UTM

UNIVERSITI TEKNOLOGI MALAYSIA

FACULTY OF COMPUTING

SECP3133-02 HIGH PERFORMANCE DATA PROCESSING

PROJECT 1 - REPORT

TITLE: OPTIMIZING HIGH-PERFORMANCE DATA PROCESSING FOR LARGE-SCALE WEB CRAWLERS

PREPARED BY: GROUP 3

NAME	MATRIC NO.
MUHAMMAD DANIEL HAKIM BIN SYAHRULNIZAM	A22EC0207
NICOLE LIM TZE YEE	A22EC0123
NUR ALEYSHA QURRATU'AINI BINTI MAT SALLEH	A22EC0241
WONG KHAI SHIAN NICHOLAS	A22EC0292

PREPARED FOR: DR. ARYATI BINTI BAKRI

DATE: 26/05/2025

Table of Contents

1.0 Introduction

This section introduces the purpose, scope, and motivation of the project, focusing on large-scale web crawling using high-performance computing (HPC) techniques.

1.1 Background of the Project

In the era of big data, web-based information has become a crucial asset for a wide range of industries and research fields. Websites, especially news portals, generate vast amounts of dynamic and continuously updated data that can offer valuable insights when collected and analyzed effectively. However, the process of gathering this data at scale introduces several technical challenges such as dynamic page rendering, data redundancy, ethical scraping, and performance bottlenecks during processing.

High-Performance Computing (HPC) techniques offer solutions to these challenges by improving the efficiency, scalability, and reliability of web crawling systems. Techniques such as multithreading, multiprocessing, and distributed computing allow data engineers to handle large volumes of web data within reasonable time and resource constraints.

This project focuses on designing and implementing a high-performance data collection and processing pipeline through a web crawler. The crawler is optimized using HPC techniques to collect structured data efficiently and effectively from a Malaysian website.

1.2 Objectives

The main objectives of this project are as follows:

- To develop a robust web crawler capable of extracting a minimum of 100,000 structured records from a Malaysian website, which is Utusan Malaysia.
- To clean, transform, and store the collected data in a structured format suitable for downstream applications or analysis.
- To evaluate the performance of the system before and after optimization using measurable performance metrics such as execution time, CPU usage, memory consumption, and throughput.

1.3 Target Website and Data to Be Extracted

The selected data source for this project is Utusan Malaysia (https://www.utusan.com.my/), a prominent and long-standing Malaysian news organization with a significant online presence. Utusan Malaysia publishes a wide array of news articles covering various categories, including national news, the economy, the Malaysian Ringgit, international news, sports, features, entertainment, and politics.

The specific data fields targeted for extraction are as follows:

- Article Title: The main title or headline of the published news article.
- Publication Date: The precise date on which the news article was published online.
- Article Category: The designated section or category under which the news article is classified on the website (e.g., Nasional, Ekonomi, Sukan).
- Article URL: The unique Uniform Resource Locator (web address) that links directly to the full content of the article.

To achieve the project's data acquisition target, the developed web crawler will be designed to navigate across multiple pages of the Utusan Malaysia website, systematically parse the HTML structure (and potentially utilize Selenium for dynamic content) to extract the relevant information for each identified article within the responsible sections. The extracted and cleaned records will be stored in a standardized Comma Separated Values (CSV) file format. Throughout the scraping process, we will adhere to ethical best practices by diligently respecting the website's robots.txt policy and implementing appropriate rate-limiting and automated retry mechanisms to prevent server overload and ensure responsible data collection.

The team will focus on the following sections based on individual responsibilities:

- Aleysha - Nasional
- Nicole - Ekonomi, Ringgit, Luar Negara
- Daniel - Sukan, Rencana, Pancaindera
- Nicholas - Gaya, Komuniti

2.0 System Design & Architecture

Building an efficient web crawler isn't just about writing scripts; it requires careful planning of how different parts work together. In this section, we break down the architecture that powers our high-performance data pipeline, explain the rationale behind our tool choices, and highlight how our team collaborated to bring the system to life.

2.1 Description of Architecture

The architecture of our high-performance web scraping system for Utusan Malaysia follows a distinct multi-stage process, emphasizing a centralized storage for raw data before cleaning and optimization.

The architecture developed for scraping and processing content from the Utusan Malaysia website is a modular and efficient pipeline tailored for structured data extraction at scale. In order to evaluate the effectiveness of optimization, performance metrics are recorded, which include execution time, CPU usage, memory usage, and throughput. The results are visualized through charts and graphs to provide insights into how HPC methods improve performance.

It begins with a request scheduler that manages a prioritized queue of URLs, implements rate limiting, and respects robots.txt directives to ensure responsible crawling behavior. The web crawling component handles HTML content downloading and parsing using tools such as BeautifulSoup, Selenium, Selectolax, and Playwright, depending on the nature of the page and the complexity of the content. This flexible crawler setup ensures compatibility with both static and dynamic web pages.

Raw data extracted from different sections of the website is saved into dedicated CSV files, each representing a specific category like national news, economy, or lifestyle. These files are then ingested into MongoDB, where both the individual CSVs and a combined dataset are stored for further processing. The data processing pipeline involves standard cleaning operations, such as removing duplicates and handling missing values, followed by data transformation to normalize fields like publication date and content category. To support high-volume data

operations, the pipeline leverages high-performance computing tools like Dask, Swifter, Modin, and Polars, enabling faster and more scalable data manipulation.

Performance metrics such as execution time, CPU and memory usage, and data throughput are monitored to assess the system's efficiency. Finally, the cleaned and standardized data is stored back in MongoDB, ready for downstream tasks such as analytics, reporting, or machine learning. This architecture provides a robust, scalable solution for continuously extracting and processing news content from a dynamic and content-rich site like Utusan Malaysia.

The system's components and data flow are illustrated in Figure 2.1 below.

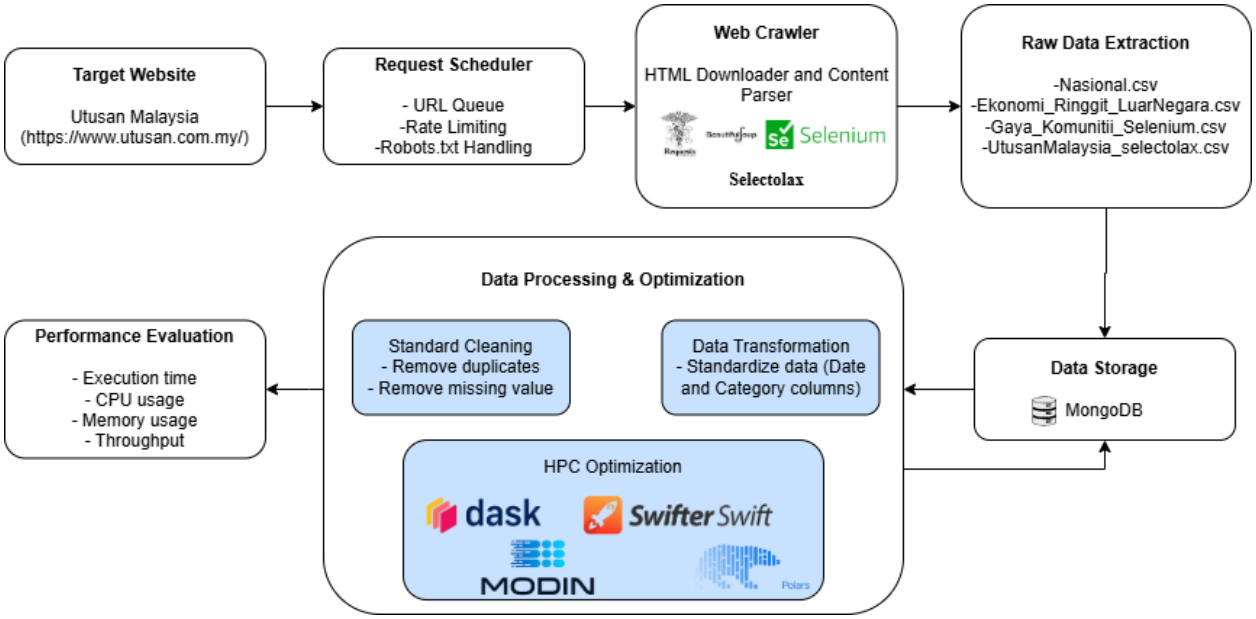


Figure 2.1: System Architecture

2.2 Tools and Frameworks Used

Our project leverages a diverse set of tools and frameworks to ensure efficient web scraping. The Requests library serves as a simple and elegant HTTP client for Python, facilitating the fetching of web pages. For parsing static HTML content, BeautifulSoup is employed due to its ease of use and straightforward syntax for extracting elements. Selenium becomes essential when dealing with dynamically loaded content or scenarios requiring user interaction, such as clicking to expand menus or paginate through results. For rapid and repetitive scraping of numerous static

pages, Selectolax is the preferred choice. Furthermore, Asyncio is utilized to enable asynchronous operations, allowing multiple page loads and content parsing processes to occur concurrently without blocking the main thread, thereby enhancing overall efficiency. Table 2.2.1 shows the libraries used to scrape the selected website, along with descriptions.

Table 2.2.1 Libraries Used

Library	Description
Request	A simple and elegant HTTP library for Python, used to send HTTP requests to fetch web pages.
BeautifulSoup	Easy-to-use HTML parser; good for static HTML; simple syntax for extracting elements by tag/class.
Selenium	Needed when content loads dynamically or requires interaction (e.g., click to expand menu or paginate).
Selectolax	Perfect for scraping many static pages quickly and repeatedly.
Asyncio	Allows multiple page loads and content parsing to occur without blocking the main thread.

2.3 Roles of Team Members

Table 2.3.1 shows the roles assigned to each member for a smooth and organized workflow.

Table 2.3.1 Roles of Team Members

Team Member	Role	Task Distribution
Nicholas	Team Leader	● Choose a Malaysian Website ● Web crawler library used: Selenium ● Optimization library used: Swifter
Nur Aleysha	Web Crawler Developer	● Web crawler library used: BeautifulSoup + Asyncio ● Create and load raw and clean data into MongoDB ● Optimization library used: Dask
Nicole	Data Handler	● Combine all scraped data to load in MongoDB ● Web crawler library used: BeautifulSoup + Selenium ● Optimization library used: Modin

Daniel Hakim	Technical Support & Tester	• Web crawler library used: Selectolax • Optimization library used: Polars

3.0 Data Collection

Gathering data from a large news site like Utusan Malaysia was not without its challenges. From inconsistent page structures to avoiding server overload, this section highlights how we navigated these hurdles and still extracted over 160,000 structured records responsibly.

3.1 Crawling Method

The data collection process involved a focused web crawling approach to gather information from the Utusan Malaysia website. To systematically navigate the website's structure, a combination of URL pattern manipulation and direct URL access was employed. The base URL for the website is https://www.utusan.com.my. For sections organized with pagination, the crawling process utilized a consistent URL structure: https://www.utusan.com.my/category/[category_name]/[subcategory_name]/page/[page_no]. This pattern allowed the crawler to iterate through multiple pages within a given subcategory by incrementing the page_no parameter, as demonstrated by the example URL: https://www.utusan.com.my/category/ringgit/hartanah/page/2/.

Conversely, certain sections of the website, specifically 'Ringgit' and 'Pancaindera', did not employ pagination. For these sections, the data was retrieved directly from the base URL associated with the category name: https://www.utusan.com.my/[category_name], as exemplified by the URL: https://www.utusan.com.my/pancaindera/. This approach ensured that all available data within these non-paginated sections was captured efficiently.

The crawler was designed to extract the following data fields from the HTML structure of the web pages:

- **Article Title:** The title of the article was extracted by locating the <h3> HTML element with the class name 'jeg_post_title'. The text content within this element, after removing any leading or trailing whitespace, was then captured as the article title.

- **Publication Date:** The publication date was obtained by identifying the element with the class name 'jeg_meta_date'. The text content within this element, after removing any leading or trailing whitespace, was extracted.
- **Article Category:** The category of the article was derived from the URL structure itself. The [category_name] part of the URL was used to assign the article to its respective category.
- **Article URL:** The full URL of the article was extracted by locating the <a> tag within the <h3> element (identified for the title). The 'href' attribute of this anchor tag provided the complete URL.

3.2 Number of Records Collected

Our crawler configurations are precisely defined to target these specific sections and subcategories to ensure comprehensive and non-overlapping data collection. The web crawler modules then fetch content from their designated sections, employing the most suitable scraping libraries:

- Crawler 1, designated for a focused subset of the Nasional section, specifically targets the subcategories of Jenayah (Crime), Tragedi (Tragedies), and Mahkamah (Courts), utilizing Playwright to handle dynamically loaded web pages and Beautiful Soup for parsing and extracting the desired information.
- Crawler 2 is responsible for the Ekonomi, Ringgit, and Luar Negara sections, employing Requests and Beautiful Soup for general content retrieval while also incorporating Selenium to handle dynamically loaded elements.
- Crawler 3 is dedicated to the Sukan, Rencana, and Pancaindera sections, leveraging the speed and efficiency of Selectolax for parsing.
- Finally, Crawler 4 is designed for the Gaya and Komuniti (subcategory of Nasional) sections, utilizing Requests and Beautiful Soup, with Selenium integrated to manage any dynamically rendered content.

