SECP3133-02

HIGH-PERFORMANCE DATA PROCESSING

# PROJECT 1: OPTIMIZING HIGH-PERFORMANCE DATA PROCESSING FOR LARGE-SCALE WEB CRAWLERS

*Prepared By:*

NURUL ERINA BINTI ZAINUDDIN  A22EC0254

ONG YI YAN A22EC0101

TANG YAN QING A22EC0109

WONG QIAO YING A22EC0118

*Lecturer Name:*

DR. ARYATI BINTI BAKRI

*Submission Date:*

*17 May 2025*

## Table of Contents

## 1.0 Introduction

In today's data-driven world, the ability to collect and process large volumes of web data is crucial across industries like business analytics, research, and data science. Web crawling, a technique used to automatically collect data from websites, plays a key role in this process. However, extracting large datasets efficiently often presents technical challenges such as slow performance, data inconsistency, and scalability issues.

This project aims to overcome these challenges by developing a high-performance web crawler designed to extract property-related data from eBay. The project will utilize High-Performance Computing (HPC) techniques, such as threading, asyncio, and multiprocessing, to optimize the speed and efficiency of data extraction. Additionally, the project will evaluate the performance of four different data processing libraries including Polars, Pandas, PySpark, and Modin by focusing on key factors like processing speed, memory usage, CPU efficiency, and throughput. This project will provide valuable insights into effective data extraction, cleansing, and performance optimization, which are critical skills in data science and analytics.

## 1.1 Background of The Project

The ability to rapidly gather and process vast amounts of web data has become crucial in the big data era, particularly in domains like analytics and data science.  Web crawling is a popular method for collecting data from websites, although it frequently has technical drawbacks like poor performance and problems with data quality.

The main goal  is to develop a web crawler capable of extracting a sizable dataset specifically from eBay, leveraging High-Performance Computing (HPC) techniques such as threading, asyncio, and multiprocessing. The project also aims to compare the performance of these techniques using four different libraries: Polars, Pandas, PySpark, and Modin. Through this process, practical experience in web data extraction, data cleansing, and performance optimization will be gained.

## 1.2 Objective

1. To develop and implement a web crawler using different libraries including Playwright, Requests, BeautifulSoup, Scrapy, and Selenium to extract at least 100,000 structured property-related records from eBay.

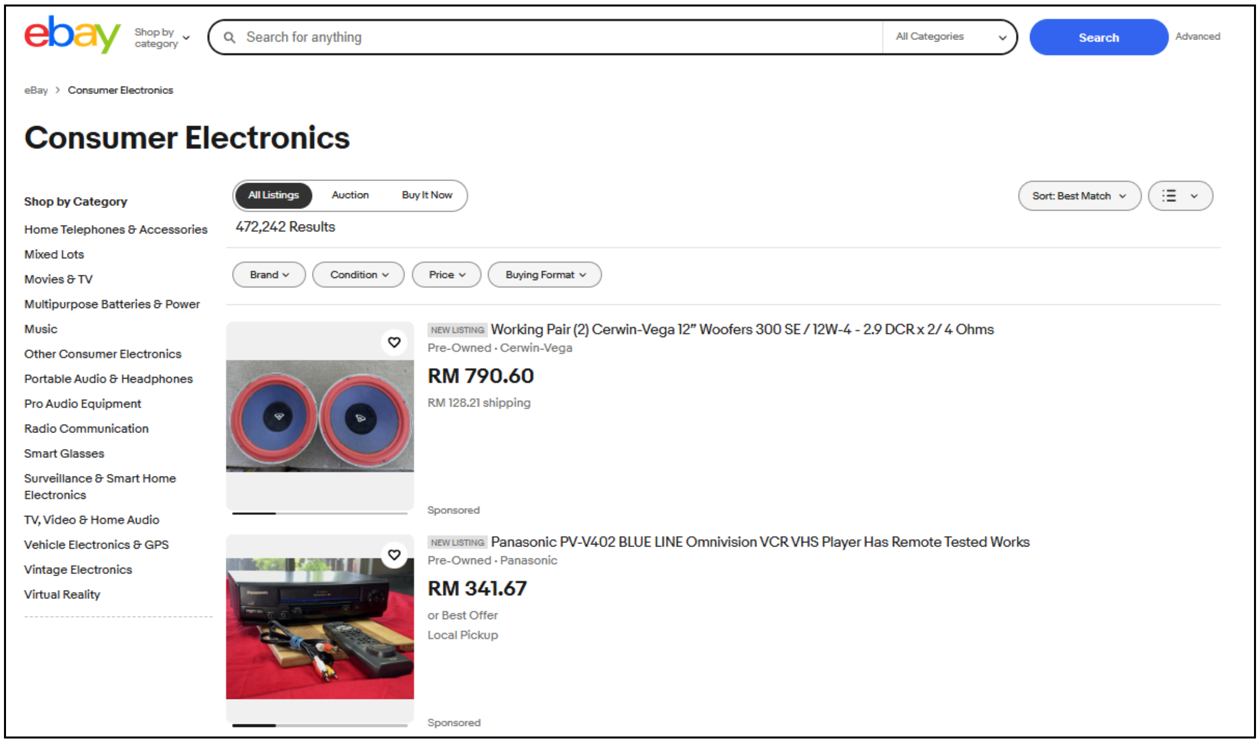
2. To process and clean the data using high-performance computing techniques using different data processing libraries including Polars, Pandas, PySpark, and Modin.

3. To perform a performance comparison between different libraries in terms of total time, memory usage, CPU efficiency, and throughput.

## 1.3 Target Website and Data to be Extracted

### 1.3.1 Target Website

Figure 1.3.1 displays the listings of eBay in Consumer Electronics category, eBay.com.my is the Malaysian portal of the global e-commerce platform eBay, where users can buy and sell a wide range of items, from electronics to fashion and collectibles. We specifically extract product listings from eBay.



472,242 Results

NEW LISTING Working Pair (2) Cerwin-Vega 12" Woofers 300 SE / 12W-4 - 2.9 DCR x 2/ 4 Ohms
Pre-Owned · Cerwin-Vega
RM 790.60
RM 128.21 shipping
Sponsored

NEW LISTING Panasonic PV-V402 BLUE LINE Omnivision VCR VHS Player Has Remote Tested Works
Pre-Owned · Panasonic
RM 341.67
or Best Offer
Local Pickup
Sponsored

**Figure 1.3.1.1 eBay website listings**

## 1.3.2 Data to be Extracted

From the eBay listings, the following key attributes are extracted for each product:

- Title: Serves as the primary identifier of the product.
- Price: Represents the listed selling price of the product.
- Shipping Fee: Indicates the delivery cost or specifies if the item is offered with free shipping.
- Link: Provides the direct URL to the product's webpage for easy access.
- Category: Identifies the product's type or classification, offering insight into its market segment.
- Brand: Specifies the manufacturer or brand of the product.

- Condition: Describes whether the item is new or used, providing information on its status.

## 2.0 System Design and Architecture

The system design and architecture section outlines the overall structure and flow of the web crawling process, detailing how the various components work together to collect, process, and store the data. This section covers the key architectural decisions, including the choice of technologies, tools, and frameworks used to build the crawler, as well as the design considerations for ensuring efficiency, scalability, and maintainability. The architecture is designed to handle large-scale data extraction while ensuring the system operates efficiently, remains resilient to errors, and adheres to ethical guidelines. The design also ensures seamless integration with the MongoDB database for data storage and facilitates the use of high-performance computing (HPC) techniques to optimize the crawling and processing tasks.

## 2.1 Description of Architecture Diagram

Figure 2.1.1 displays the architecture diagram of this project. This architecture diagram illustrates a system designed to collect, process, analyze, and visualize data from eBay. The workflow begins by pulling raw data from eBay, which could include product listings, sales figures, or other relevant transactional data available through eBay's API or data exports. This data is then ingested into MongoDB, a NoSQL database. MongoDB is chosen for its ability to handle large volumes of unstructured data and to scale efficiently, allowing for quick retrieval and storage of eBay data.

Once the data is stored, the next step is data cleaning. Data cleaning is a critical process that prepares the raw data for analysis. It involves removing duplicates, handling missing or erroneous values, and ensuring the data is consistent and structured in a way that supports accurate analysis. This ensures that the subsequent analysis and performance metrics comparison are based on clean and reliable data.

Following the cleaning process, the system uses Python with key libraries like Pandas, NumPy, and Matplotlib for further analysis. Pandas is used to manipulate and process the data efficiently, while NumPy can be employed for numerical operations, especially when calculating performance metrics. Once the analysis is complete, the next step is to compare performance metrics which involves evaluating different approaches or algorithms applied to the data. This comparison helps to identify the most effective methods for handling and interpreting the eBay data.

Finally, the results of the performance comparison and analysis are visualized using Matplotlib This step produces visual representations such as graphs, charts, or heatmaps, which make it easier to interpret the data and share insights. These visualizations help users better understand the relationships within the data, identify trends, and draw actionable conclusions from the analysis. Thus, the system not only processes and cleans the eBay data but also provides valuable insights through performance metrics and visual data representation.

