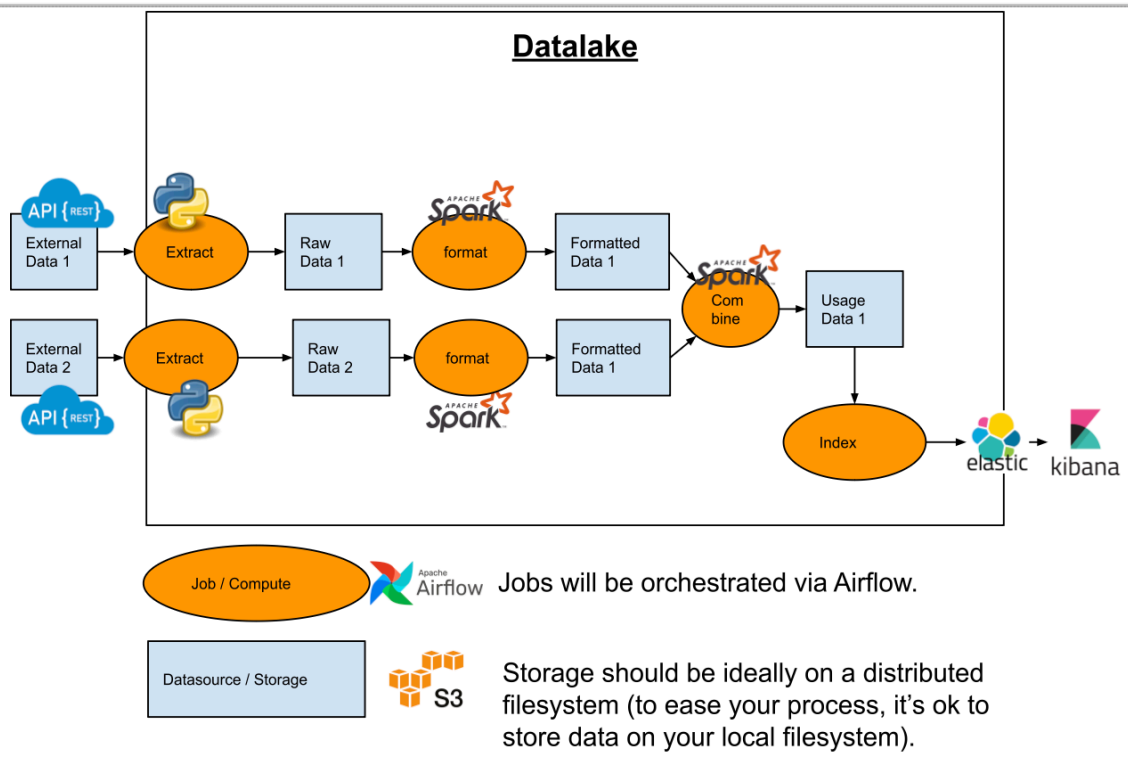
Big Data Project

Team : Abishek Thamizharasan(62733) and Megha Sivasankar(62775)

Architecture

What we did :

1. Create jobs that extract data from the Internet via REST API.
2. Create jobs that format / normalize these data to transform them into a format suitable to a Datalake / data sharing / data analysis.
3. Create a job that joins / combines your different data sources into a final interesting usage.
4. Index your data in a dashboard to expose your final output to end users.



Data Sources and Data Extraction

We have chosen cinema as our theme, There are lot of sources for cinema but we have chosen TMDb api and OMDb api for our DataLake.

