UNIVERSITI TEKNOLOGI MALAYSIA

SECP3133 HIGH PERFORMANCE DATA PROCESSING

SEMESTER 2 2024/2025

PROJECT 1

Optimizing High-Performance Data Processing for Web Crawling on PetFinder.my

SECTION	02
GROUP	GROUP 5
GROUP MEMBERS	1. NEO ZHENG WENG (A22EC0093) 2. NG SHU YU (A22EC0228) 3. MUHAMMAD SAFWAN BIN MOHD AZMI (A22EC0221) 4. NAVASARATHY A/L S.GANESWARAN (A22EC0091)
LECTURER	DR. ARYATI BINTI BAKRI
SUBMISSION DATE	25 MAY 2025

Table of Contents

1.0 Introduction

This section outlines the project's background and objectives to build a high performance pipeline to scrape and process 100,000+ records from a specific website and introduces the target data fields extracted.

1.1 Background of the Project

This project focuses on scraping and processing large-scale pet adoption data from PetFinder.my, one of Malaysia's largest pet adoption platforms. The objective is to scrape at least 100,000 pet listings, capturing details such as species, breed, age, location, and adoption fees. The data is structured, making it easier to process but still requires significant cleanup and transformation. To collect this data, a range of web scraping tools such as Scrapy, BeautifulSoup (BS4), Selenium, and lxml are employed, each chosen for their ability to efficiently handle different types of web pages, from static to dynamic content.

Once scraped, the data is stored in a MongoDB, NoSQL database, and then processed using Pandas for initial data manipulation. To optimize the data processing performance, the project compares various techniques, including PySpark, Dask, Modin, and asynchronous processing. These optimization methods are evaluated by analyzing key metrics like execution time, resource usage, and throughput, to determine which technique offers the best performance for large-scale data processing. The comparison of pre-optimization and post-optimization performance highlights the benefits of these advanced techniques in improving efficiency and scalability, ensuring that large datasets can be processed in a timely and resource-efficient manner.

1.2 Objectives

The objectives for this project are:

- To scrape a minimum of 100,000 pet records from https://www.petfinder.my/ using various web scraping tools.
- To clean and transform the extracted datasets using different Python libraries.
- To evaluate and compare the performance of data processing before and after optimization based on the execution time, memory usage, CPU utilization and throughput.

1.3 Target Website and Data to be Extracted

PetFinder.my, a well-known online resource for pet adoption and animal care in Malaysia, is the project's target website. Users can browse and adopt a variety of creatures, including birds, dogs, cats, and tiny animals. This project will concentrate on PetFinder.my's "Adoptable Pets" area, which features a list of pets up for adoption around Malaysia.

At least 100,000 pet adoption records should be extracted from the website. The following crucial details (see Appendix A, Table 1) will be included in the data that needs to be extracted.

Table 1 outlines the specific data fields targeted for extraction from PetFinder.my. These fields represent the essential attributes associated with each pet listing on the platform. The data fields were carefully selected to ensure a comprehensive analysis of the pet adoption ecosystem in Malaysia. Each field captures relevant information, such as the pet's identity (e.g., Pet ID, Name), biological characteristics (e.g., Type, Species, Body, Color), health status (e.g., Vaccinated, Dewormed, Spayed, Condition), and adoption logistics (e.g., Location, Amount, Status, Posted date). Details about the uploader (Uploader Type and Uploader Name) are also included to differentiate between individuals and organizations involved in pet adoptions. Collectively, these data fields enable detailed trend analysis and help identify factors influencing pet adoption. This structured approach ensures that the extracted data can support meaningful insights and data-driven decision-making.

This data is collected using web crawling techniques, processed, and analyzed to learn more about Malaysian pet adoption trends. The crawling procedure will follow moral guidelines, guaranteeing adherence to the terms of service of the website.

2.0 System Design and Architecture

This section outlines the system architecture of the end to end ETL data pipeline to extract raw data from the website via web scraping, transform it into a clean dataset, and load the clean dataset back to the database for further analysis. The tools and frameworks implemented are further explained and the team role assignments are also tabulated.

2.1 Description of Architecture

Figure 2.1 shows the architecture for web scraping pet details from PetFinder.my. The web scraping architecture begins with the retrieval of a list of URLs of the type https://www.petfinder.my/pets/{pet_id}/, where every pet_id represents the page of a single pet. The URLs are then provided to the web scraper, which employs libraries like BeautifulSoup, Selenium, Scrapy, or lxml to interact with the web pages and extract the desired data. The details of these libraries will be further explained in the next section, Tools and Frameworks Used. An HTTP(s) request is sent to the petfinder.my website, and the corresponding HTTPS response containing the raw HTML of the pet's page is received. The raw HTML is parsed using the chosen parsing library, and relevant data is extracted. The parsed data is written to multiple CSV files progressively. Next, all structured but uncleaned data is combined and stored in a MongoDB collection named “raw_pets”. This raw data is supplied to data processing tools such as Pandas, FastAPI, PySpark, Dask, and Modin to clean and transform the data with optimization which increases the performance. Finally, the cleaned and refined data is saved into another MongoDB collection named “cleaned_pets”. This architecture ensures scalable scraping, organized data processing, and structured storage for future analysis.

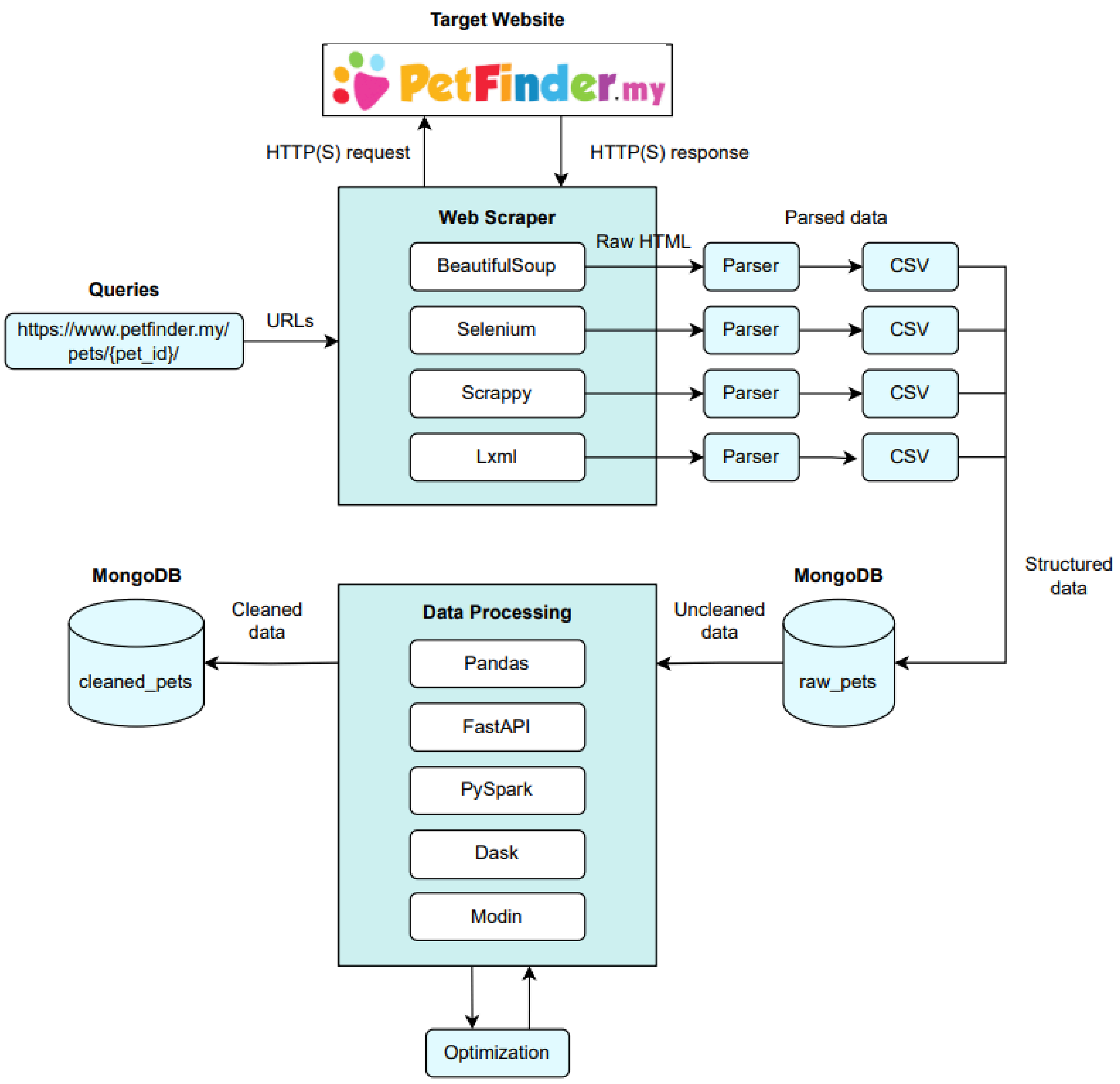


Figure 2.1. Web Scraping Architecture for PetFinder.my

2.2 Tools and Frameworks Used

The project utilizes a myriad of tools and frameworks across different categories (see Appendix B, Table 2) to provide efficient handling and processing of data. BeautifulSoup, Selenium, Scrapy, and lxml libraries are used for web scraping to obtain structured data from web pages. Throughout data processing, pandas, FastAPI, PySpark, Modin, and Dask provide scalable and high-performance data manipulation and API integration. MongoDB Atlas serves as the cloud database service for elastic and secure data storage.