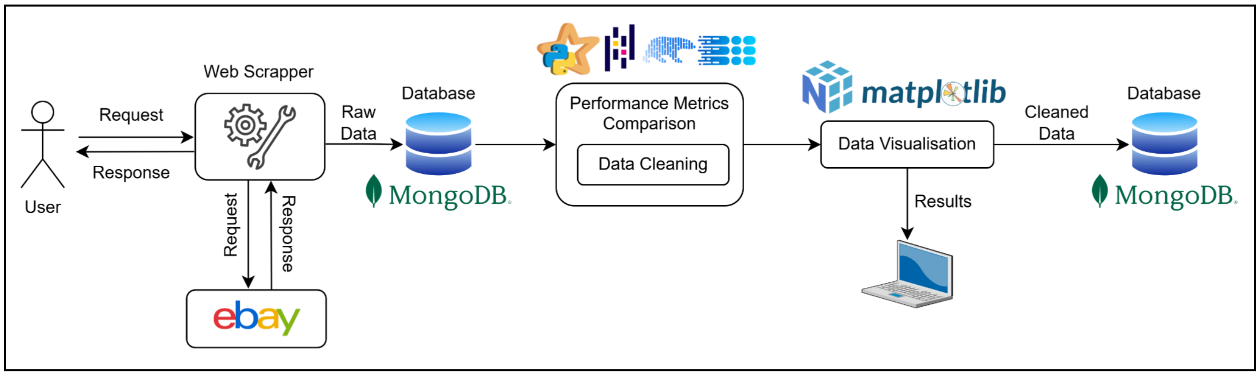


**Figure 2.1.1 Architecture Diagram**

## 2.2 Tools and Frameworks Used

This project utilizes a range of tools and frameworks for code development, web scraping, data processing, database management, architecture design, and documentation. Below is a breakdown of the key tools and their roles in the project:

- Integrated Development Environment (IDE):
    - Google Colab: Google Colab is a cloud-based Integrated Development Environment (IDE) that allows for Python development and testing in a collaborative environment. It provides an easy way to write and execute code, share projects with

collaborators, and access powerful cloud resources for running computationally expensive tasks, all without the need for local setup.

- Core Programming Language:
  - Python: Python is the core language used for this project. Known for its simplicity and extensive ecosystem, Python is highly suited for data extraction, web scraping, and analysis. Its versatility and support for multiple libraries make it the ideal language for implementing the web crawler and data processing tasks in this project.
- Python Libraries for Web Scraping:
  - Requests: Requests is a simple and intuitive library used for making HTTP requests. It is essential for fetching web pages and retrieving the raw HTML content of eBay listings. This library provides a convenient interface for interacting with web servers and handling responses.

  - BeautifulSoup: BeautifulSoup is a Python library used for parsing HTML and XML documents. It enables easy extraction of relevant data from the structured web pages. Using BeautifulSoup, we can navigate the document tree, search for specific elements, and extract information like product title, price, shipping fee, and other attributes.

  - Selenium: Selenium is a web automation tool that allows us to interact with dynamic web pages that require JavaScript rendering. It can simulate user actions like clicking, typing, or scrolling, making it valuable for scraping complex e-commerce websites such as eBay where content is dynamically loaded.

  - Playwright: Playwright is a modern web automation framework for scraping dynamic websites. It allows for more advanced browser interactions and is often faster and more reliable than Selenium for handling complex, JavaScript-heavy websites. It also supports multiple browsers, including Chromium, Firefox, and

WebKit, offering flexibility in web scraping.

  - Scrapy: Scrapy is an open-source, powerful, and efficient web crawling framework. It provides a comprehensive set of tools to manage web crawling tasks, including built-in support for handling requests, parsing HTML, following links, and exporting data. Scrapy is highly suitable for large-scale web scraping and can be used to manage multiple crawling tasks concurrently.

- Python Libraries for Data Cleaning and Processing:
  - Pandas: Pandas is one of the most widely used libraries for data manipulation and analysis in Python. It provides powerful tools for working with structured data, including support for DataFrames that allow for efficient data cleaning, transformation, and analysis. Pandas will be used to handle and clean the large dataset extracted by the crawler.

  - Polars: Polars is a high-performance DataFrame library designed for processing large datasets. Unlike Pandas, Polars leverages multithreading for faster data processing, making it suitable for situations where large volumes of data need to be handled quickly. It will be used to perform high-speed data transformations and aggregations on the scraped data.

  - PySpark: PySpark is the Python API for Apache Spark, a distributed computing framework. PySpark enables the processing of large datasets across multiple machines, which is useful when dealing with vast amounts of data. It is particularly helpful for distributed data processing tasks and will be used for handling and processing very large datasets from the web crawler in a scalable way.

  - Modin: Modin is a parallel DataFrame library that speeds up Pandas operations by utilizing all available CPU cores. It provides a seamless way to scale Pandas workflows for large datasets. Modin will be used for data processing to improve performance without requiring changes to the existing Pandas code.

- Database:
  - MongoDB Atlas: MongoDB Atlas is a cloud-based, fully-managed NoSQL database. It is used to store the extracted data in a flexible and scalable manner. The document-based architecture of MongoDB allows for storing semi-structured data, such as product listings, with ease. MongoDB Atlas ensures that the data is securely stored, backed up, and can be accessed from anywhere.
- Architecture Design:
  - Draw.io: Draw.io is a free, online diagramming tool used to create visual representations of the system architecture. It was employed to design the flow and structure of the web crawler, detailing how different components of the system interact with each other. The architecture design created using Draw.io helps ensure that the project has a well-organized and logical structure.
- Reporting & Documentation:
  - Google Docs: Google Docs is a cloud-based word processing tool used for creating and editing documentation. It was used for writing and storing the project report, maintaining detailed records of the development process, and ensuring that all findings and methodologies are clearly documented. Google Docs also facilitates collaboration, allowing multiple team members to work on the document simultaneously.

## 2.3 Roles of Team Members

Appendix 1 displays the role description of every team member based on the processes involved. This table outlines the distribution of tasks for a project on web scraping and data analysis from eBay Malaysia. The entire team is responsible for identifying eBay Malaysia as the target website for scraping.

For the web scraping process, each member is assigned specific libraries: Selenium (Nurul Erina), Scrapy (Ong Yi Yan), Playwright (Tang Yan Qing), and Requests/BeautifulSoup (Wong Qiao Ying). All team members will scrape at least 25,000 rows of data. Wong Qiao Ying also sets up the MongoDB Atlas database to store the data.

In the Data Cleaning phase, each member uses different tools: Modin (Nurul Erina), PySpark (Ong Yi Yan), Polars (Tang Yan Qing), and Pandas (Wong Qiao Ying). Ong Yi Yan will compare performance metrics using Matplotlib and NumPy.

For Reporting and Documentation, Nurul Erina will document the scraping method, number of records, conclusion, and future work; Ong Yi Yan focuses on performance evaluation; Tang Yan Qing will document data collection ethics, optimization, and challenges; and Wong Qiao Ying will cover system architecture and references. This ensures clear division of labor and thorough documentation.

## 3.0 Data Collection

Data collection is a critical aspect of this project, where the primary goal is to efficiently and responsibly extract large datasets from eBay. The data collection process is divided into distinct stages, starting with crawling methods to retrieve eBay listings and followed by necessary considerations such as rate-limiting, pagination handling, and asynchronous support. The extracted data is then stored in a MongoDB database for further analysis and processing.

In this section, we describe the methods and techniques used to collect the data from eBay, including how we ensured that the process was ethical, responsible, and within the boundaries of legal regulations.

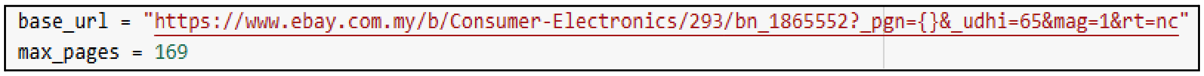
## 3.1 Crawling Method

The crawling method is designed to systematically extract data from eBay using specific techniques. These techniques are implemented to ensure that the data collection process is efficient, robust, and sustainable while also reducing the risk of detection or blocking by eBay.

### 3.1.1 Pagination Handling

Figure 3.1.1.1 displays the base URL of eBay with a dynamic page number, allowing the crawler to scrape one page of listings at a time in Consumer Electronics categories. The script is designed to handle pagination by modifying the URL with different page numbers, starting from page 1 and going up to the maximum allowed pages (in this case, page 169).

To refine the dataset, the crawler also applies a price filter using specific parameters such as _udhi=65 for a minimum price and "_udhi=150 & udlo=65 for a specified price range. This ensures that only listings within the desired price range are scraped, optimizing the data collection process and staying within the limits of available data. The crawler continues to visit each eBay page up to the maximum (page 169), ensuring a comprehensive and filtered dataset.

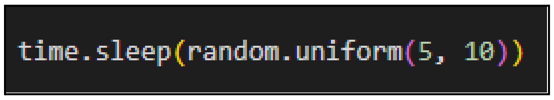


```
base_url = "https://www.ebay.com.my/b/Consumer-Electronics/293/bn_1865552?_pgn={}&_udhi=65&mag=1&rt=nc"
max_pages = 169
```

**Figure 3.1.1.1 Base URL and maximum pages to scrape**

## 3.1.2 Rate-Limiting

Figure 3.1.2.1 displays the random delay timer in web scraping. The scrapper will stop for five to ten seconds randomly after finishing scraping a page before continuing. Rate-limiting is implemented to prevent overloading eBay's servers and to avoid detection or blocking by the website. To mimic human browsing behavior, the script includes random delays between each page request. These delays help the crawler appear more like a real user, rather than an automated bot, reducing the likelihood of being blocked. By introducing random hold times between requests and adjusting the rate of requests, the crawler can operate efficiently while respecting the limitations of the eBay website.



```
time.sleep(random.uniform(5, 10))
```

**Figure 3.1.2.1 Random delay timer in web scraping**

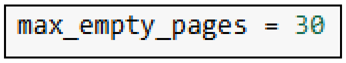
## 3.1.3 Asynchronous Support

Asynchronous support is an important consideration for improving the efficiency and speed of the web scraping process. However, in this project, the crawling process does not implement asynchronous methods. All commands, such as driver.get(url) or find_element, are executed sequentially, meaning the program waits for each command to complete before proceeding to the next one. This approach ensures that the program interacts with eBay in a controlled, step-by-step manner, retrieving one page and one listing at a time. While this method may not be as fast as asynchronous scraping, it ensures the crawler interacts with eBay in a predictable and manageable way.