Collecting APIs from TMDB (The Movie Database) and OMDB (Open Movie Database) is an excellent choice for your cinema-themed project. These APIs can provide you with valuable data related to movies, including details about films, ratings, cast and crew, genres, release dates, and more.

```python
from datetime import datetime, timedelta
import requests
import json
import logging

def collect_tmdb_data():
    try:
        logging.info("Starting TMDB data collection")
        # Fetch movie data from TMDB API for 2023 and 2022
        tmdb_api_key = 'bfd1fa0b47958850ad4d9f1606f1c15a'
        current_year = datetime.now().year
        years = [current_year - 1, current_year]  # Collect movies from previous year and current year

        movie_data = []
        for year in years:
            tmdb_url = f'https://api.themoviedb.org/3/discover/movie?api_key={tmdb_api_key}&primary_release_year={year}'
            tmdb_response = requests.get(tmdb_url)
            tmdb_response.raise_for_status()  # Raises an exception if the request fails
            tmdb_year_data = tmdb_response.json()

            # Iterate over movies and fetch IMDb ID for each movie
            for movie in tmdb_year_data['results']:
                movie_id = movie['id']
                imdb_url = f'https://api.themoviedb.org/3/movie/{movie_id}/external_ids?api_key={tmdb_api_key}'
                imdb_response = requests.get(imdb_url)
                imdb_response.raise_for_status()  # Raises an exception if the request fails
                imdb_data = imdb_response.json()
                imdb_id = imdb_data.get('imdb_id', None)

                movie['imdb_id'] = imdb_id
                movie_data.append(movie)

        # Save TMDB data in a local file
        tmdb_file_path = '/Users/abishek/datalake/tmdb_movie.json'
        with open(tmdb_file_path, 'w') as file:
            json.dump(movie_data, file)

        logging.info("Finished TMDB data collection")

    except Exception as e:
        logging.error(f"Error occurred: {e}")


def collect_omdb_data():
    try:
        logging.info("Starting OMDB data collection")
        # Fetch movie data from OMDB API for 2023 and 2022
        omdb_api_key = '2f33835b'
        current_year = datetime.now().year
        years = [current_year - 1, current_year]  # Collect movies from previous year and current year

        movie_data = []
        for year in years:
            omdb_url = f'http://www.omdbapi.com/?apikey={omdb_api_key}&s=movie&y={year}&page=1'
            omdb_response = requests.get(omdb_url)
            omdb_response.raise_for_status()  # Raises an exception if the request fails
            omdb_year_data = omdb_response.json()
            total_pages = (int(omdb_year_data['totalResults']) // 10) + 1  # Each page contains 10 results

            # Collect data for all pages
            for page in range(1, total_pages + 1):
                omdb_url = f'http://www.omdbapi.com/?apikey={omdb_api_key}&s=movie&y={year}&page={page}'
                omdb_response = requests.get(omdb_url)
                omdb_response.raise_for_status()  # Raises an exception if the request fails
                omdb_page_data = omdb_response.json()
                movie_data.extend(omdb_page_data['Search'])

        # Save OMDB data in a local file
        omdb_file_path = '/Users/abishek/datalake/omdb_movie.json'
        with open(omdb_file_path, 'w') as file:
            json.dump(movie_data, file)

        logging.info("Finished OMDB data collection")

    except Exception as e:
        logging.error(f"Error occurred: {e}")


# Run the tasks
collect_tmdb_data()
collect_omdb_data()

```

First we have created accounts in TMDb and OMDb websites and got the our own api key's. And we can use this code to get the data from the TMDb and OMDb.

But since there are lot of movies we have made a decision to just pick the movies from 2022 and this year 2023. And also our code can display the error if an error occurs while collecting the data.Since both api allows only limited number of request per day, we have also added exception handling if one of source has it limit reached.

Finally we save our two file in json format.

And we create a dag called cinema_collection in order to did you later once a day.

```python
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.bash import BashOperator

default_args = {
    'owner': 'Abishek Thamizharasan',
    'start_date': datetime(2023, 6, 9),
    'retries': 3,
    'retry_delay': timedelta(minutes=5),
}

with DAG('cinema_collection_dag', default_args=default_args, schedule_interval='@daily') as dag:
    run_movie_script = BashOperator(
        task_id='run_movie_script',
        bash_command='python /Users/abishek/airflow/dags/scripts/data_collection.py',
        dag=dag
    )

```

By inspecting the json file, here's a brief overview of the content inside tmdb_movie.json: The file contains a list of movies, where each movie is represented as a dictionary. Each dictionary has the following keys:

- "adult": A boolean indicating whether the movie is adult content or not.
- "backdrop_path": The relative URL of the backdrop image.
- "genre_ids": A list of integers representing the genres of the movie.
- "id": The TMDB ID of the movie.
- "original_language": The original language of the movie.
- "original_title": The original title of the movie.
- "overview": The overview of the movie's plot.
- "popularity": The popularity score of the movie.
- "poster_path": The relative URL of the poster image.
- "release_date": The release date of the movie.
- "title": The title of the movie.
- "video": A boolean indicating whether a video is available for the movie or not.
- "vote_average": The average rating of the movie.
- "vote_count": The number of votes for the movie.