2.3 Roles of Team Members

Table 3 outlines the roles of the team members involved in the web scraping project, which show how various web scraping and data processing libraries were assigned and managed throughout the project:

Table 3. List of Roles of Team Members

Roles		**Team Member in Charge**
Category	**Library**	
Web Scraping	BeautifulSoup	Ng Shu Yu
	Selenium	Muhammad Safwan Bin Mohd Azmi
	Scrapy	Neo Zheng Weng
	lxml	Navasarathy A/L S.Ganeswaran
Data Processing	Pandas	Navasarathy A/L S.Ganeswaran
	FastAPI	Ng Shu Yu
	PySpark	Neo Zheng Weng
	Modin	Neo Zheng Weng
	Dask	Muhammad Safwan Bin Mohd Azmi

3.0 Data Collection

This section provides a detailed explanation of the project's web crawling strategies used to retrieve over 100,000 structured pet records from PetFinder.my . It also mentioned the total number of listings extracted and outlines the fields of the raw dataset. Besides, ethical considerations are also considered with respect to the web's robots.txt to ensure responsible web crawling and scraping.

3.1 Crawling Method

Four different types of web scrapers were developed to scrape a minimum of 100,000 pet records from PetFinder Malaysia. Table 4 below summarizes the crawling methods used by the various crawlers: BeautifulSoup, Selenium, Scrapy, and lxml:

Table 4. List of Crawling Methods Used

Aspects	**Description**
Direct URL-based crawling	Pet profile pages were crawled by brute-forcing sequential pet IDs. The pages are accessed by systematically iterating through a sequence of known URL patterns.
Rate-limiting	A delay of 30 seconds is used to limit concurrent requests and not overload the server.
Asynchronous concurrent crawling	aiohttp and asyncio.gather are used to send multiple requests concurrently to speed up scraping.
Error handling	404 errors and request exceptions are caught gracefully, allowing the crawler to skip missing or broken pages without crashing.
User-Agent Spoofing	A custom User-Agent string is used to mimic real browser behavior and reduce the chance of being blocked.
Save data progressively	Data is saved progressively by opening the CSV file in append mode and writing each pet's information right after it is scraped to ensure collected data is not lost even if the scraping process is interrupted.

3.2 Number of Records Collected

A total of 101,730 records of pets were scraped successfully from the Petfinder Malaysia website through Pet ID using the URL (https://www.petfinder.my/pets/{pet_id}/) that consists of 18 features (Pet ID, Name, Type, Species, Profile, Amount, Vaccinated, Dewormed, Spayed, Condition, Body, Color, Location, Posted Date, Price, Uploader Type, Uploader Name, Status).

3.3 Ethical Considerations

Ethical web scraping is practised in this project to not infringe on the website's terms of service and not affect the website's functionality. Hence, a significant consideration is adhering to the robots.txt file that specifies how the web crawlers should access the website. The URL (https://www.petfinder.my/robots.txt) includes the following rules:

```
User-agent: *
Allow: /
Crawl-delay: 30
#Disallow: /wagazine/
```

These rules provide full access to the website with a 30-second delay between each request in order not to flood the server. Next, the /wagazine/ directory is specifically excluded from scraping. Adhering to these rules makes the web scraping ethical as well as respectful of the website infrastructure and in line with the terms of service.

The scraped data is used responsibly for research and academic project purposes, with the assurance that ethical considerations guide both the data collection and its subsequent use.

4.0 Data Processing

This section presents the end to end pipeline from transforming the raw data into a clean dataset and storing it into the Mongodb database for future use. The flow begins with data cleaning, where the missing and duplicate values are handled and text fields are standardized. Next, data structuring describes how the raw data is extracted from the Mongodb database and the cleaned data is organized back into the Mongodb database. Lastly, data transformation and formatting is explained with the creation of new features and data type conversions.

4.1 Cleaning Methods

The data cleaning process was a crucial step in preparing the raw scraped data for downstream transformation and analysis. Each member was assigned one library to process their portion of the data, ensuring that different tools, Pandas (Nava), PySpark (Neo), Dask (Safwan), Modin (Neo), and FastAPI (Shu Yu), were used while maintaining the same logic and output structure. This division allowed for fair performance comparison and effective workload distribution within the group.

Despite using different tools, all members followed a consistent set of cleaning steps to ensure standardization across implementations. The main objective was to convert inconsistent, incomplete, and messy records into a reliable, structured format. The following describes the common cleaning tasks applied across all frameworks:

1. **Handling Missing Values**
    a. Rows with all fields marked as 'N/A' or left completely blank were dropped to avoid noise in the data.
    b. Placeholder strings such as 'N/A' and 'Not Sure' were replaced with NaN (Not a Number), allowing consistent treatment of missing values across libraries.
2. **Removing Duplicates**
    a. Duplicate records were identified based on unique identifiers like Pet ID.

   b. Only the first occurrence of each duplicate was retained to preserve one clean record per listing.

3. **Whitespace and Formatting Cleanup**

   a. Leading and trailing whitespaces in string fields were stripped to ensure clean text values.
   b. Optionally, some fields were also normalized in terms of text casing (e.g., converting 'available' to 'Available') to aid consistency during grouping and filtering operations.

4. **Preparing for Transformation**

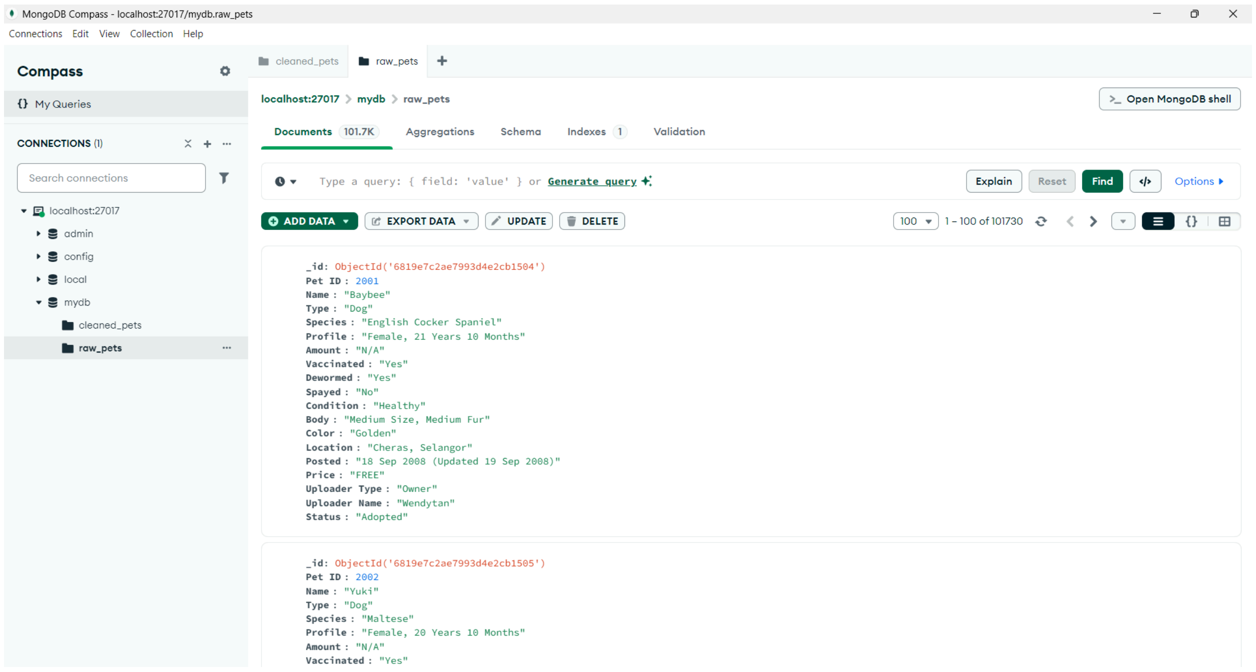
   a. Certain fields were cleaned but left in a raw textual format (e.g., "3 Pets", "RM 100") to allow precise parsing in the transformation phase.
   b. The final cleaned dataset was saved into a new file or database collection to serve as the input for structured transformation and analysis.

