UTM
UNIVERSITI TEKNOLOGI MALAYSIA

PROJECT 1: HIGH-PERFORMANCE DATA PROCESSING FOR WEB CRAWLER IN THE EDGE MALAYSIA

GROUP G

1. CHEN PYNG HAW
2. MUHAMMAD DANIAL BIN AHMAD SYAHIR
3. LOW JIE SHENG
4. NADHRAH NURSABRINA BINTI ZULAINI

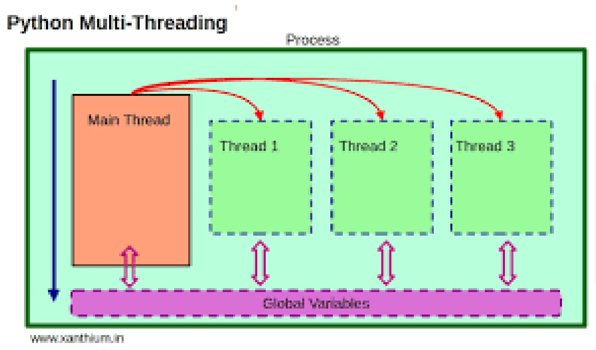


TABLE OF CONTENT

01 INTRODUCTION

OBJECTIVES

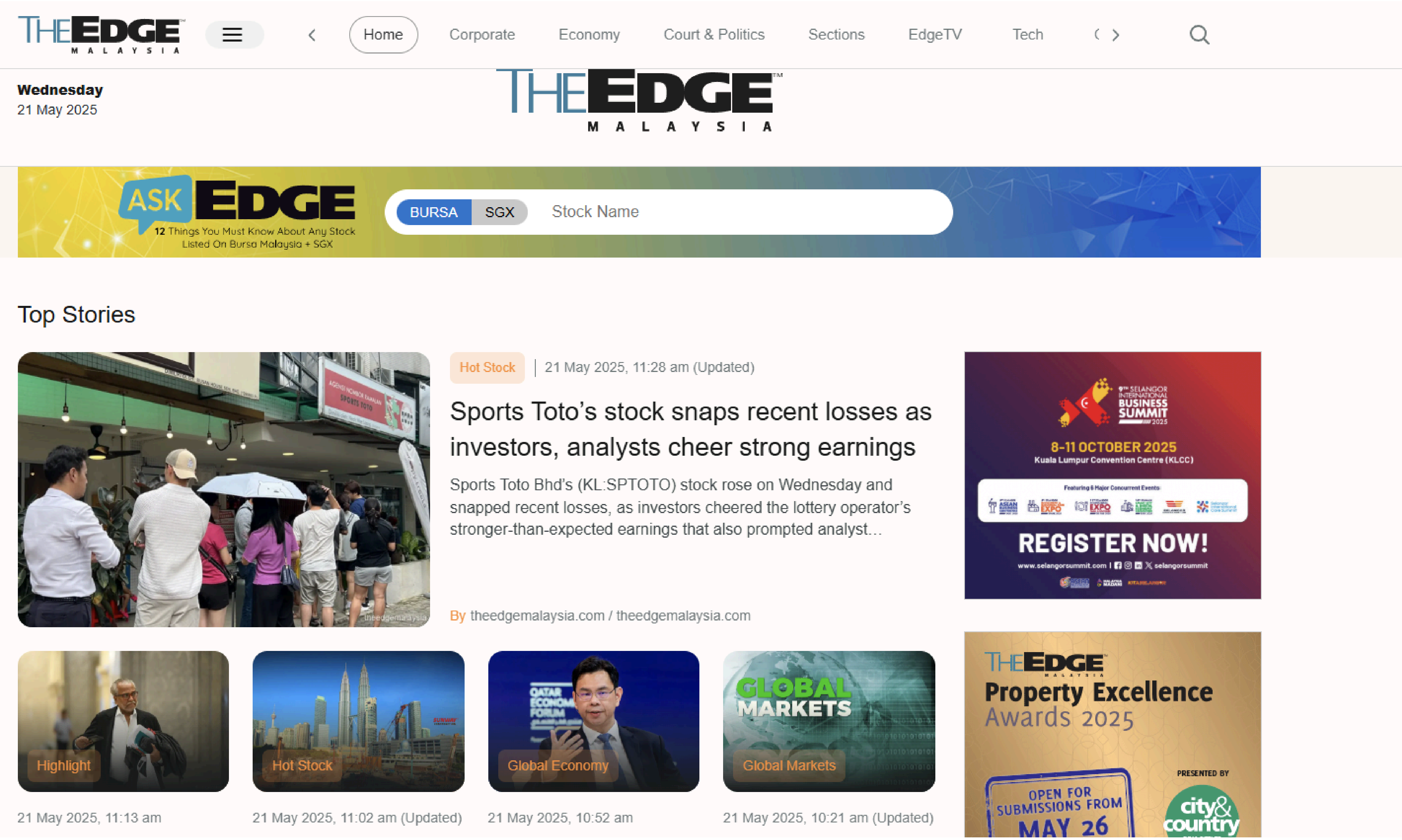
Extract 100,000 structured records from a Malaysian website

Process and clean the extracted data

Optimize the data processing pipeline

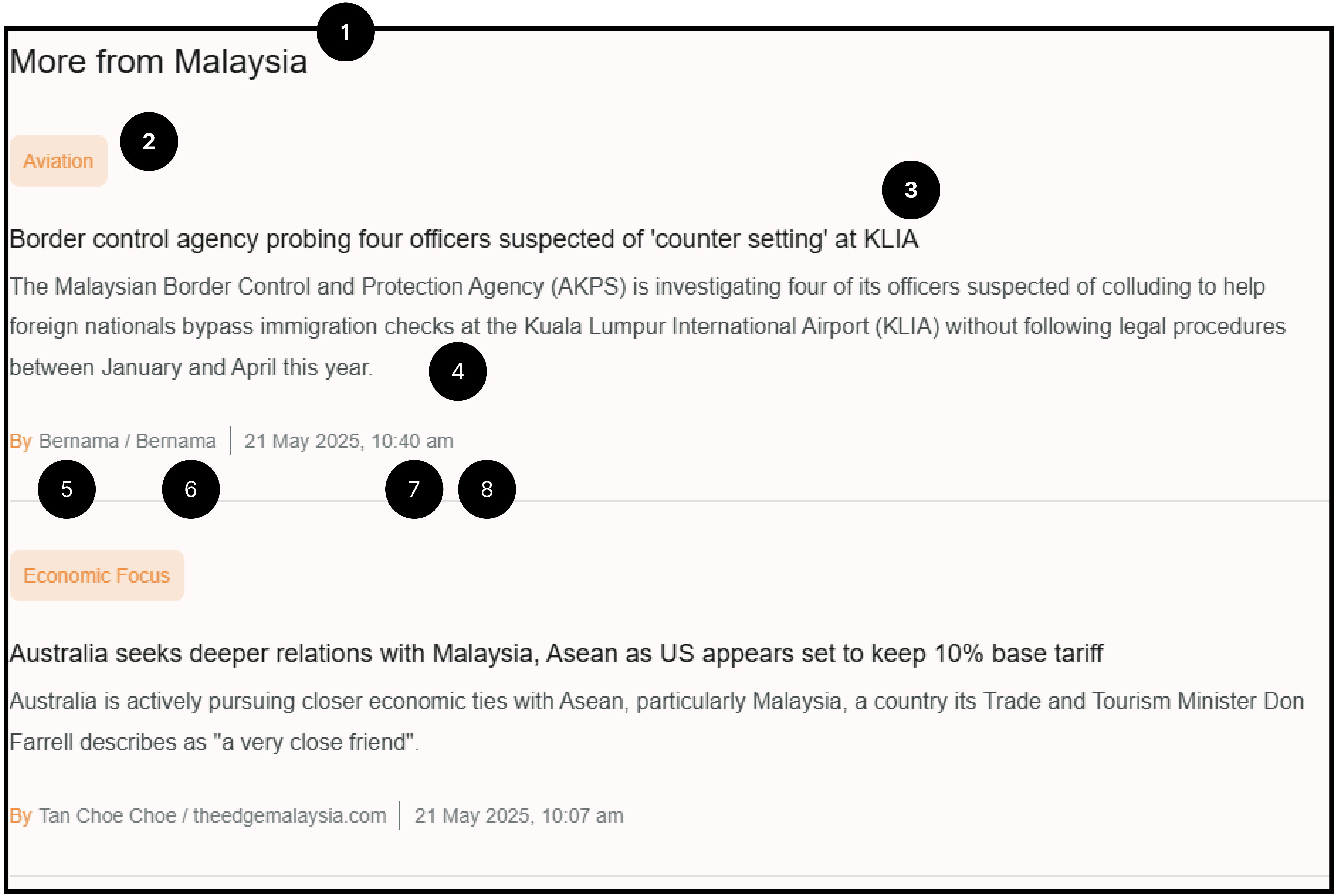
Evaluate the system's performance before and after optimization