This sequential approach aligns with the focus on responsible data collection and helps mitigate potential performance issues related to simultaneous requests, which could negatively impact the server's performance or result in the crawler being flagged for suspicious activity.

3.1.4 Empty Page Handling

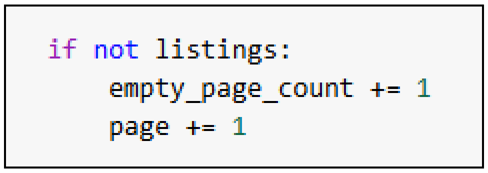
Figure 3.1.4 displays the maximum number of empty pages when scraping from the website. In some cases, eBay may return empty pages that do not contain any listings. The script checks for the presence of product listings on each page and increments an empty_page_count if no listings are found. If the empty page count exceeds a predefined limit (max_empty_pages), the script will stop scraping for that brand.



```
max_empty_pages = 30
```

**Figure 3.1.4.1 Maximum number of empty pages**

Figure 3.1.4.2 displays the looping function to record the count of empty pages. When an empty page is encountered, empty_page_count will increase by one until reaching the predefined maximum empty pages. This method optimizes the crawling process by reducing unnecessary requests and focusing the scraper's efforts on pages that are likely to contain useful data. By limiting the number of empty pages it processes, the script ensures that the data collection process remains efficient and effective.

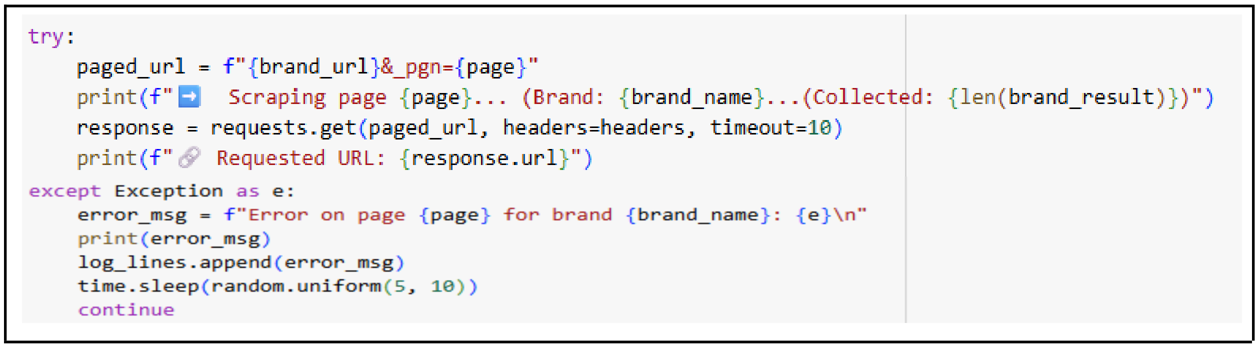


```
if not listings:
    empty_page_count += 1
    page += 1
```

**Figure 3.1.4.2 Looping through empty pages**

## 3.1.5 Error Handling

Figure 3.1.5.1 displays the try-except blocks to ensure that it continues operating smoothly even when unexpected issues occur. The use of the blocks allows the script to handle errors, such as failed HTTP requests or problems with the page structure, without crashing. If an error occurs, the script pauses for a random delay, and then attempts to scrape the next page. By catching exceptions and continuing the scraping process, the script is more resilient and can recover from minor disruptions. This ensures that the scraping process continues even if a particular page or request fails. The inclusion of error handling is vital for maintaining the stability of the crawler, especially when working with large datasets or scraping multiple pages.

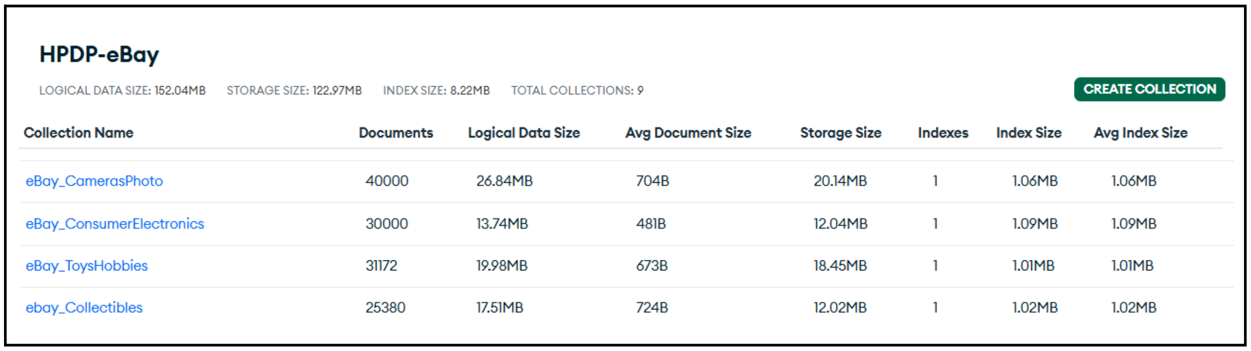


```
try:
    paged_url = f"{brand_url}&_pgn={page}"
    print(f"➡️  Scraping page {page}... (Brand: {brand_name}...(Collected: {len(brand_result)})")
    response = requests.get(paged_url, headers=headers, timeout=10)
    print(f"🔗 Requested URL: {response.url}")
except Exception as e:
    error_msg = f"Error on page {page} for brand {brand_name}: {e}\n"
    print(error_msg)
    log_lines.append(error_msg)
    time.sleep(random.uniform(5, 10))
    continue
```

**Figure 3.1.5.1 Try-except blocks for error handling**

## 3.2 Number of Records Collected

Figure 3.2.1 displays the number of data collected for each category. A total of 126552 rows of data has been extracted and stored in MongoDB across four categories.



HPDP-eBay

LOGICAL DATA SIZE: 152.04MB STORAGE SIZE: 122.97MB INDEX SIZE: 8.22MB TOTAL COLLECTIONS: 9

| Collection Name | Documents | Logical Data Size | Avg Document Size | Storage Size | Indexes | Index Size | Avg Index Size |
|---|---|---|---|---|---|---|---|
| eBay_CamerasPhoto | 40000 | 26.84MB | 704B | 20.14MB | 1 | 1.06MB | 1.06MB |
| eBay_ConsumerElectronics | 30000 | 13.74MB | 481B | 12.04MB | 1 | 1.09MB | 1.09MB |
| eBay_ToysHobbies | 31172 | 19.98MB | 673B | 18.45MB | 1 | 1.01MB | 1.01MB |
| ebay_Collectibles | 25380 | 17.51MB | 724B | 12.02MB | 1 | 1.02MB | 1.02MB |

**Figure 3.2.1 Number of data collected for each category**

## 3.3 Ethical Considerations

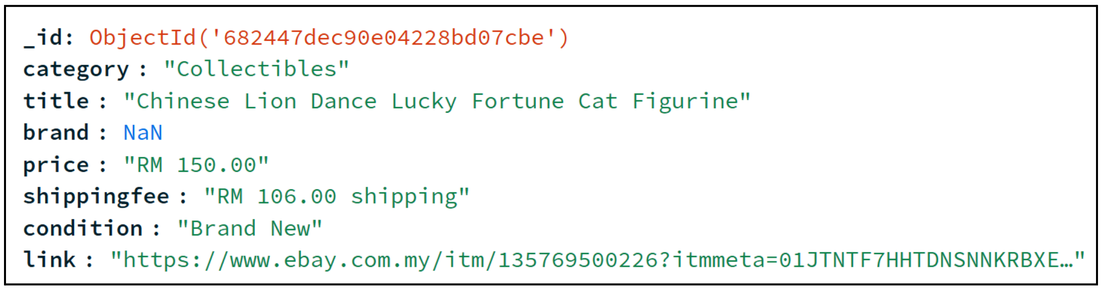
While conducting data scraping on eBay, we were highly conscious of the ethical and legal boundaries. First of all, scraping was carried out solely for research and educational purposes only. We had no intention of using it for commercial purposes or data misuse. Second, we collected only publicly published product details like brand, title, category, price and shipping fee without touching any kind of sensitive or transactional user information. Third, to ensure minimal server load and mimic real user activity, we utilized throttling techniques, including randomized user agents, hold times and controlled request rates. Fourth, we also reviewed eBay's robots.txt file to guide responsible scraping activity and kept our tool usage below acceptable access levels. Transparency and integrity were paramount throughout the duration of the project.