4.2 Data Structure

The project employs a hybrid data management approach combining CSV files and MongoDB storage. The workflow begins with web-scraped data being saved in individual CSV files (pets1.csv to pets4.csv) using standardized naming conventions, enabling efficient collaboration among team members. Each contributor was responsible for specific datasets: Shu Yu (pets1.csv), Safwan (pets2.csv), Nava (pets3.csv), and Neo (pets4.csv), all containing pet information scraped from PetFinder. These individual files were subsequently merged into a consolidated master file (pets.csv) that underwent data cleaning and validation. The finalized dataset was then transferred to MongoDB for storage, leveraging its document-oriented architecture for flexible querying and future application development. This two-tiered structure provides both human-readable intermediate files (CSV) and a scalable database solution (MongoDB), ensuring data integrity throughout the processing pipeline.

As part of the data storage process, both the raw and cleaned pet adoption datasets are imported into MongoDB for structured storage and efficient retrieval. The raw data, collected directly from the web scraping phase, is stored in the raw_pets collection, preserving the original state of the dataset. After applying cleaning and standardization procedures, the refined records are saved in the cleaned_pets collection. This dual-storage approach ensures data traceability and supports further processing, analysis, and application development.

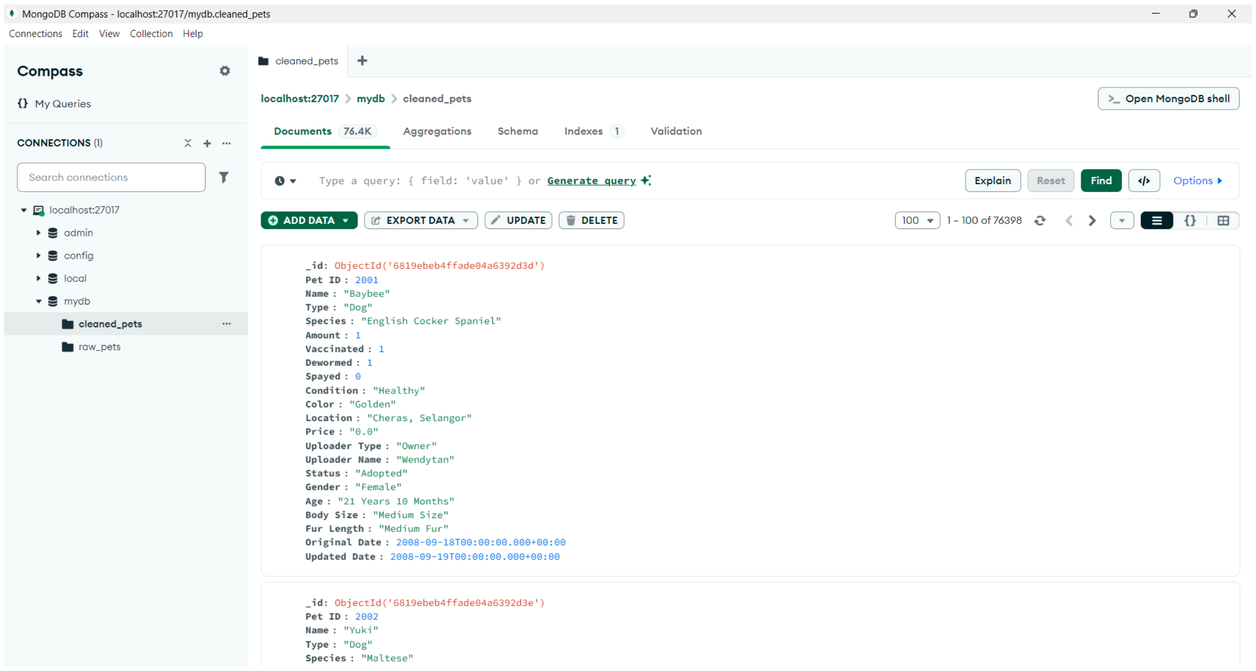
The implemented data processing pipeline follows a structured approach to manage pet adoption data efficiently. First, all raw data from various sources is combined and loaded into MongoDB Atlas as shown in Figure 4.2.1, where it is stored in its original format for preservation. Next, the data undergoes through processing including cleaning to remove inconsistencies, standardization to ensure uniform formatting, and validation to verify accuracy. Once processed, the refined dataset is loaded back into MongoDB Atlas as cleaned data as shown in Figure 4.2.2, making it immediately available for analysis and reporting while maintaining a clear separation from the original raw data. This end-to-end workflow ensures data integrity throughout the transformation process while providing both the raw records for traceability and cleaned data for reliable analytics. The two-tier storage approach in MongoDB Atlas (raw and cleaned collections) creates an efficient, auditable system for ongoing data management needs.



```
_id: ObjectId('6819e7c2ae7993d4e2cb1504')
Pet ID : 2001
Name : "Baybee"
Type : "Dog"
Species : "English Cocker Spaniel"
Profile : "Female, 21 Years 10 Months"
Amount : "N/A"
Vaccinated : "Yes"
Dewormed : "Yes"
Spayed : "No"
Condition : "Healthy"
Body : "Medium Size, Medium Fur"
Color : "Golden"
Location : "Cheras, Selangor"
Posted : "18 Sep 2008 (Updated 19 Sep 2008)"
Price : "FREE"
Uploader Type : "Owner"
Uploader Name : "Wendytan"
Status : "Adopted"
```

```
_id: ObjectId('6819e7c2ae7993d4e2cb1505')
Pet ID : 2002
Name : "Yuki"
Type : "Dog"
Species : "Maltese"
Profile : "Female, 20 Years 10 Months"
Amount : "N/A"
Vaccinated : "Yes"
```

Figure 4.2.1. MongoDB Compass displaying raw_pets collections containing the pet adoption dataset.



```
_id: ObjectId('6819ebeb4ffade04a6392d3d')
Pet ID : 2001
Name : "Baybee"
Type : "Dog"
Species : "English Cocker Spaniel"
Amount : 1
Vaccinated : 1
Dewormed : 1
Spayed : 0
Condition : "Healthy"
Color : "Golden"
Location : "Cheras, Selangor"
Price : "0.0"
Uploader Type : "Owner"
Uploader Name : "Wendytan"
Status : "Adopted"
Gender : "Female"
Age : "21 Years 10 Months"
Body Size : "Medium Size"
Fur Length : "Medium Fur"
Original Date : 2008-09-18T00:00:00.000+00:00
Updated Date : 2008-09-19T00:00:00.000+00:00
```

```
_id: ObjectId('6819ebeb4ffade04a6392d3e')
Pet ID : 2002
Name : "Yuki"
Type : "Dog"
Species : "Maltese"
```

Figure 4.2.2. MongoDB Compass displaying cleaned_pets collections containing the pet adoption dataset.

4.3 Transforming and Formatting

The following outlines the procedures performed during the data transforming and formatting phase:

1. Splitting Composite Columns

Several columns contained compound information that was separated into individual fields:

a. Profile: Split into Gender and Age
b. Body: Split into Body Size and Fur Length
c. Posted: Split into Original Data and Updated Date

2. Standardization of Date Format

All date values under Original Date and Updated Date were standardized to the format DD/MM/YYYY for uniform representation and easier processing.

3. **Standardization of Boolean Values**

   The boolean columns representing pet health status (Vaccinated, Dewormed, Spayed) were normalized as follows:

   a. "Yes" → 1
   b. "No" and missing values (NaN) → 0

4. **Value Mapping and Imputation**

   Specific fields with inconsistent or missing data were cleaned using value mapping and imputation strategies:

   a. Amount:

      i. Entries such as "3 Pets" were converted to 3
      ii. Missing values were imputed with a default value of 1

   b. Price:

      i. "FREE" was standardized to 0
      ii. Prices prefixed with "RM" (e.g., "RM100") were converted to their numeric form (e.g., 100)
      iii. Missing or non-convertible values were replaced with the string "Enquire"

5. **Correction of Data Types**

   To ensure data integrity and compatibility with downstream systems, the following data types were enforced:

   a. Pet ID, Amount, Vaccinated, Dewormed, Spayed: Integer
   b. Original Date, Updated Date: Datetime
   c. All other fields: String

5.0 Optimization Techniques

This section describes the optimization techniques applied to the data processing pipeline, such as asynchronous processing, distributed computing with PySpark and Modin, and parallel processing with Dask to maximize CPU utilization and minimize overall runtime.

