

```
df_movie = df_movie.withColumnRenamed("imdb_id", "movie_imdb_id")
```

```

junção dataframes
df_geral = df_movie.join(df_tmdb, df_movie.movie_imdb_id == df_tmdb.imdb_id, 'inner')

Remover linhas duplicadas com base na coluna movie_imdb_id para garantir que seja
uma chave primária
df_geral = df_geral.dropDuplicates(["movie_imdb_id"])

Criando uma view temporária a partir do DataFrame geral
df_geral.createOrReplaceTempView('df_geral_view')

Visualizar o esquema do DataFrame df_geral
df_geral.printSchema()

Definir as consultas para as tabelas
ids_sagas = ['tt0241527', 'tt0295297', 'tt0304141', 'tt0330373', 'tt0373889', 'tt0417741',
'tt0926084', 'tt1201607', 'tt1392170', 'tt1951264', 'tt1951265', 'tt1951266', 'tt1099212',
'tt1259571', 'tt1324999', 'tt1325004', 'tt1673434']

ids_sagas_str = ", ".join([f'"{id}"' for id in ids_sagas])
df_sagas_data_query = f"""
 SELECT anoLancamento_tmdb AS Ano_Lancamento
 FROM df_geral_view
 WHERE movie_imdb_id IN ({ids_sagas_str})
 """

Montar a parte da consulta SQL para buscar os títulos das sagas específicas
ids_sagas_str = ", ".join([f'"{id}"' for id in ids_sagas])
df_sagas_titulo_query = f"""
 SELECT tituloPrincipal AS Titulo,
 movie_imdb_id as ts_imdb_id
 FROM df_geral_view
 WHERE movie_imdb_id IN ({ids_sagas_str})
 """

Montar a parte da consulta SQL para buscar os títulos das sagas específicas
ids_sagas_str = ", ".join([f'"{id}"' for id in ids_sagas])
df_sagas_adulto_query = f"""
 SELECT adulto AS Adulto,
 movie_imdb_id as as_imdb_id
 FROM df_geral_view

```

```
WHERE movie_imdb_id IN ({ids_sagas_str})
""""
```

```
Criar DataFrames usando as consultas
df_sagas_data = spark.sql(df_sagas_data_query)
df_sagas_titulo = spark.sql(df_sagas_titulo_query)
df_sagas_adulto = spark.sql(df_sagas_adulto_query)
```

```
Criar uma janela para particionar os dados por nada, resultando em uma partição única
Definir a ordem na janela
window_data = Window.orderBy("Ano_Lancamento")
window_titulo = Window.orderBy("ts_imdb_id")
window_adulto = Window.orderBy("as_imdb_id")
```

```
Adicionar uma nova coluna de ID em cada DataFrame
Adicionar uma nova coluna de ID em cada DataFrame
df_sagas_data = df_sagas_data.withColumn("id_tabela_data", concat(lit("dt"),
row_number().over(window_data)))
df_sagas_titulo = df_sagas_titulo.withColumn("id_tabela_titulo", concat(lit("ti"),
row_number().over(window_titulo)))
df_sagas_adulto = df_sagas_adulto.withColumn("id_tabela_adulto", concat(lit("ad"),
row_number().over(window_adulto)))
```

```
Converter id_adulto para string
df_sagas_data = df_sagas_data.withColumn("id_tabela_data",
col("id_tabela_data").cast("string"))
df_sagas_titulo = df_sagas_titulo.withColumn("id_tabela_titulo",
col("id_tabela_titulo").cast("string"))
df_sagas_adulto = df_sagas_adulto.withColumn("id_tabela_adulto",
col("id_tabela_adulto").cast("string"))
```

```
Montar a parte da consulta SQL para buscar os dados das sagas específicas
ids_sagas_str = ", ".join([f"{id}" for id in ids_sagas])
df_fato_sagas_query = f"""
```

```
SELECT DISTINCT
 df_geral_view.movie_imdb_id as fs_movie_imdb_id,
 orcamento,
 receita,
 tempoMinutos,
 popularidade,
```

```

 contagem_de_voto,
 votacao_media,
 data_coleta_tmdb,
 anoLancamento_tmdb
FROM df_geral_view
WHERE df_geral_view.movie_imdb_id IN ({ids_sagas_str})
"""

```

```
df_fato_sagas = spark.sql(df_fato_sagas_query)
```

```
df_fato_sagas.printSchema()
```

```

Joining and dropping redundant columns
df_fato_sagas = df_fato_sagas.join(df_sagas_data.select("id_tabela_data",
"Ano_Lancamento"),
 df_fato_sagas.anoLancamento_tmdb ==
df_sagas_data.Ano_Lancamento, "inner")
#df_fato_sagas = df_fato_sagas.drop("anoLancamento_tmdb", "Ano_Lancamento")

```

```

df_fato_sagas = df_fato_sagas.join(df_sagas_titulo.select('id_tabela_titulo', 'ts_imdb_id'),
 df_fato_sagas.fs_movie_imdb_id==df_sagas_titulo.ts_imdb_id, 'inner')
#df_fato_sagas = df_fato_sagas.drop('ts_imdb_id')

```

```

df_fato_sagas = df_fato_sagas.join(df_sagas_adulto.select('id_tabela_adulto',
'as_imdb_id'),
 df_fato_sagas.fs_movie_imdb_id==df_sagas_adulto.as_imdb_id, 'inner')
#df_fato = df_fato.drop('adulto', 'Adulto')

```

```

Remover colunas redundantes após o join
df_fato_sagas = df_fato_sagas.drop("anoLancamento_tmdb", "Ano_Lancamento",
"adulto", "ts_imdb_id", "as_imdb_id")

```

```

Eliminar linhas duplicadas
df_fato_sagas = df_fato_sagas.dropDuplicates(["fs_movie_imdb_id"])

```

```

df_fato_sagas.orderBy('fs_movie_imdb_id').show
df_fato_sagas.printSchema()

```

```
Escrever DataFrames como tabelas no AWS Glue
```

```
def salvar_tabela(df, tabela_nome):
```

```
 df.write.format("parquet") \
```

```
 .mode("overwrite") \
```

```
 .save(output_path + "/" + tabela_nome)
```

```
Salvando as tabelas
```

```
salvar_tabela(df_fato_sagas, "tabela_fato_sagas")
```

```
salvar_tabela(df_sagas_data, "tabela_sagas_data")
```

```
salvar_tabela(df_sagas_titulo, "tabela_sagas_titulo")
```

```
salvar_tabela(df_sagas_adulto, "tabela_sagas_adulto")
```

```
Encerrar a sessão do Spark
```

```
spark.stop()
```

```
job.commit()
```

```
...
```

Criada as tabelas fatos e dimensões é necessário rodar o crawler, que se encontra na página do glue na lateral esquerda.