Table 3.2.1 summarizes the number of records collected by each of the four crawlers used in the project.

Table 3.2.1 Numbers of Records Collected

Crawler	Records Collected
Crawler 1	28544
Crawler 2	48847
Crawler 3	42450
Crawler 4	41854
Total	161695

3.3 Ethical Considerations

To ensure responsible data collection, the web scraper was developed following ethical guidelines:

- The target website, Utusan Malaysia, is a public news portal, and only data that was openly accessible was collected. The web scraping process strictly adhered to the website's robots.txt to avoid accessing restricted areas, adhering to the site usage policy.
- No private data or personal information was collected or stored. The content collected consisted only of article metadata like titles, publish dates, categories, and URLs, which were already publicly accessible from the publisher.
- The scrapers were set with proper rate-limiting and delays to avoid putting too much load on the host server. The approach reduced opportunities for service interruption or damage to the operationality of the website.
- The data collected was solely utilized for research and academic purposes for the undertaking of the project. There is no redistribution of the scraped content as intended.

4.0 Data Processing

Before any meaningful analysis can take place, raw data must be refined. This section outlines the systematic cleaning, transformation, and structuring of our collected data to ensure accuracy, consistency, and compatibility across various high-performance processing tools. From handling nulls and duplicates to formatting dates and categories, these steps prepare the dataset for high-quality analysis and storage.

4.1 Cleaning Methods

To ensure data quality and consistency, the collected dataset underwent a series of cleaning procedures, effectively transforming it into a refined format suitable for analysis. Key procedures include the identification and removal of missing values and duplicate records.

The process starts with the identification and removal of null values. Table 4.1.1 explains the functions used to handle null values.

Table 4.1.1 Handling null values

Python Code	Explanation
`df.isnull().any()`	This function is designed to identify the presence of null values, returning "True" for any column containing missing data. Based on Figure 4.1.1, it was confirmed that the dataset does not contain any null values. `# Check null value` `df.isnull().any()` 0 Title False Category False Date False URL False dtype: bool **Figure 4.1.1 Null Values Output**
`df2 = df2[['Title', 'Category', 'Date', 'URL']].dropna()`	The line of code removes rows containing null values from the DataFrame.

Subsequently, the process was continued by removing duplicate values. A detailed explanation is provided in Table 4.1.2 below.

Table 4.1.2 Remove Duplicate Records

Python Code	Explanation
df2['URL'] = df2['URL'].apply(lambda x: str(x).strip() if isinstance(x, str) else x) df2.drop_duplicates(subset='URL', keep='first', inplace=True)	The code removes any leading or trailing spaces from URL strings to standardize the data. Then, it eliminates duplicate rows based on the 'URL' column, keeping only the first occurrence of each unique URL. This step prevents skewed analysis results due to repeated data.

4.2 Transformation and Formatting

The collected data also underwent transformation to ensure its quality and consistency, processing the data into a usable form for subsequent data analysis. These steps included standardizing the ‘Category’ and ‘Date’ columns.

The process begins by standardizing the ‘Category’ column, with the specific methodology detailed in Table 4.2.1.

Table 4.2.1 Standardize ‘Category’ Columns

Python Code	Explanation
df2['Category'] = df2['Category'].astype(str).str.capitalize()	The code standardizes the 'Category' column by first converting all values to strings and then capitalizing the first letter of each category while lowercasing the rest (e.g. “Jenayah”, “Ekonomi”). This ensures consistency and avoids issues caused by variations in capitalization.

Next, rows with ‘No Date’ values are identified and removed as they are not suitable for time-based analysis. The line of code is explained in Table 4.2.2.

Table 4.2.2 Remove Unnecessary Rows

Python Code	Explanation
date['Date_Format'] = date['Date'].apply(identify_date_pattern)	The function identifies each entry in the 'Date' column and stores the result in a new column, Date_Format.
format_counts = date['Date_Format'].value_counts()	Counts the occurrences of each unique date format in the Date_Format column.
print(format_counts)	Displays the number of occurrences for each unique date format, including rows labeled as 'no date'. Based on Figure 4.2.1, the dataset contains 15 rows with 'no date' values in the 'Date' column. dd jun dddd, dd:dd am 3270 dd jun dddd, dd:dd 3004 no date 15 **Figure 4.2.1 Rows with 'no date'**
df2 = df2[~df2['Date'].astype(str).str.strip().str.lower().eq('no date')]	This code filters out rows with "no date" in the 'Date' column. It converts the column to strings, removes extra spaces, makes it lowercase, and then filters to keep only rows where the date is not "no date".

Consequently, the 'Date' column is reformatted to allow accurate sorting and filtering. The string format DD Month YYYY, HH:MM AM/PM is converted into the standard format YYYY-MM-DD HH:MM:SS (e.g. 30 April 2025, 3:13 PM is converted into the standard format 2025-04-30 15:13:00). The step-by-step transformation process is detailed in Table 4.2.3.

Table 4.2.3 Reformat 'Date' Column

Python Code	Explanation
 **Figure 4.2.2 Reformat 'Date' column**	This function, shown in Figure 4.2.2, standardizes date strings. It uses a predefined dictionary (month_map) to convert month names (in Malay) to numbers and a function (convert_date) to reformat dates to "YYYY-MM-DD HH:MM:00". The function parses the date string using a regular expression, handles AM/PM, and includes error handling.

4.3 Data Structure

After completing the data cleaning processes using five different libraries (Pure Python, Polars, Modin, Dask, and Swifter), the resulting cleaned datasets were exported into CSV format and uploaded into separate collections in MongoDB for structured storage and future retrieval. The raw datasets were also uploaded into a collection in MongoDB before cleaning (Raw_Data).

Each cleaned dataset maintains a consistent structure, as shown in Table 4.3.1:

Table 4.3.1 Data Structure and Storage of Cleaned Datasets

Field Name	Data Type	Description
Title	String	News article title
Category	String	Section or category (e.g., Ekonomi, Agama) – standardized with capitalization
Date	Datetime	Publication date – cleaned and standardized to YYYY-MM-DD HH:MM:SS format
URL	String	Unique article link

Storage Details:

- CSV Files: Each library's cleaned dataset is exported as a CSV file (e.g., cleaned_polars.csv, cleaned_modin.csv) for comparison and external use.

- MongoDB Collections: Cleaned data is also stored in MongoDB under the database webcrawler_project, with separate collections per library, which are, cleaned_python_data, cleaned_polars_data, cleaned_modin_data, cleaned_dask_data, and cleaned_swifter_data.

Figure 4.3 shows the webcrawler_project database in MongoDB, which consists of six collections for the raw dataset and clean datasets. The Raw_Data collection stores the unprocessed data obtained directly from the web crawling process, with a total of 161,695 data and the remaining five collections are the cleaned datasets, which collectively contain 126,294 data after data processing and optimization.

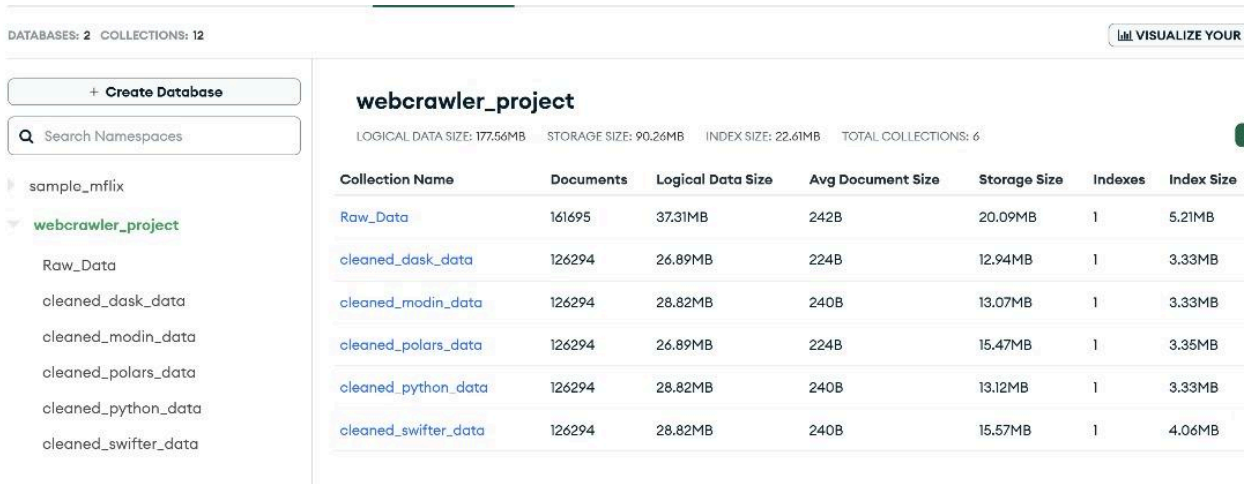


Figure 4.3 Summary of Data Collection in webcrawler_project in MongoDB

5.0 Optimization Techniques

In the field of data processing and analysis, various Python libraries have emerged to address performance challenges associated with large datasets. This section outlines the primary optimization methods utilized, including Polars, Modin, Dask, and Swifter. These techniques aim to leverage parallel processing and distributed computing to expedite computations and handle large datasets more effectively.

5.1 Optimization Methods Used

In order to enhance the performance and efficiency of data processing, several optimization techniques can be employed.

Swifter is a Python library designed to accelerate the performance of pandas.apply() operations by intelligently choosing the most efficient execution strategy. Depending on the size of the dataset and the complexity of the function, Swifter automatically decides whether to use pandas, Dask, or Numba. This makes it particularly useful for large-scale data transformations, as it enables parallel execution with minimal changes to the existing code, thereby maximizing CPU utilization efficiently.

Modin is another library aimed at enhancing the performance of pandas by distributing computations across all available CPU cores. It allows users to significantly speed up their pandas workflows without needing to modify much of their existing code. Modin is especially effective for working with large datasets, as it parallelizes operations to allow for faster and more efficient data manipulation and analysis.

Dask is a flexible parallel computing library in Python that extends the capabilities of pandas and NumPy to handle larger and more complex workloads. It enables parallel computing both on single machines and distributed clusters. Dask is well-suited for working with datasets that exceed memory limits by dividing them into smaller chunks and processing them in parallel. It also offers high-level abstractions for arrays, dataframes, and machine learning workflows, simplifying the parallelization of complex computations.

Polars is an open-source library designed for high-performance data manipulation and is recognized as one of the fastest data processing tools available for single-machine environments. It employs lazy evaluation, which means it constructs a query plan before executing any operations, allowing for comprehensive optimization of the data pipeline. Additionally, Polars leverages a multithreaded execution engine that parallelizes tasks efficiently, contributing to its exceptional processing speed and performance.

A brief description of these optimization methods is provided in **Table 5.1.1**, located in **Section 10.3.1 Description of Optimization Methods Used**, under the appendices section.

5.2 Code Overview of Techniques Applied

This section provides an overview of key data processing techniques implemented in the project. Each technique is supported with relevant code snippets and explanations, highlighting how these techniques optimize the dataset for further analysis.