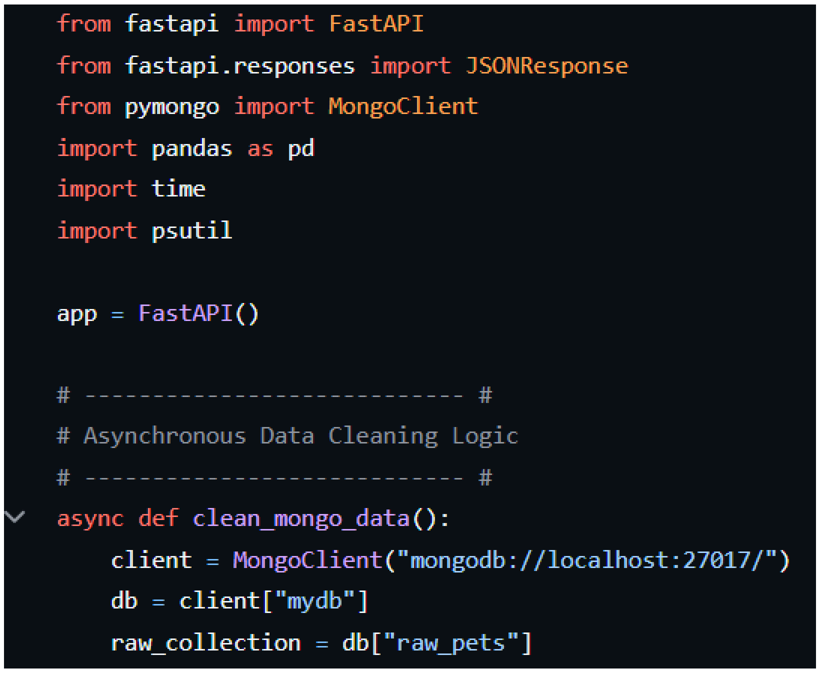
5.1 Methods Used

Table 5 in Appendix C shows the high-performance optimization techniques implemented with various data processing libraries, including FastAPI (Async), PySpark, Dask, and Modin.

5.2 Code Overview or Pseudocode of Techniques Applied

5.2.1 Fast API (Async)

Figure 5.2.1.1 shows the asynchronous data cleaning function. The line ‘async def clean_mongo_data()’ defines an asynchronous function using the async def syntax. Although the operations inside (like MongoDB queries via ‘pymongo’ and pandas data manipulation) are synchronous, defining the function inside ‘async def’ allows FastAPI to execute it non-blockingly when awaited. It makes the function compatible with an asynchronous event loop for scaling and concurrency.



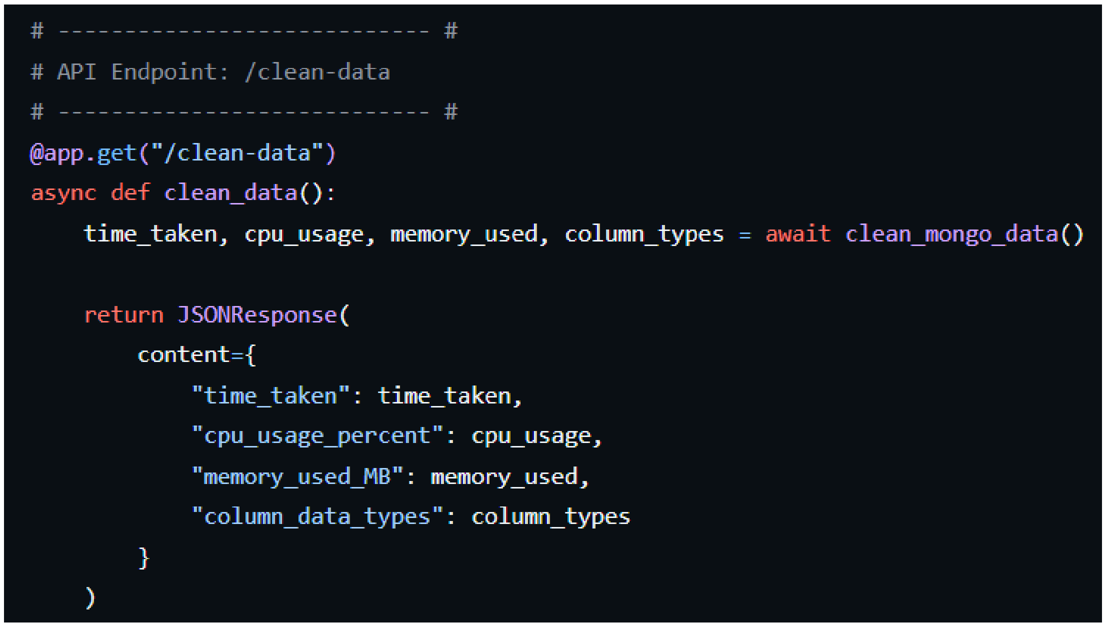
```python
from fastapi import FastAPI
from fastapi.responses import JSONResponse
from pymongo import MongoClient
import pandas as pd
import time
import psutil

app = FastAPI()

# --------------------------- #
# Asynchronous Data Cleaning Logic
# --------------------------- #
async def clean_mongo_data():
    client = MongoClient("mongodb://localhost:27017/")
    db = client["mydb"]
    raw_collection = db["raw_pets"]
```

Figure 5.2.1.1. Asynchronous Data Cleaning Function

Figure 5.2.1.2 shows the definition of asynchronous FastAPI endpoint. This is an asynchronous endpoint handler in FastAPI. Declaring the endpoint with 'async def' tells it that it will run in an asynchronous event loop. This enables the server to process multiple requests concurrently without having to wait for one request to complete before serving another. It keeps the endpoint responsive for long operations. Also, the 'await' keyword suspends the running of the current function until 'clean_mongo_data()' is complete. It is a building block of asynchronous programming, allowing cooperative multitasking—other requests are being processed while this one waits. That prevents blocking the entire thread during I/O-bound operations like database lookup and allows for better performance under heavy traffic.



```
# ---------------------------- #
# API Endpoint: /clean-data
# ---------------------------- #
@app.get("/clean-data")
async def clean_data():
    time_taken, cpu_usage, memory_used, column_types = await clean_mongo_data()

    return JSONResponse(
        content={
            "time_taken": time_taken,
            "cpu_usage_percent": cpu_usage,
            "memory_used_MB": memory_used,
            "column_data_types": column_types
        }
    )
```

Figure 5.2.1.2. Asynchronous FastAPI Endpoint Definition

5.2.2 PySpark

The script coded (see Appendix D) maximizes the processing of a big pet adoption dataset with PySpark. It carries out data cleaning and transformation operations like replacing empty strings with nulls, deleting rows with missing required data, removing leading and trailing whitespace, eliminating duplicates, and splitting certain columns like ‘Profile’ and ‘Body’. It also converts boolean and numeric columns to their respective data types and manipulates date columns to a uniform format.

Performance is then measured through psutil and tracemalloc for memory consumption, CPU, and wall clock. Throughput (number of records per second) is then computed from these metrics and the efficiency of the PySpark data pipeline is determined.

5.2.3 Dask

This Dask pipeline (see Appendix E) efficiently processes a pet adoption dataset by initializing a parallel processing environment, loading and partitioning the data, then performing comprehensive cleaning operations including column splitting, datetime conversion, categorical mapping, and numeric standardization, all optimized through lazy evaluation that builds an execution graph triggered by compute() to maximize performance, with metrics tracking time, memory, CPU usage, and throughput to validate the scalable processing of medium-to-large datasets while maintaining Pandas-like simplicity.