O serviço AWS Glue Crawler é uma ferramenta que automatiza a descoberta e classificação de metadados em diversas fontes de dados. Ele é especialmente útil em ambientes de Big Data e Data Lakes, onde há uma grande variedade de fontes de dados e os metadados precisam ser gerenciados de forma eficiente.

Os metadados coletados pelo Crawler são armazenados no AWS Glue Data Catalog, um catálogo centralizado de metadados que pode ser compartilhado e utilizado por outras ferramentas e serviços da AWS, e é com ele que vamos carregar os dados no serviço da QuickSight para a elaboração dos gráficos.

Imagens da criação do crawlers:

Step 1

**Set crawler properties**

Step 2

[Choose data sources and classifiers](#)

Step 3

[Configure security settings](#)

Step 4

[Set output and scheduling](#)

Step 5

[Review and update](#)

## Set crawler properties

**Crawler details** [Info](#)

Name

df\_crawler-copy

Name can be up to 255 characters long. Some character set including control characters are prohibited.

Description - *optional*

Enter a description

Descriptions can be up to 2048 characters long.

**► Tags - optional**

Use tags to organize and identify your resources.

Cancel

Next

Step 1

[Set crawler properties](#)

Step 2

**Choose data sources and classifiers**

Step 3

[Configure security settings](#)

Step 4

[Set output and scheduling](#)

Step 5

[Review and update](#)

## Choose data sources and classifiers

**Data source configuration**

Is your data already mapped to Glue tables?

☒ Not yet

Select one or more data sources to be crawled.

☐ Yes

Select existing tables from your Glue Data Catalog.

**Data sources (8)** [Info](#)

The list of data sources to be scanned by the crawler.

Edit

Remove

Add a data source

	Type	Data source	Parameters
<input type="radio"/>	S3	s3://bucketdesafiofinal/Refined3/tabela_data/	Recrawl all
<input type="radio"/>	S3	s3://bucketdesafiofinal/Refined3/tabela_fato_sagas/	Recrawl all
<input type="radio"/>	S3	s3://bucketdesafiofinal/Refined3/tabela_fato/	Recrawl all
<input type="radio"/>	S3	s3://bucketdesafiofinal/Refined3/tabela_sagas_adulto/	Recrawl all
<input type="radio"/>	S3	s3://bucketdesafiofinal/Refined3/tabela_sagas_data/	Recrawl all
<input type="radio"/>	S3	s3://bucketdesafiofinal/Refined3/tabela_sagas_titulo/	Recrawl all
<input type="radio"/>	S3	s3://bucketdesafiofinal/Refined3/tabela_titulo/	Recrawl all
<input type="radio"/>	S3	s3://bucketdesafiofinal/Refined3/tabela_adulto/	Recrawl all

**► Custom classifiers - optional**

A classifier checks whether a given file is in a format the crawler can handle. If it is, the classifier creates a schema in the form of a StructType object that matches that data format.

Cancel

Previous

Next

[AWS Glue](#) > [Crawlers](#) > Edit crawler

Step 1  
[Set crawler properties](#)

Step 2  
[Choose data sources and classifiers](#)

Step 3  
**Configure security settings**

Step 4  
[Set output and scheduling](#)

Step 5  
[Review and update](#)

## Configure security settings

**IAM role** [Info](#)

Existing IAM role  

AWSGlueServiceRole

▼

↺

View [↗](#)

Create new IAM role

Update chosen IAM role

Only IAM roles created by the AWS Glue console and have the prefix "AWSGlueServiceRole-" can be updated.

**Lake Formation configuration - optional**

Allow the crawler to use Lake Formation credentials for crawling the data source. [Learn more.](#) [↗](#)

☐ Use Lake Formation credentials for crawling S3 data source

Checking this box will allow the crawler to use Lake Formation credentials for crawling the data source. If the data source is registered in another account, you must provide the registered account ID. Otherwise, the crawler will crawl only those data sources associated to the account. Only applicable to S3, Glue Catalog, Iceberg, and Hudi data sources.

**► Security configuration - optional**

Enable at-rest encryption with a security configuration.

Cancel

Previous

Next

[AWS Glue](#) > [Crawlers](#) > Edit crawler

Step 1  
[Set crawler properties](#)

Step 2  
[Choose data sources and classifiers](#)

Step 3  
[Configure security settings](#)

Step 4  
**Set output and scheduling**

Step 5  
[Review and update](#)

## Set output and scheduling

**Output configuration** [Info](#)

Target database  

dfdfdf

▼

↺

Clear selection

Add database [↗](#)

Table name prefix - optional

Type a prefix added to table names

Maximum table threshold - optional

This field sets the maximum number of tables the crawler is allowed to generate. In the event that this number is surpassed, the crawl will fail with an error. If not set, the crawler will automatically generate the number of tables depending on the data schema.

Type a number greater than 0

**▼ Advanced options**

S3 schema grouping

☒ Create a single schema for each S3 path

By default, when a crawler defines tables for data stored in S3, it considers both data compatibility and schema similarity. Select this check box to group compatible schemas into a single table definition across all S3 objects under the provided include path. Other criteria will still be considered to determine proper grouping.

Table level - optional

Nesta última tela é necessário criar um database, ou já ter criado antes, e nas opções avançadas selecionar o “Crie um único esquema para cada caminho S3”. Criado o Crawler selecione a opção de rodar “run” na tela inicial.

[AWS Glue](#) > Crawlers

## Crawlers

A crawler connects to a data store, progresses through a prioritized list of classifiers to determine the schema for your data, and then creates metadata tables in your data catalog.

**Crawlers (1/2) Info**
Last updated (UTC)  
April 26, 2024 at 17:43:47
Refresh
Action ▾
Run
Create crawler

View and manage all available crawlers.