5.2.1 Swifter

The performance improvements observed in this part of the data cleaning pipeline stem from Swifter's ability to automatically optimize .apply() operations using parallel processing or vectorized execution, depending on the workload size and complexity. All these codes can be referred to **Appendix** in **Table 10.1.2 Data Processing and Optimization Code Snippets.**

1. Figure 5.2.1.1 introduces the Swifter library, a lightweight yet powerful tool designed to speed up Pandas' .apply() operations without requiring major code changes. Swifter intelligently determines the best execution strategy which could be vectorized, Dask-based, or multiprocessing, to accelerate row-wise operations. The import is seamless and integrates directly with the standard Pandas workflow.

```
import swifter
```

Figure 5.2.1.1 Import Swifter library

2. The code shown in Figure 5.2.1.2 is the core performance enhancement line. In the non-optimized version, the .apply() function is executed sequentially, which can be slow for large datasets. By replacing .apply() with .swifter.apply(), the convert_date function is applied in parallel across multiple cores, improving speed without changing the function logic.

```
df['Date'] = df['Date'].swifter.apply(convert_date)
```

Figure 5.2.1.2 swifter.apply()

3. To complement the optimization, detailed performance tracking was added to benchmark Swifter's efficiency shown in Figure 5.2.1.3. By isolating timing and computing throughput (i.e., rows processed per second), we were able to quantify the performance

gains relative to the unoptimized Pandas version. This provided valuable insights into the resource savings and speedups achieved through parallelism.

```
swifter_end_time = time.time()
swifter_execution_time = swifter_end_time - swifter_start_time
swifter_cpu_percent = psutil.cpu_percent(interval=1)
swifter_memory_usage = process.memory_info().rss / (1024 ** 2)
swifter_throughput = len(df) / swifter_execution_time
```

Figure 5.2.1.3 Performance Tracking Benchmark

5.2.2 Modin

The performance enhancements observed in this code are largely due to Modin's capability to parallelize operations, a feature that is effectively enabled by its integration with the Ray execution engine. All these codes can be referred to **Appendix** in **Table 10.1.2 Data Processing and Optimization Code Snippets.**

1. Instead of the conventional Pandas library, Modin is imported as 'pd' as shown in Figure 5.2.2.1. Modin is designed to be a drop-in replacement for Pandas, offering an identical API but with significantly improved performance, especially for larger datasets. This performance gain is achieved through Modin's ability to parallelize Pandas operations across multiple CPU cores.

```
import modin.pandas as pd
```

Figure 5.2.2.1 Import Modin as pd

2. The line in Figure 5.2.2.2 explicitly instructs Modin to utilize the Ray framework as its execution engine. Ray is a powerful distributed computing framework that facilitates the parallel execution of tasks. By setting the 'MODIN_ENGINE' environment variable to 'ray', we enable Modin to leverage Ray's capabilities to distribute both data and computations. This allows Modin to efficiently utilize available system resources, leading to substantial speedups in data processing.

```
import os
os.environ["MODIN_ENGINE"] = "ray"
```

Figure 5.2.2.2 Utilize Ray Framework

5.2.3 Dask

Dask's ability to handle large-scale data processing allows for optimizations through parallelism which makes it suitable for datasets that exceed memory limits or require high throughput processing. All these codes can be referred to **Appendix** in **Table 10.1.2 Data Processing and Optimization Code Snippets.**

1. Imports the Dask library, aliasing it to dd for easier use as shown in Figure 5.2.3.1. The dataset is converted into a Dask dataFrame, which splits the data into multiple smaller partitions and processes them in parallel.

```
import dask.dataframe as dd
```

Figure 5.2.3.1 Import Dask Library as dd

2. Figure 5.2.3.2 illustrates the number of partitions is dynamically set based on the available CPU cores, ensuring optimal parallelism. This enables each partition to be processed concurrently, significantly reducing execution time.

```
# Optimal partition count
    npartitions = multiprocessing.cpu_count()
    dask_df = dd.from_pandas(df, npartitions=npartitions)
```

Figure 5.2.3.2 Optimal Partition Count

3. According to Figure 5.2.3.3, "map_partitions" applied the cleaning logic independently to each partition, which preserves the lazy evaluation model and ensures that operations like string parsing and datetime conversion are only performed when compute() is explicitly called.

```
# Apply cleaning in partitions
    dask_df = dask_df.map_partitions(clean_partition)
# Drop duplicates by URL
    dask_df = dask_df.drop_duplicates(subset='URL')
 # Final compute
    cleaned_dask_df = dask_df.compute()
```

Figure 5.2.3.3 Apply "map_paritions"

5.2.4 Polars

Leverages lazy evaluation through its LazyFrame. Instead of executing operations immediately, Polars builds an optimized query plan. The actual computation only happens when you call .collect() on the LazyFrame. This allows Polars to analyze the entire sequence of operations and potentially optimize them for speed and memory efficiency. All these codes can be referred to **Appendix** in **Table 10.1.2 Data Processing and Optimization Code Snippets.**

1. Imports the Polars library, aliasing it to pl for easier use as shown in Figure 5.2.4.1.

```
import polars as pl
```

Figure 5.2.4.1 Import Polars Library as pl

2. The Polars code segment in Figure 5.2.4.2 focuses on efficiently cleaning and preparing a DataFrame using lazy evaluation. It begins by converting the input DataFrame into a LazyFrame, which delays the execution of operations to allow Polars to optimize the entire query plan. The subsequent chained operations define this plan: rows with null values are removed, and rows with the string "no date" in the "Date" column are filtered out. The "Category" column is standardized by capitalizing the first letter of each word, and the "Date" column undergoes transformation to handle Malay month names and is parsed into a datetime format. Finally, rows with parsing failures in the "Date" column are removed, and the data is deduplicated based on the "URL" column. The collect() function then executes this optimized query plan, producing the cleaned DataFrame.

```
# Convert to LazyFrame for optimization
lazy_df = df_polars.lazy()

# Clean & prepare columns
lazy_df = (
    lazy_df
    .drop_nulls()
    .filter(~pl.col("Date").str.to_lowercase().str.contains("no date"))
    .with_columns([
        pl.col("Category").str.to_titlecase().alias("Category"),
        pl.col("Date").map_elements(replace_malay_months, return_dtype=pl.Utf8)
    ])
    .with_columns([
        pl.col("Date")
        .str.replace_all(r"(am|pm)", "")
        .str.replace_all(",", "")
        .str.strip_chars()
        .str.strptime(pl.Datetime, format="%d %m %Y %I:%M %p", strict=False)
        .alias("Date")
    ])
    .drop_nulls(subset=["Date"])  # Remove failed datetime parsing rows
    .unique(subset=["URL"])       # Deduplicate by URL after date validation
)

# Execute optimized plan
df_cleaned = lazy_df.collect()
```

Figure 5.2.4.2 Lazy Evaluation

6.0 Performance Evaluation

This section presents the performance comparison of the data processing before and after optimization. The performance metrics measured include execution time, CPU usage, memory usage, and throughput across different libraries and optimization methods. There are five libraries used to do the cleaning process which are Pure Python (Pandas), Polars, Modin, Dask, and Swifter. Pure Python served as the baseline with no optimization, while the other four libraries are optimized libraries.

Based on Figure 6.1, the running times of different libraries were compared: Pandas (without optimization), Polars, Modin, Dask, and Swifter. Pandas with no optimization had the longest running time, and Modin had the shortest, which indicates that Modin is the most efficient as the optimized library in this case. Polars presents a minimal improvement with only 0.01 seconds faster than Pandas. However, Modin, Dask, and Swifter show significant decreases in running time most likely due to the parallelized and distributed data processing. The relatively poor performance of Polars suggests that it may not be best suited for dealing with smaller data sets in the range of 100,000 to 200,000 rows, likely due to the overhead of its distributed computing environment, which is typically optimized for dealing with much larger data sets.

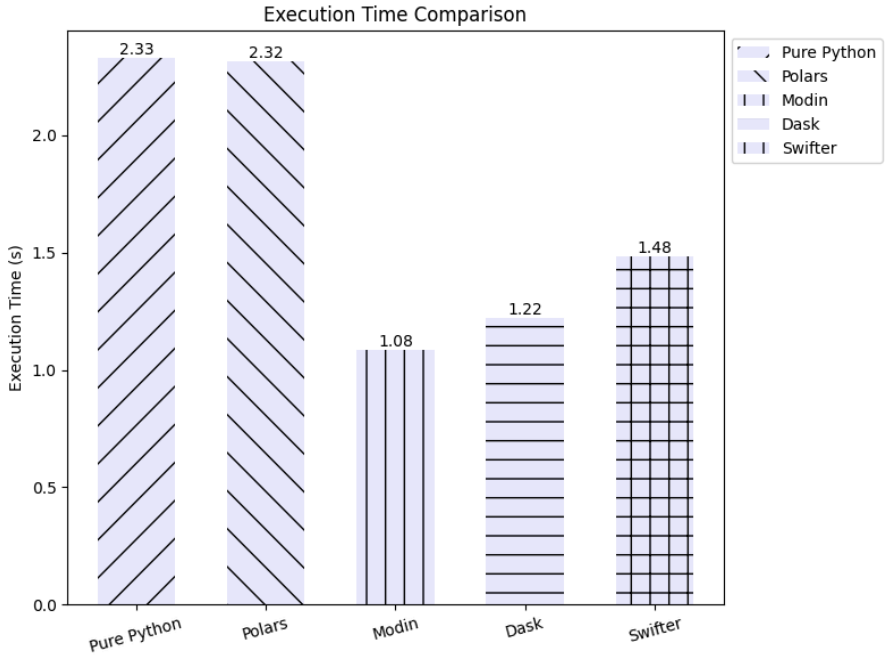


Figure 6.1 Comparison of Execution Time between Libraries

Based on Figure 6.2, the bar chart presents consistent consumption across all libraries. The memory usage values range from approximately 1524 MB to 1537 MB. This result indicates that despite the optimized library significantly improving speed and efficiency, memory usage remains fairly constant, which may be attributed to the uniform size and structure of the dataset.

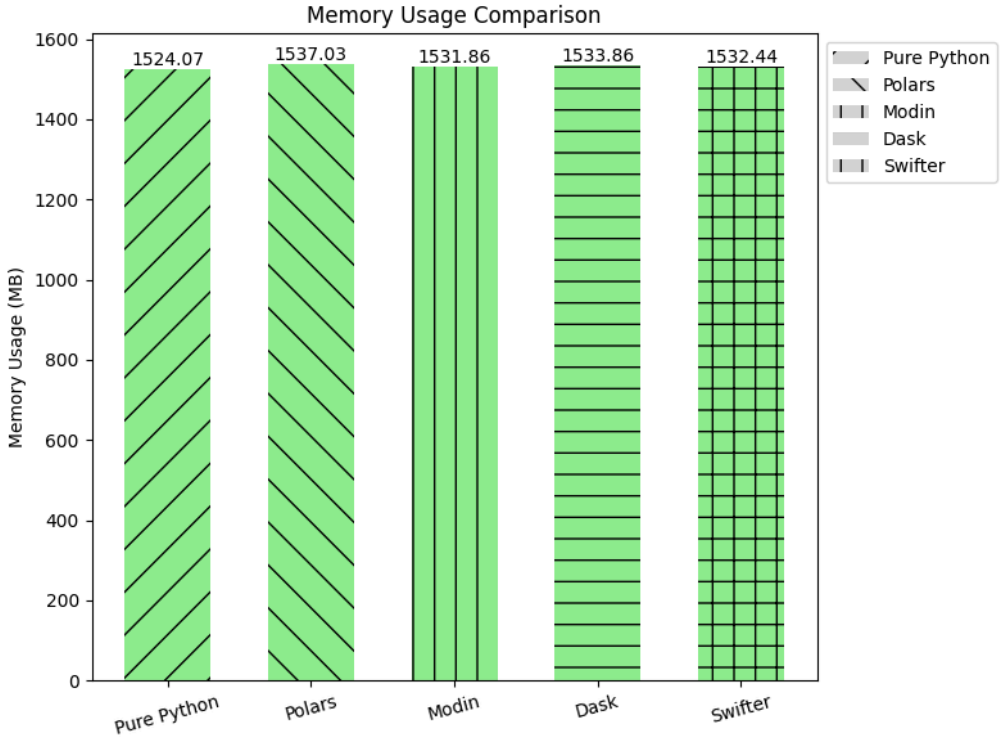


Figure 6.2 Comparison of Memory Usage between Libraries

Figure 6.3 illustrates the CPU usage comparison was made among different libraries. Pure Python (Pandas) showed the highest CPU usage, representing that without optimization, it is inefficient for big data processing. However, Swifter showed high CPU usage (36.9%) but still slower than Modin (16.8%) and Dask (74.4%). Additionally, Polars also showed high CPU usage with high execution time which can appear it is less efficient for local or small-scale setups. Modin's low CPU usage due to the efficient distributed task across cores using Ray. Whereas the slightly more CPU consumption for Dask is most likely due to task scheduling across its partitions while parallelized.

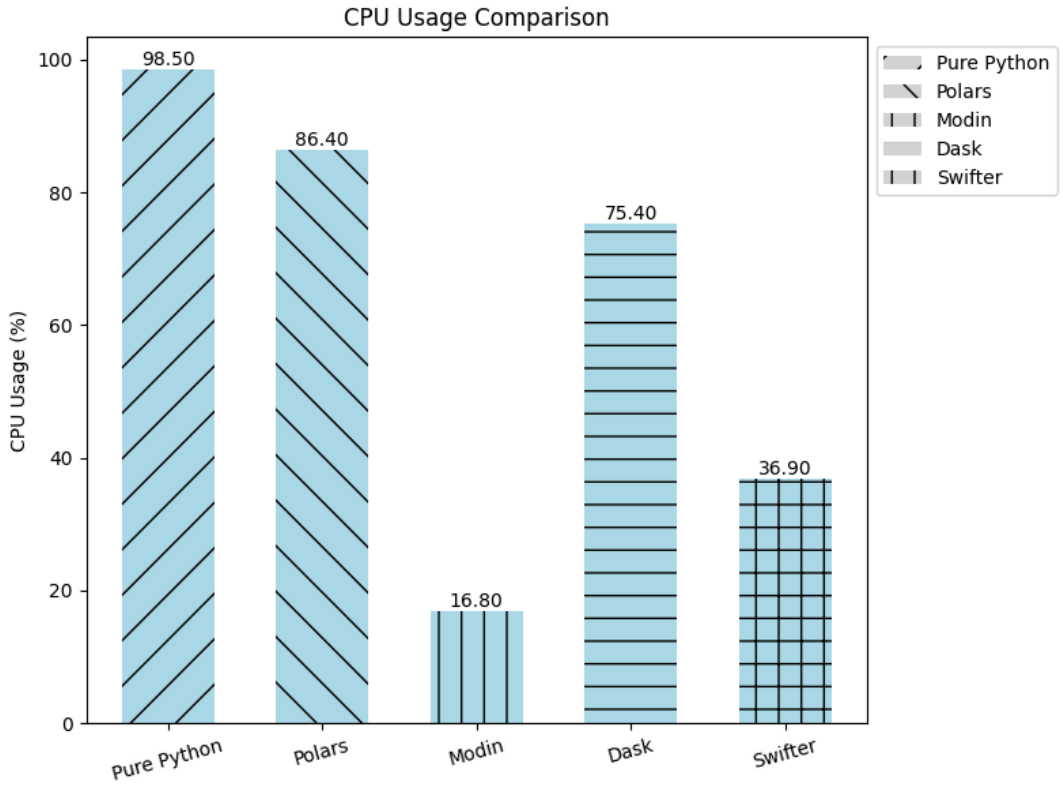


Figure 6.3 Comparison of CPU Usage between Libraries

Based on Figure 6.4, it demonstrates the comparison of throughput of different libraries. As illustrated by the chart, Modin achieved the highest throughput, followed by Dask and Swifter. These findings prove that parallel computing libraries can increase the rate of data processing compared to single-threaded execution. Pure Python (Pandas) and Polars showed the lowest throughput, around 54,000 rows per second, aligning with their slower overall execution time. Therefore, Polars as an optimized library used in this case appear to be the most inefficient in overall performance.