5.2.4 Modin

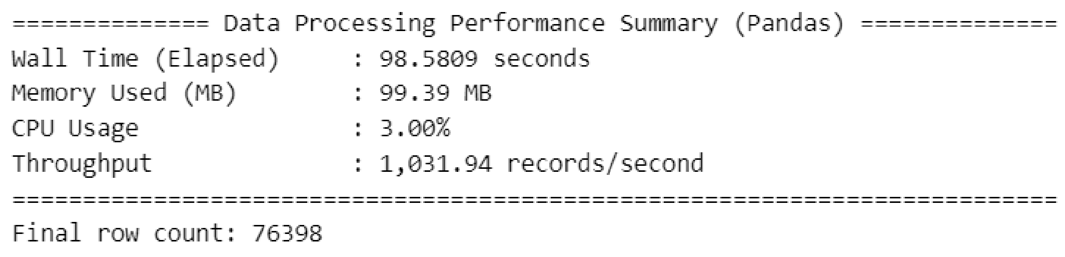
This script shown in Appendix F utilizes Modin, a parallelized implementation of Pandas, to accelerate data processing by using multiple CPU cores to do things in parallel and quicker. It is refactored to process large data efficiently by doing operations like replacing missing values with `NaN`, stripping whitespaces, dropping duplicates, and splitting columns on regular expressions in parallel. Modin is also employed for date parsing, boolean conversion, and price normalization. For situations where Modin cannot perform some functionality (e.g., `to_numeric`), the code automatically falls back to Pandas for seamless processing. The performance is measured in terms of execution time, memory, CPU, and throughput and exhibits remarkable improvement in terms of processing efficiency and scalability over conventional methods.

6.0 Performance Evaluation

This section presents the evaluation of performance before and after optimization of each library or technique, benchmarked on execution time, memory usage, CPU utilization, and throughput with data visualization (chart/graph) to identify the best approach for large-scale data processing.

6.1 Before vs After Optimization

To evaluate the performance improvement, the data processing workflow was initially implemented using Pandas, which acted as the baseline. Pandas is widely used for its simplicity and expressive syntax, but it operates on a single thread, making it less efficient for large-scale datasets. As a result, operations such as data cleaning, type conversion, and transformation were slower, and system resources like CPU and memory were underutilized. Figure 6.1.1 shows the result of Pandas data processing performance:

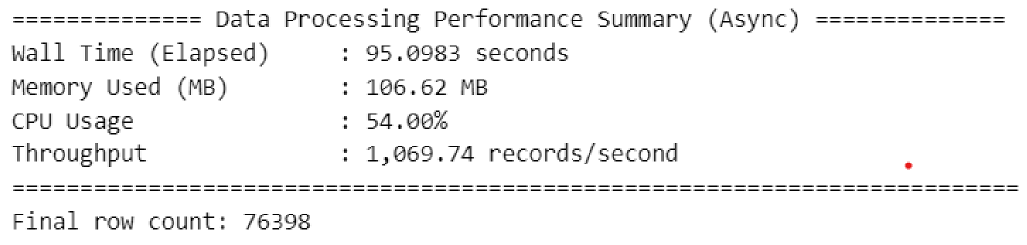


```
============== Data Processing Performance Summary (Pandas) ==============
Wall Time (Elapsed)     : 98.5809 seconds
Memory Used (MB)        : 99.39 MB
CPU Usage               : 3.00%
Throughput              : 1,031.94 records/second
==========================================================================
Final row count: 76398
```

Figure 6.1.1. Result of Pandas Data Processing Performance

To optimize this process, the same logic and dataset were used with the following high-performance frameworks:

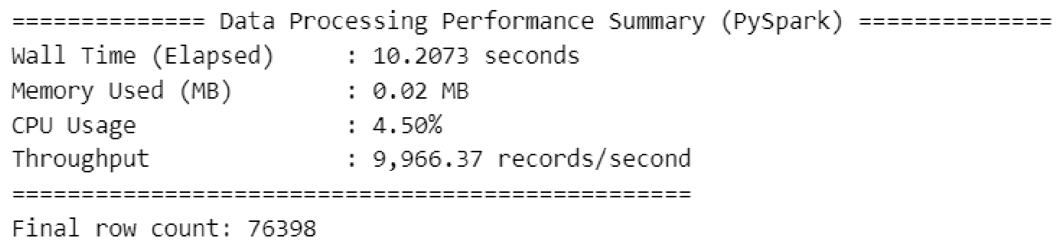
A. **FastAPI (async):** Integrated asynchronous functions to avoid blocking I/O during data retrieval and writing, enabling non-blocking and scalable performance in server-based tasks. Figure 6.1.2 shows the result of FastAPI data processing performance:



```
============== Data Processing Performance Summary (Async) ==============
Wall Time (Elapsed)      : 95.0983 seconds
Memory Used (MB)         : 106.62 MB
CPU Usage                : 54.00%
Throughput               : 1,069.74 records/second
==========================================================================
Final row count: 76398
```

Figure 6.1.2. Result of FastAPI Data Processing Performance

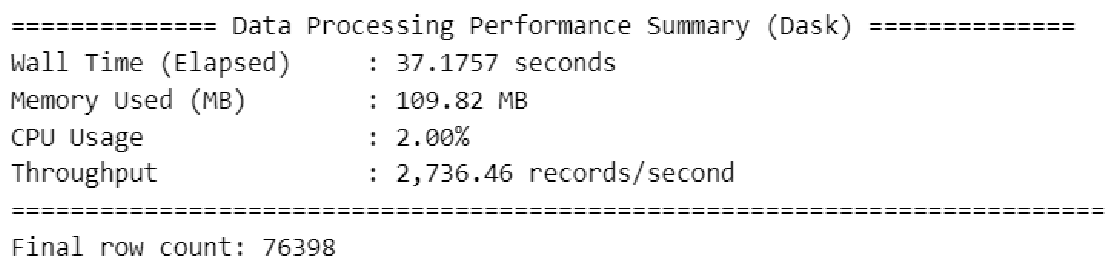
B. **PySpark:** Used distributed processing across multiple cores or clusters. It transformed and cleaned large datasets efficiently with built-in parallelism and fault tolerance. Figure 6.1.3 shows the result of PySpark data processing performance:



```
============== Data Processing Performance Summary (PySpark) ==============
Wall Time (Elapsed)      : 10.2073 seconds
Memory Used (MB)         : 0.02 MB
CPU Usage                : 4.50%
Throughput               : 9,966.37 records/second
==================================================
Final row count: 76398
```

Figure 6.1.3. Result of PySpark Data Processing Performance

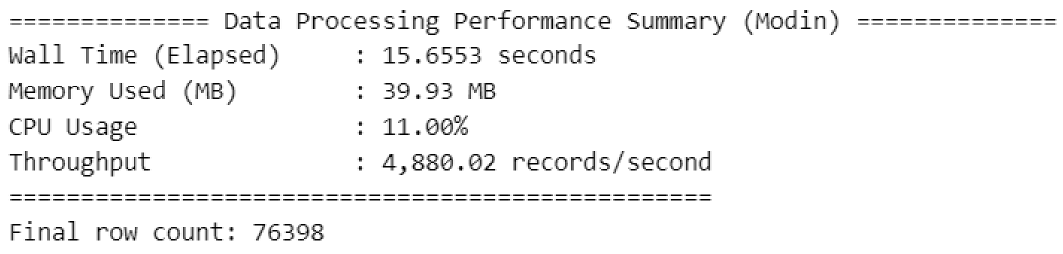
C. **Dask:** Enabled parallel and out-of-core processing by breaking data into chunks. It performed similar operations to Pandas but distributed them across multiple processes. Figure 6.1.4 shows the result of Dask data processing performance:



```
============== Data Processing Performance Summary (Dask) ==============
Wall Time (Elapsed)      : 37.1757 seconds
Memory Used (MB)         : 109.82 MB
CPU Usage                : 2.00%
Throughput               : 2,736.46 records/second
=========================================================================
Final row count: 76398
```

Figure 6.1.4. Result of Dask Data Processing Performance

D. **Modin:** Served as a drop-in replacement for Pandas, it accelerated Pandas operations by utilizing all available CPU cores through parallelization. Figure 6.1.5 shows the result of Modin data processing performance:



```
============= Data Processing Performance Summary (Modin) =============
Wall Time (Elapsed)     : 15.6553 seconds
Memory Used (MB)        : 39.93 MB
CPU Usage               : 11.00%
Throughput              : 4,880.02 records/second
==================================================
Final row count: 76398
```

Figure 6.1.5. Result of Modin Data Processing Performance

The performance improvements were noticeable in terms of time, memory usage, CPU utilization, and throughput. While Pandas struggled with long processing times and low CPU usage, tools like PySpark and Modin processed the same dataset significantly faster.