## 4.0 Data Processing

The data processing phase serves as a critical component of this project, wherein data retrieved from four distinct collections within a MongoDB database is refined and prepared for subsequent analysis and performance benchmarking. The data processing workflow is systematically divided into three key stages: **Data Loading, Data Cleaning, and Data Transformation.**

## 4.1 Data Structure

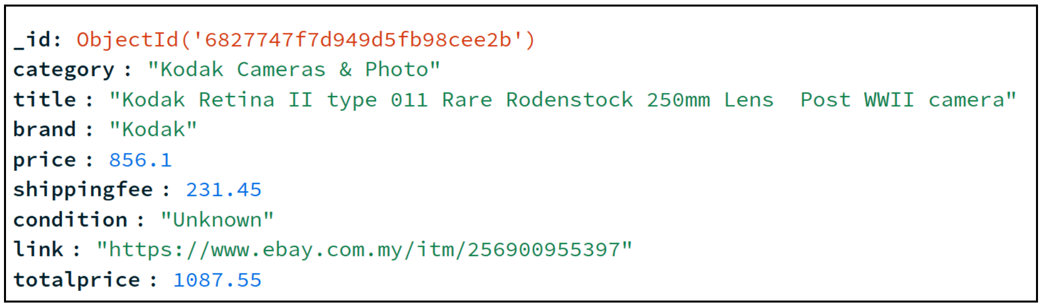
Prior to the data processing phase, the scraped data was organized into four separate collections within the MongoDB database named **HPDP_eBay**. These collections—**eBay_CamerasPhoto**, **eBay_Collectibles**, **eBay_ToysHobbies**, and **eBay_ConsumerElectronics**—each stored data specific to a product category. Every collection contained documents with **eight attributes**: _id, category, title, brand, condition, price, shipping fee, and link. Figure 4.1.1 illustrates the data structure of one collection, eBay_CamerasPhoto, as an example of the initial format.



```
_id: ObjectId('682447dec90e04228bd07cbe')
category : "Collectibles"
title : "Chinese Lion Dance Lucky Fortune Cat Figurine"
brand : NaN
price : "RM 150.00"
shippingfee : "RM 106.00 shipping"
condition : "Brand New"
link : "https://www.ebay.com.my/itm/135769500226?itmmeta=01JTNTF7HHTDNSNNKRBXE…"
```

**Figure 4.1.1: Data structure of the eBay_CamerasPhoto collection before data processing.**

Following the data processing steps—which included loading, cleaning, and transformation—the data from all four collections were merged into a single consolidated collection named **cleaned_All**. This unified dataset contains **nine attributes**, with the addition of a new column, totalprice, which represents the sum of the cleaned price and shipping fee fields. Figure 4.1.2 depicts the structure of the cleaned and consolidated dataset in its final form.



```
_id: ObjectId('6827747f7d949d5fb98cee2b')
category : "Kodak Cameras & Photo"
title : "Kodak Retina II type 011 Rare Rodenstock 250mm Lens  Post WWII camera"
brand : "Kodak"
price : 856.1
shippingfee : 231.45
condition : "Unknown"
link : "https://www.ebay.com.my/itm/256900955397"
totalprice : 1087.55
```

**Figure 4.1.2: Data structure of the cleaned_All collection after data processing.**

## 4.2 Data Loading

In the initial step of the data processing phase, the previously scraped data—stored in four separate collections within MongoDB Atlas—is retrieved using the pymongo client in Python. Each collection is loaded and merged into a unified dataframe. This process is conducted independently for each of the four selected data processing libraries—Pandas, Polars, PySpark, and Modin—to ensure consistency in the data structure and enable a reliable performance comparison across frameworks.

## 4.3 Data Cleaning

The data cleaning phase is essential for enhancing the integrity and consistency of the dataset prior to analysis. The process begins by **dropping the _id column**, which is automatically generated by MongoDB and is not pertinent to the analysis objectives. Following this, **rows that contain three or more null values are** identified and **removed**, as such records are considered insufficiently complete for reliable analysis.

After pruning incomplete entries, the dataset undergoes targeted imputation to handle the remaining missing values. For categorical attributes such as brand, category, condition, and link, **missing values are filled** with the placeholder 'Unknown'. A corrected 'Unknown' value is also applied to the title field. In the case of numerical fields such as shippingfee and price, missing values are replaced with 0.0 to maintain numerical consistency.

To further improve data quality, **duplicate records are removed** based on a combination of the title, brand, price, shippingfee, and condition fields. Additionally, all string-type columns are trimmed to eliminate leading and trailing whitespaces, ensuring textual uniformity across entries. The **link** column is normalized by extracting the canonical eBay item ID, thereby standardizing URL formats. Lastly, the **title** field is refined by removing "NEW LISTING" prefixes, sanitizing the text to eliminate superfluous symbols and non-alphanumeric characters, and applying a final whitespace trim.

## 4.4 Data Transformation

The data transformation stage focuses on refining key numerical attributes to enable accurate computation and analysis. The **shippingfee** column is first transformed by extracting numeric values from textual representations. For instance, strings such as "RM 23.00 shipping" are parsed to extract the numeric amount 23.00, while entries labeled as "free shipping" are standardized to 0.0. This conversion ensures that the entire column is represented in a consistent numeric format suitable for further processing.

Similarly, the **price** column undergoes formatting to remove currency symbols or textual prefixes—such as "rm"—resulting in clean numerical values like 23.00. These cleaned entries are then explicitly converted into a numeric data type to facilitate numerical operations.

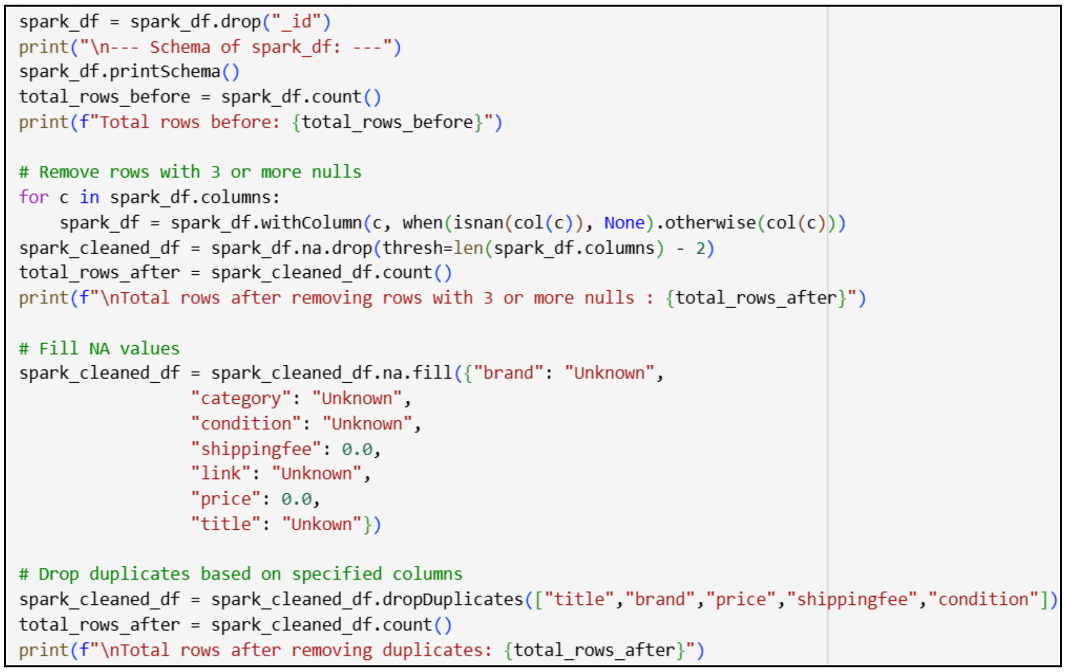
A new column named **totalprice** is introduced. This field is computed by summing the formatted values of the price and shippingfee columns, providing a unified representation of the total item cost inclusive of shipping.

## 5.0 Optimization Techniques

Several optimization techniques were implemented in this project to improve data processing speed, memory efficiency, and scalability. The main libraries used were **PySpark, Modin, and Polars**. Each of them provides unique performance-enhancing mechanisms, allowing the project to manage large volumes of scraped data more effectively.

**PySpark** enhances large-scale big data processing by distributing tasks across multiple CPU cores or machines. This enables parallel execution and in-memory computation, significantly enhancing performance when working with big data. Operations such as deduplication, management of missing values and the addition of new columns are done lazily, which means they are only processed when needed. This enables Spark's Catalyst optimizer to develop optimal plans of execution. Additionally, complex operations with regular expressions such as replacements and extractions are performed directly on Spark DataFrames, reducing unnecessary intermediate computations and improving memory efficiency.

As illustrated in Figure 5.0.1, PySpark's lazy evaluation enables optimized execution plans that enhance distributed data processing efficiency.