Targeted Website - The Edge



The Edge Malaysia main page

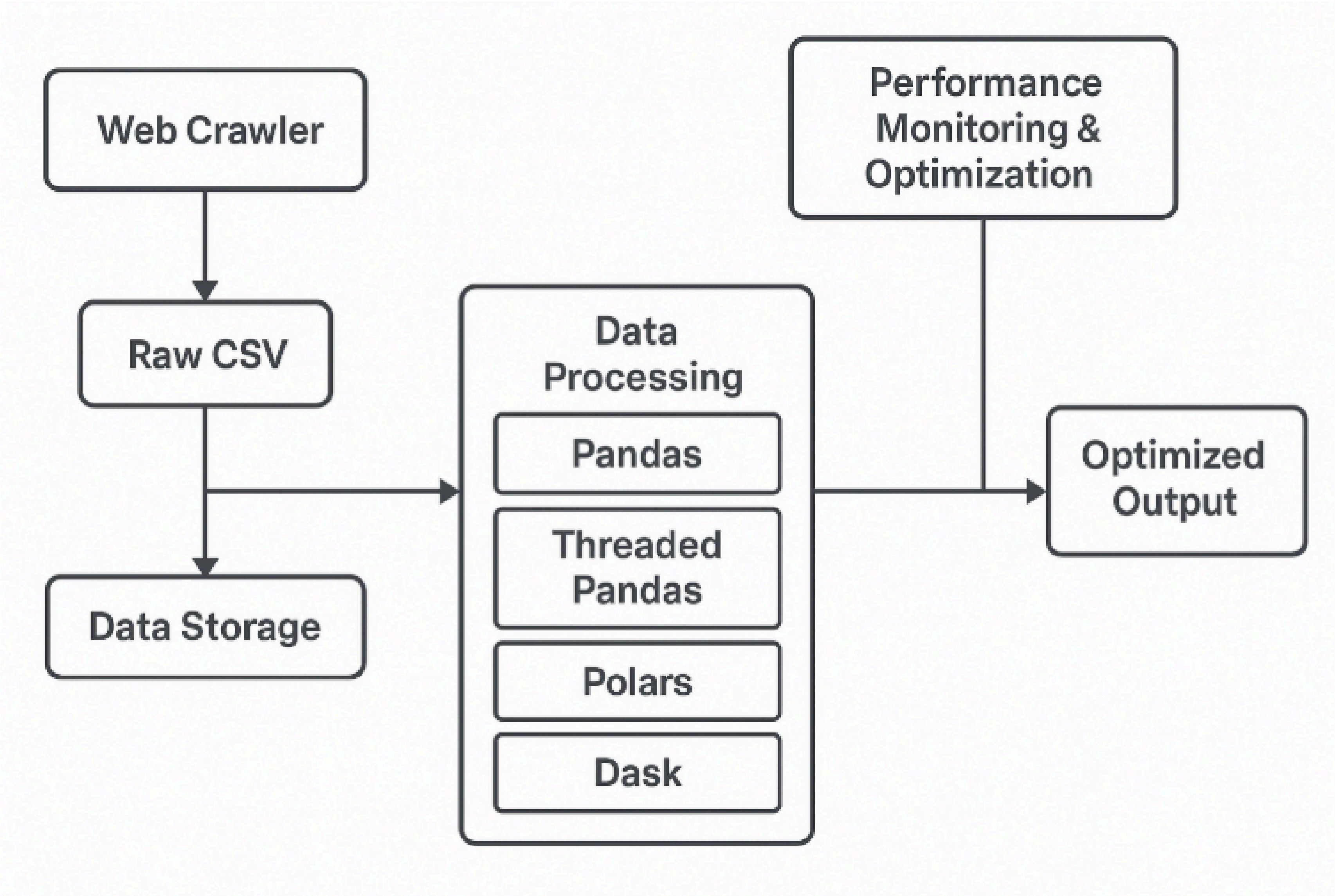
Data To Be Extracted



The Edge Malaysia, Malaysia section

1. Category
2. Sub-category
3. Title
4. Author
5. Source
6. Summary
7. Created date
8. Updated date

02 SYSTEM DESIGN AND ARCHITECTURE



The Edge Malaysia, Malaysia section

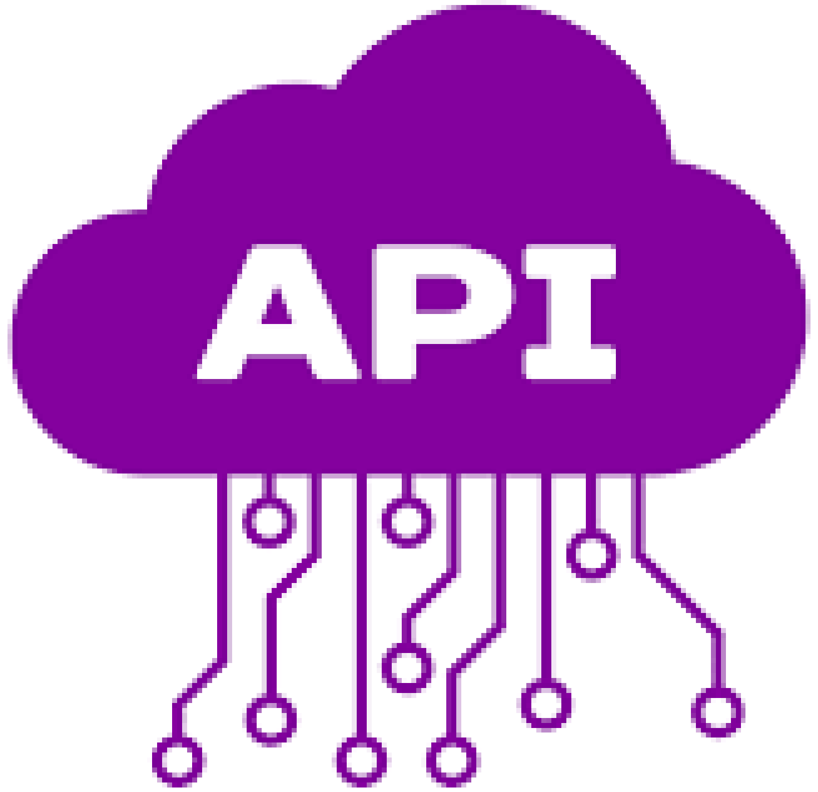
- Crawling data via The Edge Malaysia's public API using Python requests.
- Pagination handled by offset parameter (10 articles/request).
- Rate limiting: 0.3 seconds delay between requests to avoid server overload.
- Error handling to manage failed requests.
- Progressive saving: Data saved in CSV files every 10,000 records.