Q Filter crawlers

<input type="checkbox"/>	Name ▾	State ▾	Schedule	Last run ▾	Last run time... ▾	Log	Table changes fr...
<input type="checkbox"/>	<a href="#">df_crawler</a>	Ready		Succeeded	April 22, 2024 at ...	<a href="#">View log</a>	8 created
<input checked="" type="checkbox"/>	<a href="#">df_crawler-copy</a>	Ready		Succeeded	April 24, 2024 at ...	<a href="#">View log</a>	8 created

Depois verifique se deu certo a criação das tabelas na parte de DataBases e Tables, ainda na página do glue no canto esquerdo.

[AWS Glue](#) > Tables

## Tables

A table is the metadata definition that represents your data, including its schema. A table can be used as a source or target in a job definition.

**Tables (8)**
Last updated (UTC)  
April 26, 2024 at 17:41:22
Refresh
Delete
Add tables using crawler
Add table

View and manage all available tables.

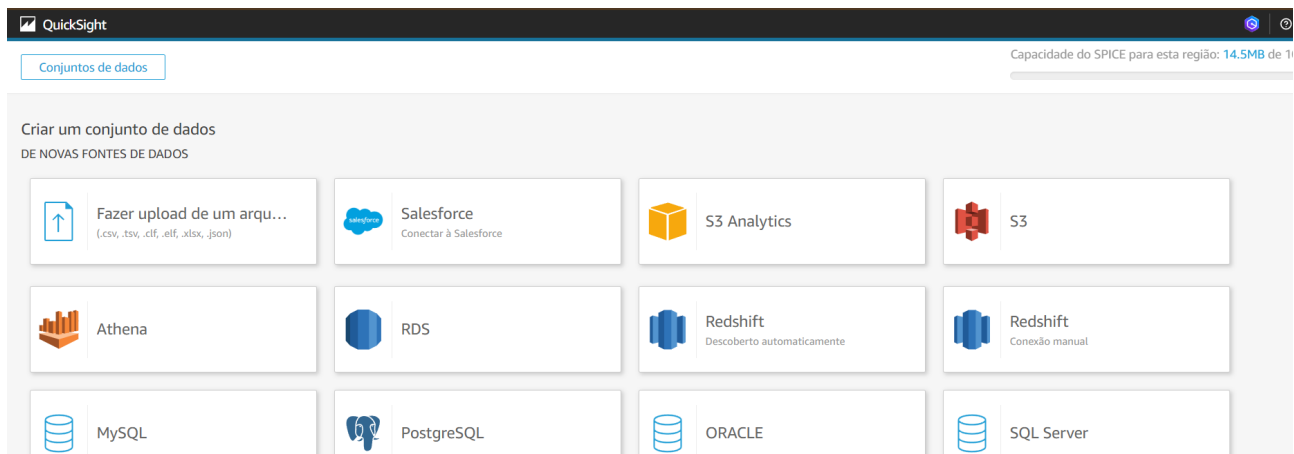
Q Filter tables

<input type="checkbox"/>	Name ▲	Database ▾	Location ▾	Classification ▾	Deprecated ▾	View data	Data quality
<input type="checkbox"/>	tabela_adulto	dfdfdf	s3://bucketdesafiofina	Parquet	-	<a href="#">Table data</a>	<a href="#">View data quality</a>
<input type="checkbox"/>	tabela_data	dfdfdf	s3://bucketdesafiofina	Parquet	-	<a href="#">Table data</a>	<a href="#">View data quality</a>
<input type="checkbox"/>	tabela_fato	dfdfdf	s3://bucketdesafiofina	Parquet	-	<a href="#">Table data</a>	<a href="#">View data quality</a>
<input type="checkbox"/>	tabela_fato_sagas	dfdfdf	s3://bucketdesafiofina	Parquet	-	<a href="#">Table data</a>	<a href="#">View data quality</a>
<input type="checkbox"/>	tabela_sagas_adulto	dfdfdf	s3://bucketdesafiofina	Parquet	-	<a href="#">Table data</a>	<a href="#">View data quality</a>
<input type="checkbox"/>	tabela_sagas_data	dfdfdf	s3://bucketdesafiofina	Parquet	-	<a href="#">Table data</a>	<a href="#">View data quality</a>
<input type="checkbox"/>	tabela_sagas_titulo	dfdfdf	s3://bucketdesafiofina	Parquet	-	<a href="#">Table data</a>	<a href="#">View data quality</a>
<input type="checkbox"/>	tabela_titulo	dfdfdf	s3://bucketdesafiofina	Parquet	-	<a href="#">Table data</a>	<a href="#">View data quality</a>

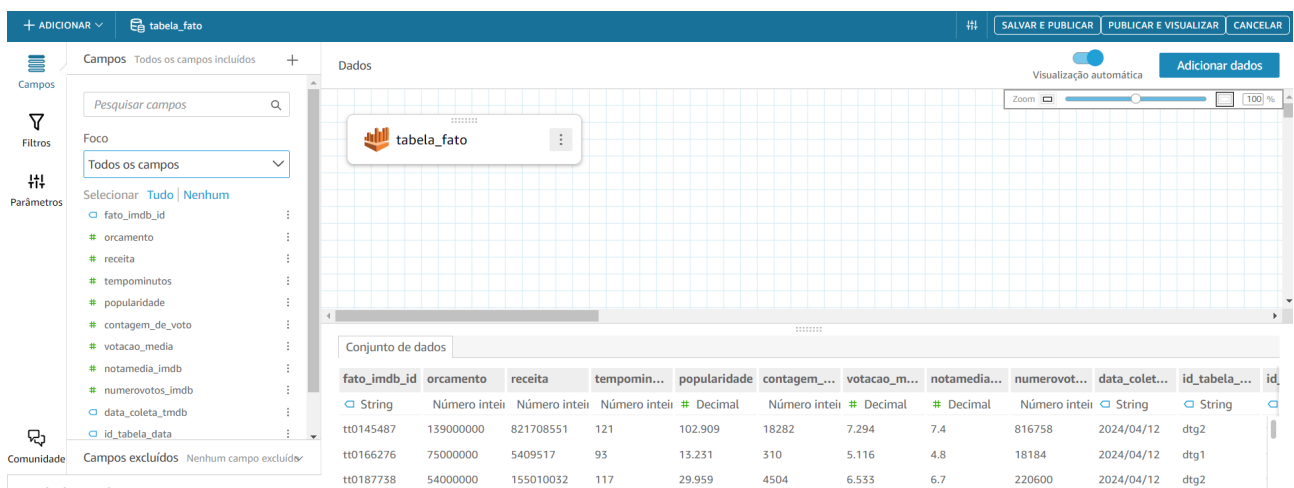
**QUINTO PASSO.** Configuração do QuickSight no AWS. Escolha a versão Standart que está no final da janela de aviso, em uma linha pequena, esta é a versão gratuita.