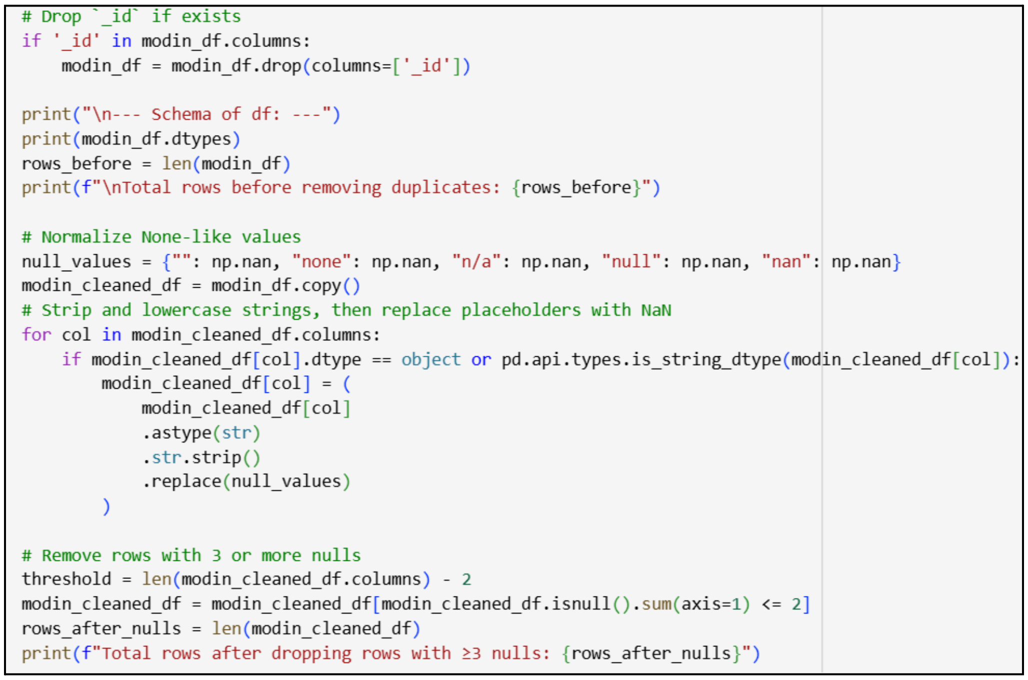


```
spark_df = spark_df.drop("_id")
print("\n--- Schema of spark_df: ---")
spark_df.printSchema()
total_rows_before = spark_df.count()
print(f"Total rows before: {total_rows_before}")

# Remove rows with 3 or more nulls
for c in spark_df.columns:
    spark_df = spark_df.withColumn(c, when(isnan(col(c)), None).otherwise(col(c)))
spark_cleaned_df = spark_df.na.drop(thresh=len(spark_df.columns) - 2)
total_rows_after = spark_cleaned_df.count()
print(f"\nTotal rows after removing rows with 3 or more nulls : {total_rows_after}")

# Fill NA values
spark_cleaned_df = spark_cleaned_df.na.fill({"brand": "Unknown",
                 "category": "Unknown",
                 "condition": "Unknown",
                 "shippingfee": 0.0,
                 "link": "Unknown",
                 "price": 0.0,
                 "title": "Unkown"})

# Drop duplicates based on specified columns
spark_cleaned_df = spark_cleaned_df.dropDuplicates(["title","brand","price","shippingfee","condition"])
total_rows_after = spark_cleaned_df.count()
print(f"\nTotal rows after removing duplicates: {total_rows_after}")
```

**Figure 5.0.1: PySpark's lazy evaluation**

**Modin** improves the performance of pandas operations by distributing them across multiple CPU cores using multiprocessing frameworks like Ray or Dask. Pandas operations such as reading the data, combining DataFrames, dropping duplicates and substituting strings are automatically parallelized, which makes the execution faster with minimal changes to existing code. Modin preserves the familiar pandas interface while internally managing data partitioning and task scheduling. It also defers execution and optimizes memory usage by chunking operations, providing a scalable and efficient solution for large datasets in multi-core environments.

Figure 5.0.2 demonstrates how Modin parallelizes pandas operations using Ray or Dask, providing scalable performance on multi-core systems.

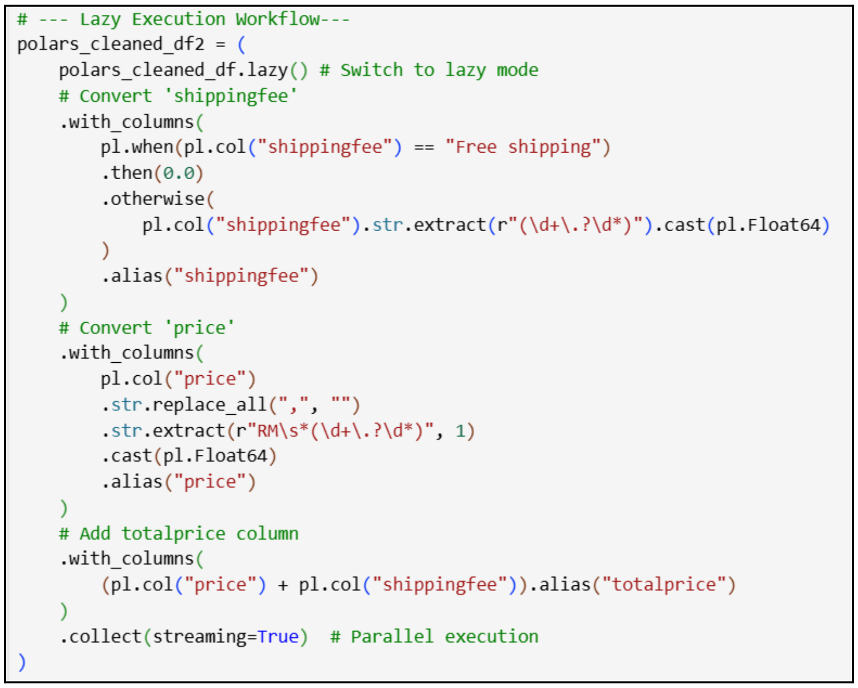


```python
# Drop `_id` if exists
if '_id' in modin_df.columns:
    modin_df = modin_df.drop(columns=['_id'])

print("\n--- Schema of df: ---")
print(modin_df.dtypes)
rows_before = len(modin_df)
print(f"\nTotal rows before removing duplicates: {rows_before}")

# Normalize None-like values
null_values = {"": np.nan, "none": np.nan, "n/a": np.nan, "null": np.nan, "nan": np.nan}
modin_cleaned_df = modin_df.copy()
# Strip and lowercase strings, then replace placeholders with NaN
for col in modin_cleaned_df.columns:
    if modin_cleaned_df[col].dtype == object or pd.api.types.is_string_dtype(modin_cleaned_df[col]):
        modin_cleaned_df[col] = (
            modin_cleaned_df[col]
            .astype(str)
            .str.strip()
            .replace(null_values)
        )

# Remove rows with 3 or more nulls
threshold = len(modin_cleaned_df.columns) - 2
modin_cleaned_df = modin_cleaned_df[modin_cleaned_df.isnull().sum(axis=1) <= 2]
rows_after_nulls = len(modin_cleaned_df)
print(f"Total rows after dropping rows with ≥3 nulls: {rows_after_nulls}")
```

**Figure 5.0.2: Modin parallelizes pandas operations**

**Polars** utilizes multithreading and SIMD (Single Instruction, Multiple Data) to accelerate data processing on a single machine. Through its lazy evaluation API, multiple operations such as column transformations, filtering, and regex-based cleaning are combined into a single optimized execution plan and run in parallel. This minimizes overhead from intermediate computations. Polars supports streaming execution as well, so it can handle large datasets efficiently without loading the entire dataset into memory. Its zero-copy memory design further enhances speed and memory efficiency.

The approach taken by Polars, shown in Figure 5.0.3, highlights its streaming and lazy evaluation capabilities for high-performance single-machine processing.



```
# --- Lazy Execution Workflow---
polars_cleaned_df2 = (
    polars_cleaned_df.lazy() # Switch to lazy mode
    # Convert 'shippingfee'
    .with_columns(
        pl.when(pl.col("shippingfee") == "Free shipping")
        .then(0.0)
        .otherwise(
            pl.col("shippingfee").str.extract(r"(\d+\.?\d*)").cast(pl.Float64)
        )
        .alias("shippingfee")
    )
    # Convert 'price'
    .with_columns(
        pl.col("price")
        .str.replace_all(",", "")
        .str.extract(r"RM\s*(\d+\.?\d*)", 1)
        .cast(pl.Float64)
        .alias("price")
    )
    # Add totalprice column
    .with_columns(
        (pl.col("price") + pl.col("shippingfee")).alias("totalprice")
    )
    .collect(streaming=True)  # Parallel execution
)
```

**Figure 5.0.3: Polars's lazy evaluation and streaming**

Overall, PySpark, Modin, and Polars each offer unique optimization strategies that improve data processing performance in various contexts: distributed computing, parallelized pandas operations, and multithreaded streaming execution, respectively.

## 6.0 Performance Evaluation

This project begins by using **Pandas** to carry out core data processing tasks, including data loading, cleaning, and transformation. To enhance processing efficiency and reduce execution time, three additional high-performance libraries were subsequently utilized: **PySpark**, which capitalizes on distributed computing; **Modin**, which accelerates Pandas workflows through parallelization with minimal code modifications; and **Polars**, a modern, Rust-based framework designed for fast, multi-threaded data operations. The objective of this evaluation is to compare the performance of these libraries in executing identical tasks and to analyze their relative strengths and trade-offs. The evaluation relies on metrics such as total execution time, CPU usage, peak memory consumption, and throughput.

## 6.1 Evaluation Metrics

To enable a comprehensive and objective performance comparison among Pandas, PySpark, Modin, and Polars, four key metrics are employed—each reflecting different aspects of computational efficiency and system resource utilization.

**Total Execution Time** represents the duration required to complete each individual phase of the data processing workflow—data loading, data cleaning, and data transformation. Recording execution time per step identifies which phases are more time-consuming for each framework.

**CPU Usage** measures the average percentage of CPU resources consumed during each specific processing step, which identifies how computationally intensive each phase is independently.

**Peak Memory Usage** captures the maximum memory consumption during each stage of data processing, which is crucial for understanding memory demands and management throughout the workflow.

**Throughput** is defined as the number of records processed per second within each processing phase. Measuring throughput stepwise provides a clearer picture of data handling efficiency and potential bottlenecks at different points in the pipeline.

## 6.2 Results and Discussion

The detailed numerical results for all evaluated metrics across each data processing step and library are presented in Appendix 2. This comprehensive table reports execution time, CPU usage, peak memory usage, and throughput for the three main data processing steps: Data

Loading, Basic Cleaning, and Data Transformation, evaluated using Pandas, PySpark, Modin, and Polars.

