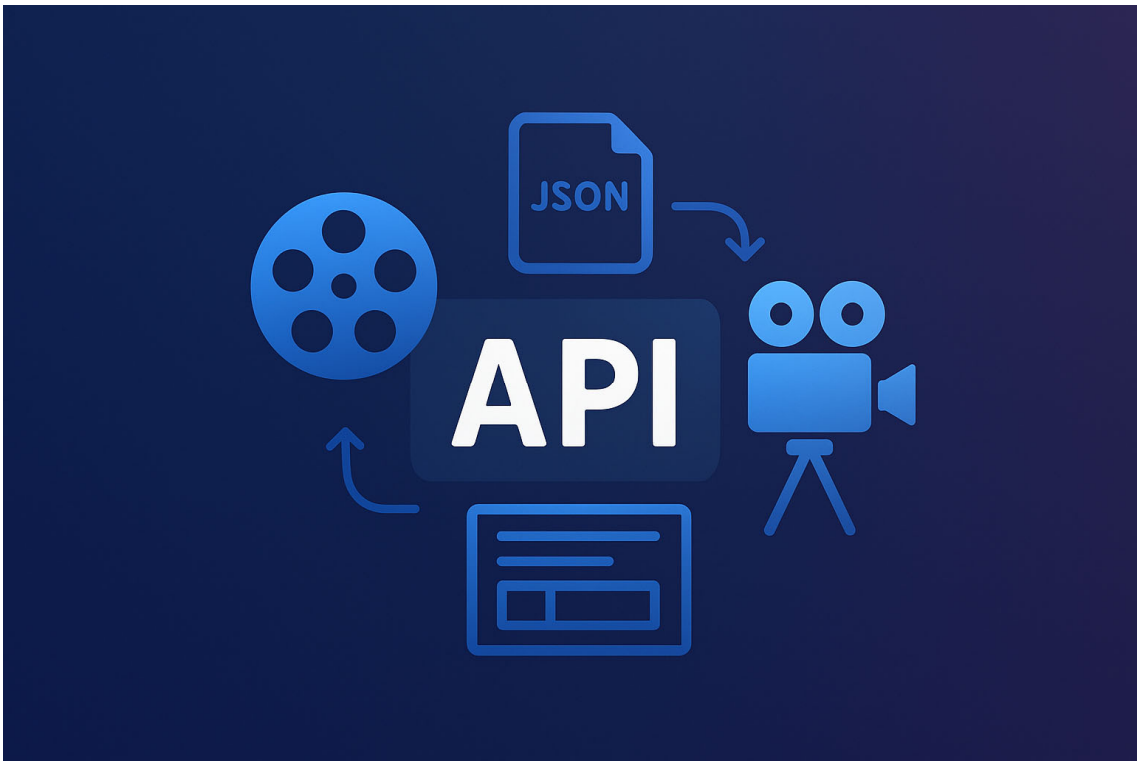
Harvesting Local and Global Movie Data at Scale: From TMDB Async API to Google BigQuery



How I optimized fetching 10,000 movie records in parallel using Python AsyncIO and built a streamlined ETL pipeline to launch it on Google Looker Studio.

In the world of Data Engineering and Data Science, fetching data from APIs is our daily bread and butter. But what happens when you need to retrieve thousands of pages of data? If you stick to traditional *synchronous* methods, you might find yourself waiting hours for a script to finish.

For this portfolio project, I challenged myself to fetch weekly Trending Movies data from TMDB (The Movie Database). The goal wasn't just to get the data, but to do it fast, efficiently, clean it, and load it into Google BigQuery for further analysis then I visualize it in Google Looker Studio.

Here is my journey in building this high-performance async pipeline.

---

1. The Setup: Why Async?

Typically, when we request data using the standard Python requests library, the computer works sequentially: *Send Request 1 -> Wait for Response -> Send Request 2 -> Wait for Response*, and so on. This "blocking" behavior is incredibly slow when you need to scrape 500 pages of pagination.