Dentro do QuickSight escolha “novo conjunto de dados” no canto superior direito. Selecione “Athena”.





Entregue um nome para seu conjunto de dados e na próxima página selecione o database criado no momento de definição do crawler, após selecione a sua tabela fato e “editar/pré-visualizar dados”.



Você será capaz de observar os nomes das colunas da tabela a esquerda e elementos da tabela em baixo. Mas é necessário ainda adicionar as demais tabelas, as dimensões. Então encontre o botão “adicionar dados” no canto superior direito., em fonte de dados selecione o nome do conjunto de dados e selecione as demais tabelas.

Dados

Visualização automática

Adicionar dados

Zoom 100 %

Configuração de junção

Juntar cláusulas

+ Adicionar uma nova cláusula de junção

Tipo de junção

Inner Left Right Full

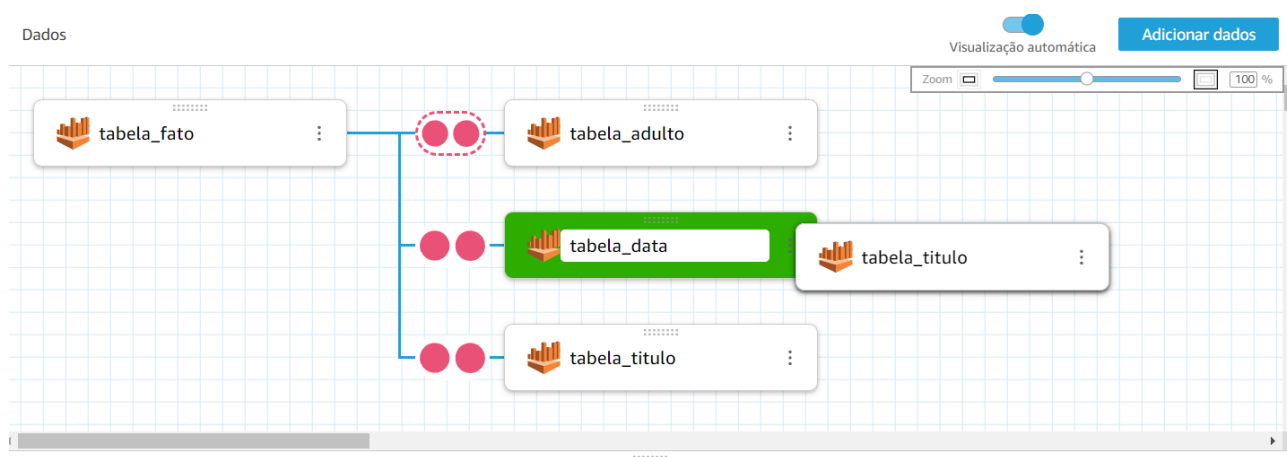
tabela\_fato

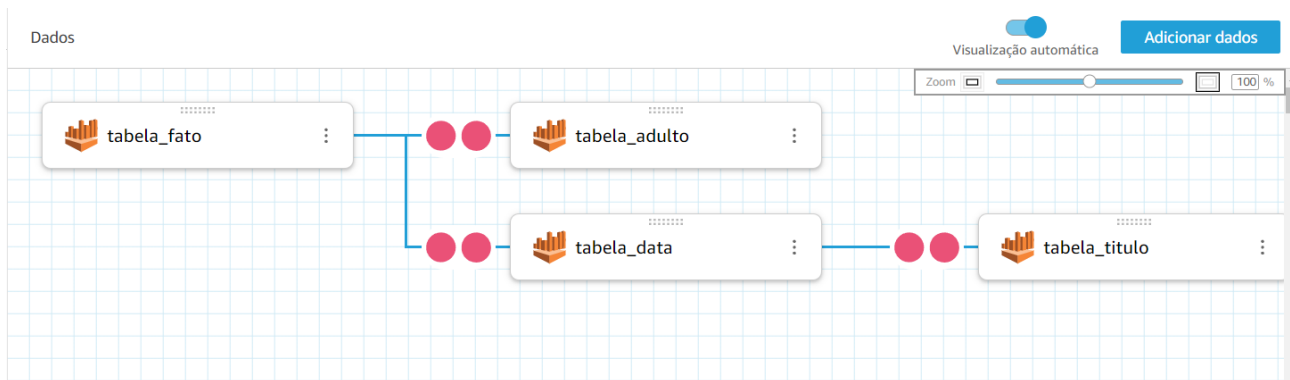
tabela\_adulto

id\_tabela\_adulto = adulto\_imdb\_id

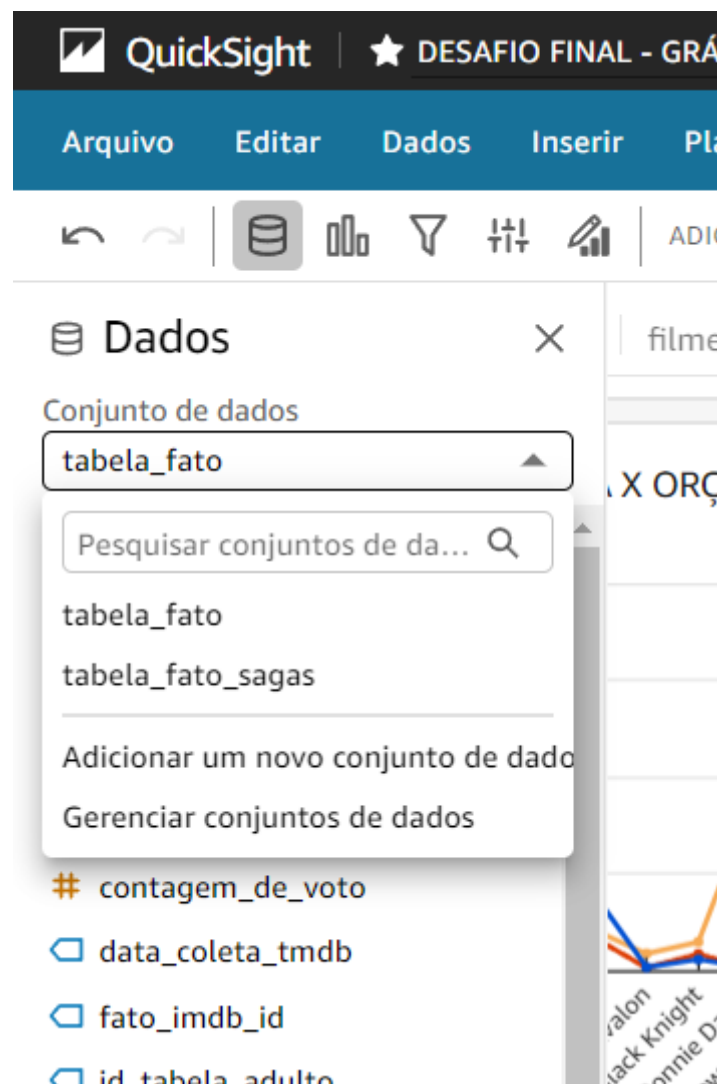
Para cada tabela escolha o tipo de join e qual a cláusula de junção, apertando as imagens das bolinhas.

Se você tiver um diagrama no estilo estrela, carregue todas as tabelas e depois arraste ela para as respectivas tabelas selecionando a parte pontilhada acima do nome. Fiz um como exemplo:






Tudo finalizado, aparecerá no canto superior direito a opção “Salvar e Publicar” e “Publicar e Visualizar”, aperte cada um nessa ordem. Após isso abra o meu conjunto de dados geral e no canto esquerdo pedi para adicionar o conjunto das sagas, que já havia preparado.



Vou conseguir, portanto, utilizar os dois conjuntos numa página de gráficos. Em campo calculado realizei funções de media de nota, receita, orçamento e porcentagem de lucro para utilizar nos gráficos.

As funções são bem intuitivas, lembra o excel, e possui uma explicação da sintaxe no canto inferior direito.

média filmes por ano 

```
1 distinct_count(título) / distinct_count({Ano_Lançamento})
2
```

Cancelar Salvar

Campos

Parâmetros

Funções

Funções de pesquisa

difference

distinctCountOver

distinct\_count

distinct\_countIf

endsWith

epochDate

como uma função agregada LAC.

SINTAXE

distinct\_count(expression)

SINTAXE PARA LAC

distinct\_count(expression, [calculation\_dimension])

**SEXTO PASSO.** Vou apresentar os gráficos, que fiz pra minha apresentação.

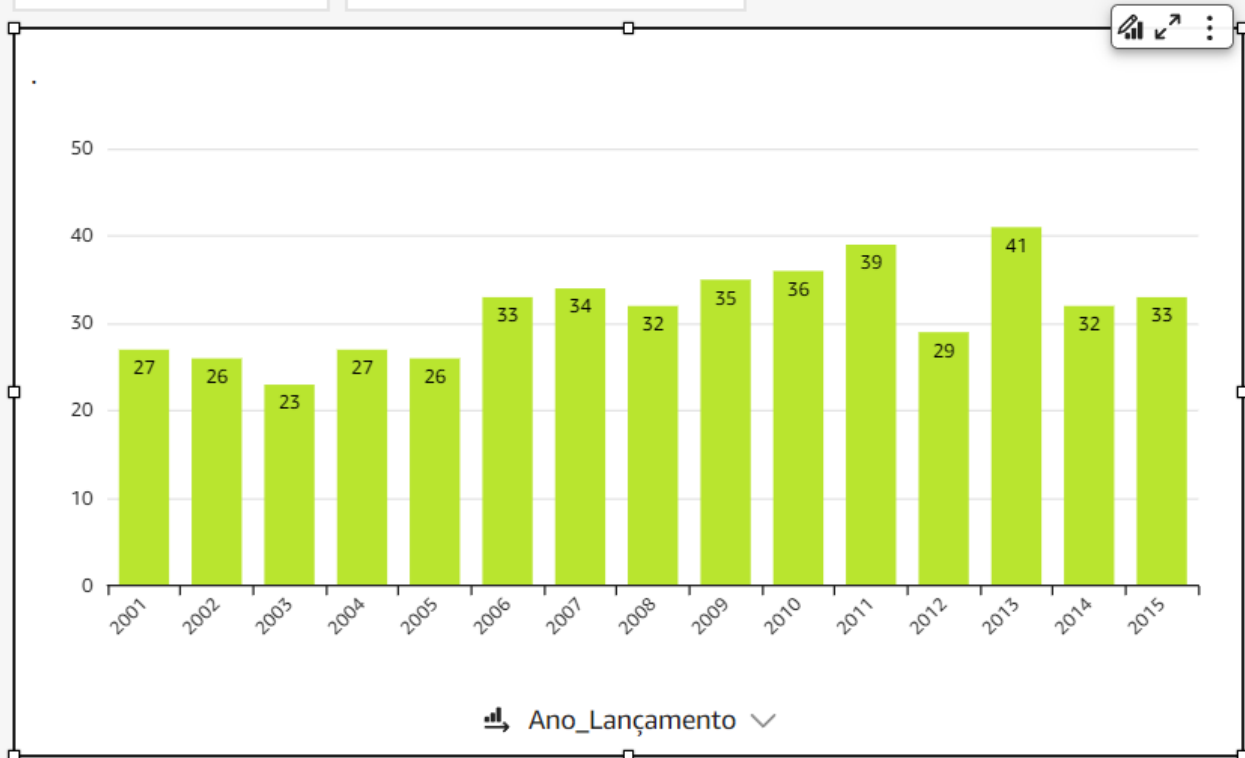
1º – Demonstrei o número total de filmes no geral, média por ano e lista de filmes das sagas.