### 6.2.1 Total Execution Time

To visualize execution time differences, Figure 6.2.1.1 presents a bar chart comparing total execution times of the four libraries across the three primary data processing operations. As shown in Figure 6.2.1.1, Polars consistently delivers the fastest execution times, with especially notable improvements during Basic Cleaning (0.86 seconds) and Data Transformation (0.17 seconds). Pandas exhibits moderate performance in Data Loading (12.86 seconds) and generally outperforms Modin, which shows slower speeds across all processing steps. PySpark demonstrates the longest execution times, largely due to overhead from initialization.

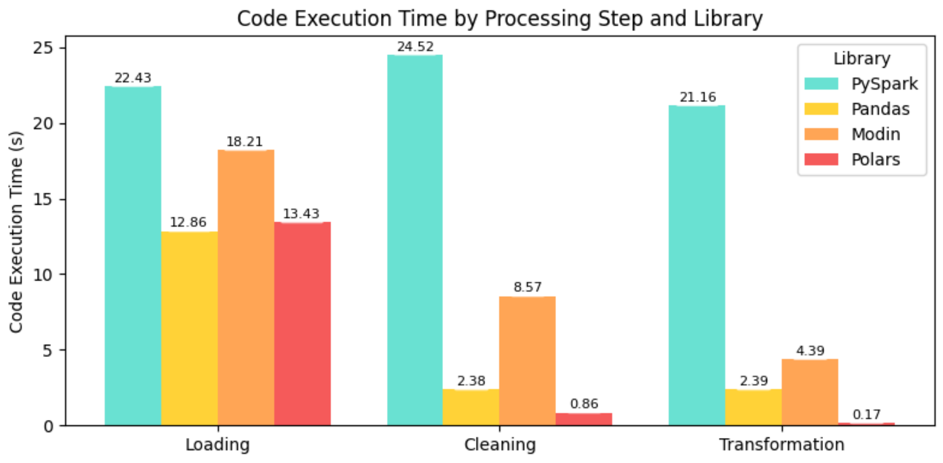


**Figure 6.2.1.1: Bar chart comparing total execution times (in seconds) across data processing steps for four libraries.**

### 6.2.2 CPU Usage

A deeper look into CPU resource utilization is provided in Figure 6.2.2.1, which presents a line chart depicting average CPU usage (%) across each data processing step for Pandas, PySpark, Modin and Polars. It highlights that Polars and Pandas utilize CPU resources more intensively during cleaning and transformation, with Polars reaching nearly 48% CPU usage during transformation. PySpark, in contrast, maintains low CPU usage in these steps, likely due to overheads and distributed execution delays. Modin's CPU usage remains modest throughout but higher than PySpark, reflecting its parallelized Pandas approach.

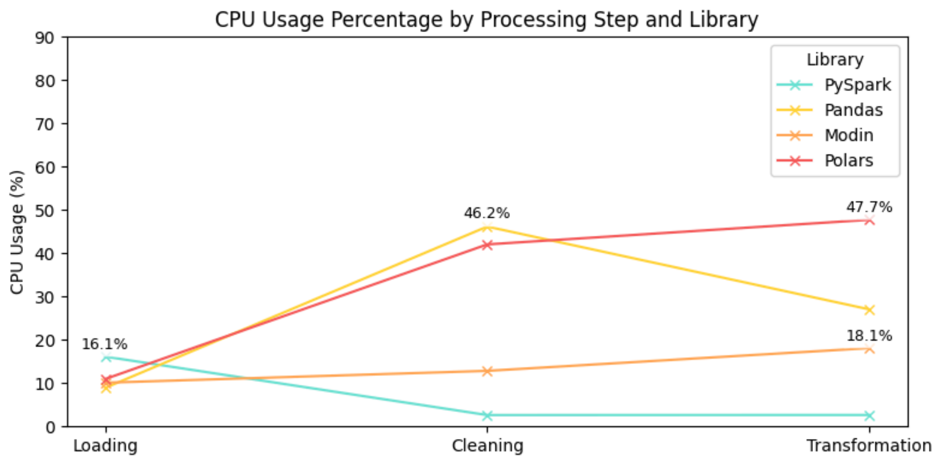


**Figure 6.2.2.1: Line chart comparing average CPU usage (%) across data processing steps for four libraries.**

### 6.2.3 Peak Memory Usage

Memory consumption patterns are visualized in Figure 6.2.3.1 via a heatmap illustrating peak memory usage (in MB) across the three data processing steps for the four libraries. From Figure 6.2.3.1, PySpark consistently consumes the highest memory, approximately 818 MB across all operations, reflecting its distributed in-memory data model. Modin uses the most memory during cleaning and transformation (over 1200 MB), indicative of its parallelization overhead. Polars maintains a relatively high but stable memory footprint around 1000–1200 MB, balancing resource usage and performance. Pandas uses the least memory among all, averaging around 500–580 MB.

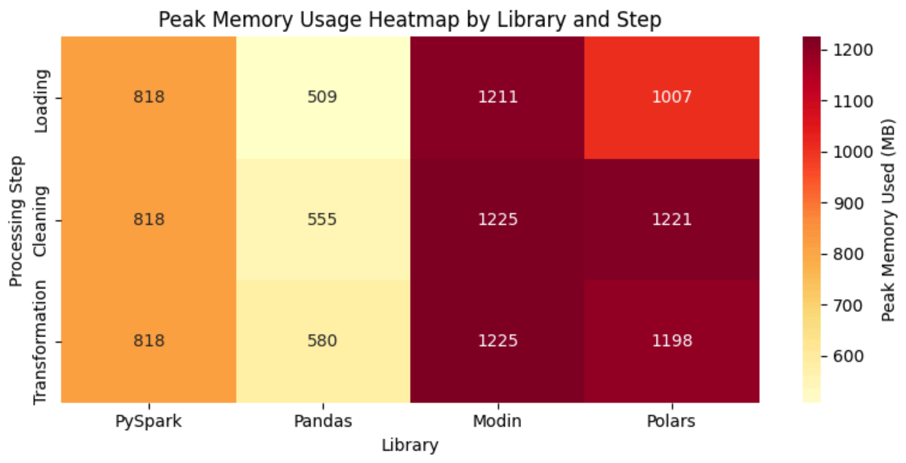


**Figure 6.2.3.1: Heatmap illustrating peak memory usage (in MB) across each data processing step for four libraries.**

## 6.2.4 Throughput

To evaluate throughput, Figure 6.2.4.1 provides a horizontal bar chart depicting throughput (records per second) across each data processing step for four libraries.Polars demonstrates its edge by processing 125,323 records per second in cleaning and 640,177 in transformation. Pandas and Modin achieve moderate throughput rates, with Pandas leading in Data Loading (9,844 records/s) but significantly lower throughput in other steps. PySpark records the lowest throughput across all operations, hindered by distributed overhead and latency.

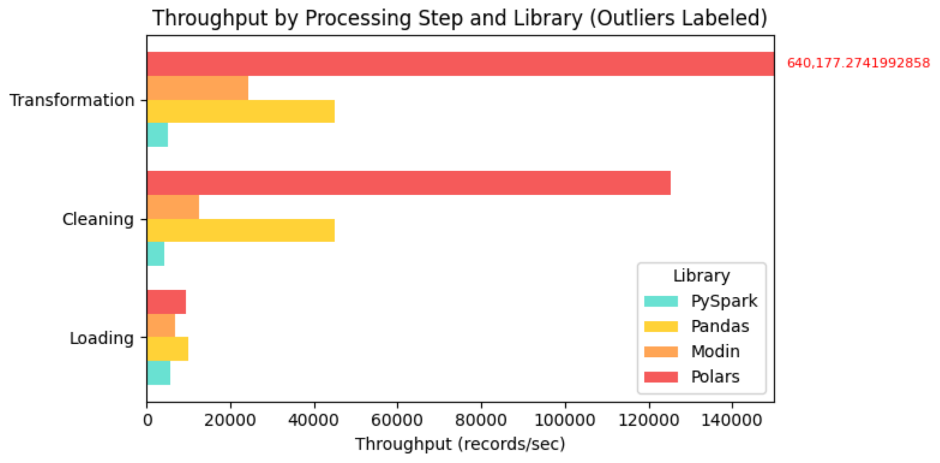


**Figure 6.2.4.1: Horizontal bar chart comparing throughput (records/second) across data processing steps for four libraries.**

### 6.2.5 Conclusion

Overall, Polars demonstrated the most efficient performance across the data processing workflow. While its data loading speed was competitive, it excelled in basic cleaning and data transformation, delivering unmatched speed and throughput. Pandas achieved the fastest data loading time but was significantly slower than Polars in subsequent tasks, with notably lower throughput. Modin did not outperform Pandas and Polars in any phase, showing slower speeds across all steps. PySpark consistently exhibited the slowest performance overall, with the longest execution times and lowest throughput across all data processing stages.

## 7.0 Challenges and Limitations

This section introduces the challenges encountered and the limitations of the proposed solution during the project. Although the project achieved its objectives, the development process revealed several practical issues that affected data quality, processing efficiency, and overall scalability.

### 7.1 Challenges Encountered

Despite careful planning, several unexpected challenges emerged during the data scraping and processing stages. These issues affected data quality, processing efficiency, and overall project accuracy.

One of the key obstacles encountered was inconsistent website behavior. Different libraries may experience different levels of reliability in retrieving complete data during web scraping. For example, sometimes some pages may fail to load using Selenium or Playwright due to bot detection on eBay. Such conditions affected the scraping accuracy and completeness.