6.2 Time, Memory, CPU Usage, Throughput

Table 6 shows the performance comparison in data processing for each technique, including Pandas, PySpark, Dask, FastAPI, and Modin. The performance metrics such as processing time, memory capacity, CPU usage, and throughput are used to compare how efficient and scalable each tool is. The additional analysis and graphical representation of these results are shown in the next section, Charts and Graphs.

Table 6. Comparison of Performance between Data Processing Techniques

Metric	Pandas	PySpark	Dask	FastAPI	Modin
Time (s)	98.58	10.21	37.1	95.10	15.66
Memory (MB)	99.39	0.02	109.82	106.2	39.93
CPU Usage (%)	3.00	4.50	2.00	54.00	11.00
Throughput (No of Records/s)	1031.94	9966.37	2736.46	1069.74	4880.02

6.3 Charts and Graphs

Our comprehensive evaluation of five data processing tools reveals significant performance differences, as illustrated in Figure 6.3. Pandas, the popular Python data analysis library, demonstrated limitations in large-scale processing despite its user-friendly interface. Benchmark results showed Pandas processed 1,031.94 records per second while consuming 99.39 MB of memory, completing tasks in 98.58 seconds with only 3% CPU utilization due to its single-threaded architecture.

In contrast, PySpark exhibited remarkable efficiency as Apache Spark's Python implementation. The distributed computing framework achieved an impressive 9,966.37 records per second throughput while using minimal memory (0.02 MB) and completing operations in just 10.21 seconds with 4.5% CPU usage. This makes PySpark particularly suitable for big data applications requiring high performance.

Dask provided intermediate results, processing 2,736.46 records per second with 109.82 MB memory consumption and finishing in 37.18 seconds. Its 2% CPU utilization suggests potential for further optimization in its parallel computing capabilities. Asynchronous processing methods showed limited effectiveness for data-intensive tasks, achieving 1,069.74 records per second with 106.62 MB memory usage and high CPU utilization (54%), requiring 95.10 seconds for completion.

Modin emerged as a compelling alternative, delivering Pandas-like usability with enhanced performance. It processed 4,880.02 records per second using 39.93 MB memory and completed tasks in 15.66 seconds with moderate CPU usage (11%). These results position Modin as an excellent choice for users needing improved scalability without sacrificing familiar Pandas syntax.

In conclusion, our performance analysis shows that PySpark is the best package for handling large datasets, utilizing distributed computing to achieve great throughput and short execution times. With notable performance enhancements and a recognizable Pandas-like UI, Modin is a formidable competitor that is perfect for those looking for increased scalability without compromising usability. For parallel processing on a single machine or small cluster, Dask is still a potent Python-native solution that strikes a fair compromise between flexibility and performance. Async may still be appropriate for some use cases involving asynchronous I/O or latency-bound processes even if it was not intended for CPU-bound tasks. Despite having the lowest performance in this comparison, Pandas' user-friendliness and well-established ecosystem make it a dependable tool for smaller datasets.

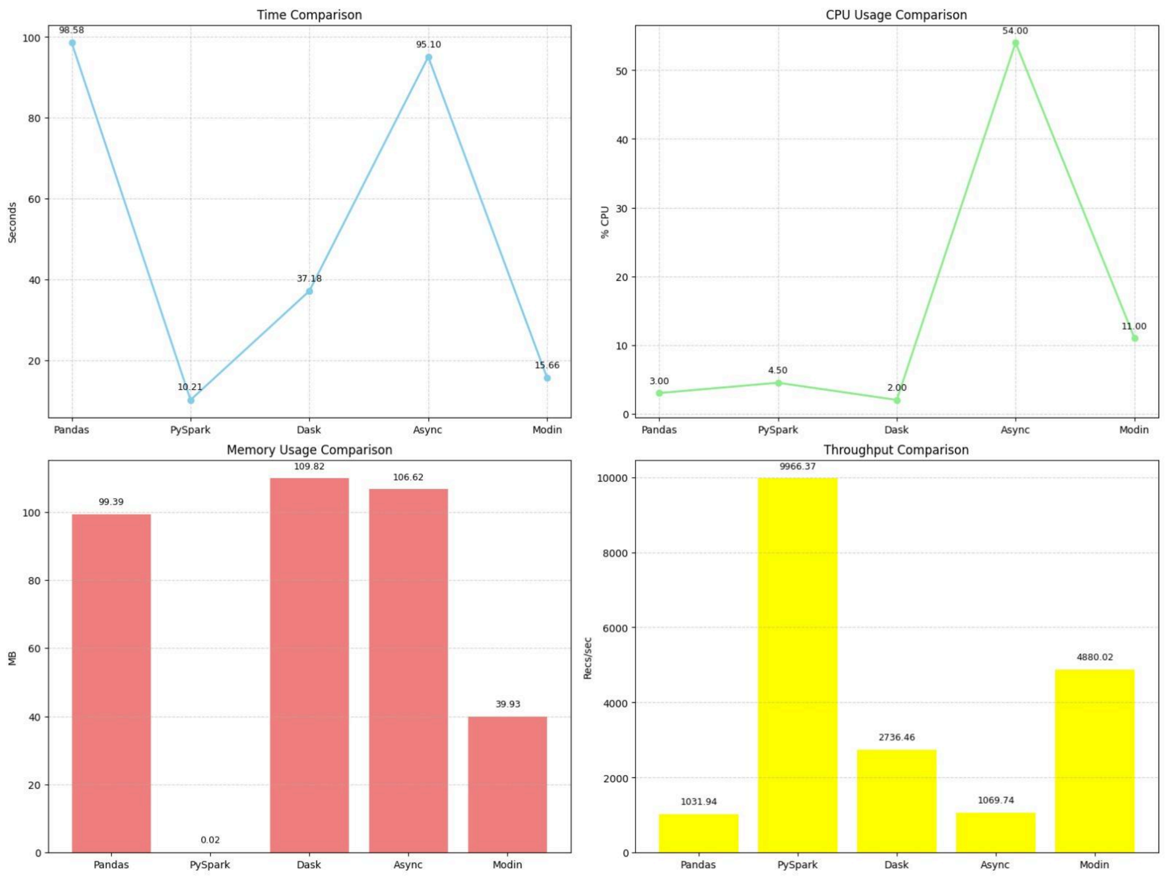


Figure 6.3. Charts and Graphs of Data Processing Performance

7.0 Challenges and Limitations

This section outlines the challenges encountered all along the project and how it was solved accordingly. Next, the limitations of the project solution are also discussed to reflect the project scope and boundaries.

7.1 What Didn’t Go as Planned

The project originally intended to scrape information from Mudah Malaysia, but since the robots.txt file disallowed scraping, we had to change targets. With the permission of Dr. Aryati, we changed the target website to PetFinder Malaysia.

Besides, in data processing, our group came across an issue where we crawled over 100,000 pet posts, yet after cleaning the data, our final dataset included less than 100,000 rows. It is because among the pet postings, some got removed by uploaders after data crawling. With confirmation from Dr. Aryati, we proceeded with the work, realizing the cleaned dataset continued to have worthwhile information to conduct high-performance data processing on it.

Also, we faced a problem while loading data from MongoDB to PySpark because of a Java version compatibility problem. Therefore, we have omitted the data loading steps (both to and from MongoDB) from the performance consideration. We have considered the performance evaluation only on the data processing steps, from the initial step of cleaning and transformation to the last processing step, to give a fair evaluation of the processing performance.

7.2 Limitations of Solution

Despite the successful implementation of scraping and data processing pipelines, several limitations were encountered. Firstly, dynamic or JavaScript-rendered content on PetFinder.my posed challenges for certain scraping tools like BeautifulSoup and lxml, requiring fallback to tools like Selenium in specific cases. Secondly, IP blocking and rate limits occasionally interrupted the crawling process, especially during extended scraping sessions, despite implementing delays and user-agent spoofing.

On the data side, some fields such as price and uploader information were inconsistently structured, leading to parsing difficulties and occasional missing values. While efforts were made to clean and standardize the data, complete accuracy could not be guaranteed due to inconsistencies in how information was presented across listings.

Lastly, although optimization techniques were applied through tools like FastAPI, PySpark, and Dask, limited hardware sometimes affected performance, especially during large read/write operations. These limitations highlight the trade-offs between cost, speed, and accuracy in real-world high-volume data processing.

8.0 Conclusion and Future Work

This section concludes the findings of the project and discusses the future work in order to improve or overcome the limitations of the project.