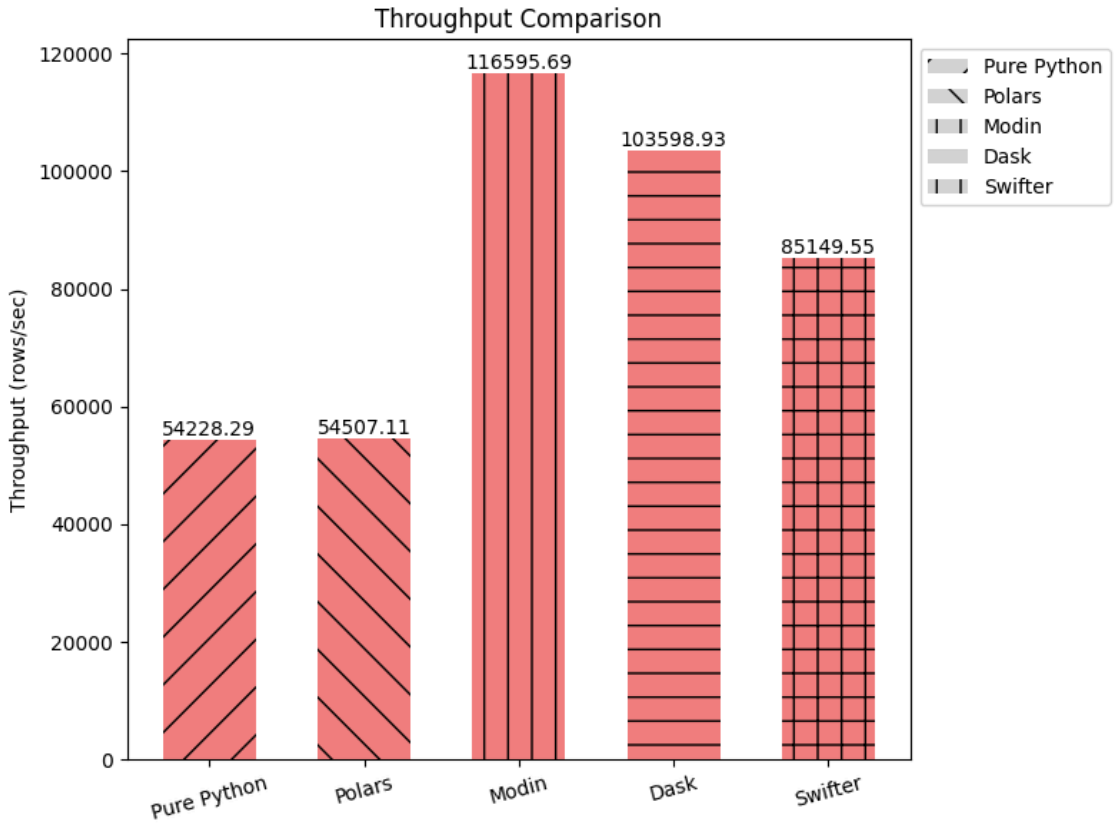


Figure 6.4 Comparison of Throughput between Libraries

Figure 6.5 illustrates a summary of performance metrics across five libraries in a table for easier comparison. The optimization techniques clearly demonstrated a significant improvement in execution time and processing efficiency. As shown in the performance metrics table, Modin consistently outperformed other methods in both speed and throughput, making it appear as the best approach for efficient data cleaning with minimal code changes. Dask also showed strong performance, but it will be more useful when scaling to larger datasets. Swifter offered a lightweight optimization path with reasonable performance improvements, especially when code simplicity is prioritized. On the other hand, Pure Python and Polars demonstrated less compelling results for the dataset size tested, with Polars likely constrained by overhead rather than computational inefficiency.

Comparison Table (Performance Metrics by Library):

Library	Final Row Count	Execution Time (s)	CPU Usage (%)	Memory Usage (MB)	Throughput (rows/sec)
Pure Python	126294	2.328932	98.5	1524.066406	54228.288957
Polars	126294	2.317019	86.4	1537.031250	54507.106233
Modin	126294	1.083179	16.8	1531.855469	116595.687643
Dask	126294	1.219067	75.4	1533.863281	103598.932118
Swifter	126294	1.483202	36.9	1532.441406	85149.548984

Figure 6.5 Comparison Table of Performance Metrics

7.0 Challenges and Limitations

Throughout the development of this project, several challenges were encountered that influenced the overall approach. Initially, the focus was on scraping data from the CompAsia website, followed by The Star. However, CompAsia's content was primarily limited to e-commerce products with minimal structural variety, restricting the volume and richness of data available for analysis. Meanwhile, The Star presented access limitations and pagination issues, which significantly affected the scale and speed of data extraction. Due to these obstacles, the project shifted its focus to Utusan Malaysia, which provided a well-organized archive, systematic category tagging, and sufficient data volume to support a comprehensive large-scale web crawling and analysis effort.

A notable limitation arose from the use of BeautifulSoup for web scraping. While BeautifulSoup is highly effective for static web content extraction, it proved inadequate when dealing with dynamic content that relies heavily on JavaScript rendering. To overcome this challenge, Selenium was integrated into the crawling framework to enable rendering of dynamic pages and ensure complete data capture. This integration improved the crawler's ability to handle modern web technologies and enhanced the overall robustness of the data collection process.

Finally, during the evaluation of available web crawling frameworks, Scrapy was considered for its powerful scraping capabilities and scalability. However, Scrapy was ultimately found to be overly complex for the scope of this project, and its numerous dependencies created compatibility issues with the Google Colab environment, complicating setup and deployment. As a result, the team opted for a more modular approach, combining Requests, BeautifulSoup, Selenium, and Selectolax. This approach offered a flexible, lightweight, and easier-to-manage solution that met the project's functional requirements without unnecessary overhead.

8.0 Conclusion and Future Work

This project showcased the successful integration of scalable crawling and high-performance data processing techniques to extract over 100,000 news articles from Utusan Malaysia. Tools like Selenium, Modin, and Polars enabled efficient data collection and transformation, while multithreading and benchmarking further improved performance. Looking ahead, the system can be enhanced with distributed crawlers, cloud-based processing, smarter error handling, and deeper analytical capabilities, which lay a strong foundation for future research and applications.

8.1 Summary of Findings

Throughout this project, our team set out to build an efficient and scalable solution to extract and process large-scale data from the Utusan Malaysia website. With a clear target of collecting over 100,000 news records, we combined both traditional and modern tools, including Selenium and Playwright for crawling and BeautifulSoup for parsing. For data cleaning and processing, we incorporated performance-oriented libraries like Modin, Dask, PySpark, and Swifter.

One of our main achievements was optimizing the crawler to systematically extract articles across multiple sections under the "Gaya" category, such as *agama*, *fesyen*, and *kesihatan*. With multithreading, we significantly cut down crawling time, and through intelligent page navigation and parsing strategies, we ensured consistency in the collected records.

On the processing side, we benchmarked different HPC techniques and observed that:

- Modin and Dask helped speed up typical Pandas operations without needing much code change.
- Swifter gave moderate speedups for row-wise operations, useful during data cleaning.
- Polars provided significant performance improvements, particularly for columnar operations and when optimizing query-like operations on DataFrames.

Overall, the project successfully demonstrated how high-performance computing tools can be integrated into real-world data crawling tasks, providing both scalability and efficiency.

8.2 Future Work

Although the project successfully achieved its core objectives, there remain several opportunities to enhance the system's robustness and scalability. Currently, the crawler operates on a single machine utilizing multithreading, which may limit its capacity when handling larger or more complex websites. Future work could involve implementing distributed crawling frameworks, such as Scrapy Cluster or Kafka pipelines, to enable parallel data collection across multiple machines. This approach would facilitate the crawling of larger websites or multiple news sources simultaneously, thereby improving scalability.

Another area for improvement is fault tolerance. Certain sections of the website occasionally failed to load properly due to JavaScript-heavy content or connection timeouts. Incorporating smarter retry mechanisms and more comprehensive error logging would help minimize data loss and simplify debugging processes. Such enhancements would make the crawling system more resilient and reliable during extended operations.

Incremental crawling is also a potential enhancement. At present, the crawler collects all articles in a single run, which may result in redundant data and unnecessary processing. Supporting incremental updates where only newly published or updated articles are crawled regularly would help maintain an up-to-date dataset while reducing resource consumption. Additionally, although all processing was performed locally during this project, future deployments could leverage cloud-based infrastructures, such as PySpark or Dask clusters on AWS or Google Cloud Platform, to improve scalability and reduce runtime when processing very large datasets.

Finally, the rich dataset generated through this project opens up many possibilities for post-crawling analysis. Future research could explore trend identification in Malaysian lifestyle topics, sentiment analysis over time, or topic clustering using advanced natural language processing techniques. This project has provided valuable hands-on experience in large-scale crawling and high-performance data processing, laying a strong foundation for building more sophisticated and scalable data pipelines in the future.

9.0 References

An introduction to DASK: The python data scientist's power tool. KDnuggets. (2024, December 16). https://www.kdnuggets.com/introduction-dask-python-data-scientist-power-tool#:~:text=Dask%20splits%20these%20large%20datasets,process%20of%20handling%20big%20data

Blazingly Fast Dataframe Library. Index - Polars user guide. (n.d.). https://docs.pola.rs/#key-features

Coditation. (2023, November 28). *High-performance data analysis with Polars: A comprehensive guide*. LinkedIn. https://www.linkedin.com/pulse/high-performance-data-analysis-polars-comprehensive-r0rdf/

Imran. (2022, March 30). *Faster Pandas operation with Swifter*. Abdullah Al Imran. https://www.imranabdullah.com/2022-03-30/faster-pandas-operation-using-swifter

Scale your pandas workflow by changing a single line of code#. Scale your pandas workflow by changing a single line of code - Modin 0+untagged.50.g8600760 documentation. (n.d.). https://modin.readthedocs.io/en/latest/index.html

Using Pandas on Ray (Modin)#. Using Pandas on Ray (Modin) - Ray 2.46.0. (n.d.). https://docs.ray.io/en/latest/ray-more-libs/modin/index.html

10.0 Appendices

This section includes sample code snippets, outputs, and screenshots that demonstrate the implementation and results of web scraping, data processing, and optimization techniques used in the project. These appendices serve as visual and technical references for the methodology described in earlier sections.

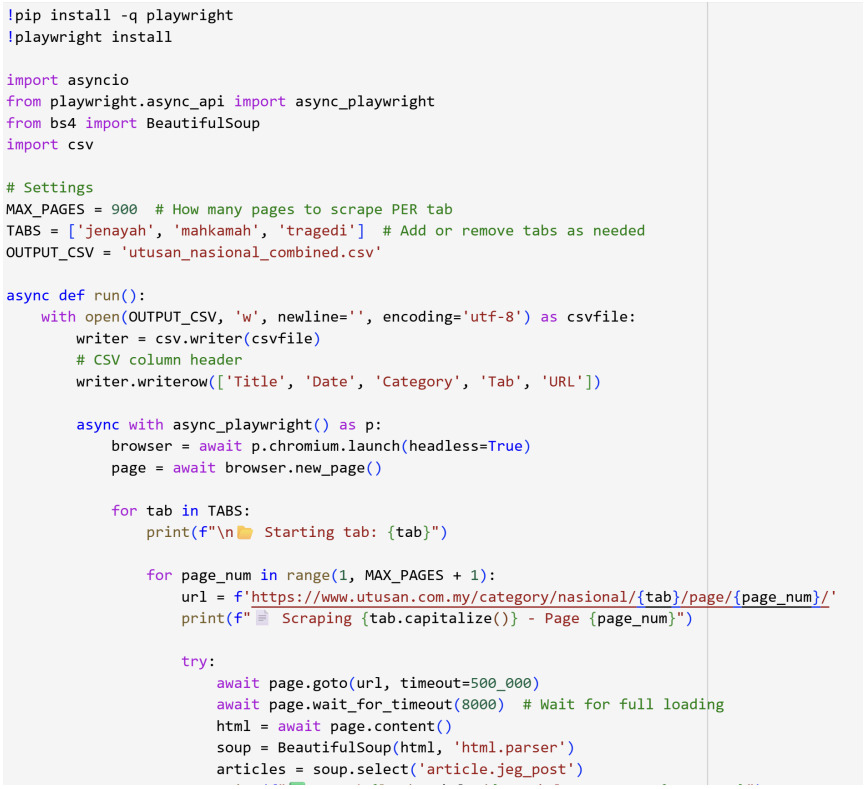
10.1 Sample Code Snippets

This section provides selected portions of the code used in the project to illustrate how specific tasks were performed, including scraping, data cleaning, and optimization.