Another issue was rate limiting and fake results. eBay only allows loading up to 169 pages or around 10,000 results per search. Beyond that, the platform may return repeated or fake listings, making it difficult to retrieve comprehensive datasets. This introduced data duplication and skewed results.

The project also faced problems due to varying HTML structures. In eBay, the HTML structure for similar fields was not uniform across items. For instance, some shipping fees were inside a <span> while others were within a <div> with class <s-item>. This led to massive null values during scraping and required additional conditional parsing logic to handle them.

Furthermore, there were issues with the data processing library initially selected. Dask was used for data processing, but it performed poorly with datasets larger than 100k rows. It took extremely long or failed to complete when doing data transformation. After further research, Dask was replaced by Modin, which showed better performance in this scenario.

Lastly, inconsistent null handling across libraries presented a challenge. Data processing across Polars, Pandas, PySpark, and Modin were more complex than expected. Since each library handles null values and missing data differently, the results when applying the same logic were affected. For example, Polars treats nulls more strictly, while PySpark may preserve them unless explicitly filtered. This required writing library-specific data cleaning code to ensure consistency.

## 7.2 Constraints of The Proposed Solution

While the solution achieved its primary objectives, several limitations were identified that could impact its scalability, efficiency, and completeness. These constraints highlight areas for future improvement and should be considered when extending or deploying the solution in other contexts.

Firstly, single-machine execution posed a limitation. Some of the data processing libraries used like PySpark and Modin are designed for distributed computing environments. However, all testing in this project was done on a single machine. This limits their ability to perform true scalability and high-performance processing.

Secondly, processing overhead was another constraint. Libraries like PySpark take extra time and system resources to initialize before running any operations. For medium-sized datasets (approximately 100,000 records), this setup time slowed down the workflow compared to more lightweight tools like Pandas or Polars.

In addition, the solution does not support real-time data handling. All data scraping and processing were done in batch mode. It means that we scraped everything first, then processed it. This makes the solution unsuitable for real-time use cases like live product tracking or continuous updates.

Lastly, there was the possibility of missed data due to complex web features. Although we added logic to handle dynamic elements like the shipping fee, we still couldn't manually verify every item. This means that there is a chance that some values were still missed or recorded as null, especially for listings with unusual layouts or hidden content.

# 8.0 Conclusion and Future Works

## 8.1 Summary of Findings

The evaluation tested the performance of Pandas, PySpark, Polars, and Modin using total execution time, CPU usage, memory utilization, and how fast these libraries handle each of three key operations. loading the data, basic cleaning of the data, and transforming the data.

Pandas is an excellent choice for handling light workloads such as basic data cleaning and transformation. It performs exceptionally well with small to medium-sized datasets, particularly when running on a single machine. Due to its high CPU utilization and efficient single-threaded performance, Pandas offers great throughput for smaller datasets. It is also ideal for quick prototyping and exploratory data analysis (EDA), making it a go-to tool for many data scientists during the initial stages of data exploration.

PySpark, in contrast, is designed for big data scenarios. It handles large datasets efficiently through distributed processing across clusters, making it highly suitable for use in cloud or cluster environments. PySpark maintains stable memory usage across various tasks, which is especially important in large-scale ETL pipelines. It is the best fit when dealing with very large data volumes that go beyond the capabilities of single-machine tools like Pandas.

Polars and Modin address the need for higher performance and scalability. Polars delivers the fastest execution and highest throughput, particularly in transformation tasks, due to its efficient multi-threaded processing and balanced CPU and memory usage. It is ideal for performance-critical applications and suitable for real-time or near real-time processing environments. Modin, on the other hand, offers parallelized execution with a Pandas-like syntax, providing a smoother learning curve for users familiar with Pandas. It performs well on moderate workloads and multi-core machines, making it a practical solution for scaling up existing Pandas code with minimal changes.

## 8.2 Recommendations for Future Improvements

Although each framework is useful in its own way, there are situations where their limits play a role. Understanding of such areas allows one to choose suitable tools for the right task .

1. Better Resource Utilization in PySpark

Despite being designed for large-scale data processing, PySpark showed very low CPU usage during basic tasks like cleaning and transformation. This underutilization suggests that PySpark is inefficient for small to medium-sized datasets and may introduce unnecessary overhead. Improving how PySpark handles lighter workloads or offering a lightweight mode could make it more versatile.

2. More Consistent Speed Gains in Modin

Modin aims to speed up Pandas by enabling parallel processing, but the results show inconsistent performance gains across different operations. For example, while it improves transformation speed compared to Pandas, it still lags behind Polars. This suggests room for optimization in how Modin distributes and executes tasks internally, especially in basic cleaning.

3. Expanded Feature Support in Polars

Polars delivered the best throughput and speed across nearly all tasks, especially transformations. However, it’s still a relatively new library and doesn’t yet match Pandas in terms of ecosystem or advanced functionality. Improving compatibility with existing Python tools and adding support for more complex operations could help Polars gain broader adoption.

4. Enhanced Scalability in Pandas

Pandas performed well in smaller tasks with high throughput and efficiency, but it doesn't scale well with growing data sizes. Memory usage and single-threaded processing become limitations for larger workloads. Incorporating built-in support for parallelism or native scaling features could significantly extend its usability.

## 9.0 References

[1] Python Software Foundation, "Python Language Reference, version 3.11", [Online]. Available: https://www.python.org

[2] The Pandas Development Team, "Pandas Documentation", [Online]. Available: https://pandas.pydata.org/docs/

[3] MongoDB Inc., "MongoDB Manual", [Online]. Available: https://www.mongodb.com/docs/manual/

[4] PyMongo Developers, "PyMongo Documentation", [Online]. Available: https://pymongo.readthedocs.io/en/stable/

[5] NumPy Developers, "NumPy Documentation", [Online]. Available: https://numpy.org/doc/

[6] Python Software Foundation, "re — Regular Expression Operations", [Online]. Available: https://docs.python.org/3/library/re.html

[7] eBay Inc., "eBay Malaysia - Electronics, Cars, Fashion, Collectibles & More", [Online]. Available: https://www.ebay.com.my

[8] Polars Developers, "Polars User Guide", [Online]. Available: https://pola-rs.github.io/polars/

[9] Apache Software Foundation, "PySpark Documentation", [Online]. Available: https://spark.apache.org/docs/latest/api/python/

[10] Modin Contributors, "Modin: Speed up your Pandas workflows by changing a single line of code", [Online]. Available: https://modin.readthedocs.io/en/latest/

[11] Scrapy Developers, "Scrapy Documentation", [Online]. Available: https://docs.scrapy.org/en/latest/

[12] Apache Software Foundation, "Apache Spark Documentation", [Online]. Available: https://spark.apache.org/docs/latest/

[13] L. Richardson et al., "BeautifulSoup: Python library for pulling data out of HTML and XML files", [Online]. Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/

[14] PSUtil Developers, "psutil – Process and System Utilities", [Online]. Available: https://psutil.readthedocs.io/en/latest/

[15] R. Mitchell, Web Scraping with Python: Collecting More Data from the Modern Web, 2nd ed., O'Reilly Media, 2018.

[16] M. Lutz, Learning Python, 5th ed., Sebastopol, CA: O'Reilly Media, 2013.

## 10.0 Appendices

**Appendix 1: Roles Description of Team Member**

| **Process** | **Member Name** | **Outcome/ Description** |
|---|---|---|
| Website identification | Everyone | eBay Malaysia |
| Identify libraries for web scraping | Nurul Erina Binti Zainuddin | Selenium |
| | Ong Yi Yan | Scrapy |
| | Tang Yan Qing | Playwright |
| | Wong Qiao Ying | Requests, BeautifulSoup |
| Web Scraping | Everyone | Each member scrape at least 25,000 rows of data |
| Database Setup | Wong Qiao Ying | MongoDB Atlas |
| Data Cleaning | Nurul Erina Binti Zainuddin | Modin |
| | Ong Yi Yan | PySpark |
| | Tang Yan Qing | Polars |
| | Wong Qiao Ying | Pandas |
| Performance Metrics Comparison | Ong Yi Yan | Matplotlib, Numpy |
| Reporting and Documentation | Nurul Erina Binti Zainuddin | ● Crawling Method and Number of Records<br>● Conclusion & Future Work<br>● Appendices |
| | Ong Yi Yan | ● Performance Evaluation<br>● Data Processing |
| | Tang Yan Qing | ● Data Collection (Ethical Considerations)<br>● Optimization Techniques<br>● Challenges and Limitations |
| | Wong Qiao Ying | ● Description of Architecture |

| | | |
|---|---|---|
| | | ● Data Processing<br>● References |

**Appendix 2: Performance Metrics for Data Processing Steps Across Different Libraries**

| **Operations** | **Metrics** | **Libraries** | | | |
|---|---|---|---|---|---|
| | | **Pandas** | **Pyspark** | **Modin** | **Polars** |
| Data Loading | Total Execution Time (s) | 12.86 | 22.43 | 18.21 | 13.43 |
| | CPU usage (%) | 8.91 | 16.14 | 10.11 | 10.95 |
| | Peak Memory Usage (MB) | 508.68 | 818.06 | 1211.25 | 1007.21 |
| | Throughput (records/s) | 9843.63 | 5641.18 | 6950.08 | 9423.28 |
| Data Cleaning | Total Execution Time (s) | 2.38 | 24.52 | 8.57 | 0.86 |
| | CPU usage (%) | 46.18 | 2.63 | 12.84 | 42.06 |
| | Peak Memory Usage (MB) | 555.21 | 818.09 | 1224.61 | 1220.97 |
| | Throughput (records/s) | 45031.67 | 4374.18 | 12521.91 | 125322.79 |
| Data Transformation | Total Execution Time (s) | 2.39 | 21.16 | 4.39 | 0.17 |
| | CPU usage (%) | 27.04 | 2.65 | 18.10 | 47.74 |