Open Movie Database (OMDb.json):

  - Title: The title of the movie.
  - Year: The release year of the movie.
  - imdbID: The unique identifier of the movie in the IMDb database.
  - Type: The type of the media, in this case, all are "movie".
  - Poster: The URL to the movie's poster image.
- totalResults: Total number of results that match the search criteria. In this case, it's 5581.
- Response: Indicates if the search operation was successful or not. In this case, it's "True", meaning the operation was successful.

Data Transformation

In the context of the data you provided from omdb_movie.json and TMDb.json, let's go through the steps for data transformation and cleaning:

- Select Important Fields: For many applications, the important fields are likely to be Title, Year, imdbID, and Poster. The Type field might not be necessary in this context since all entries are movies.
- Handle Missing Data: The data you provided seems to be quite clean, but in a real-world scenario, there might be cases where some fields are missing. For instance, if the Poster link is missing, you might want to provide a default image. For missing Year, you could either omit the movie from the dataset or set the year as 'Unknown'.
- Data Type Conversion: The Year field is a string but for some applications, you might need it to be an integer for sorting or comparison purposes. You would need to convert this string to an integer.
- Normalization: In this dataset, normalization may not be required since all the fields are either strings or discrete values that don't have a wide range. Normalization is typically used for numerical data that has varying scales.
- Aggregation: In this case, aggregation doesn't seem to be required as each entry in the dataset represents an individual movie. However, if you had additional data, like box office revenue, you could aggregate movies by year to see total revenues per year.

And finally we went though the steps of :

- Created a Spark Session to handle data in Spark.
- Converted loaded JSON to Spark DataFrame directly.
- For TMDB data, converted the release_date to UTC timestamp.
- For OMDB data, converted the Year to integer type.
- Replaced the missing values with respective default values.
- Wrote the cleaned data to parquet files.

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, to_timestamp
from pyspark.sql.types import StringType

# Create Spark Session
spark = SparkSession.builder.appName("DataTransformation").getOrCreate()

# Load TMDB JSON data
tmdb_json_file_path = "/Users/abishek/datalake/tmdb_movie.json"
tmdb_schema = spark.read.json(tmdb_json_file_path).schema
tmdb_df = spark.read.schema(tmdb_schema).json(tmdb_json_file_path)

# Load OMDB JSON data
omdb_json_file_path = "/Users/abishek/datalake/omdb_movie.json"
omdb_schema = spark.read.json(omdb_json_file_path).schema
omdb_df = spark.read.schema(omdb_schema).json(omdb_json_file_path)

# Formatting and transformation for TMDB data
tmdb_df = tmdb_df.withColumn("release_date", to_timestamp(col("release_date"), "yyyy-MM-dd"))
tmdb_df = tmdb_df.withColumn("release_date", col("release_date").cast("timestamp"))
tmdb_df = tmdb_df.withColumn("release_date", col("release_date").cast("long"))
tmdb_df = tmdb_df.withColumn("release_date_utc", col("release_date") * 1000)  # Convert to UTC timestamp

# Formatting and transformation for OMDB data
omdb_df = omdb_df.withColumn("Year", col("Year").cast("integer"))

# Normalize all other column values
tmdb_columns = tmdb_df.columns
omdb_columns = omdb_df.columns

for column in tmdb_columns:
    tmdb_df = tmdb_df.withColumn(column, when(col(column).isNull(), "").otherwise(col(column).cast(StringType())))

for column in omdb_columns:
    omdb_df = omdb_df.withColumn(column, when(col(column).isNull(), "").otherwise(col(column).cast(StringType())))

# Select the required columns, including imdb_id
tmdb_df = tmdb_df.select("id", "adult", "backdrop_path", "genre_ids", "imdb_id", "original_language", "original_title",
                         "overview", "popularity", "poster_path", "release_date", "title", "video", "vote_average",
                         "vote_count", "release_date_utc")

# Write the cleaned data to Parquet files
output_path = "/Users/abishek/datalake/formatted"
tmdb_df.write.mode("overwrite").parquet(output_path + "/tmdb.parquet")
omdb_df.write.mode("overwrite").parquet(output_path + "/omdb.parquet")

# Stop Spark Session
spark.stop()
```

And saved it in a folder called formatted, Here is the code for the following, and we created a dag called cinema_data_formatting for running this script.