8.1 Summary of Findings

In order to collect and examine more than 100,000 pet adoption records from PetFinder.my, the team successfully created and refined a high-performance data processing pipeline. The crawler was created to properly gather data through sequential Pet ID URLs using tools like BeautifulSoup, Selenium, Scrapy, and lxml, while adhering to the site's robots.txt directives and a 30-second crawl delay. Several performance-tested Python modules, such as Pandas, FastAPI, PySpark, Dask, Async, and Modin, were used to clean and modify the collected data. Because of its distributed architecture, PySpark was able to process about 10,000 records per second with the least amount of memory utilization among these. By parallelizing Pandas operations across several cores, Modin also produced impressive results, delivering high throughput with low memory usage. This makes it the perfect choice for customers who are accustomed to the Pandas interface but require improved performance. With strong parallelism and decent scalability, Dask offered a dependable compromise. Async approaches worked rather well in I/O-bound situations but were less appropriate for CPU-bound workloads. For asynchronous operations, FastAPI enhances responsiveness. Despite being popular and simple to use, pandas has trouble effectively managing big datasets. In spite of obstacles including inconsistent data structures, changing content, and compatibility problems, the team consistently upheld ethical scraping procedures and preserved the quality of the data. For large-scale data processing jobs, these results highlight the value and efficacy of distributed and parallel computing tools like PySpark, Modin, and Dask.

8.2 What Could be Improved

To further improve this project, two significant improvements are going to be made. Increasing the range of data sources by web scraping sites with richer and more heterogeneous information will yield more comprehensive information that can lead to improved findings and insights in pet adoption trends. Second, technical limitations such as IP blocking, rate capping, and

JavaScript-generated content can be overcome using more advanced scraping components and proxy management to ensure smoother and more consistent data collection. Finally, overcoming compatibility issues, such as Java version incompatibility with MongoDB and PySpark, by containerizing the data pipeline with Docker, ensuring an isolated environment, would allow for end-to-end performance analysis across the steps of data loading and data storage.

9.0 References

Apache Spark. (n.d.). *PySpark overview — PySpark 3.5.5 documentation*. https://spark.apache.org/docs/latest/api/python/index.html

Dask. (n.d.). *Dask | Scale the Python tools you love*. https://www.dask.org/

GeeksforGeeks. (2025, April 28). *asyncio in Python*. https://www.geeksforgeeks.org/asyncio-in-python/

Modin. (n.d.). *Scale your pandas workflow by changing a single line of code — Modin 0.32.0+0.g3e951a6.dirty documentation*. https://modin.readthedocs.io/en/stable/

ProjectPro. (2024, October 28). *7 Python libraries for web scraping to master data extraction*. https://www.projectpro.io/article/python-libraries-for-web-scraping/625#mcetoc_1gb5hj7o81a

10.0 Appendices

10.1 Appendix A

Table 1. List of Data Fields to be Extracted

No	Data Field	Description
1	Pet ID	A unique identifier automatically assigned to each pet listing for tracking purposes.
2	Name	The given name of the pet, often chosen by the current owner or shelter.
3	Type	The general classification of the pet, such as dog, cat, rabbit, or bird.
4	Species	The specific breed or species of the pet, e.g., Golden Retriever, Persian Cat.
5	Profile	A short written summary describing the pet’s personality, background, or story.
6	Amount	The stated adoption fee or donation amount,if required by the uploader
7	Vaccinated	Indicates whether the pet has received necessary vaccinations (Yes/No)
8	Dewormed	Indicates whether the pet has been treated for worms and parasites (Yes/No).
9	Spayed	States if the pet has been neutered/spayed to prevent breeding (Yes/No)
10	Condition	Notes any special health conditions or physical impairments the pet may have
11	Body	Describes the pet’s physical attributes such as size (small, medium, large).
12	Color	The dominant color(s) of the pet’s fur, feathers, or skin
13	Location	The geographic location (city and state) of the pet's current residence.

14	Posted	The date when the pet listing was first made available on the platform.
15	Price	Additional costs beyond adoption (e.g., medical, transport), if applicable.
16	Uploader Type	Identifies the type of user listing the pet — e.g., individual, shelter, or rescue.
17	Uploader Name	The name of the person, shelter, or organization responsible for the listing.
18	Status	The current adoption status of the pet (e.g., available, adopted, pending).

10.2 Appendix B

Table 2. List of Tools and Frameworks Used

Category	Tools	Description
Web Scraping	BeautifulSoup	BeautifulSoup is a Python library used to parse HTML and extract data from web pages. It is used to parse the HTML response from PetFinder and extract pet details like names, adoption fees, and other information to be saved to a CSV file for further processing.
	Selenium	Selenium is a browser automation tool used for scraping data from websites that load content dynamically using JavaScript. Unlike BeautifulSoup or Scrapy (which work mainly with static HTML), Selenium can interact with elements like buttons, forms, and dropdowns — just like a human user would
	Scrapy	Scrapy is an open-source Python framework for efficient web scraping. It handles crawling, parsing, and storing data, using asynchronous requests for fast, large-scale scraping. It's known for its speed, flexibility, and ease of use.
	lxml	lxml is a high-performance library used to parse and navigate HTML and XML documents. In this project, it was used to build a lightweight and efficient web crawler that scrapes structured data directly from PetFinder.my's static HTML pages. XPath expressions were used to extract pet information which were then stored in CSV files for further processing.
Data Processing	Pandas	Pandas is a Python library used for data manipulation and analysis. It was used to combine multiple CSV files, remove duplicates, clean and standardize fields like price, status, and body type, and prepare the dataset for further analysis. It also enabled efficient handling of missing values and type conversions during the data cleaning phase.
	FastAPI	FastAPI is a modern, fast, web framework for building APIs with Python. It is used to create an API endpoint that starts an asynchronous pipeline of processing data,

		where pet data is read and cleaned to be exported to a new database and summary statistics are returned as a JSON response.
	PySpark	PySpark is the Python API for Apache Spark, enabling distributed data processing across multiple nodes. It’s used for handling big data, offering scalable and fast data manipulation, analytics, and machine learning.
	Modin	Modin is a high-performance library that accelerates Pandas operations by parallelizing tasks across multiple CPU cores or machines. It allows users to scale their Pandas workflows without modifying existing code, making it ideal for faster data processing on large datasets.
	Dask	Dask is a Python library that enables parallel and distributed computing. It can handle larger-than-memory datasets by breaking them into smaller chunks and processing them in parallel. Similar to Pandas but scalable across CPUs and clusters.
Database Storage	MongoDB Atlas	MongoDB Atlas is a cloud-based NoSQL database that stores and processes high volumes of data in a document-oriented format. It is commonly used in new mobile and web applications because of its scalability, high performance, and ease of integration with both programming languages and cloud platforms.

10.3 Appendix C

Table 5. List of Optimization Techniques Used with Each Library

Libraries	Optimization Techniques	Explanation
FastAPI (Async)	Asynchronous processing	**Async** uses asynchronous programming to perform non-blocking I/O operations, enhancing the system’s responsiveness. This allows other tasks to run while waiting for I/O operations to complete, making the system more scalable when handling multiple requests.
PySpark	Distributed processing	**PySpark** enables processing across multiple cores or machines, speeding up operations like cleaning, aggregating, and normalizing data. This greatly improves performance and scalability, reducing processing time compared to single-threaded methods like Pandas.
Dask	Distributed / Parallel processing	**Dask** breaks large datasets into smaller chunks and processes them in parallel across multiple CPU cores, efficiently handling large-scale data that exceeds available memory. It uses lazy evaluation and task scheduling to optimize resources and processing time.
Modin	Parallel processing	**Modin** accelerates **Pandas** operations by automatically distributing tasks across multiple cores or workers. It offers a seamless scaling solution for large datasets, providing a faster alternative to Pandas without requiring major changes to the existing code.