Web Scraping

Table 10.1.1 includes sample code and logic used for extracting data from the target website, demonstrating the use of libraries and techniques applied in the web scraping process.

Table 10.1.1 Web Scraping Code Snippets

Section	Screenshot
Nasional_Scrape(BeautifulSoup_Playwright).ipynb	

```
!pip install -q playwright
!playwright install

import asyncio
from playwright.async_api import async_playwright
from bs4 import BeautifulSoup
import csv

# Settings
MAX_PAGES = 900  # How many pages to scrape PER tab
TABS = ['jenayah', 'mahkamah', 'tragedi']  # Add or remove tabs as needed
OUTPUT_CSV = 'utusan_nasional_combined.csv'

async def run():
    with open(OUTPUT_CSV, 'w', newline='', encoding='utf-8') as csvfile:
        writer = csv.writer(csvfile)
        # CSV column header
        writer.writerow(['Title', 'Date', 'Category', 'Tab', 'URL'])

        async with async_playwright() as p:
            browser = await p.chromium.launch(headless=True)
            page = await browser.new_page()

            for tab in TABS:
                print(f"\n📂 Starting tab: {tab}")

                for page_num in range(1, MAX_PAGES + 1):
                    url = f'https://www.utusan.com.my/category/nasional/{tab}/page/{page_num}/'
                    print(f"📄 Scraping {tab.capitalize()} - Page {page_num}")

                    try:
                        await page.goto(url, timeout=500_000)
                        await page.wait_for_timeout(8000)  # Wait for full loading
                        html = await page.content()
                        soup = BeautifulSoup(html, 'html.parser')
                        articles = soup.select('article.jeg_post')
```

Ekonomi_Ringgit_LuarNegara.ipynb	

```python
def scrape_page(url):
    for attempt in range(3):
        try:
            print(f"Loading: {url} (Attempt {attempt + 1})")
            response = requests.get(url, timeout=MAX_PAGE_TIMEOUT)
            if response.status_code == 200:
                return response.text
            print(f"⚠️ Error: {response.status_code}")
        except Exception as e:
            print(f"⚠️ Exception: {e}")
        time.sleep(RETRY_WAIT)
    print(f"❌ Failed to load {url}")
    return None

def scrape_category(relative_path, category_name, writer, max_records):
    total_scraped = 0
    page_num = 1

    while total_scraped < max_records:
        # Handle pagination logic
        url = f"{BASE_URL}{relative_path}" if category_name in NO_PAGINATION else f"{BASE_URL}{relative_path}page/{page_num}/"
        html = scrape_page(url)
        if not html:
            break

        soup = BeautifulSoup(html, 'html.parser')
        articles = soup.find_all('article', class_='jeg_post')
        if not articles:
            break

        for article in articles:
            if total_scraped >= max_records:
                break
            title_tag = article.find('h3', class_='jeg_post_title')
            title = title_tag.get_text(strip=True) if title_tag else 'No Title'
            link_tag = title_tag.find('a') if title_tag else None
            full_url = link_tag['href'] if link_tag and 'href' in link_tag.attrs else 'No URL'

            date_tag = article.find('div', class_='jeg_meta_date')
            date = date_tag.get_text(strip=True) if date_tag else 'No Date'

            # Use provided dictionary name as the category
            cat = category_name.capitalize()

            writer.writerow([title, full_url, date, cat])
            total_scraped += 1


        print(f"✅ {category_name.capitalize()} - Page {page_num} done. Total scraped: {total_scraped}")
```

Politik_Sukan_Rencana_Pancaindera_Selectolax.ipynb	

```python
###Use Selectolax library

import requests
import pandas as pd
import time
from selectolax.parser import HTMLParser

# Subcategories under 'Pancaindera' and 'Politik' and 'Sukan' and 'Rencana'
subcategories = {
    'cover': 'https://www.utusan.com.my/category/pancaindera/cover/',
    'khabar-lagenda': 'https://www.utusan.com.my/category/pancaindera/khabar-lagenda/',
    'rondivu': 'https://www.utusan.com.my/category/pancaindera/rondivu/',
    'hangat-seminggu': 'https://www.utusan.com.my/category/pancaindera/hangat-seminggu/',
    'jawablah': 'https://www.utusan.com.my/category/pancaindera/jawablah/',
    'jejak-ulama': 'https://www.utusan.com.my/category/pancaindera/jejak-ulama/',
    'sastera': 'https://www.utusan.com.my/category/gaya/sastera/',
    'seram': 'https://www.utusan.com.my/category/pancaindera/seram/',
    'politik': 'https://www.utusan.com.my/category/nasional/politik/',
     'bola-sepak': 'https://www.utusan.com.my/category/sukan/bola-sepak/',
    'badminton': 'https://www.utusan.com.my/category/sukan/badminton/',
    'e-sukan': 'https://www.utusan.com.my/category/sukan/e-sukan/',
    'lumba-basikal': 'https://www.utusan.com.my/category/sukan/basikal/',
    'lumba-kereta': 'https://www.utusan.com.my/category/sukan/lumba-kereta/',
    'sepak-takraw': 'https://www.utusan.com.my/category/sukan/sepak-takraw/',
    'surat-pembaca': 'https://www.utusan.com.my/category/rencana/surat-pembaca/',
    'wawancara': 'https://www.utusan.com.my/category/rencana/wawancara/',
}

# Function to scrape a single page using selectolax
def scrape_page(url, category_name):
    response = requests.get(url)
    response.raise_for_status()
    tree = HTMLParser(response.text)

    articles = tree.css('article.jeg_post')
    page_data = []

    for article in articles:
        title_node = article.css_first('h3.jeg_post_title a')
        title = title_node.text(strip=True) if title_node else None
        link = title_node.attrs.get('href') if title_node else None

        img_node = article.css_first('img')
        img_url = img_node.attrs.get('src') if img_node else None
```

	`` category_node = article.css_first('div.jeg_post_category') category = category_node.text(strip=True) if category_node else None date_node = article.css_first('div.jeg_meta_date') date = date_node.text(strip=True) if date_node else None page_data.append({ 'Title': title, 'Link': link, 'Image_URL': img_url, 'Category': category, 'Subcategory': category_name, 'DateTime': date }) return page_data # Scraping logic all_data = [] for sub_name, sub_url in subcategories.items(): print(f"--- Scraping subcategory: {sub_name.upper()} ---") page = 1 while True: page_url = sub_url if page == 1 else f"{sub_url}page/{page}/" print(f"Scraping page {page}: {page_url}") try: page_data = scrape_page(page_url, sub_name) if not page_data: print(f"No more articles found on page {page}. Stopping.") break all_data.extend(page_data) page += 1 time.sleep(1) except Exception as e: print(f"Failed to scrape page {page} of {sub_name}: {e}") break # Save results to CSV df = pd.DataFrame(all_data) df.to_csv('utusan_pancaindera_all_pages.csv', index=False) print("✅ Scraping completed! Saved to 'utusan_pancaindera_all_pages.csv'.") ```
Gaya_Komuniti_Selenium.ipynb	``` from selenium import webdriver from selenium.webdriver.chrome.service import Service from selenium.webdriver.chrome.options import Options from selenium.webdriver.common.by import By import csv import time import random import os # Setup Selenium ChromeDriver (headless mode) def setup_driver(): options = Options() options.add_argument('--headless') # Comment this out to see browser window options.add_argument('--disable-gpu') options.add_argument('--no-sandbox') service = Service() driver = webdriver.Chrome(service=service, options=options) return driver # Scrape a single subcategory def scrape_subcategory(driver, category, subcategory, writer, max_pages=1400, max_records=50000): base_url = f'https://www.utusan.com.my/category/{category}/{subcategory}/page/' page_num = 1 total_scraped = 0 ```

```
while total_scraped < max_records and page_num <= max_pages:
    url = f"{base_url}{page_num}/"
    print(f"🔄 Loading {category}/{subcategory} - Page {page_num}: {url}")
    try:
        driver.get(url)
        time.sleep(random.uniform(1, 3))  # Allow page to load

        articles = driver.find_elements(By.CLASS_NAME, 'jeg_post')
        if not articles:
            print(f"⚠️ No articles found on page {page_num} for {subcategory}. Stopping.")
            break

        for article in articles:
            if total_scraped >= max_records:
                break

            try:
                title_elem = article.find_element(By.CLASS_NAME, 'jeg_post_title')
                title = title_elem.text.strip()
                link = title_elem.find_element(By.TAG_NAME, 'a').get_attribute('href')
            except:
                title, link = 'No Title', 'No URL'

            try:
                date = article.find_element(By.CLASS_NAME, 'jeg_meta_date').text.strip()
            except:
                date = 'No Date'
```

```
                writer.writerow([title, link, date, category, subcategory])
                total_scraped += 1

            print(f"✅ Done page {page_num} of {subcategory}. Total scraped: {total_scraped}")
            page_num += 1

        except Exception as e:
            print(f"❌ Error on {category}/{subcategory} page {page_num}: {e}")
            page_num += 1
            time.sleep(1)

def run_full_scraper():
    driver = setup_driver()

    filename = 'utusan_full_scrape.csv'
    csv_exists = os.path.exists(filename)

    with open(filename, 'a', newline='', encoding='utf-8') as csvfile:
        writer = csv.writer(csvfile)
        if not csv_exists:
            writer.writerow(['Title', 'URL', 'Date', 'Category', 'Subcategory'])

        # Gaya category and its subcategories
        gaya_subcategories = [
            'agama', 'agro', 'anakku', 'deko', 'fesyen',
            'gajet', 'hiburan', 'keluarga-wanita', 'kesihatan', 'pendidikan'
        ]
```

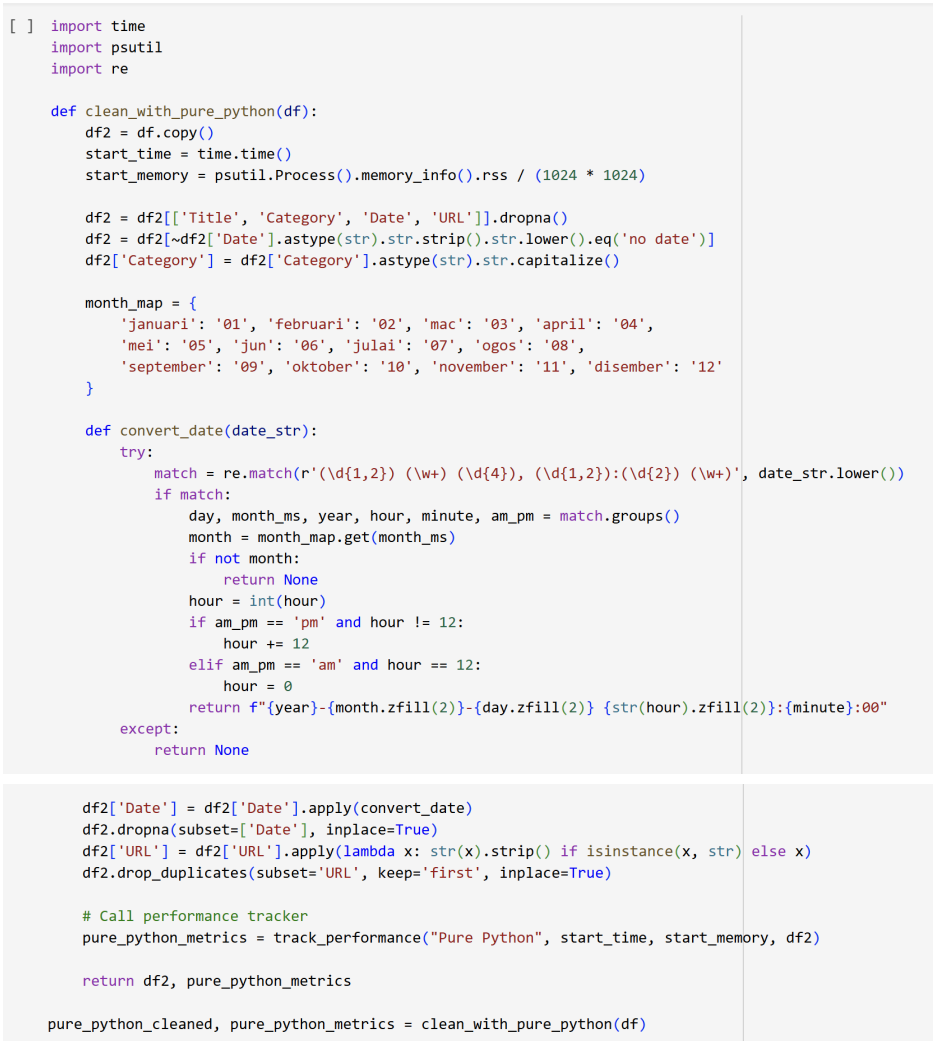
```
        for subcat in gaya_subcategories:
            scrape_subcategory(driver, 'gaya', subcat, writer)

        # Nasional > Komuniti
        scrape_subcategory(driver, 'nasional', 'komuniti', writer)

    driver.quit()
    print("🎉 Full scraping complete!")

# Run it
run_full_scraper()
```

Export to MongoDB.ipynb - combine all scraped data and load into MongoDB	``` utusan_articles = pd.concat([nasional, erl, srp, gk], ignore_index=True) utusan_articles ``` ``` from pymongo.mongo_client import MongoClient from pymongo.server_api import ServerApi uri = "mongodb+srv://nuraleyshaqurratuaini:Leysha18@cluster0.dsmzjrk.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0" # Create a new client and connect to the server client = MongoClient(uri, server_api=ServerApi('1')) # Send a ping to confirm a successful connection try: client.admin.command('ping') print("Pinged your deployment. You successfully connected to MongoDB!") except Exception as e: print(e) ``` ``` db = client["webcrawler_project"] collection = db["Raw_Data"] # Convert to list of dicts (MongoDB format) data_dict = utusan_articles.to_dict("records") # Insert into MongoDB collection.insert_many(data_dict) print("✅ Inserted", len(data_dict), "rows into MongoDB.") ```

Data Processing and Optimization

Table 10.1.2 presents code related to data cleaning, transformation, and performance optimization, including handling missing values, removing duplicates, and parallel processing.