Quantidade Total

473

Média Filmes Por Ano

31.4



## FILMES SAGAS

Harry Potter and the Sorcerer's Stone	2001
Harry Potter and the Chamber of Secrets	2002
Harry Potter and the Prisoner of Azkaban	2004
Harry Potter and the Goblet of Fire	2005
Harry Potter and the Order of the Phoenix	2007
Twilight	2008
Harry Potter and the Half-Blood Prince	2009
The Twilight Saga: New Moon	2009
Harry Potter and the Deathly Hallows: Part 1	2010
The Twilight Saga: Eclipse	2010
Harry Potter and the Deathly Hallows: Part 2	2011
The Twilight Saga: Breaking Dawn - Part 1	2011
The Hunger Games	2012
The Twilight Saga: Breaking Dawn - Part 2	2012
The Hunger Games: Catching Fire	2013
The Hunger Games: Mockingjay - Part 1	2014
The Hunger Games: Mockingjay - Part 2	2015

2º – Fiz uma correlação de popularidade e nota média dos filmes gerais e das sagas, sendo que nos gerais ao selecionar um dos itens o nome do filme aparecia na nuvem em baixo, para melhor visualização.

- tt0035423
- tt0120667
- tt0120804
- tt0121765
- tt0121766
- tt0133152
- tt0145487
- tt0146316
- tt0160399
- tt0163025
- tt0166276
- tt0167190

[illegible]

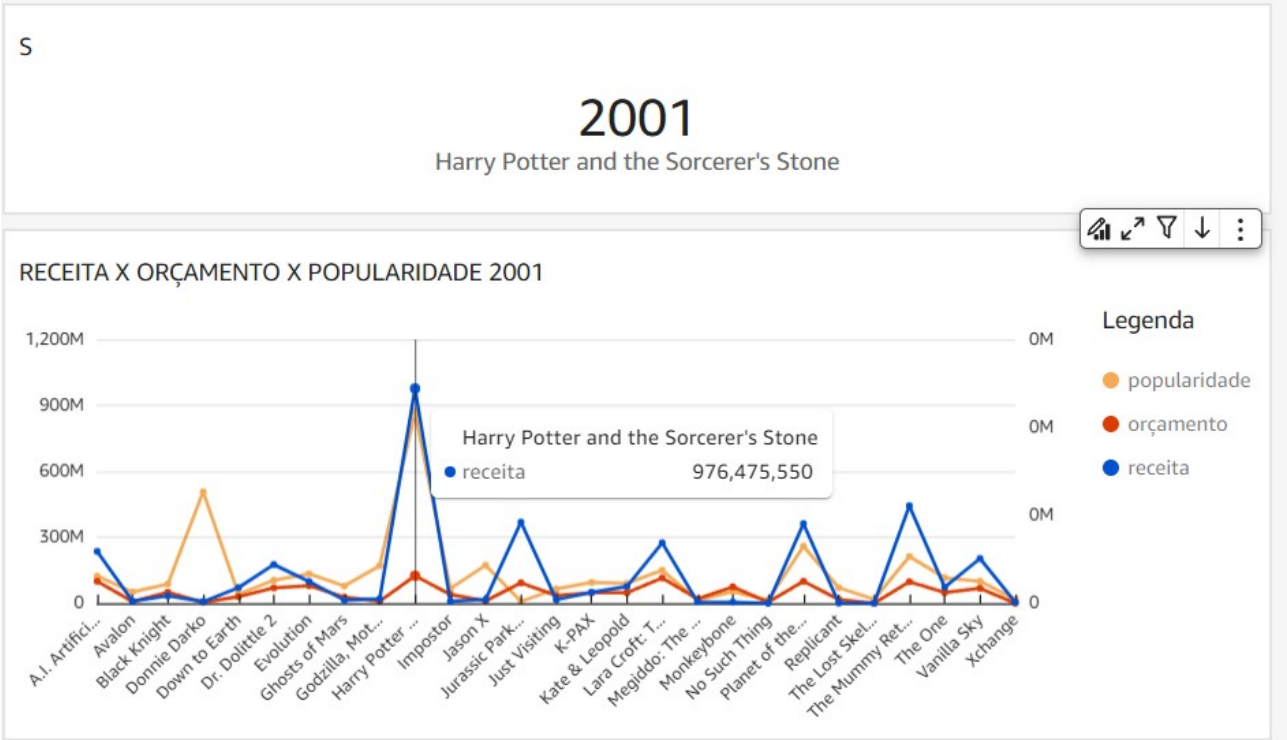
A scatter plot showing the relationship between 'votacao\_media' (media vote) on the x-axis and 'popularidade' (popularity) on the y-axis. The x-axis ranges from 0 to 250, and the y-axis ranges from 0 to 10. Data points are colored circles representing different movies. A tooltip for 'The Twilight Saga: New Moon' is displayed, showing its media vote of 5.99, title, and popularity of 81.85.

Movie Title	votacao_media	popularidade
The Twilight Saga: New Moon	5.99	81.85
Other Movies (approximate values)	35, 65, 85, 95, 105, 115, 125, 135, 185, 215	6.0, 6.2, 6.3, 6.5, 6.8, 7.2, 7.4, 7.6, 7.7, 7.9

- Harry Pott...
- Harry Pott...
- Harry Pott...
- Harry Pott...
- Harry Pott...
- Harry Pott...
- Harry Pott...
- Harry Pott...
- The Hunge...
- The Hunge...
- The Hunge...
- The Hunge...

3º – Utilizando as funções de média de receita, orçamento e  
 e um gráfico de cada ano (de 2001 a 2015), demonstrando os

orçamentos parecidos, porém a diferença gritante de popularidade, o que aumenta a venda de produtos relacionados ao título.



4º – Com relação as sagas estudei e entreguei uma explicação para a variação das notas medias.