Tools and Technologies Used

- Python Requests – for API communication
- AsyncIO – to handle asynchronous tasks
- Dask – for distributed and parallel data processing
- Pandas / CSV / JSON – for data handling, cleaning, and storage

The Edge Malaysia, Malaysia section

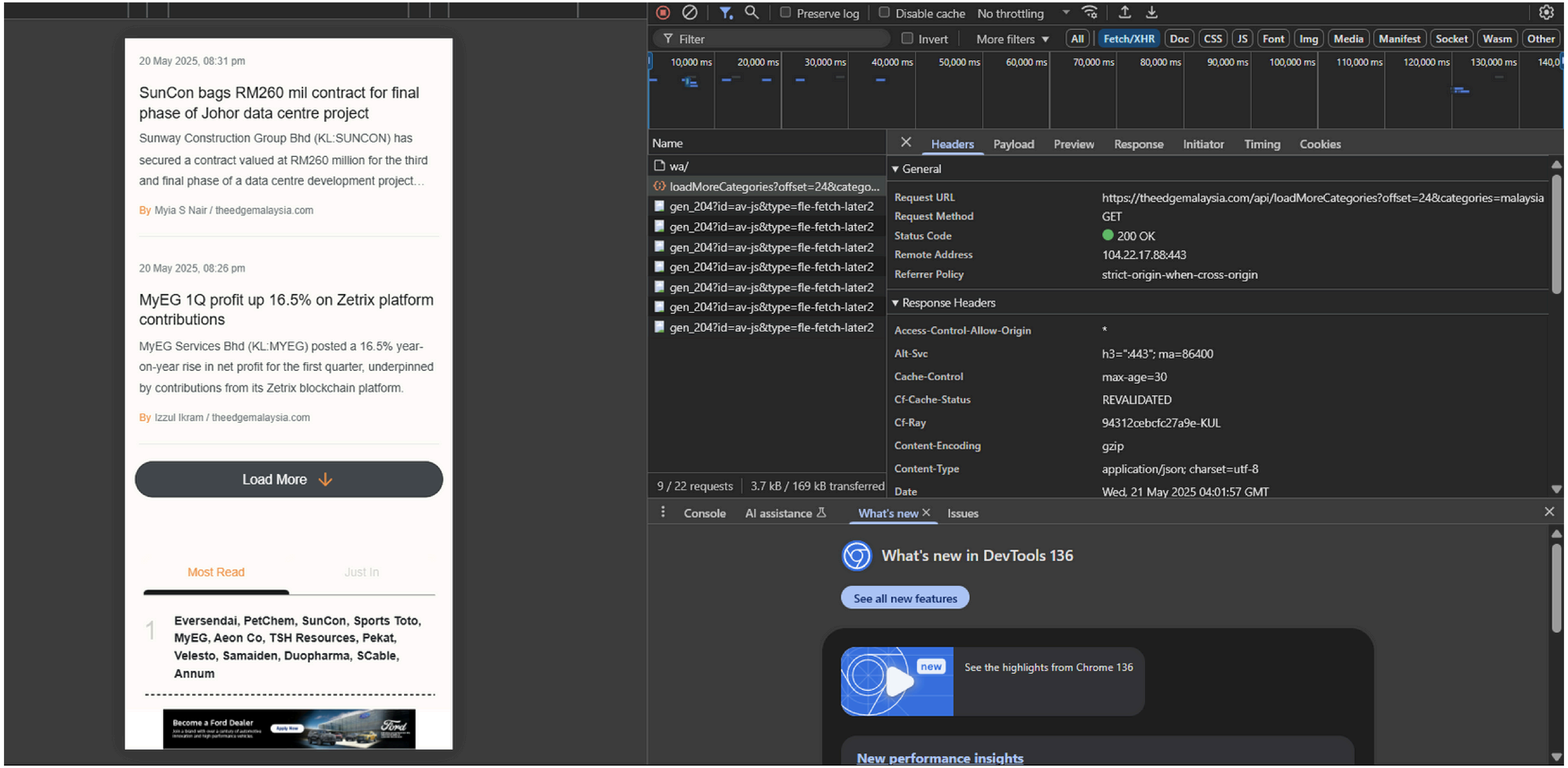
03 DATA COLLECTION



Requests

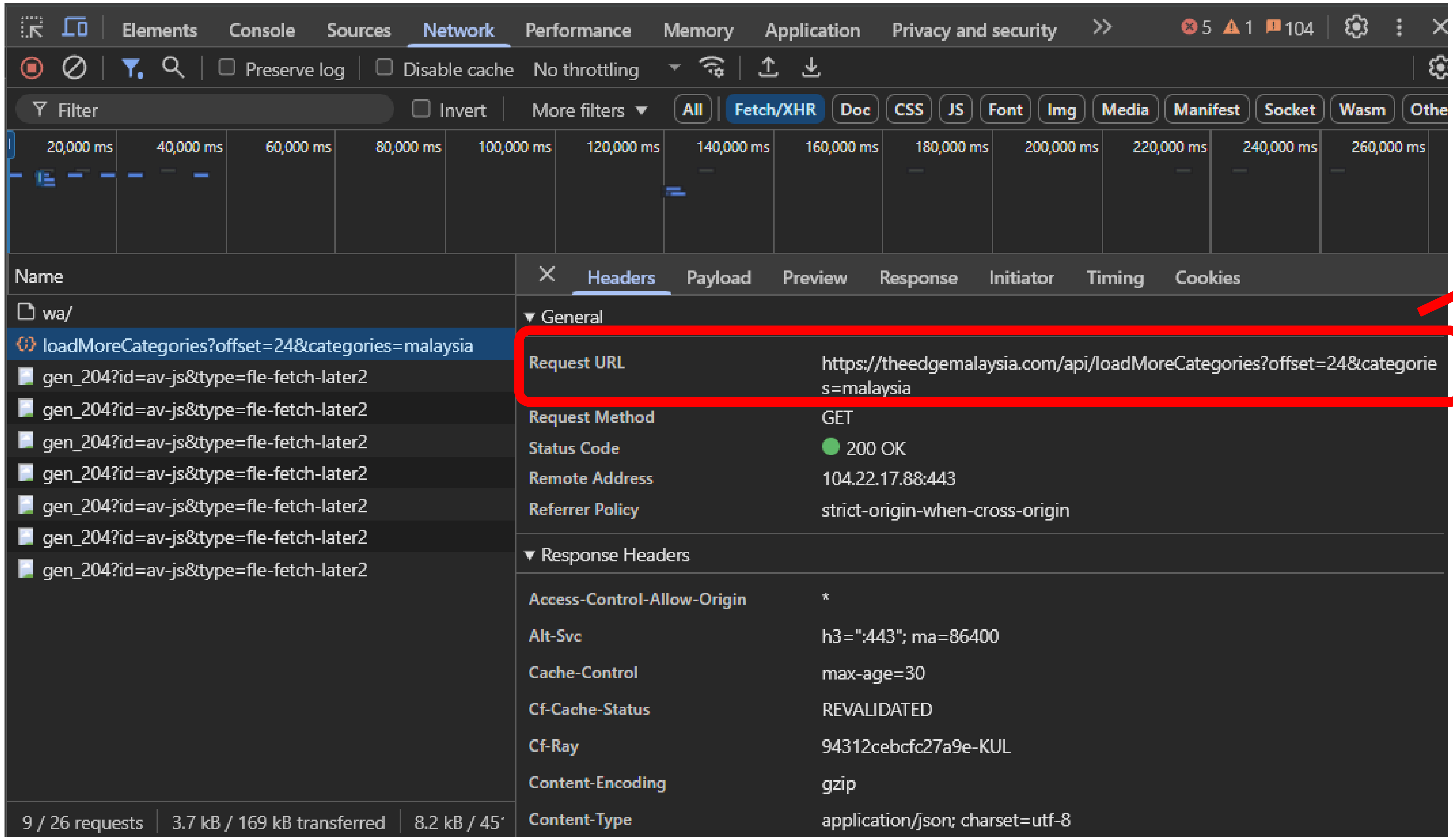
Python library

Look for API



The Edge Malaysia, inspect page

Look for API

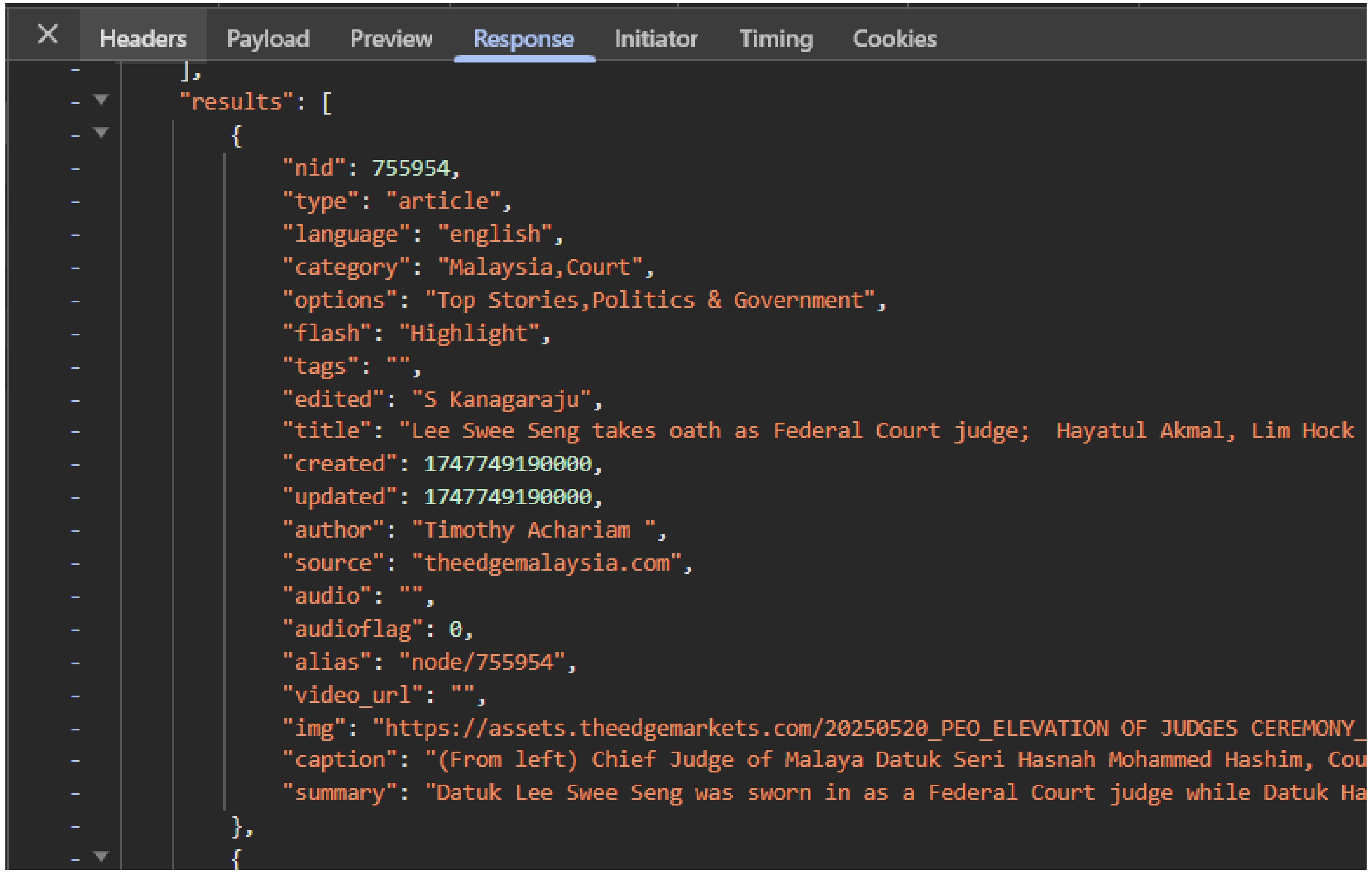


‘GET’ Request URL

The Edge Malaysia, network tab in inspect page

Understand the data

The Edge Malaysia, response tab in inspect page



The Edge Malaysia, response tab in inspect page

Each URL contain 10 articles

20 attributes in each article

Web Scraping

Requests

```python
import requests
import csv
import time
from datetime import UTC, datetime

def convert_timestamp(ms):
    if ms is None:
        return ""
    return datetime.fromtimestamp(ms / 1000, tz=UTC).strftime('%Y-%m-%d %H:%M:%S')

def fetch_articles(offset):
    url = f"https://theedgemalaysia.com/api/loadMoreCategories?offset={offset}&categories=malaysia"
    headers = {
        "User-Agent": "Mozilla/5.0"
    }
    response = requests.get(url, headers=headers)
    if response.status_code == 200:
        return response.json().get("results", [])
    else:
        print(f"Request failed: {response.status_code}")
        return []
```

Request the URL

Web scraping code Part A

Web Scraping

Requests

```
for i in range(1, 10000):
    offset = i * 10
    print(f"Fetching articles {offset}...")
    articles = fetch_articles(offset)

    if not articles:
        print("No more articles found.")
        break

    for item in articles:
        article = {
            "category": item.get("category", ""),
            "sub-category": item.get("flash", ""),
            "title": item.get("title", ""),
            "author": item.get("author", ""),
            "source": item.get("source", ""),
            "summary": item.get("summary", ""),
            "created date": convert_timestamp(item.get("created", 0)),
            "updated date": convert_timestamp(item.get("updated", 0))
        }
        all_articles.append(article)

    time.sleep(0.3)  # avoid overloading the server