Table 10.1.2 Data Processing and Optimization Code Snippets

Section	Screenshot
Cleaning Code in Pandas Library	

```
[ ]  import time
     import psutil
     import re

     def clean_with_pure_python(df):
         df2 = df.copy()
         start_time = time.time()
         start_memory = psutil.Process().memory_info().rss / (1024 * 1024)

         df2 = df2[['Title', 'Category', 'Date', 'URL']].dropna()
         df2 = df2[~df2['Date'].astype(str).str.strip().str.lower().eq('no date')]
         df2['Category'] = df2['Category'].astype(str).str.capitalize()

         month_map = {
             'januari': '01', 'februari': '02', 'mac': '03', 'april': '04',
             'mei': '05', 'jun': '06', 'julai': '07', 'ogos': '08',
             'september': '09', 'oktober': '10', 'november': '11', 'disember': '12'
         }

         def convert_date(date_str):
             try:
                 match = re.match(r'(\d{1,2}) (\w+) (\d{4}), (\d{1,2}):(\d{2}) (\w+)', date_str.lower())
                 if match:
                     day, month_ms, year, hour, minute, am_pm = match.groups()
                     month = month_map.get(month_ms)
                     if not month:
                         return None
                     hour = int(hour)
                     if am_pm == 'pm' and hour != 12:
                         hour += 12
                     elif am_pm == 'am' and hour == 12:
                         hour = 0
                     return f"{year}-{month.zfill(2)}-{day.zfill(2)} {str(hour).zfill(2)}:{minute}:00"
             except:
                 return None

         df2['Date'] = df2['Date'].apply(convert_date)
         df2.dropna(subset=['Date'], inplace=True)
         df2['URL'] = df2['URL'].apply(lambda x: str(x).strip() if isinstance(x, str) else x)
         df2.drop_duplicates(subset='URL', keep='first', inplace=True)

         # Call performance tracker
         pure_python_metrics = track_performance("Pure Python", start_time, start_memory, df2)

         return df2, pure_python_metrics

     pure_python_cleaned, pure_python_metrics = clean_with_pure_python(df)
```

Cleaning Code in Polars Library	

```
[ ]  import polars as pl
     import re
     import time
     import psutil

     def clean_polars_dataframe_with_malay(df_polars):
         # Step 1: Track performance start
         start_time = time.time()
         start_memory = psutil.Process().memory_info().rss / (1024 * 1024)

         # Malay month mapping
         month_map = {
             "januari": "01", "februari": "02", "mac": "03", "april": "04",
             "mei": "05", "jun": "06", "julai": "07", "ogos": "08",
             "september": "09", "oktober": "10", "november": "11", "disember": "12"
         }
         malay_months_pattern = re.compile(r'\b(' + '|'.join(month_map.keys()) + r')\b', re.IGNORECASE)

         def replace_malay_months(date_str):
             if isinstance(date_str, str):
                 date_str = date_str.lower()
                 date_str = malay_months_pattern.sub(lambda m: month_map[m.group()], date_str)
                 date_str = re.sub(r'\s*,', '', date_str)
                 date_str = date_str.replace('am', 'AM').replace('pm', 'PM')
                 return date_str.split("@")[0].strip()
             return date_str

         # Convert to LazyFrame for optimization
         lazy_df = df_polars.lazy()

         # Clean & prepare columns
         lazy_df = (
             lazy_df
             .drop_nulls()
             .filter(~pl.col("Date").str.to_lowercase().str.contains("no date"))
             .with_columns([
                 pl.col("Category").str.to_titlecase().alias("Category"),
                 pl.col("Date").map_elements(replace_malay_months, return_dtype=pl.Utf8)
             ])
             .with_columns([
                 pl.col("Date")
                 .str.replace_all(r"(am|pm)", "")
                 .str.replace_all(",", "")
                 .str.strip_chars()
                 .str.strptime(pl.Datetime, format="%d %m %Y %I:%M %p", strict=False)
                 .alias("Date")
             ])
             .drop_nulls(subset=["Date"])  # Remove failed datetime parsing rows
             .unique(subset=["URL"])       # Deduplicate by URL after date validation
         )

         # Execute optimized plan
         df_cleaned = lazy_df.collect()

         # Call performance tracker
         polars_metrics = track_performance("Polars (Optimized)", start_time, start_memory, df_cleaned)

         return df_cleaned, polars_metrics

     df_polars = pl.from_pandas(df)
     cleaned_polars_df, polars_metrics = clean_polars_dataframe_with_malay(df_polars)
```

Cleaning Code in Modin Library	

```
import modin.pandas as pd
import numpy as np
import re
import time
import psutil

import os
os.environ["MODIN_ENGINE"] = "ray"


def clean_with_modin(df):

    start_time = time.time()
    start_memory = psutil.Process().memory_info().rss / (1024 * 1024)

    # Initial cleaning
    modin = df[['Title', 'Category', 'Date', 'URL']].dropna().copy()
    modin = modin[~modin['Date'].astype(str).str.strip().str.lower().eq('no date')]
    modin['Category'] = modin['Category'].astype(str).str.capitalize()

    month_map = {
        'januari': '01', 'februari': '02', 'mac': '03', 'april': '04',
        'mei': '05', 'jun': '06', 'julai': '07', 'ogos': '08',
        'september': '09', 'oktober': '10', 'november': '11', 'disember': '12'
    }

    def convert_date(s):
        try:
            m = re.match(r'(\d{1,2}) (\w+) (\d{4}), (\d{1,2}):(\d{2}) (\w+)', str(s).lower())
            if m:
                day, month, year, hour, minute, ampm = m.groups()
                if ampm == 'pm' and int(hour) != 12:
                    hour = str(int(hour) + 12)
                elif ampm == 'am' and int(hour) == 12:
                    hour = '00'
                return f"{year}-{month_map.get(month)}-{day.zfill(2)} {hour.zfill(2)}:{minute}:00"
        except:
            return np.nan
        return np.nan
```

```
    modin['Date'] = modin['Date'].apply(convert_date)
    modin.dropna(subset=['Date'], inplace=True)

    modin.drop_duplicates(subset='URL', keep='first', inplace=True)

    # Track performance
    modin_metrics = track_performance("Modin", start_time, start_memory, modin)

    return modin, modin_metrics

modin_cleaned, modin_metrics = clean_with_modin(df)
```

Cleaning Code in Dask Library	

```
[ ] import pandas as pd
    import dask.dataframe as dd
    import multiprocessing
    import time
    import psutil
    import re

    def clean_with_dask(df):
        # Optimal partition count
        npartitions = multiprocessing.cpu_count()
        dask_df = dd.from_pandas(df, npartitions=npartitions)

        # Start tracking
        start_time = time.time()
        start_memory = psutil.Process().memory_info().rss / (1024 * 1024)

        # Drop nulls
        dask_df = dask_df.dropna()
        dask_df = dask_df[dask_df["Date"].str.lower() != "no date"]

        def clean_partition(df_partition):
            month_map = {
                "januari": "01", "februari": "02", "mac": "03", "april": "04",
                "mei": "05", "jun": "06", "julai": "07", "ogos": "08",
                "september": "09", "oktober": "10", "november": "11", "disember": "12"
            }

            def convert_date(s):
                try:
                    m = re.match(r'(\d{1,2}) (\w+) (\d{4}), (\d{1,2}):(\d{2}) (\w+)', str(s).lower())
                    if m:
                        day, month, year, hour, minute, ampm = m.groups()
                        if ampm == 'pm' and int(hour) != 12:
                            hour = str(int(hour) + 12)
                        elif ampm == 'am' and int(hour) == 12:
                            hour = '00'
                        return f"{year}-{month_map.get(month)}-{day.zfill(2)} {hour.zfill(2)}:{minute}:00"
                except:
                    return pd.NaT
                return pd.NaT

            df_partition = df_partition.copy()
            df_partition["Date"] = df_partition["Date"].apply(convert_date)
            df_partition["Date"] = pd.to_datetime(df_partition["Date"], errors="coerce")
            df_partition = df_partition.dropna(subset=["Date"])
            df_partition["Category"] = df_partition["Category"].str.title()
            return df_partition

        # Apply cleaning in partitions
        dask_df = dask_df.map_partitions(clean_partition)

        # Drop duplicates by URL
        dask_df = dask_df.drop_duplicates(subset='URL')

        # Final compute
        cleaned_dask_df = dask_df.compute()

        # Call performance tracker
        dask_metrics = track_performance("Dask", start_time, start_memory, cleaned_dask_df)

        return cleaned_dask_df, dask_metrics


    cleaned_dask_df, dask_metrics = clean_with_dask(df)
```

Cleaning Code in Swifter Library

```
import pandas as pd
import time
import psutil
import re
import swifter

def clean_with_swifter(df):
    start_time = time.time()
    start_memory = psutil.Process().memory_info().rss / (1024 * 1024)

    df = df.dropna(subset=['Title', 'Category', 'Date', 'URL'])
    df = df[~df['Date'].astype(str).str.strip().str.lower().eq('no date')]
    df['Category'] = df['Category'].astype(str).str.capitalize()

    month_map = {
        'januari': '01', 'februari': '02', 'mac': '03', 'april': '04',
        'mei': '05', 'jun': '06', 'julai': '07', 'ogos': '08',
        'september': '09', 'oktober': '10', 'november': '11', 'disember': '12'
    }

    def convert_date(date_str):
        try:
            match = re.match(r'(\d{1,2}) (\w+) (\d{4}), (\d{1,2}):(\d{2}) (\w+)', date_str.lower())
            if match:
                day, month_ms, year, hour, minute, am_pm = match.groups()
                month = month_map.get(month_ms)
                if not month:
                    return None
                hour = int(hour)
                if am_pm == 'pm' and hour != 12:
                    hour += 12
                elif am_pm == 'am' and hour == 12:
                    hour = 0
                return f"{year}-{month.zfill(2)}-{day.zfill(2)} {str(hour).zfill(2)}:{minute}:00"
        except:
            return None

    df['Date'] = df['Date'].swifter.apply(convert_date)
    df.dropna(subset=['Date'], inplace=True)
    df['URL'] = df['URL'].astype(str).str.strip()
    df.drop_duplicates(subset='URL', keep='first', inplace=True)

    # Call performance tracker
    swifter_metrics = track_performance("Swifter", start_time, start_memory, df)

    return df, swifter_metrics

cleaned_swifter_df, swifter_metrics = clean_with_swifter(df)
```

Cleaning_Optimization_Comparison.ipynb

```
from pymongo.mongo_client import MongoClient
from pymongo.server_api import ServerApi
import os

# Connect to MongoDB
client = MongoClient("mongodb+srv://nuraleyshaqurratuaini:Leysha18@cluster0.dsmzjrk.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0")
db = client["webcrawler_project"]
# collection = db["cleaned_data_db"]

# Output folder
output_dir = "cleaned_exports"
os.makedirs(output_dir, exist_ok=True)

# List of (DataFrame, CSV name, MongoDB collection)
exports = [
    (pure_python_cleaned, "cleaned_python.csv", "cleaned_python_data"),
    (cleaned_polars_df.to_pandas(), "cleaned_polars.csv", "cleaned_polars_data"),
    (modin_cleaned, "cleaned_modin.csv", "cleaned_modin_data"),
    (cleaned_dask_df, "cleaned_dask.csv", "cleaned_dask_data"),
    (cleaned_swifter_df, "cleaned_swifter.csv", "cleaned_swifter_data")
]

for df, filename, collection_name in exports:
    # Save to CSV
    csv_path = os.path.join(output_dir, filename)
    df.to_csv(csv_path, index=False)
    print(f"✅ Saved: {filename}")

    # Insert into MongoDB
    mongo_collection = db[collection_name]
    mongo_collection.delete_many({})  # Clear existing records
    mongo_collection.insert_many(df.to_dict(orient="records"))
    print(f"📤 Uploaded to MongoDB collection: {collection_name}")
```