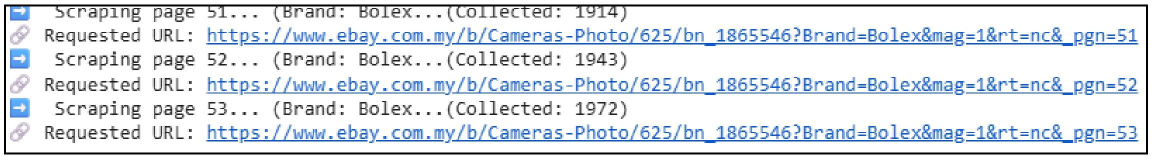
| | | | | | |
|---|---|---|---|---|---|
| | Peak Memory Usage (MB) | 580.05 | 818.13 | 1224.61 | 1197.61 |
| | Throughput (records/s) | 44976.16 | 5068.56 | 24420.35 | 640177.27 |

## 10.1 Screenshots of Output

This section provides visual evidence of the program's results by displaying screenshots of the output generated during the execution of the project.

- Web Crawler

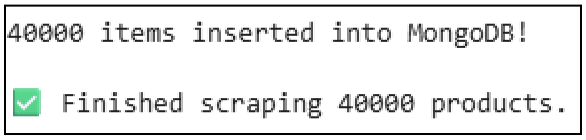
Printing the current page number, brand, and total items collected after each page



```
➡ Scraping page 51... (Brand: Bolex...(Collected: 1914)
🔗 Requested URL: https://www.ebay.com.my/b/Cameras-Photo/625/bn_1865546?Brand=Bolex&mag=1&rt=nc&_pgn=51
➡ Scraping page 52... (Brand: Bolex...(Collected: 1943)
🔗 Requested URL: https://www.ebay.com.my/b/Cameras-Photo/625/bn_1865546?Brand=Bolex&mag=1&rt=nc&_pgn=52
➡ Scraping page 53... (Brand: Bolex...(Collected: 1972)
🔗 Requested URL: https://www.ebay.com.my/b/Cameras-Photo/625/bn_1865546?Brand=Bolex&mag=1&rt=nc&_pgn=53
```

**Figure 10.1.1: Illustrates the console output that displays the current page number, brand name, and the total number of items collected after each page is crawled**

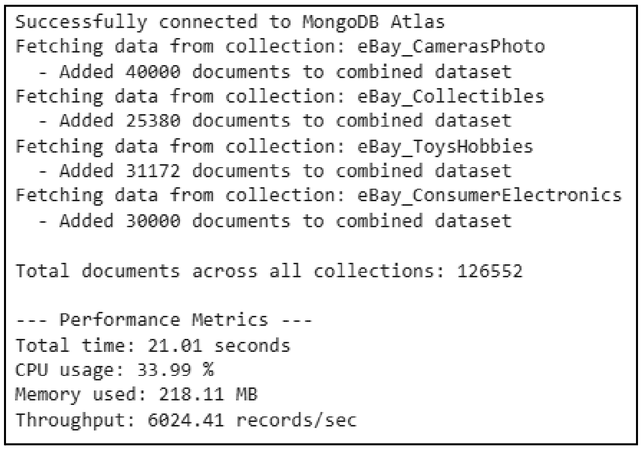
Finished scraping 40k rows and inserted into MongoDB



```
40000 items inserted into MongoDB!

✅ Finished scraping 40000 products.
```

**Figure 10.1.2: Shows the final output indicating the successful completion of the scraping process, with 40,000 rows inserted into MongoDB**

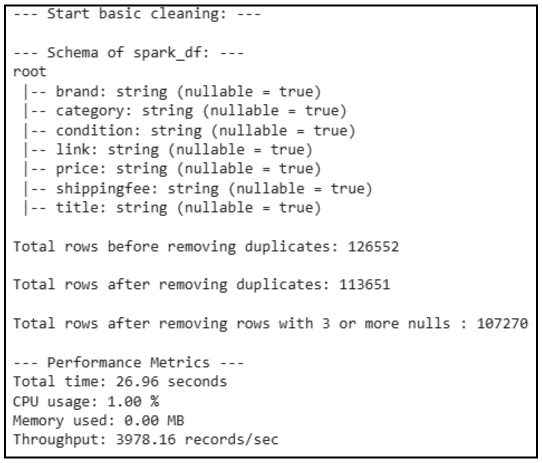
- Data Loading



```
Successfully connected to MongoDB Atlas
Fetching data from collection: eBay_CamerasPhoto
  - Added 40000 documents to combined dataset
Fetching data from collection: eBay_Collectibles
  - Added 25380 documents to combined dataset
Fetching data from collection: eBay_ToysHobbies
  - Added 31172 documents to combined dataset
Fetching data from collection: eBay_ConsumerElectronics
  - Added 30000 documents to combined dataset

Total documents across all collections: 126552

--- Performance Metrics ---
Total time: 21.01 seconds
CPU usage: 33.99 %
Memory used: 218.11 MB
Throughput: 6024.41 records/sec
```

**Figure 10.1.3: Shows the final output indicating the successful completion of Data Loading**

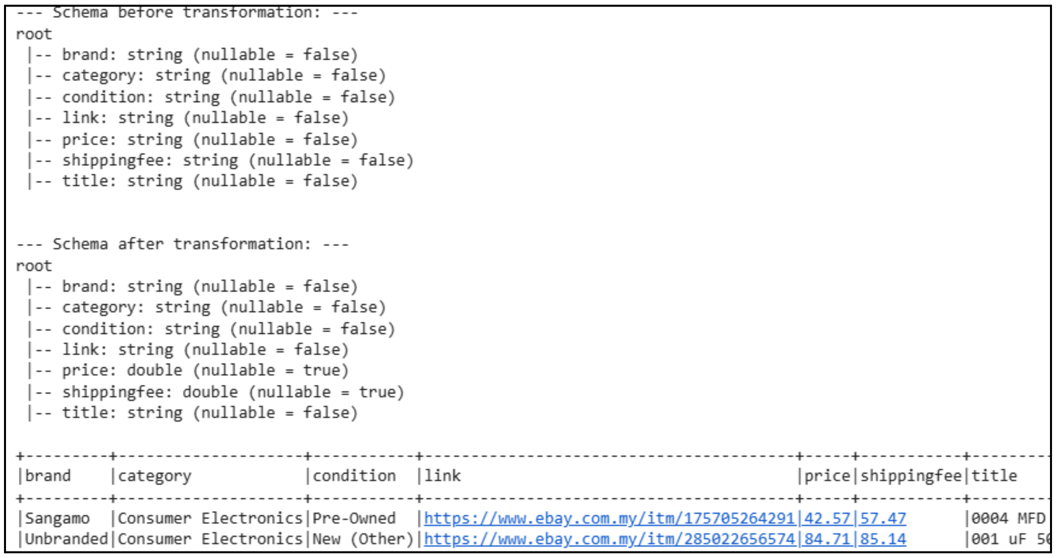
- Basic Cleaning



```
--- Start basic cleaning: ---

--- Schema of spark_df: ---
root
 |-- brand: string (nullable = true)
 |-- category: string (nullable = true)
 |-- condition: string (nullable = true)
 |-- link: string (nullable = true)
 |-- price: string (nullable = true)
 |-- shippingfee: string (nullable = true)
 |-- title: string (nullable = true)

Total rows before removing duplicates: 126552

Total rows after removing duplicates: 113651

Total rows after removing rows with 3 or more nulls : 107270

--- Performance Metrics ---
Total time: 26.96 seconds
CPU usage: 1.00 %
Memory used: 0.00 MB
Throughput: 3978.16 records/sec
```

**Figure 10.1.4: Shows the final output indicating the successful completion of Basic Cleaning**

- Data Transformation



```
--- Schema before transformation: ---
root
 |-- brand: string (nullable = false)
 |-- category: string (nullable = false)
 |-- condition: string (nullable = false)
 |-- link: string (nullable = false)
 |-- price: string (nullable = false)
 |-- shippingfee: string (nullable = false)
 |-- title: string (nullable = false)


--- Schema after transformation: ---
root
 |-- brand: string (nullable = false)
 |-- category: string (nullable = false)
 |-- condition: string (nullable = false)
 |-- link: string (nullable = false)
 |-- price: double (nullable = true)
 |-- shippingfee: double (nullable = true)
 |-- title: string (nullable = false)

+---------+--------------------+-----------+----------------------------------------+-----+-----------+--------
|brand    |category            |condition  |link                                    |price|shippingfee|title
+---------+--------------------+-----------+----------------------------------------+-----+-----------+--------
|Sangamo  |Consumer Electronics|Pre-Owned  |https://www.ebay.com.my/itm/175705264291|42.57|57.47      |0004 MFD
|Unbranded|Consumer Electronics|New (Other)|https://www.ebay.com.my/itm/285022656574|84.71|85.14      |001 uF 5
```

**Figure 10.1.5: Shows the final output indicating the successful completion of Data Transformation**

## 10.2 Links to full code repo or dataset

- Github Repositories
- Project Proposal
- System Architecture
- Crawler Source Code
- Data Processing and Optimization technique Source Code