    if len(all_articles) >= 10000:
        batch_start = offset - len(all_articles) + 10  # starting offset for this chunk
        batch_end = offset + 9                         # ending offset for this chunk
        filename = f"theedge_{batch_start}_{batch_end}.csv"
        with open(filename, "w", newline="", encoding="utf-8") as csvfile:
            fieldnames = ["category", "sub-category", "title", "author", "source", "summary", "created date", "updated date"]
            writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
            writer.writeheader()
            writer.writerows(all_articles)
        print(f"💾 Saved chunk: {filename}")
        all_articles = []  # ✅ reset buffer
```

Scrape data for every URL request

Save articles when it reaches 10k every time

Web scraping code Part B

Save to final csv file

theege_articles.csv

theedge_datasets
- merge.py
- theedge_0_9999.csv
- theedge_10000_19999.csv
- theedge_20000_29999.csv
- theedge_30000_39999.csv
- theedge_40000_49999.csv
- theedge_50000_59999.csv
- theedge_60000_69999.csv
- theedge_70000_79999.csv
- theedge_80000_89999.csv
- theedge_90000_99999.csv
- theedge_100000_109999.csv

pandas_performance_log.csv
the-edge-scrape.py
theedge_articles.csv

Saved siles from scraping

theedge_articles.csv is the final combined dataset

```python
import pandas as pd
import glob

# Match all your chunk files
file_list = glob.glob("theedge_*_*.csv")

# List to hold DataFrames
df_list = []

# Load each file and append to list
for file in file_list:
    print(f"Loading {file}...")
    df = pd.read_csv(file)
    df_list.append(df)

# Concatenate all DataFrames
combined_df = pd.concat(df_list, ignore_index=True)

# Save to a new single CSV
combined_df.to_csv("theedge_articles.csv", index=False, encoding='utf-8-sig')

print(f"✅ Done! Combined {len(file_list)} files into 'theedge_combined.csv'")
```

Saving files code

Combine all the theedge datasets

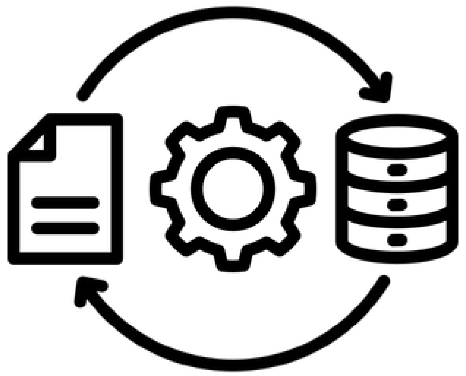
04 DATA PROCESSING & CLEANING METHODS

Cleaning Methods:

- Removed Duplicates: Ensured each article appears only once based on title.
- Handled Missing Data:
- Deleted articles without summaries (critical for understanding).
- Filled missing author and source fields with “Unknown” to preserve useful data.