The solution? Asynchronous Programming.

```
import os
from dotenv import load_dotenv
import nest_asyncio
import asyncio
import aiohttp
import numpy as np
import pandas as pd
import warnings
from datetime import datetime, timedelta
from google.oauth2 import service_account
import pandas_gbq
warnings.filterwarnings("ignore")
pd.set_option("display.max_columns", None)
load_dotenv()
nest_asyncio.apply()
BASE_URL = "https://api.themoviedb.org/3"
ENDPOINT_TRENDING_MOVIE_WEEK = "/trending/movie/week"
TMDB_IMAGE_BASE = "https://image.tmdb.org/t/p/"
POSTER_SIZE = "w185"
```

By leveraging the aiohttp and asyncio libraries, we can fire off multiple requests simultaneously without waiting for the previous ones to complete. Think of it as opening 4 checkout lanes at a supermarket instead of forcing everyone through a single line.

```
async def fetch_page(session: aiohttp.ClientSession, page: int) -> dict:
    url = BASE_URL + ENDPOINT_TRENDING_MOVIE_WEEK
    params = {"api_key": API_KEY, "page": page}

    async with session.get(url, params=params) as resp:
        status = resp.status
        data = await resp.json()

        print(f"Ambil page {page} ... status code: {status}")

        if status != 200:
            raise RuntimeError(f"Gagal ambil page {page}: status {status} | body: {data}")

        return data
```

Here, I set up the environment, loaded my API keys securely, and prepared the base fetching function.

---

2. The Engine: Parallel Data Fetching

```python
async def fetch_trending_movie_week_async(
    max_pages: int | None = None,
    concurrency: int = 4
) -> pd.DataFrame:
    connector = aiohttp.TCPConnector(limit=None)  # biarkan semaphore yang batasi
    async with aiohttp.ClientSession(connector=connector) as session:

        first_data = await fetch_page(session, page=1)
        total_pages_api = first_data["total_pages"]
        total_results = first_data["total_results"]

        if max_pages is None:
            n_pages = total_pages_api
        else:
            n_pages = min(max_pages, total_pages_api)

        print(f"\nTotal pages di API : {total_pages_api}")
        print(f"Total results di API : {total_results}")
        print(f"Pages yang di-fetch : 1..{n_pages}\n")

        all_results = list(first_data["results"])

        if n_pages == 1:
            df = pd.DataFrame(all_results)
            return df
```

This is the core of the project. I created a function called fetch_trending_movie_week_async. However, simply blasting the API with 500 simultaneous requests is a bad idea—it's a quick way to get your IP banned for resembling a DDoS attack.

```python
semaphore = asyncio.Semaphore(concurrency)

async def bound_fetch(page: int) -> dict:
    async with semaphore:
        return await fetch_page(session, page)

tasks = [
    asyncio.create_task(bound_fetch(page))
    for page in range(2, n_pages + 1)
]

pages_data = await asyncio.gather(*tasks)

for page_data in pages_data:
    all_results.extend(page_data["results"])

df = pd.DataFrame(all_results)
return df
```

In accordance for tackle the DDos attach is to solve it, which I implemented a Semaphore.

What is a Semaphore? Think of it as a "bouncer" at a club. I set the concurrency=4, meaning only 4 active requests are allowed at any single moment. This keeps the script respectful of TMDB's rate limits while still being exponentially faster than a synchronous loop.

The flow is:

1. **Fetch Page 1 to determine the total_pages (which turned out to be 500 pages!).**
2. **Prepare tasks for pages 2 through 500.**
3. **Execute all tasks using asyncio.gather.**

The result? I successfully retrieved 500 pages (approx. 10,000 movie records) in a fraction of the time.

---

3. Data Cleaning: Handling Duplicates

Once the data was collected, I encountered a classic data engineering problem: Dirty Data.