	`` # Stats for each library (ensure these variables are already defined) stats = { "Pure Python": pure_python_metrics, "Polars": polars_metrics, "Modin": modin_metrics, "Dask": dask_metrics, "Swifter": swifter_metrics } # Convert to DataFrame for tabular display df = pd.DataFrame.from_dict(stats, orient='index') df.index.name = 'Library' df = df.reset_index() # Display the table print("\nComparison Table (Performance Metrics by Library):\n") print(df.to_string(index=False)) ```
Comparison_PatternedGraph.ipynb	``` # Performance metrics data for different libraries/methods stats = { "Pure Python": { "Execution Time (s)": 2.328932, "CPU Usage (%)": 98.5, "Memory Usage (MB)": 1524.066406, "Throughput (rows/sec)": 54228.288957 }, "Polars": { "Execution Time (s)": 2.317019, "CPU Usage (%)": 86.4, "Memory Usage (MB)": 1537.031250, "Throughput (rows/sec)": 54507.106233 }, "Modin": { "Execution Time (s)": 1.083179, "CPU Usage (%)": 16.8, "Memory Usage (MB)": 1531.855469, "Throughput (rows/sec)": 116595.687643 }, "Dask": { "Execution Time (s)": 1.219067, "CPU Usage (%)": 75.4, "Memory Usage (MB)": 1533.863281, "Throughput (rows/sec)": 103598.932118 }, "Swifter": { "Execution Time (s)": 1.483202, "CPU Usage (%)": 36.9, "Memory Usage (MB)": 1532.441406, "Throughput (rows/sec)": 85149.548984 } } # You might also want to define labels and x-axis positions here as they are used in all plots labels = list(stats.keys()) x = np.arange(len(labels)) # Positions for the bars on the x-axis width = 0.6 # Width of each bar hatch_patterns = ['/', '\', '|', '-', '+'] # Define hatch patterns here too ```

Import Libraries

```
import matplotlib.pyplot as plt
import numpy as np
from google.colab import drive
```

Performance Evaluation

```
# Extract performance metrics for plotting
execution_times = [stats[lib]["Execution Time (s)"] for lib in labels]
cpu_usages = [stats[lib]["CPU Usage (%)"] for lib in labels]
memory_usages = [stats[lib]["Memory Usage (MB)"] for lib in labels]
throughputs = [stats[lib]["Throughput (rows/sec)"] for lib in labels]

# Define unique hatch patterns for each library to differentiate them within a subplot
hatch_patterns = ['/', '\', '|', '-', '+']

# Define distinct colors for each of the four subplots (metrics)
subplot_colors = [
    '#E6E6FA',  # Light Lavender for Execution Time
    '#90EE90',  # Light Green for Memory Usage
    '#ADD8E6',  # Light Blue for CPU Usage
    '#F08080'   # Light Coral for Throughput
]

# Create a figure and a 2x2 grid of subplots
fig, axs = plt.subplots(2, 2, figsize=(14, 10))

# Function to plot bars with a specified color for the subplot and unique hatch patterns
def plot_bars(ax, data, title, ylabel, bar_color):
    """
    Plots a bar chart on the given axes.

    Args:
        ax (matplotlib.axes.Axes): The axes object to draw the plot on.
        data (list): The list of data values for the bars.
        title (str): The title of the subplot.
        vlabel (str): The label for the v-axis.
```

```
    Args:
        ax (matplotlib.axes.Axes): The axes object to draw the plot on.
        data (list): The list of data values for the bars.
        title (str): The title of the subplot.
        ylabel (str): The label for the y-axis.
        bar_color (str): The color to apply to all bars in this subplot.
    """
    # Create the bars, all with the same 'bar_color' for this subplot
    bars = ax.bar(x, data, width, color=bar_color)
    # Apply unique hatch patterns to each individual bar to differentiate libraries
    for bar, hatch in zip(bars, hatch_patterns):
        bar.set_hatch(hatch) # Set the hatch pattern for the current bar
    ax.set_title(title) # Set the title of the subplot
    ax.set_ylabel(ylabel) # Set the y-axis label
    ax.set_xticks(x) # Set x-axis ticks at the positions of the bars
    ax.set_xticklabels(labels, rotation=15) # Set x-axis tick labels (library names) with rotation
    ax.bar_label(bars, fmt='%.2f') # Add data labels on top of the bars, formatted to two decimal places

# Call the plot_bars function for each subplot, assigning a unique color
plot_bars(axs[0, 0], execution_times, 'Execution Time Comparison', 'Execution Time (s)', subplot_colors[0])
plot_bars(axs[0, 1], memory_usages, 'Memory Usage Comparison', 'Memory Usage (MB)', subplot_colors[1])
plot_bars(axs[1, 0], cpu_usages, 'CPU Usage Comparison', 'CPU Usage (%)', subplot_colors[2])
plot_bars(axs[1, 1], throughputs, 'Throughput Comparison', 'Throughput (rows/sec)', subplot_colors[3])

# Create a custom legend to explain the hatch patterns for each library
# The facecolor for the legend elements is set to 'lightgray' to be neutral,
# as the main bar color is now determined by the subplot.
legend_elements = [
    plt.Rectangle(
        (0, 0), 1, 1, # Rectangle dimensions for the legend swatch
        facecolor='lightgray', # Neutral background color for the legend swatch
        hatch=hatch_patterns[i], # Apply the specific hatch pattern
        label=labels[i] # Label for the legend entry (library name)
    ) for i in range(len(labels)) # Iterate through all libraries
]
# Add the legend to the figure
fig.legend(
    handles=legend_elements, # List of legend handles created above
    loc='upper center', # Position the legend at the upper center of the figure
    ncol=len(labels), # Arrange legend entries in a single row
    bbox_to_anchor=(0.5, 1.05) # Adjust legend position to be above the subplots
)
```

```
# Adjust subplot parameters for a tight layout and make space for the legend
plt.tight_layout()
plt.subplots_adjust(top=0.9) # Ensure there's enough space at the top for the legend

# --- Google Colab Specific Operations ---
# This section handles mounting Google Drive and saving the plot.
# It will prompt for authentication if not already mounted.
try:
    drive.mount('/content/drive')
    # Save the generated figure as a PNG image to Google Drive
    plt.savefig('/content/drive/MyDrive/ColabNotebooks/HPDP_Project/Patterned_Graph.png', bbox_inches='tight')
except Exception as e:
    print(f"Could not mount Google Drive or save file: {e}")
    print("The plot will still be displayed in the output.")

# Display the plot
plt.show()
```

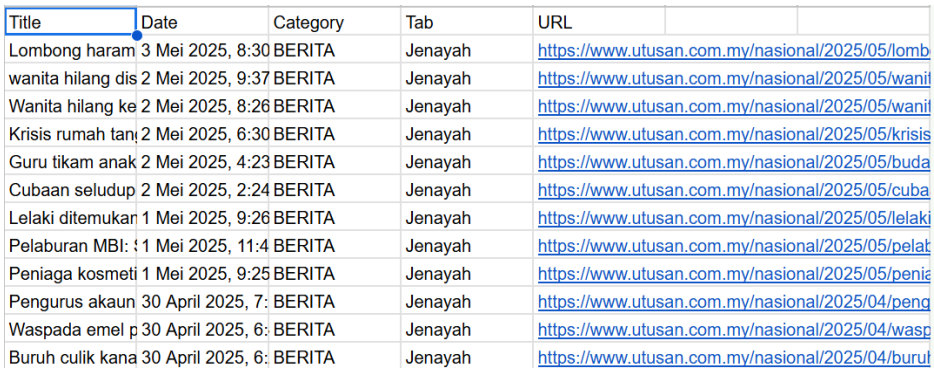
10.2 Screenshots of Output

This section provides visual screenshots of program execution results to validate the successful completion of tasks.

Web Scraping Output

Table 10.2.1 includes execution results in Google Collab and samples of the raw data collected through web scraping process, demonstrating the format and structure of the initial dataset.

Table 10.2.1 Web Scraping Output and Sample of Raw Data

Section	Screenshot
Nasional.ipynb	
Nasional.csv	

Ekonomi_Ringgit_LuarNegara.ipynb	
Ekonomi_Ringgit_LuarNegara.csv	
Gaya_Komuniti_Selenium.ipynb	

Gaya_Komuniti_Selenium.csv

	Title	URL	Date	Category	Subcategory
2	Kerja! Jangan mencuri!	https://www.utusan.com.my/gaya/2025/05/kerja-jangan-mencuri/	2 Mei 2025, 8:00 am	gaya	agama
3	Putus tangan pertahan bendera	https://www.utusan.com.my/gaya/2025/04/putus-tangan-pertahan-bende	18 April 2025, 08:00	gaya	agama
4	Penduduk Melayu Katanning, Australia Barat perlu guru agama	https://www.utusan.com.my/nasional/2025/04/penduduk-melayu-katanni	15 April 2025, 15:02	gaya	agama
5	Gabung dua program dekatkan masyarakat dengan masjid	https://www.utusan.com.my/nasional/2025/04/gabung-dua-program-deka	12 April 2025, 08:45	gaya	agama
6	Seputar mengenai al-Quran	https://www.utusan.com.my/gaya/2025/04/seputar-mengenai-al-quran/	11 April 2025, 11:00	gaya	agama
7	Musim sakura dengan al-Quran	https://www.utusan.com.my/gaya/2025/04/musim-sakura-dengan-al-qura	4 April 2025, 08:00	gaya	agama
8	Contohi diplomasi, kebijaksanaan kepemimpinan Nabi Sulaiman	https://www.utusan.com.my/gaya/2025/03/contohi-diplomasi-kebijaksana	28 Mac 2025, 10:30 am	gaya	agama
9	Dari langit turun ke jantung	https://www.utusan.com.my/gaya/2025/03/dari-langit-turun-ke-jantung/	21 Mac 2025, 9:00 am	gaya	agama
10	Al-Quran dan hadis qudsi	https://www.utusan.com.my/gaya/2025/03/al-quran-dan-hadis-qudsi/	14 Mac 2025, 10:00 am	gaya	agama
11	Usah jadikan Ramadan 'bulan pesta makan' – Mufti	https://www.utusan.com.my/nasional/2025/03/usah-jadikan-ramadan-bul	13 Mac 2025, 7:30 am	gaya	agama
12	JAIS serah surat kebenaran solat kepada Masjid Al-Falah	https://www.utusan.com.my/nasional/2025/03/jais-serah-surat-kebenara	7 Mac 2025, 11:13 am	gaya	agama
13	Usah perlekeh kedatangan ramai jemaah ke masjid	https://www.utusan.com.my/gaya/agama/2025/03/usah-perlekeh-kedata	3 Mac 2025, 7:30 am	gaya	agama
14	Mengapa logo halal Jakim penting dalam industri makanan?	https://www.utusan.com.my/gaya/2025/02/mengapa-logo-halal-jakim-pe	28 Februari 2025, 10:00 am	gaya	agama
15	Fahaman bertentangan ASWJ di Perak terkawal	https://www.utusan.com.my/berita/2025/02/fahaman-bertentangan-aswj-	21 Februari 2025, 6:28 am	gaya	agama
16	Solat duha tanda syukur pada Allah	https://www.utusan.com.my/gaya/2025/02/solat-duha-tanda-syukur-pada	14 Februari 2025, 8:00 am	gaya	agama
17	Masjid Al-Ikhlas tamatkan penantian lebih 30 tahun penduduk	https://www.utusan.com.my/nasional/2025/02/masjid-al-ikhlas-tamatkan-	13 Februari 2025, 8:15 am	gaya	agama
18	16,000 Orang Asli di Pahang peluk Islam	https://www.utusan.com.my/berita/2025/02/16000-orang-asli-di-pahang-j	13 Februari 2025, 7:30 am	gaya	agama
19	Bukan halangan hadiri majlis bukan Islam	https://www.utusan.com.my/nasional/2025/02/bukan-halangan-hadiri-ma	10 Februari 2025, 7:30 am	gaya	agama
20	Berlian suci berbau harum di celahan lumpur	https://www.utusan.com.my/gaya/2025/02/berlian-suci-berbau-harum-di-	7 Februari 2025, 9:00 am	gaya	agama
21	Bayar zakat 33 tahun 'sucikan harta'	https://www.utusan.com.my/berita/2025/02/bayar-zakat-33-tahun-sucika	6 Februari 2025, 8:45 am	gaya	agama