```python
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta

default_args = {
    'owner': 'Abishek Thamizharasan',
    'start_date': datetime(2023, 6, 11),
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

with DAG('cinema_data_formatting_dag',
         default_args=default_args,
         schedule_interval='@daily') as dag:

    run_python_task = BashOperator(
        task_id='run_python_file',
        bash_command='python /Users/abishek/airflow/dags/scripts/data_transformation.py',
        dag=dag
    )

run_python_task
```

Data Combination

In this step we need to combine the TMDb and OMDb data. Consider what you want our final dataset to look like and how we can join the data to achieve this. Apache Spark is a good tool for this.

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

spark = SparkSession.builder.getOrCreate()

# Read the cleaned TMDB and OMDB data from parquet files
tmdb_df = spark.read.parquet("/Users/abishek/datalake/formatted/tmdb.parquet")
omdb_df = spark.read.parquet("/Users/abishek/datalake/formatted/omdb.parquet")

# Perform a left outer join using imdb_id and imdbID columns
joined_df = tmdb_df.join(omdb_df, tmdb_df.imdb_id == omdb_df.imdbID, "left_outer") \
                   .select(tmdb_df["*"], omdb_df["title"].alias("omdb_title"))

# Save the joined DataFrame as a combined Parquet file
output_path = "/Users/abishek/datalake/combined.parquet"
joined_df.write.mode("overwrite").parquet(output_path)

# Stop Spark Session
spark.stop()
```

- The code imports the necessary modules: SparkSession from pyspark.sql and col from pyspark.sql.functions.
- It creates a SparkSession named "spark" using the SparkSession.builder API. This is the entry point for using Spark functionality.
- The code reads the cleaned TMDB (The Movie Database) and OMDB (Open Movie Database) data from Parquet files using the spark.read.parquet() method. The TMDB data is assigned to the variable tmdb_df and the OMDB data to omdb_df.
- It performs a left outer join between the tmdb_df and omdb_df DataFrames using the join() method. The join condition is specified as tmdb_df.imdb_id == omdb_df.imdbID, indicating that the join should be performed based on the equality of the "imdb_id" column in tmdb_df and the "imdbID" column in omdb_df. The result of the join is assigned to the variable joined_df.
- The joined DataFrame is further processed using the select() method. It selects all columns from tmdb_df using tmdb_df["*"] and renames the "title" column from omdb_df as "omdb_title" using the alias() function. The resulting DataFrame is assigned back to joined_df.
- The code specifies the output path for the combined Parquet file as output_path.
- It writes the joined DataFrame to the specified output path as a Parquet file using the write() method. The mode is set to "overwrite" to replace any existing files at the output path. The data is saved in the Parquet format using the parquet() method.
- The SparkSession is stopped using spark.stop() to release the resources.

In summary, this code loads two DataFrames from Parquet files, performs a left outer join based on a specific join condition, selects desired columns, and saves the resulting joined DataFrame as a Parquet file named as “combined.parquet”.

Finally our comined.parquet schema is like :
```
 |-- id: string (nullable = true)
 |-- adult: string (nullable = true)
 |-- backdrop_path: string (nullable = true)
 |-- genre_ids: string (nullable = true)
 |-- imdb_id: string (nullable = true)
 |-- original_language: string (nullable = true)
 |-- original_title: string (nullable = true)
 |-- overview: string (nullable = true)
 |-- popularity: string (nullable = true)
 |-- poster_path: string (nullable = true)
 |-- release_date: string (nullable = true)
 |-- title: string (nullable = true)
 |-- video: string (nullable = true)
 |-- vote_average: string (nullable = true)
 |-- vote_count: string (nullable = true)
 |-- release_date_utc: string (nullable = true)
 |-- omdb_title: string (nullable = true)
```

And finally we run this code through a dag called cinema_data_joining.