Data Structure

- Raw data was imported from a CSV file.
- Cleaned and formatted data was saved into a new structured file for analysis.



Transformation & Formatting

- Date Conversion: Converted article dates into proper datetime format for sorting and analysis.
- Category Structuring: Split multiple categories into separate fields for clarity.
- Text Normalization: Standardized casing and cleaned up extra whitespace across key text fields.

The Edge Malaysia, Malaysia section

05 OPTIMIZATION TECHNIQUES

pandas

Polars

dask

Optimization Techniques-Multithreading-pandas with threading

```python
import pandas as pd
import time
import psutil
import threading
from concurrent.futures import ThreadPoolExecutor

# Global flag to stop monitoring
monitoring = True

# Function to monitor perform
def monitor_perf
    process =          ess()
    while        ing:
        _usage = process.memory_info().rss / (1024 * 1024)  # MB
        cpu_usage = process.cpu_percent(interval=None)        # %
        log_list.append((time.time(), mem_usage, cpu_usage))
        time.sleep(interval)

# List to store performance logs
performance_logs = []

# Start monitoring in a separate thread
monitor_thread = threading.Thread(target=monitor_performance, args=(performance_logs, 0.5))
monitor_thread.start()

# Track time
start_time = time.time()

print("==============================")
 int("🔄 Starting cleaning process (fully threaded column cleaning)...")
   ("==============================")

# 📥 Lo        taset
df = pd.read_          rticles.csv', low_memory=Fal
print(f"✅ Loaded {len

# 🔍 Check for missing values
print("Missing values per column:")
print(df.isnull().sum())


# 🧹 Fill missing values (threaded)
print("✅ Filling missing values (threaded)...")

def fill_missing(col_value):
    col, value = col_value
    df[col] = df[col].fillna(value)

fill_values = {
    'sub-category': 'General',
    'author': 'Unknown',
    'source': 'Unknown',
    'summary': 'N/A',
    'updated date': 'NaT'
}

with ThreadPoolExecutor(max_workers=len(fill_values)) as executor:
    executor.map(fill_missing, fill_values.items())

# 🧾 Remove exact duplicates
initial_len = len(df)
df = df.drop_duplicates().copy()
print(f"Removed {initial_len - len(df)} exact duplicate records.")

# 🧾 Remove duplicates based on 'title'
initial_len = len(df)
df = df.drop_duplicates(subset=['title']).copy()
print(f"Removed {initial_len - len(df)} duplicate records based on title.")

# 🕒 Convert date columns to datetime (threaded)
print("✅ Converting date columns to datetime (threaded)")

def convert_date(col):
```

Multithreading process

We used Python's ThreadPoolExecutor to run multiple column operations like filling missing values or converting dates in parallel. This helped reduce the time taken, especially for I/O-bound tasks.

Optimization Techniques-Distributed processing-Dask

```python
import dask.dataframe as dd
import time
import psutil
import threading

# Global flag to stop monitoring
monitoring = True

def monitor_performance(log_list, interval=1):
    process = psutil.Process()
    while monitoring:
        mem_usage = process.memory_info().rss / (1024 * 1024)  # MB
        cpu_usage = process.cpu_percent(interval=None)
        log_list.append((time.time(), mem_usage, cpu_usage))
        time.sleep(interval)

performance_logs = []
monitor_thread = threading.Thread(target=monitor_performance, args=(performance_logs, 0.5))
monitor_thread.start()

start_time = time.time()
print("================================")
print("🔄 Starting Dask cleaning process...")
print("================================")

# ✅ Tell Dask the exact type of each column
ddf = dd.read_csv('theedge_articles.csv', low_memory=False)
print(f"✅ Loaded {len(ddf)} records.")

# Cleaning steps (same as before)
print("🔄 Filling missing values...")
ddf['sub-category'] = ddf['sub-category'].fillna('General')
ddf['author'] = ddf['author'].fillna('Unknown')
ddf['source'] = ddf['source'].fillna('Unknown')
ddf['summary'] = ddf['summary'].fillna('')
ddf['updated date'] = ddf['updated date'].fillna('NaT')

# Remove duplicates
ddf = ddf.drop_duplicates().drop_duplicates(subset=['title'])
print("✅ Removed duplicates.")

# Strip whitespace
text_cols = ['title', 'summary', 'category', 'sub-category', 'author', 'source']
for col in text_cols:
    if col in ddf.columns:
        ddf[col] = ddf[col].astype(str).str.strip()
print("✅ Stripped whitespace from text columns")

# Save result
ddf.compute().to_csv('theedge_cleaned_dask.csv', index=False, encoding='utf-8-sig', na_rep='N/A')
print("✅ Saved cleaned dataset to 'theedge_cleaned_dask.csv'.")

# Stop monitoring
monitoring = False
monitor_thread.join()

end_time = time.time()

total_time = end_time - start_time
num_records = ddf.shape[0].compute()
throughput = num_records / total_time

print("================================")
print("🚀 Performance Summary")
print("================================")
print(f"Total time taken: {end_time - start_time:.2f} seconds")

# Peak + average usage
peak_mem = max([m for _, m, _ in performance_logs])
peak_cpu = max([c for _, _, c in performance_logs])
avg_mem = sum([m for _, m, _ in performance_logs]) / len(performance_logs)
avg_cpu = sum([c for _, _, c in performance_logs]) / len(performance_logs)
print(f"Average memory: {avg_mem:.2f} MB")
print(f"Peak memory: {peak_mem:.2f} MB")
print(f"Average CPU: {avg_cpu:.2f}%")
print(f"Peak CPU: {peak_cpu:.2f}%")
print(f"✅ Processed {num_records} records in {total_time:.2f} seconds.")
print(f"🚀 Throughput: {throughput:.2f} records per second.")
```

Dask triggering execution happen

Dask allowed us to process large datasets by splitting them into partitions and handling each in parallel across CPU cores. It's especially useful when the dataset is too big to fit into memory.