Because the "Trending" list on TMDB is dynamic (rankings can shift in real-time), it's possible for the same movie to appear on different pages while the scraping script is running. I discovered about 200 duplicate records in the dataset.

```python
before = len(df_trending_all)
df_trending_all = (
    df_trending_all
    .drop_duplicates(subset="id", keep="first")
    .reset_index(drop=True)
)
after = len(df_trending_all)

print(f"Baris sebelum : {before}")
print(f"Baris sesudah : {after}")
print(f"Duplikat dibuang: {before - after}")
print("Masih ada duplikat id?", df_trending_all["id"].duplicated().any())

kolom_penting = [
    "id",
    "title",
    "original_title",
    "overview",
    "release_date",
    "vote_average",
    "vote_count",
    "popularity",
    "media_type",
    "original_language",
    "poster_path",
]

kolom_penting_ada = [k for k in kolom_penting if k in df_trending_all.columns]

df_trending_clean = df_trending_all[kolom_penting_ada].copy()
```

I cleaned this by dropping duplicates based on the unique movie id and resetting the index.

```python
def make_poster_url(path):
    if not path or pd.isna(path):
        return None
    return f"{TMDB_IMAGE_BASE}{POSTER_SIZE}{path}"

df_trending_clean["poster_url"] = df_trending_clean["poster_path"].apply(make_poster_url)

df_trending_clean.head()
```

I also added a helper function, make_poster_url, to convert the partial image paths provided by the API into fully accessible URLs.

---

4. Enriching Data: Genre Mapping

The TMDB API provides genres as a list of IDs (e.g., [28, 12]) rather than human-readable text (e.g., "Action", "Adventure"). While efficient for computers, this isn't great for analysis.

```python
async def get_genre_mapping_async(language: str = "en-US") -> dict:
    """
    Ambil daftar genre movie dari TMDB.

    Return:
        dict {genre_id: genre_name}
    """
    url = BASE_URL + "/genre/movie/list"
    params = {"api_key": API_KEY, "language": language}

    async with aiohttp.ClientSession() as session:
        async with session.get(url, params=params) as resp:
            status = resp.status
            data = await resp.json()
            print(f"Ambil genre list ... status code: {status}")

            if status != 200:
                raise RuntimeError(
                    f"Gagal ambil genre list: status {status} | body: {data}"
                )

    genres = data.get("genres", [])
    return {g["id"]: g["name"] for g in genres}

genre_map = await get_genre_mapping_async(language="en-US")

genre_map
```

I built an additional async function, get_genre_mapping_async, to fetch the official genre reference list from TMDB.

```python
def map_genre_ids_to_names(genre_ids):
    """
    genre_ids: list of int, misalnya [28, 12]
    return: string, misalnya "Action, Adventure"
    """
    if not isinstance(genre_ids, (list, tuple)):
        return None
    names = [genre_map.get(gid) for gid in genre_ids if gid in genre_map]
    if not names:
        return None
    return ", ".join(names)

df_trending_all["genres"] = df_trending_all["genre_ids"].apply(map_genre_ids_to_names)

df_trending_all[["title", "genre_ids", "genres"]].head()
```

I then mapped these IDs in my main DataFrame to create a readable "genres" column.

---

5. Segmentation: Local vs. Global Markets

A key part of my analysis plan was to compare the performance of domestic (Indonesian) films against the global market.

I split the cleaned DataFrame into two segments based on original_language:

1. **Indonesian Market: Movies where the language is 'id'.**
2. **Global Market: All other movies.**

```
# Movie Indonesia
df_movie_id = df_trending_clean[
    df_trending_clean["original_language"] == "id"
].copy()
df_movie_id["segment"] = "indonesia"

# Movie Global (selain Indonesia)
df_movie_global = df_trending_clean[
    df_trending_clean["original_language"] != "id"
].copy()
df_movie_global["segment"] = "global"

print("Jumlah movie Indonesia :", len(df_movie_id))
print("Jumlah movie global    :", len(df_movie_global))

display(df_movie_id.head())
display(df_movie_global.head())
```

```
Jumlah movie Indonesia : 53
Jumlah movie global    : 9732
```

Interestingly, out of 10,000 trending movies this week, about 53 Indonesian films made the cut, competing alongside thousands of global blockbusters!