### NOTA MÉDIA HARRY POTTER

titulo	notamedia_imdb	votacao_media
Harry Potter and the Sorcerer's Stone	7.6	7.91
Harry Potter and the Chamber of Secrets	7.4	7.72
Harry Potter and the Prisoner of Azkaban	7.9	8.02
Harry Potter and the Goblet of Fire	7.7	7.81
Harry Potter and the Order of the Phoenix	7.5	7.68
Harry Potter and the Half-Blood Prince	7.6	7.69
Harry Potter and the Deathly Hallows: Part 1	7.7	7.76
Harry Potter and the Deathly Hallows: Part 2	8.1	8.1



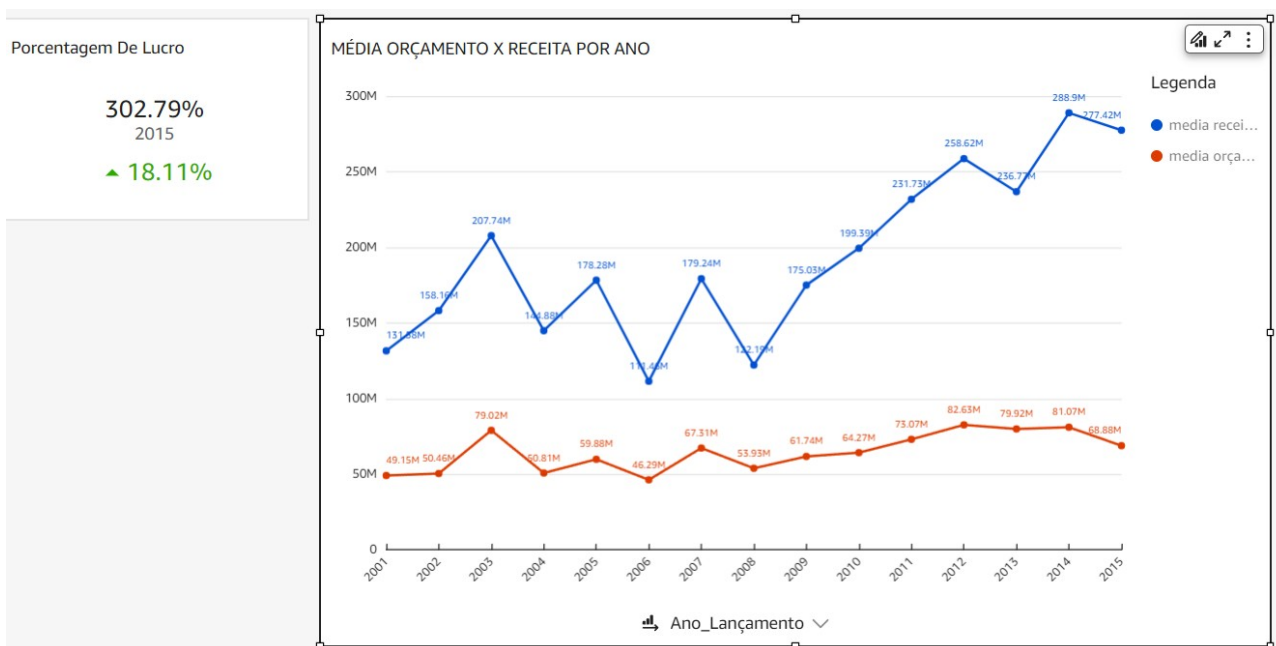
## NOTA MÉDIA CREPÚSCULO

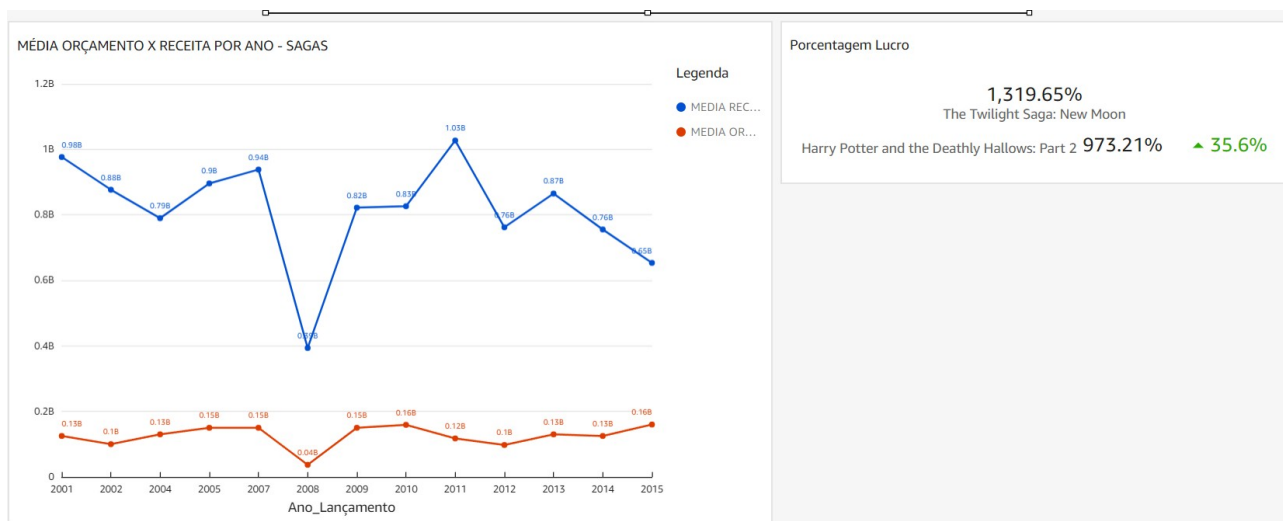
titulo	notamedia_imdb	votacao_media
Twilight	5.3	6.29
The Twilight Saga: New Moon	4.7	5.99
The Twilight Saga: Eclipse	5	6.21
The Twilight Saga: Breaking Dawn - Part 1	4.9	6.2
The Twilight Saga: Breaking Dawn - Part 2	5.5	6.46