Data Uploading and Data Indexing

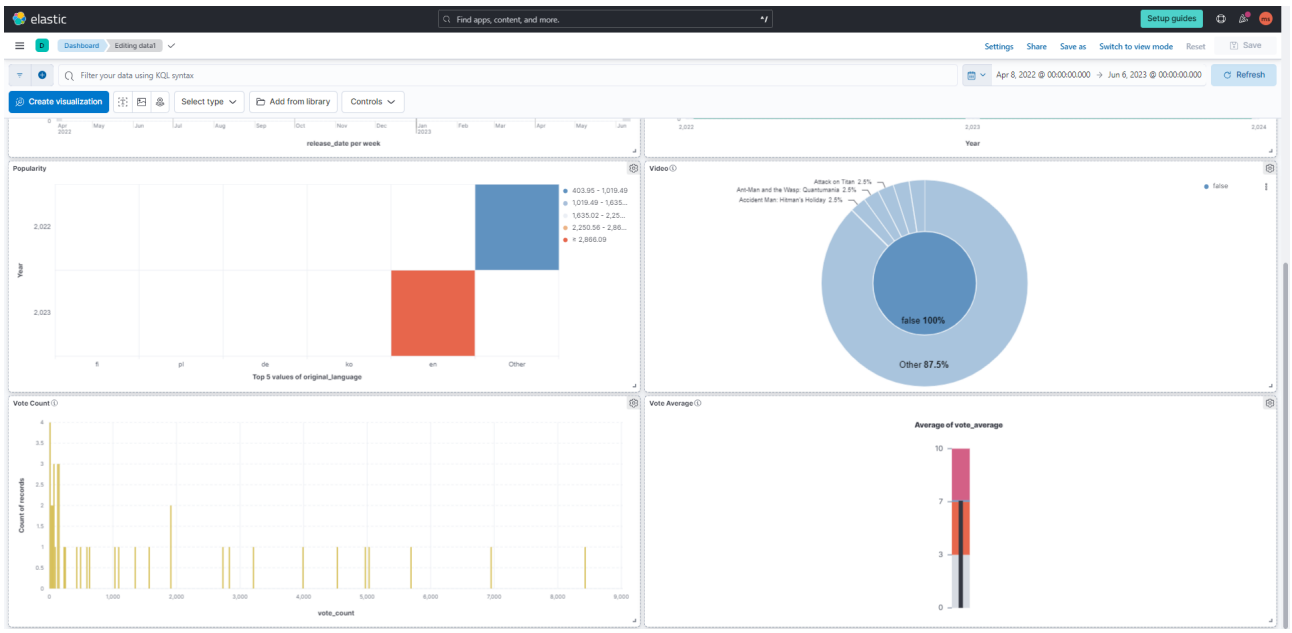
Before we can index your data, we need to create an Elasticsearch index to hold our data. The index should be designed to match the schema of our data.