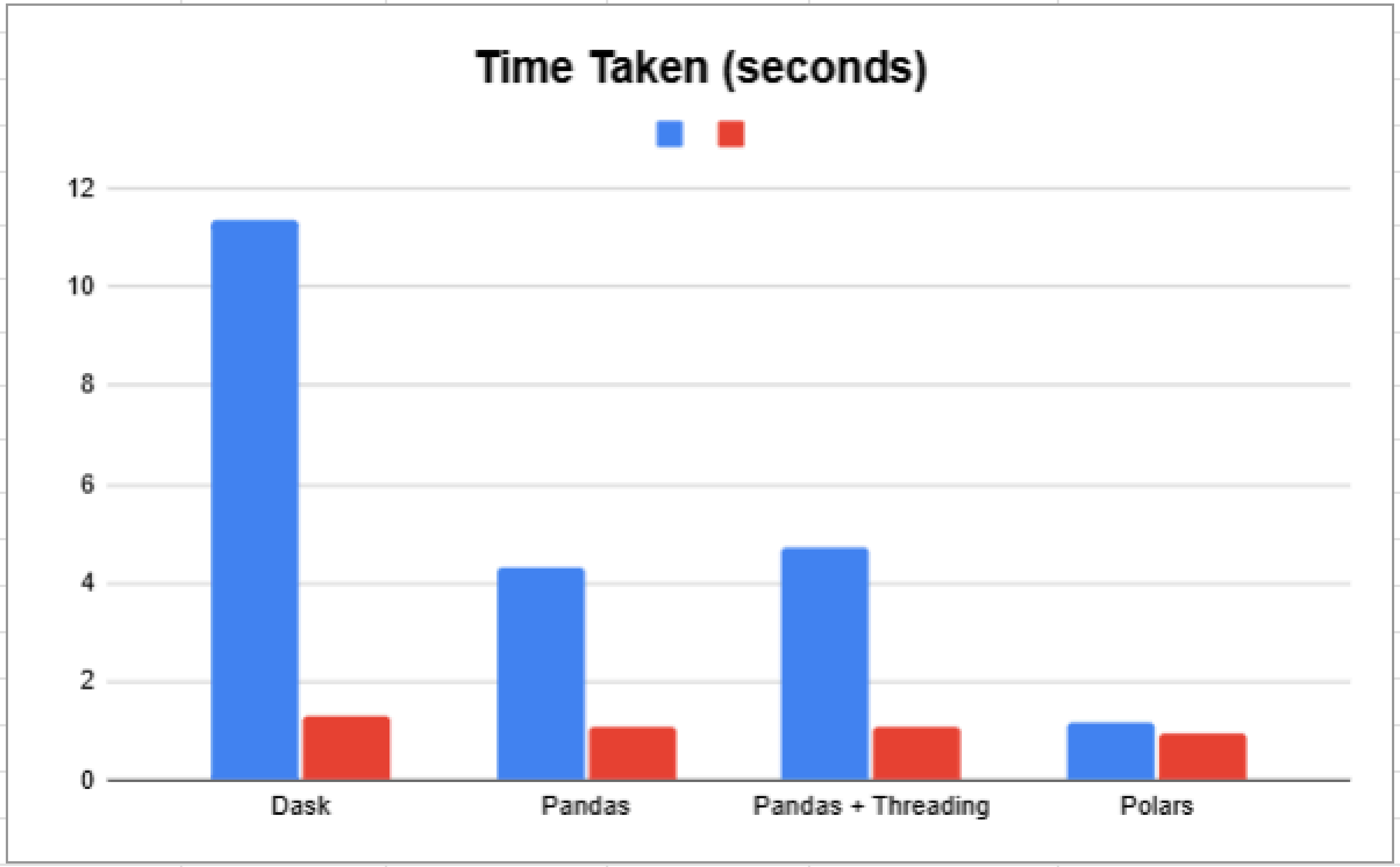
Optimization Techniques-Library-Polars

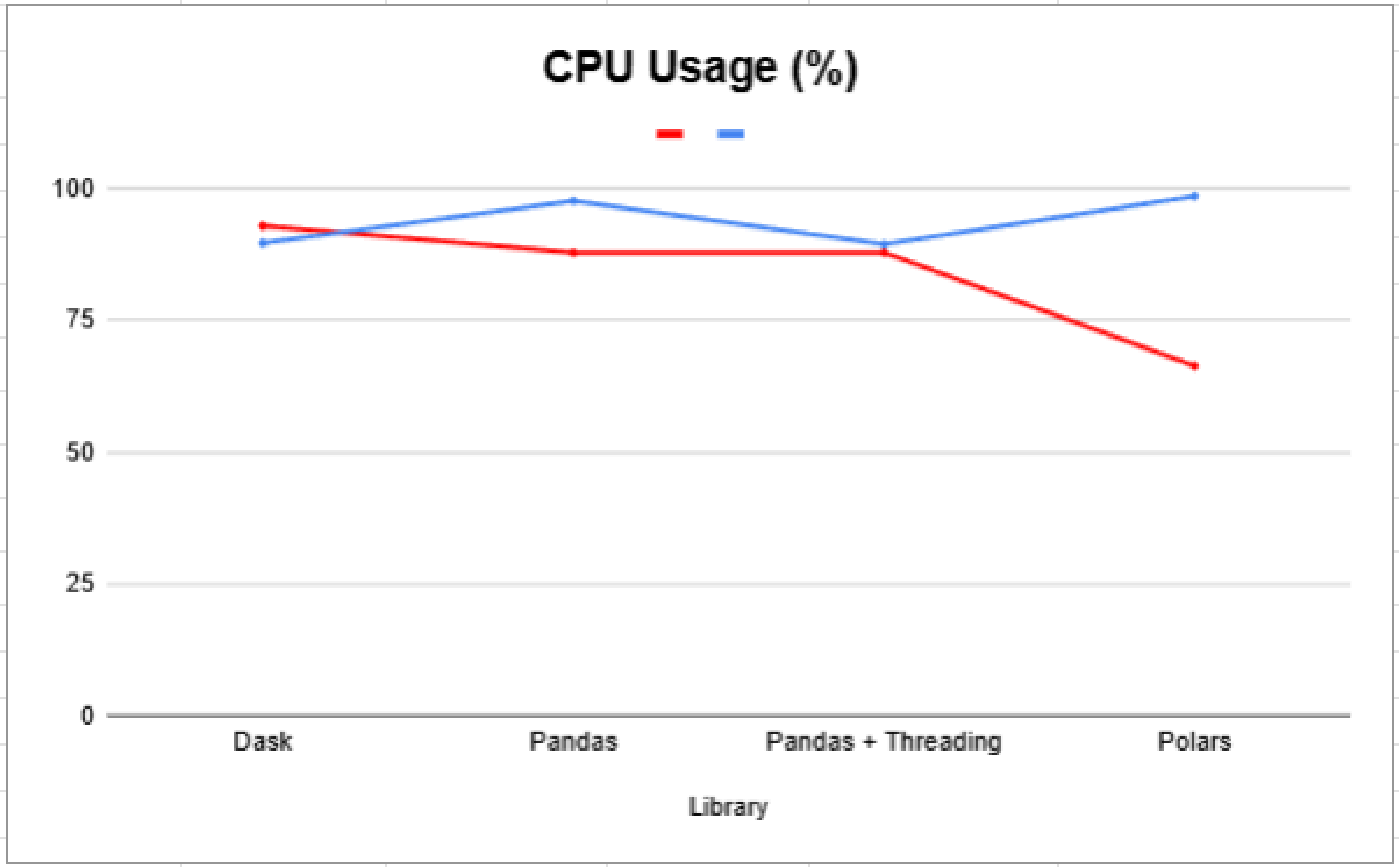
```python
import polars as pl
import time
import psutil
import threading

# Global flag to stop monitoring
monitoring = True

# Function to monitor performance DURING the process
def monitor_performance(log_list, interval=1):
    process = psutil.Process()
    while monitoring:
        mem_usage = process.memory_info().rss / (1024 * 1024)  # MB
        cpu_usage = process.cpu_percent(interval=None)         # %
        log_list.append((time.time(), mem_usage, cpu_usage))
        time.sleep(interval)

# List to store performance logs
performance_logs = []

# Start monitoring in a separate thread
monitor_thread = threading.Thread(target=monitor_performance, args=(performance_logs, 0.5))
monitor_thread.start()

# [illegible]
[illegible] = time.time()

print("==============================")
print("🔄 Starting Polars cleaning process...")
print("==============================")

# 📥 Load the dataset
df = pl.read_csv('theedge_articles.csv', low_memory=False)
print(f"✅ Loaded {df.shape[0]} records.")

# 🔍 Check for missing values
null_counts = df.null_count()
print("Missing values per column:")
print(null_counts)

# 🧹 Fill missing values
fill_values = {
    'sub-category': 'General',
    'author': 'Unknown',
    'source': 'Unknown',
    'summary': '',
    '[illegible]ed date': 'NaT'
}
for column, value in [illegible]():
    if column in df.columns:
        df = df.with_columns(
            pl.col(column).fill_null(value)
        )
print("✅ Filled missing values.")

# 🗑️ Remove exact duplicates
initial_len = df.shape[0]
df = df.unique()
print(f"Removed {initial_len - df.shape[0]} exact duplicate records.")

# 🗑️ Remove duplicates based on 'title'
initial_len = df.shape[0]
if 'title' in df.columns:
    df = df.unique(subset=['title'])
    print(f"Removed {initial_len - df.shape[0]} duplicate records based on title.")
else:
    print("⚠️ 'title' column not found; skipping duplicate removal based on title.")

# 🕒 Convert date columns to datetime
if 'created date' in df.columns:
    df = df.with_columns(
        pl.col('created date').str.strip_chars().str.strptime(pl.Datetime, strict=False).alias('created date')
    )
if 'updated date' in df.columns:
    df = df.with_columns(
        pl.col('updated date').str.strip_chars().str.strptime(pl.Datetime, strict=False).alias('updated date')
    )
print("✅ Converted date columns to datetime.")

# 🎀 Strip whitespace from text columns
text_cols = ['title', 'summary', 'category', 'sub-category', 'author', 'source']
for col in text_cols:
    if col in df.columns:
        df = df.with_columns(
            pl.col(col).cast(pl.Utf8).str.strip_chars()
        )
print("✅ Stripped whitespace from text columns.")

# 💾 Save cleaned data
df.write_csv('theedge_cleaned_polars.csv')
print(f"✅ Saved cleaned dataset: {df.shape[0]} records to 'theedge_cleaned_polars.csv'.")

# Stop monitoring
monitoring = False
monitor_thread.join()

end_time = time.time()

total_time = end_time - start_time
num_records = df.shape[0]
throughput = num_records / total_time

# ==========================
# 🚀 Performance Summary
# ==========================
```