Politik_Sukan_Rencana_Pancaindera_Selectolax.ipynb

```
--- Scraping subcategory: SASTERA ---
Scraping page 1: https://www.utusan.com.my/category/gaya/sastera/
Scraping page 2: https://www.utusan.com.my/category/gaya/sastera/page/2/
Scraping page 3: https://www.utusan.com.my/category/gaya/sastera/page/3/
Scraping page 4: https://www.utusan.com.my/category/gaya/sastera/page/4/
Scraping page 5: https://www.utusan.com.my/category/gaya/sastera/page/5/
Scraping page 6: https://www.utusan.com.my/category/gaya/sastera/page/6/
Scraping page 7: https://www.utusan.com.my/category/gaya/sastera/page/7/
Scraping page 8: https://www.utusan.com.my/category/gaya/sastera/page/8/
Scraping page 9: https://www.utusan.com.my/category/gaya/sastera/page/9/
Scraping page 10: https://www.utusan.com.my/category/gaya/sastera/page/10/
Scraping page 11: https://www.utusan.com.my/category/gaya/sastera/page/11/
Scraping page 12: https://www.utusan.com.my/category/gaya/sastera/page/12/
Scraping page 13: https://www.utusan.com.my/category/gaya/sastera/page/13/
Scraping page 14: https://www.utusan.com.my/category/gaya/sastera/page/14/
Scraping page 15: https://www.utusan.com.my/category/gaya/sastera/page/15/
Scraping page 16: https://www.utusan.com.my/category/gaya/sastera/page/16/
Scraping page 17: https://www.utusan.com.my/category/gaya/sastera/page/17/
Scraping page 18: https://www.utusan.com.my/category/gaya/sastera/page/18/
Scraping page 19: https://www.utusan.com.my/category/gaya/sastera/page/19/
Scraping page 20: https://www.utusan.com.my/category/gaya/sastera/page/20/
Failed to scrape page 20 of sastera: 404 Client Error: Not Found for url: https://www.utusan.com.my/category/gaya/sastera/page/20/
--- Scraping subcategory: SERAM ---
Scraping page 1: https://www.utusan.com.my/category/pancaindera/seram/
Scraping page 2: https://www.utusan.com.my/category/pancaindera/seram/page/2/
Scraping page 3: https://www.utusan.com.my/category/pancaindera/seram/page/3/
Scraping page 4: https://www.utusan.com.my/category/pancaindera/seram/page/4/
Scraping page 5: https://www.utusan.com.my/category/pancaindera/seram/page/5/
Failed to scrape page 5 of seram: 404 Client Error: Not Found for url: https://www.utusan.com.my/category/pancaindera/seram/page/5/
--- Scraping subcategory: POLITIK ---
Scraping page 1: https://www.utusan.com.my/category/nasional/politik/
Scraping page 2: https://www.utusan.com.my/category/nasional/politik/page/2/
Scraping page 3: https://www.utusan.com.my/category/nasional/politik/page/3/
Scraping page 4: https://www.utusan.com.my/category/nasional/politik/page/4/
Scraping page 5: https://www.utusan.com.my/category/nasional/politik/page/5/
Scraping page 6: https://www.utusan.com.my/category/nasional/politik/page/6/
Scraping page 7: https://www.utusan.com.my/category/nasional/politik/page/7/
```

UtusanMalaysia_selectolax.csv

Title	URL	Category	Date
Aura Walid â€¯one takeâ€™	https://www.utusan.com.m	cover	27 April 2025, 9:00 am
Terumbang ambing mencipta nama -Sheila Abdull	https://www.utusan.com.m	cover	20 April 2025, 6:50 am
Terima kasih Maya pinjamkan â€¯Babahâ€™ untuk saya	https://www.utusan.com.m	cover	13 April 2025, 9:00 am
â€¯Daripada jawab soal jodoh, saya rela duduk dapurâ€™ â€“	https://www.utusan.com.m	cover	6 April 2025, 6:50 am
Memori raya dengan Dee? Hanya Allah yang tahu- Farid Kamil	https://www.utusan.com.m	cover	30 Mac 2025, 8:00 am
Dulu ada yang kata muzik Mohram tak boleh jual	https://www.utusan.com.m	cover	23 Mac 2025, 9:10 am
â€¯Kami berbeza agama, bukan semua orang sukaâ€™	https://www.utusan.com.m	cover	23 Mac 2025, 6:50 am
Kalau semua betul, kita tidak belajar apa-apa â€“ Mira Filzah	https://www.utusan.com.m	cover	16 Mac 2025, 8:00 am
Datanglah berjumpa orang yang dia zalimi ini	https://www.utusan.com.m	cover	9 Mac 2025, 7:50 am
Tiada lagi gulai ikan talang ibu	https://www.utusan.com.m	cover	2 Mac 2025, 9:00 am
Saya menangis â€¯mencariâ€™ Tracie Sinidol	https://www.utusan.com.m	cover	23 Februari 2025, 8:00 am
Bagai ada beban yang harus dipikul	https://www.utusan.com.m	cover	16 Februari 2025, 9:00 am
Liza Aziz lunas impian setelah lebih empat dekad	https://www.utusan.com.m	cover	9 Februari 2025, 8:00 am
Umai mahu bersinar dengan identiti sendiri	https://www.utusan.com.m	cover	2 Februari 2025, 6:17 am
Penat lelah empat dekad terbayar	https://www.utusan.com.m	cover	26 Januari 2025, 6:39 am
Hun Haqeem pakai mask naik LRT lihat manusia	https://www.utusan.com.m	cover	19 Januari 2025, 6:21 am
Populariti tidak pernah jatuh menjunam	https://www.utusan.com.m	cover	12 Januari 2025, 6:33 am
Puaskan hati semua orang? Saya tidak akan mampu â€“ Noki	https://www.utusan.com.m	cover	5 Januari 2025, 6:40 am
â€¯Terima kasih Siti, si gadis bergaun merahâ€™	https://www.utusan.com.m	cover	29 Disember 2024, 6:29 am
Dulu saya hanya mampu makan telur sahaja â€“ Nadeera Zair	https://www.utusan.com.m	cover	22 Disember 2024, 6:09 am
â€¯Saya korbankan usia remaja, demi muzikâ€™	https://www.utusan.com.m	cover	15 Disember 2024, 6:11 am
Sebar manfaat sebelum menutup mata	https://www.utusan.com.m	cover	8 Disember 2024, 6:23 am
Yang muda kena perbaiki salah faham ini â€“ Harissa Adlynn	https://www.utusan.com.m	cover	1 Disember 2024, 6:30 am
Pernah bernyanyi di kafe sebelum popular	https://www.utusan.com.m	cover	24 November 2024, 6:50 am
â€¯Aktor paling mahal? Bukan sayaâ€¦â€™	https://www.utusan.com.m	cover	10 November 2024, 7:45 am

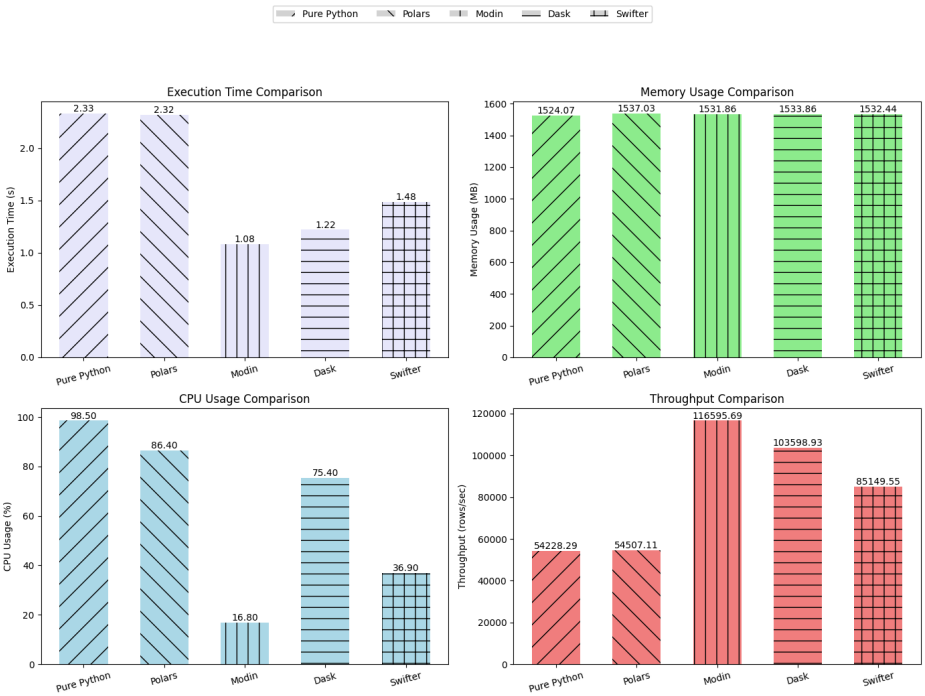
Data Processing and Optimization Output

Table 10.2.2 shows the output of performance metrics for Pandas, Polars, Modin, Dask, and Swifter, as well as comparison of performance between libraries in the bar chart and table run in Google Collab.

Table 10.2.2 Data Processing and Optimization Performance Metrics

Section	Screenshot
Pure Python (Pandas)	Performance Report [Pure Python] Final Row Count: 126,294 Time Taken: 2.3289 seconds Throughput: 54228.29 rows/sec CPU Usage: 98.50% Memory Used: 1524.07 MB
Polars	Performance Report [Polars (Optimized)] Final Row Count: 126,294 Time Taken: 2.3170 seconds Throughput: 54507.11 rows/sec CPU Usage: 86.40% Memory Used: 1537.03 MB
Modin	Performance Report [Modin] Final Row Count: 126,294 Time Taken: 1.0832 seconds Throughput: 116595.69 rows/sec CPU Usage: 16.80% Memory Used: 1531.86 MB

Dask	Performance Report [Dask] Final Row Count: 126,294 Time Taken: 1.2191 seconds Throughput: 103598.93 rows/sec CPU Usage: 75.40% Memory Used: 1533.86 MB
Swifter	Performance Report [Swifter] Final Row Count: 126,294 Time Taken: 1.4832 seconds Throughput: 85149.55 rows/sec CPU Usage: 36.90% Memory Used: 1532.44 MB
Save Cleaned Data and Load to MongoDB	Saved: cleaned_python.csv Uploaded to MongoDB collection: cleaned_python_data Saved: cleaned_polars.csv Uploaded to MongoDB collection: cleaned_polars_data Saved: cleaned_modin.csv Uploaded to MongoDB collection: cleaned_modin_data Saved: cleaned_dask.csv Uploaded to MongoDB collection: cleaned_dask_data Saved: cleaned_swifter.csv Uploaded to MongoDB collection: cleaned_swifter_data

Comparison in Bar Charts	
Comparison Table	Comparison Table (Performance Metrics by Library): Library Final Row Count Execution Time (s) CPU Usage (%) Memory Usage (MB) Throughput (rows/sec) Pure Python 126294 2.328932 98.5 1524.066406 54228.288957 Polars 126294 2.317019 86.4 1537.031250 54507.106233 Modin 126294 1.083179 16.8 1531.855469 116595.687643 Dask 126294 1.219067 75.4 1533.863281 103598.932118 Swifter 126294 1.483202 36.9 1532.441406 85149.548984

10.3 Tables

10.3.1 Description of Optimization Methods Used

Table 5.1.1 summarizes the key optimization libraries used to enhance data processing performance. It highlights Swifter's dynamic strategy for accelerating pandas .apply() operations, Modin's parallelization of pandas workflows across multiple CPUs, Dask's ability to scale computations for large or distributed datasets, and Polars' fast, multithreaded data manipulation using lazy evaluation. Each method offers distinct advantages for handling large-scale data efficiently.

Table 5.1.1 Description of Optimization Methods Used

Library/Methods Used	Description
	• Swifter is a Python library that accelerates the performance of pandas.apply() operations by dynamically selecting the most efficient execution strategy. • It automatically decides whether to use pandas, Dask, or Numba, depending on the size of the dataset and the complexity of the function. • Swifter is especially useful for large-scale data transformations, enabling parallel execution with minimal code changes and maximizing CPU utilization efficiently.

	• Modin is a library that accelerates Pandas by automatically distributing the computation across all of the system's CPUs. • It provides an effortless way to speed up Pandas workflows without requiring significant code changes. • Particularly effective for large datasets, as it parallelizes Pandas operations, allowing for faster data manipulation and analysis.
	• Dask is a flexible parallel computing library for Python that scales Pandas and NumPy workflows. It enables parallel computing on single machines or distributed clusters. • Dask excels at handling larger-than-memory datasets by breaking them into smaller chunks and processing them in parallel. • It provides high-level abstractions for parallel arrays, dataframes, and machine learning, making it easier to parallelize complex computations.

	• Polars is an open-source library for data manipulation, known for being one of the fastest data processing solutions on a single machine. • Polars uses lazy evaluation, which allows it to build a query plan first then optimize the entire pipeline before execution. • Its execution engine uses multithreading under the hood to parallelize operations.

10.4 Links to Full Code Repository

Below is the Github link which includes all scripts and files related to the project. Each task is organized in dedicated folders for ease of reference.

GitHub Repository link:

https://github.com/Jingyong14/HPDP02/tree/a6d028845e5040da68b99a245802950ec122314c/2425/project/p1/Group%203