---

6. Query Processing: Google BigQuery

The next step is the Load phase of the ETL process. Clean, structured data is useless if it sits in a Jupyter Notebook. I utilized pandas_gbq and a Google Cloud Service Account to upload the data directly into BigQuery.

```
pandas_gbq.to_gbq(
    df_movie_id,
    destination_table=table_indo,
    project_id=project_id,
    if_exists="replace"
)
```

```
pandas_gbq.to_gbq(
    df_movie_global,
    destination_table=table_global,
    project_id=project_id,
    if_exists="replace"
)
```

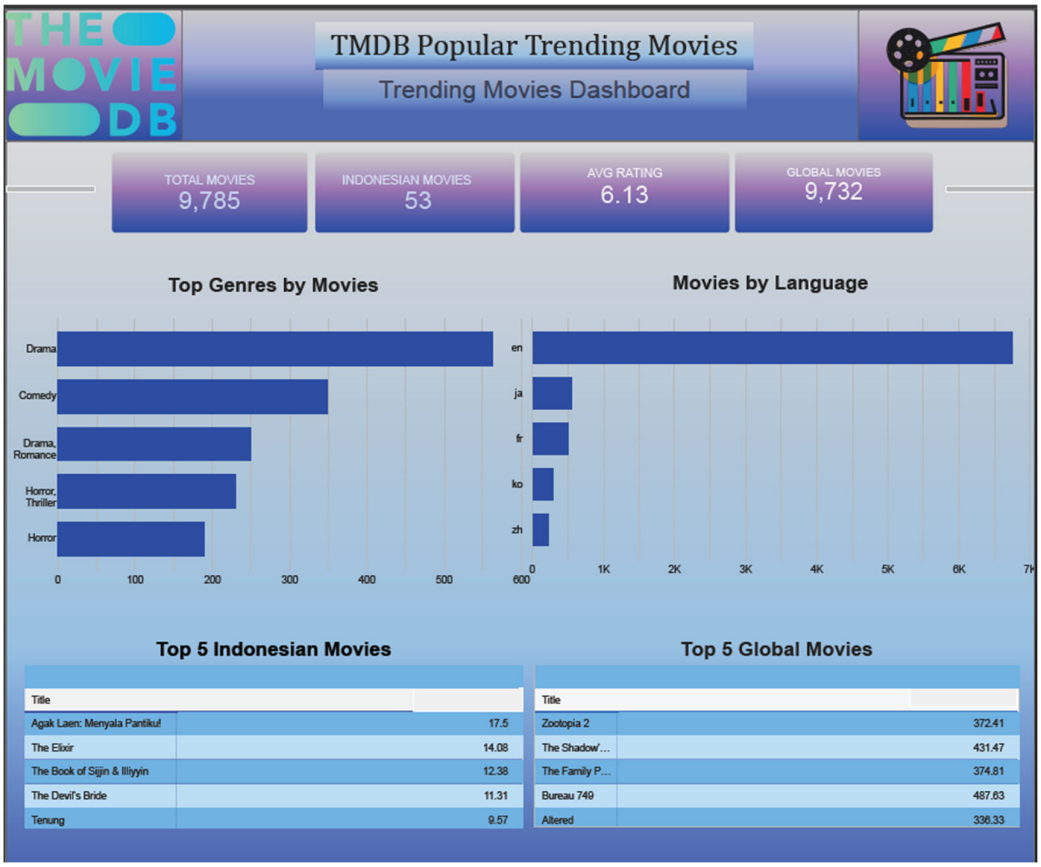
By doing this, the data is securely stored in a cloud data warehouse, ready to be connected to visualization tools like Google Looker Studio for real-time dashboarding.