## NOTA MÉDIA JOGOS VORAZES

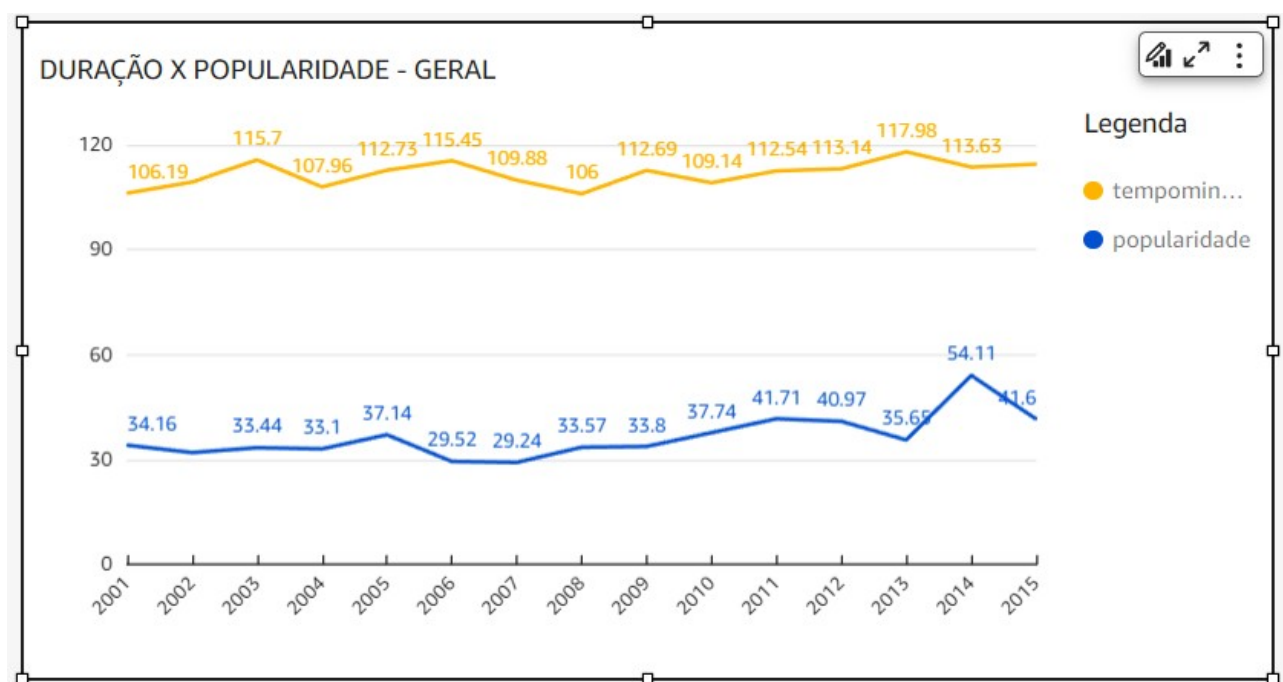
titulo	notamedia_imdb	votacao_media
The Hunger Games	7.2	7.21
The Hunger Games: Catching Fire	7.5	7.43
The Hunger Games: Mockingjay - Part 1	6.6	6.81
The Hunger Games: Mockingjay - Part 2	6.5	6.9

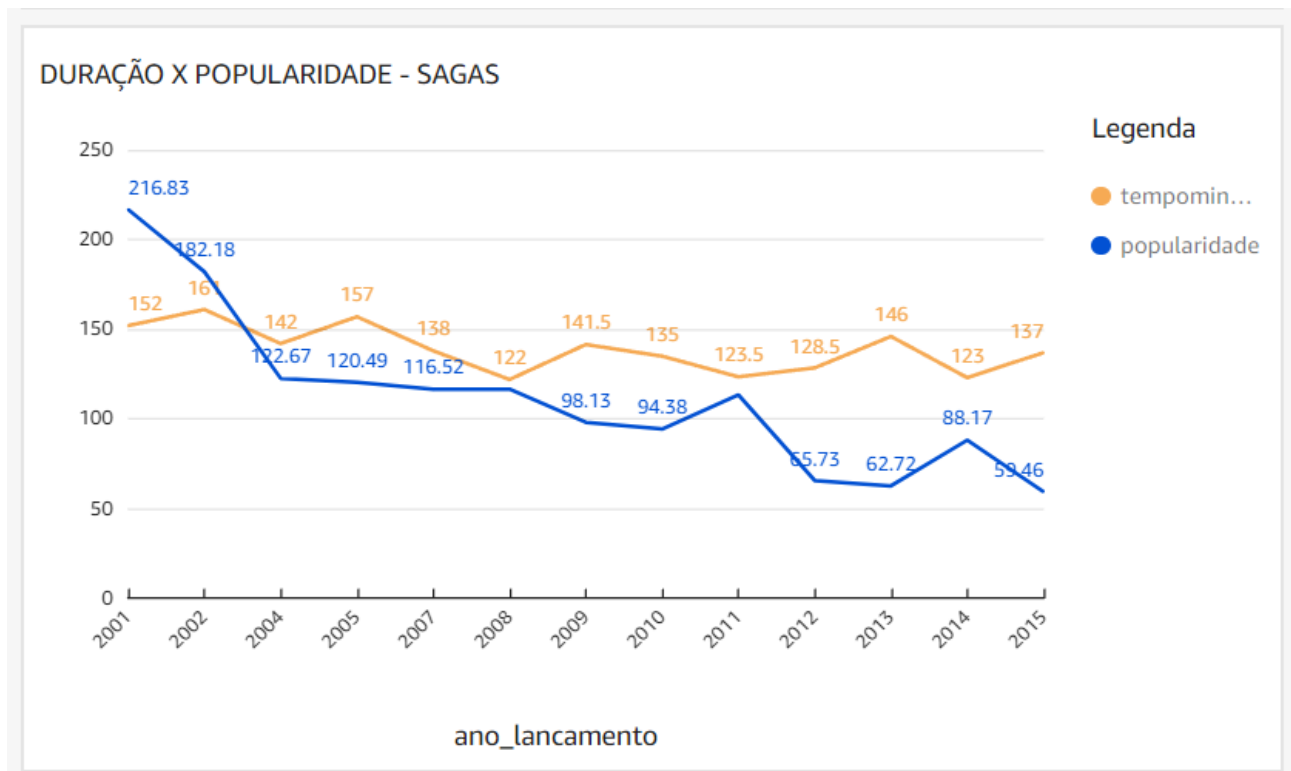
5º – Fiz um gráfico da media de orçamento e receita, sendo que para cada ano selecionado aparece a porcentagem de lucro, realizei isso com o geral e as sagas.





6º – Por fim, analisei a questão da popularidade e o tempo de duração dos filmes, realizei isso com o geral e as sagas.





Este foi projeto final, espero que algum ponto do texto ajude outras pessoas.