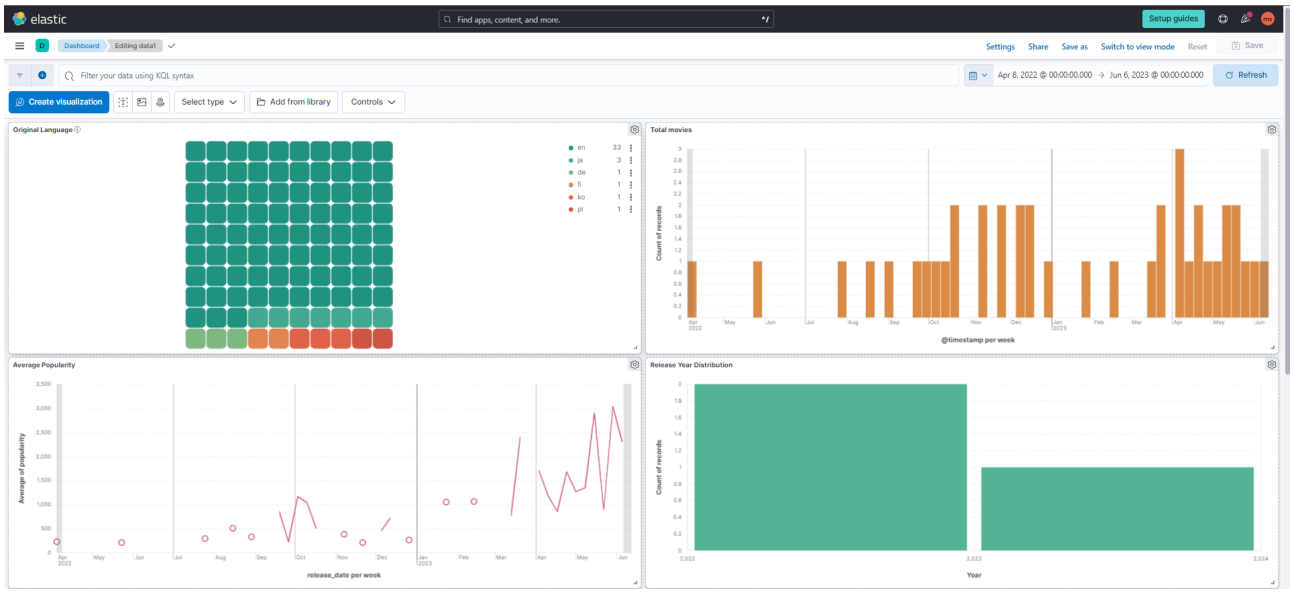
Ours look likes :

```
{
  "mappings": {
    "properties": {
      "id": { "type": "text" },
      "adult": { "type": "boolean" },
      "backdrop_path": { "type": "text" },
      "genre_ids": { "type": "keyword" },
      "imdb_id": { "type": "text" },
      "original_language": { "type": "text" },
      "original_title": { "type": "text" },
      "overview": { "type": "text" },
      "popularity": { "type": "float" },
      "poster_path": { "type": "text" },
      "release_date": { "type": "date" },
      "video": { "type": "boolean" },
      "vote_average": { "type": "float" },
      "vote_count": { "type": "integer" },
      "release_date_utc": { "type": "date" },
      "Poster": { "type": "text" },
      "Type": { "type": "text" },
      "Year": { "type": "date" },
      "imdbID": { "type": "text" }
    } }}'
```

And next we need oo index our data from the Parquet file into Elasticsearch, we need a method to convert and load our data. PySpark is a powerful tool for this job because it can read your Parquet file, and then you can use the Elasticsearch Spark connector to save your DataFrame to Elasticsearch.

Now that our data is indexed in Elasticsearch, we can use Kibana to visualize our data.





Popularity:

The heat map provides a visual representation of the median popularity values for different language-year combinations, with the horizontal axis representing the top 5 values of the original_language field and the vertical axis representing the optional field of Year.
The cell values in the heat map represent the median of popularity within specific popularity range intervals, defined as 403.95 - 1,019.49, 1,019.49 - 1,635.02, 1,635.02 - 2,250.56, 2,250.56 - 2,866.09, and 2,866.09.
Original Language:
The waffle chart visualization displays the distribution of movies based on their original language. It is grouped by the original_language field, which represents the language in which the movies were originally produced. The metric used is the count of records, which represents the number of movies available for each language. English (en) has 33 movies, Japanese (ja) has 3 movies, German (de) has 1 movie, Finnish (fi) has 1 movie, Korean (ko) has 1 movie, and Polish (pl) has 1 movie.

Total Counts:

The vertical stacked bar chart visualization is used to display the total counts of movies over time, represented by the @timestamp field. The horizontal axis is the @timestamp field, which tracks the changes in movie counts over time. The vertical axis is the count of records, which indicates the total number of movies available for each time interval. By analyzing this chart, users can observe trends and patterns in the count of movies over time, allowing them to identify periods with high or low movie counts and understand how the movie collection evolves over different time intervals.
Release year distribution:
The vertical stacked bar chart visualizes the distribution of movies based on their release years. The horizontal axis represents the release years and the vertical axis represents the count of records. Each bar represents a specific release year and the height of the bar corresponds to the total count of movies released in that year. By analyzing this chart, users can observe the distribution of movies across different release years and gain insights into the temporal pattern of movie releases.