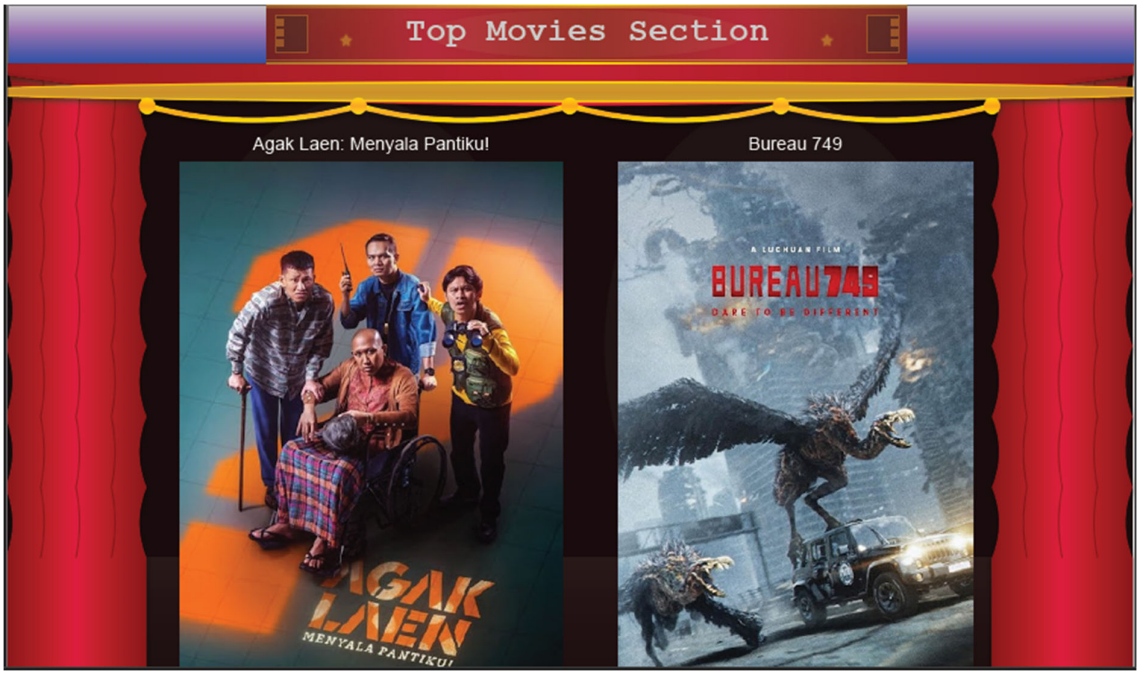
---

7. The Result: Bringing Data to Life with Looker Studio

Clean data is useless if it doesn't "speak" to us. The final step of this ETL pipeline is Visualization.

Since the data was already safely stored in BigQuery, I connected it directly to Google Looker Studio. This allowed me to build an interactive dashboard to monitor weekly movie trends without writing any extra code.





Looking at the dashboard, all the previous data cleaning and transformation steps pays off:

- **Data Integrity: The dashboard displays a Total Movies count of 9,785. This matches our scraping results (post-deduplication), validating that our pipeline is reliable.**
- **Segmentation in Action: Remember when I split the data into "Indo" vs. "Global"? Here, we can visualize them separately. The movie "Agak Laen: Menyala Pantiku!" leads the local Indonesian chart, while global blockbusters like "Bureau 749" dominate the global chart with massive popularity scores.**
- **Genre Insights: The bar chart confirms that Drama and Comedy are currently the most dominant genres in the market.**
- **Poster Visualization: Recall the make_poster_url function from the cleaning stage? It proves vital here. The movie posters render perfectly at the bottom of the dashboard, transforming a boring dataset into an engaging visual experience.**

Key Takeaways

Building this module taught me several valuable lessons:

1. **The Power of AsyncIO: The speed difference between synchronous and asynchronous execution is massive for I/O-bound tasks like API fetching. It is a must-have skill for modern data engineers.**
2. **Rate Limiting Awareness: Speed is good, but respecting the data provider is better. Using Semaphores is the ethical and safe way to scrape data without getting blocked.**
3. **Data Quality Control: Never trust raw API data blindly. Always implement checks for duplicates, missing values, and data types before storage.**
4. **Cloud Integration: Integrating Python with Google BigQuery is seamless with the right libraries, opening the door to scalable data analytics.**
5. **End-to-End Flow: This project demonstrates the complete data engineering lifecycle: from fetching raw data (API), processing it (Python), storing it (Cloud Warehouse), to finally presenting visual insights (Dashboard).**

This project proves that with the right code architecture, we can build robust, scalable data pipelines efficiently.

Thanks for reading!