Polars happen during loading the dataset and it is highly optimized compared to pandas

Polars is built in Rust and automatically runs tasks in parallel under the hood. It's optimized for performance, uses lazy execution, and was the fastest among the tools we tested for cleaning and transforming data.

06 PERFORMANCE EVALUATION

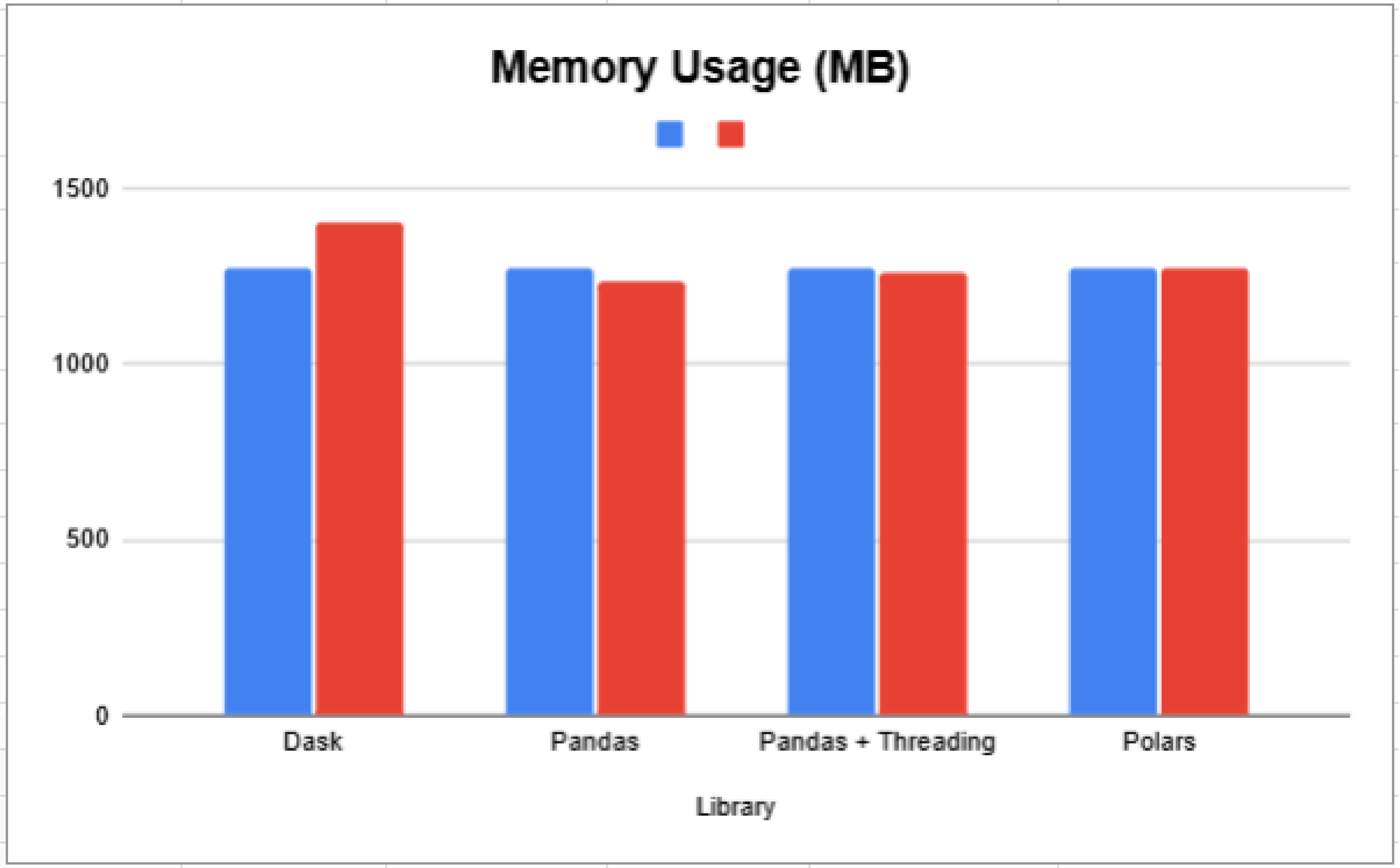


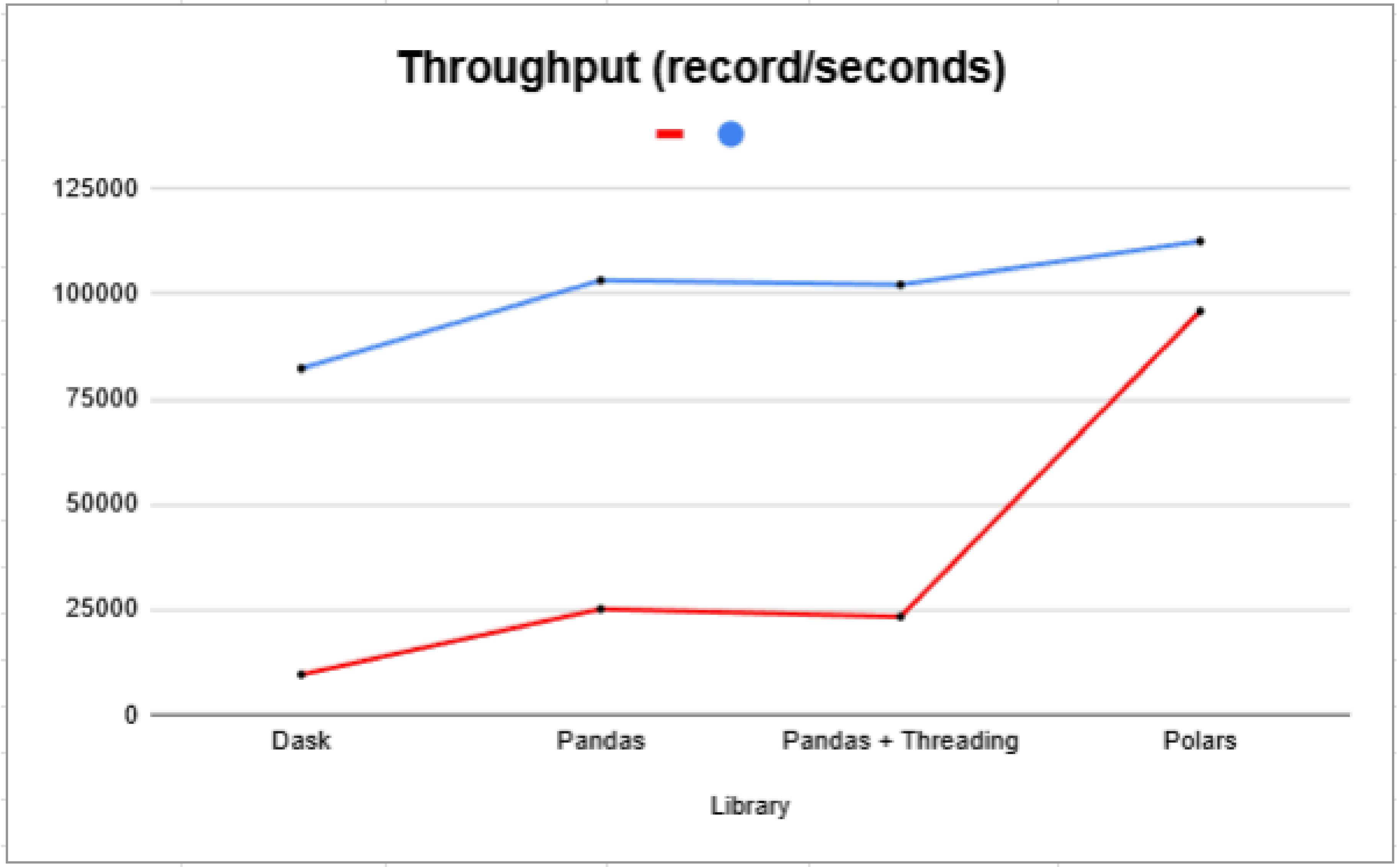


- **Polars is fastest**
- **all optimized versions are better.**

- **Polars and Pandas used CPU more efficiently after optimization.**

06 PERFORMANCE EVALUATION





- **Memory usage slightly improved or stayed stable after optimization.**

Polars achieved the best throughput after optimization.

Analysis

POLARS > PANDAS = PANDAS + THREADING > DASK

- **Polars** dominated in speed and throughput, proving to be the most efficient for our scale.
- **Pandas + Threading** showed only a minor performance gain — threading helped, but not significantly.
- **Dask**, while powerful, was slower before optimization and only caught up slightly after. It shines more with very large datasets.

07 CHALLENGES

WE WILL SOLVE THE PROBLEMS

01 Challenges Encountered

Data was loaded dynamically via JavaScript, which required tools like Selenium, increasing processing time and system load.

02 IP Blocking from Crawling

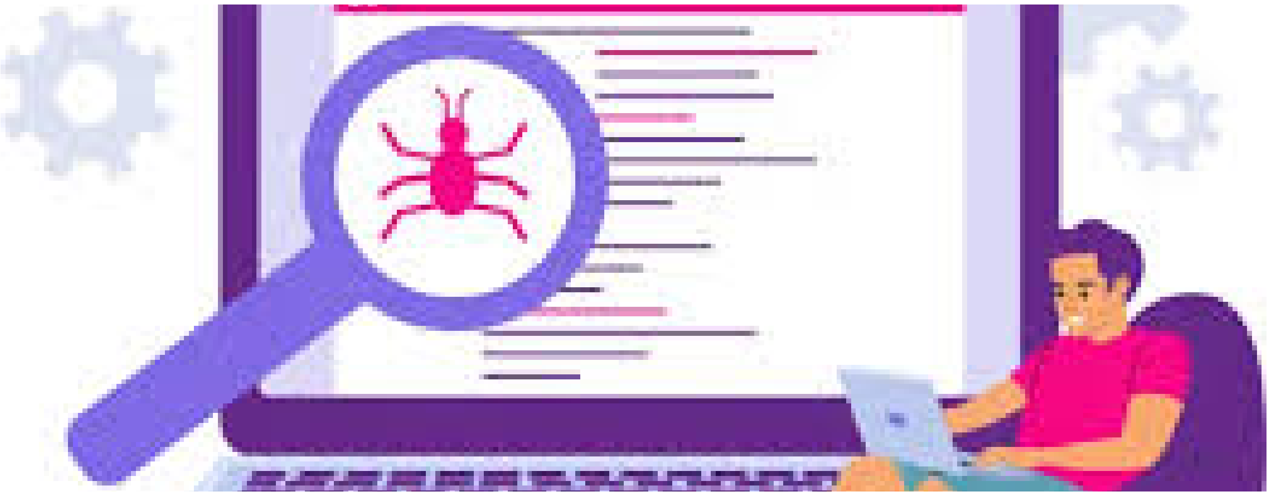