Average Popularity:

The line chart visualization is used to display the average popularity of movies over time. It uses a horizontal axis to establish a timeline and order the movies based on their release dates, and a vertical axis to show the average popularity score calculated by averaging the popularity values of all movies released on a particular date. The chart uses a line to connect the average popularity values across different release dates, each point representing the average popularity score for a specific release date. This chart provides a visual representation of the average popularity's variation, helping to track changes and fluctuations in movie popularity over time.

Video:

The "videos" visualization in the pie chart provides an overview of the availability of videos for the movies in the dataset. It categorizes the movies into two groups: those with videos available and those without videos available. The "true" category indicates the percentage of movies with videos available, while the "false" category indicates the percentage of movies without videos available. The "Other" category accounts for the remaining movies that are not part of the top 3 values of the "original_title" field. Overall, the pie chart gives an overview of the distribution of videos availability among movies in the dataset.

Vote count:

The "vote count" visualization in a vertical bar chart provides information about the distribution of votes or ratings received by movies in the dataset. Each bar on the chart represents a range of vote counts, and the height of the bar indicates the number of movies falling within that range. By examining the distribution of vote counts, users can gain insights into the engagement and popularity of movies among viewers.

Vote average:

The "vote average" KPI in the given vertical gauge chart represents the average rating given to the movies in the dataset. The chart is visualized with a vertical gauge ranging from the minimum value (0) to the maximum value (10). The average value of the "vote_average" field is shown by the position of the gauge pointer or indicator.
The gauge provides a single value representation of the average rating, allowing you to quickly assess the overall rating of the movies. For example, if the gauge pointer is positioned at 7, it indicates that the average rating of the movies is 7 out of 10.
By monitoring the changes in the average rating over time or comparing it with other metrics, you can evaluate the overall quality or reception of the movies in terms of user ratings.

Using Cloud Storage

For this project I have decided to use Google Cloud Storage, I have created a bucket in it and I have a dag to upload the files to the bucket.

Let's break down the code:

- It imports necessary modules: datetime and timedelta from datetime, DAG from airflow, PythonOperator from airflow.operators.python, and storage from google.cloud.
- The upload_file_to_bucket() function defines the task logic. It performs the following steps:
  - Specifies the Google Cloud project ID.
  - Creates a storage client using the project ID.
  - Retrieves the specified bucket using the bucket name.
  - Specifies the file path and name of the file to be uploaded.
  - Creates a blob (object) in the bucket.
  - Uploads the file to the blob using upload_from_file().
  - Prints a message confirming the successful upload.
- It defines the default_args dictionary, which sets various properties of the DAG:
  - 'owner' specifies the owner/author of the DAG.
  - 'start_date' sets the date from which the DAG will start running.
  - 'retries' specifies the number of times the task should be retried if it fails.
  - 'retry_delay' sets the time delay between retries.
- It creates a DAG named 'upload_file_to_bucket_dag' using the DAG class. The default_args dictionary is passed as an argument, and the 'schedule_interval' parameter is set to '@daily' to schedule the DAG to run once daily.
- It creates a PythonOperator named 'upload_task' that executes the upload_file_to_bucket function.
- Finally, it adds the upload_task to the DAG, which signifies the dependency between the task and the DAG.

```python
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.python import PythonOperator
from google.cloud import storage

def upload_file_to_bucket():
    # Specify your project ID
    project_id = 'singular-vector-389317'

    # Create a storage client with the project ID
    client = storage.Client(project=project_id)

    # Get the bucket
    bucket_name = 'movies_db'
    bucket = client.get_bucket(bucket_name)

    # Specify the file path and name
    file_path = '/Users/abishek/datalake/combined.parquet'
    blob_name = 'combined.parquet'

    # Create a blob (object) in the bucket
    blob = bucket.blob(blob_name)

    # Upload the file to the blob
    with open(file_path, 'rb') as f:
        blob.upload_from_file(f)

    print(f"File uploaded to bucket: {bucket.name}, Blob: {blob_name}")


default_args = {
    'owner': 'Abishek Thamizharasan',
    'start_date': datetime(2023, 6, 9),
    'retries': 3,
    'retry_delay': timedelta(minutes=5),
}

with DAG('upload_file_to_bucket_dag', default_args=default_args, schedule_interval='@daily') as dag:
    upload_task = PythonOperator(
        task_id='upload_file_to_bucket',
        python_callable=upload_file_to_bucket
    )

    upload_task
```