High request volume triggered anti-bot measures. Used delays and user-agent rotation to avoid bans, but this slowed scraping.

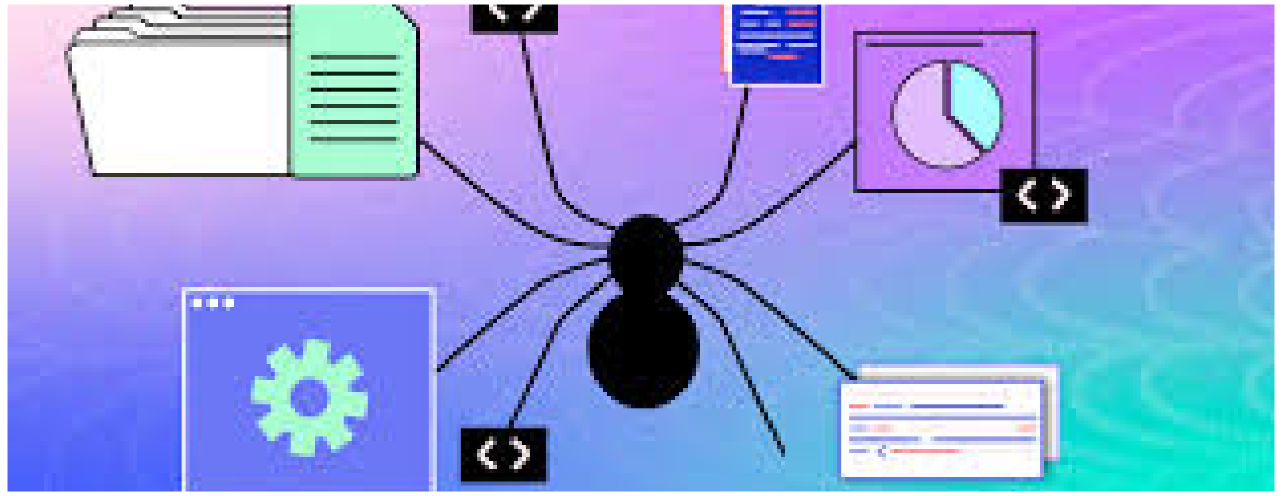
03 Inconsistent Data Structure

Page layouts varied, causing parsing issues. Required extra handling for missing or mislabeled fields.

07 LIMITATIONS

SYSTEM LIMITATIONS





01 Site Structure Dependency

Scraper depends on specific HTML structure. Any website update could break it and require script adjustments

02 Local Hardware Constraints

Processing speed was limited by CPU and memory. JavaScript-heavy pages and large datasets were resource-intensive.

08 CONCLUSION

Summary of Findings

- **Optimization** significantly **reduced processing time** across all libraries
- **CPU and memory usage** became **more efficient** after optimization
- **Throughput increased by 10x** or more in most libraries
- **Successfully processed over 100,000** valid records without major bottlenecks

Conclusions

We proved that optimizing our crawler made a big difference. It worked faster, used less CPU and memory, and handled over 100,000 records easily.

- **Polars** was the **fastest**,
- **Pandas with threading** was **slightly better** than normal Pandas,
- **Dask** is better for very big data, **not for this small scale.**

PRESENTED BY GROUP G

1. CHEN PYNG HAW
2. MUHAMMAD DANIAL BIN AHMAD SYAHIR
3. LOW JIE SHENG
4. NADHRAH NURSABRINA BINTI ZULAINI