10.4 Appendix D

PySpark Data Processing

```
import os
import time
import psutil
import tracemalloc
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, trim, regexp_extract, 
regexp_replace, to_date, lit, upper
from pyspark.sql.types import IntegerType, StringType

# Initialize Spark session
spark = SparkSession.builder \
    .appName("PetFinderMy") \
    .getOrCreate()

# Convert pandas DataFrame to PySpark DataFrame
df_spark = spark.createDataFrame(df)

# ========== Performance Tracking ==========
process = psutil.Process(os.getpid())
tracemalloc.start()
start_time = time.perf_counter()
mem_start = process.memory_info().rss / (1024 * 1024)
initial_count = df_spark.count()

# ========== Data Processing ==========
# 1. Replace empty values with None (null)
str_cols = [field.name for field in df_spark.schema.fields if field.dataType == 
StringType()]
for col_name in str_cols:
    df_spark = df_spark.withColumn(
        col_name,
          when(trim(col(col_name)) == "NaN", None)   # Convert empty strings to 
null
        .otherwise(col(col_name))
    )

# 2. Drop rows where all columns (except 'Pet ID') are NULL
columns_to_check = [c for c in df_spark.columns if c != 'Pet ID']
df_spark = df_spark.dropna(subset=columns_to_check, how='all')

# 3. Strip whitespace from string columns
str_cols = [field.name for field in df_spark.schema.fields if field.dataType == 
StringType()]
for col_name in str_cols:
    df_spark = df_spark.withColumn(col_name, trim(col(col_name)))
```

```
# 4. Remove duplicates
df_spark = df_spark.dropDuplicates()

# 5. Process 'Profile' column: Split into 'Gender' and 'Age'
if 'Profile' in df_spark.columns:
    df_spark = df_spark.withColumn(
        "Gender",
        regexp_extract(col("Profile"), r"^([^,]+)", 1)
    ).withColumn(
        "Age",
        regexp_extract(col("Profile"), r",\s*(.+)$", 1)
    ).drop("Profile")

# 6. Process 'Body' column: Split into 'Body Size' and 'Fur Length'
if 'Body' in df_spark.columns:
    df_spark = df_spark.withColumn(
        "Body Size",
        regexp_extract(col("Body"), r"^([^,]+)", 1)
    ).withColumn(
        "Fur Length",
        regexp_extract(col("Body"), r",\s*(.+)$", 1)
    ).drop("Body")

# 7. Process 'Posted' column: Extract dates
if 'Posted' in df_spark.columns:
    df_spark = df_spark.withColumn(
        "Original",
        regexp_extract(col("Posted"), r"^([^(]+)", 1)
    ).withColumn(
        "Updated",
        regexp_extract(col("Posted"), r"Updated\s([^)]+)", 1)
    )

    # Clean and parse dates
    df_spark = df_spark.withColumn(
        "Original Date",
        to_date(trim(col("Original")), "dd MMM yyyy")
    ).withColumn(
        "Original Date",
        when(col("Original Date").isNull(),
             to_date(trim(col("Original")), "dd-MMM-yy"))
        .otherwise(col("Original Date"))
    ).withColumn(
        "Updated Date",
        to_date(trim(col("Updated")), "dd MMM yyyy")
    ).withColumn(
        "Updated Date",
        when(col("Updated Date").isNull(),
            to_date(trim(col("Updated")), "dd-MMM-yy"))
        .otherwise(col("Updated Date"))
    )

    # Fill Updated Date with Original Date if null
    df_spark = df_spark.withColumn(
        "Updated Date",
                            when(col("Updated   Date").isNull(),   col("Original
Date")).otherwise(col("Updated Date"))
    ).drop("Posted", "Original", "Updated")

# 8. Process boolean columns
bool_cols = ['Vaccinated', 'Dewormed', 'Spayed']
for col_name in bool_cols:
    if col_name in df_spark.columns:
        df_spark = df_spark.withColumn(
            col_name,
            when(col(col_name) == 'Yes', 1).otherwise(0).cast(IntegerType())
        )

# 9. Process 'Amount' column
if 'Amount' in df_spark.columns:
    df_spark = df_spark.withColumn(
        "Amount",
        regexp_extract(col("Amount").cast("string"), r"(\d+)", 1)
        .cast(IntegerType())
    )
    df_spark = df_spark.fillna(1, subset=["Amount"])

# 10. Process 'Price' column
if 'Price' in df_spark.columns:
    df_spark = df_spark.withColumn(
        "Price",
        trim(upper(col("Price").cast("string")))
    ).withColumn(
        "Price",
        when(col("Price") == "FREE", "0")
        .otherwise(regexp_replace(col("Price"), "^RM", ""))
    )

    # Try to convert to numeric, keep as string if fails
    df_spark = df_spark.withColumn(
        "Price",
        when(col("Price").rlike("^\d+$"), col("Price").cast("double"))
        .otherwise(lit("Enquire"))
    )

# ========== Performance Tracking ==========
end_time = time.perf_counter()
```

```python
mem_end = process.memory_info().rss / (1024 * 1024)
cpu_usage = psutil.cpu_percent(interval=1)

elapsed_time = end_time - start_time
mem_used = mem_end - mem_start
throughput = initial_count / elapsed_time

# ========== Metrics ==========
print("\n==============   Data   Processing   Performance   Summary   (PySpark)
==============")
print(f"Wall Time (Elapsed)     : {elapsed_time:.4f} seconds")
print(f"Memory Used (MB)        : {mem_used:.2f} MB")
print(f"CPU Usage               : {cpu_usage:.2f}%")
print(f"Throughput              : {throughput:,.2f} records/second")
print("=================================================")

print(f"Final row count: {df_spark.count()}")
# PySpark: Show top 3 rows
df_spark.show(3)
spark.stop()
```

10.5 Appendix E

Dask Data Processing

```
import dask.dataframe as dd
import pandas as pd
import numpy as np
import time
import psutil
import os

# Convert existing pandas DataFrame to Dask
df_dask = dd.from_pandas(df, npartitions=4)

# ========== Performance Tracking ==========
process_dask = psutil.Process(os.getpid())
tracemalloc.start()
start_time_dask = time.perf_counter()
mem_start_dask = process_dask.memory_info().rss / (1024 * 1024)
initial_count_dask = len(df_dask)

# ========== Data Processing ==========

df_dask = df_dask.replace(["", "NaN", "N/A", "Not Sure"], np.nan)
df_dask = df_dask.dropna(subset=[col for col in df_dask.columns if col != 'Pet
ID'], how='all')
str_cols = df_dask.select_dtypes(include='object').columns
df_dask[str_cols]  = df_dask[str_cols].map(lambda  x:  str(x).strip(),  meta=('x',
'str'))
df_dask = df_dask.drop_duplicates()

if 'Profile' in df_dask.columns:
             df_dask['Gender']   =   df_dask['Profile'].str.extract(r'^([^,]+)',
expand=False)
    df_dask['Age'] = df_dask['Profile'].str.extract(r',\s*(.+)', expand=False)
    df_dask = df_dask.drop('Profile', axis=1)

if 'Body' in df_dask.columns:
            df_dask['Body   Size']   =   df_dask['Body'].str.extract(r'^([^,]+)',
expand=False)
           df_dask['Fur   Length']   =   df_dask['Body'].str.extract(r',\s*(.+)',
expand=False)
    df_dask = df_dask.drop('Body', axis=1)

if 'Posted' in df_dask.columns:
            df_dask['Original']   =   df_dask['Posted'].str.extract(r'^([^(]+)',
expand=False)
       df_dask['Updated']  =  df_dask['Posted'].str.extract(r'Updated\s([^)]+)',
expand=False)
```

```
            df_dask['Original   Date']   =   dd.to_datetime(df_dask['Original'],
errors='coerce')
             df_dask['Updated   Date']   =   dd.to_datetime(df_dask['Updated'],
errors='coerce')
     df_dask['Updated Date'] = df_dask['Updated Date'].fillna(df_dask['Original
Date'])
    df_dask = df_dask.drop(['Posted', 'Original', 'Updated'], axis=1)

for col in ['Vaccinated', 'Dewormed', 'Spayed']:
    if col in df_dask.columns:
          df_dask[col] = df_dask[col].map(lambda x: 1 if str(x).strip().lower()
== 'yes' else 0, meta=(col, 'int64'))

if 'Amount' in df_dask.columns:
        df_dask['Amount']  =  df_dask['Amount'].map(lambda  x:  pd.to_numeric(x,
errors='coerce'), meta=('Amount', 'float64'))
    df_dask['Amount'] = df_dask['Amount'].fillna(1).astype('int64')

if 'Price' in df_dask.columns:
    def map_price(x):
        x = str(x).strip().upper()
        if x == 'FREE':
            return 0
        elif x.startswith('RM'):
            try:
                return int(x.replace('RM', '').strip())
            except:
                return np.nan
        return np.nan
          df_dask['Price']   =   df_dask['Price'].map(map_price,   meta=('Price',
'float64'))
    df_dask['Price'] = df_dask['Price'].fillna('Enquire')

# Trigger Dask computation
df_cleaned_dask = df_dask.compute(scheduler="threads") # Multithreads

# ========== Performance Tracking ==========
end_time_dask = time.perf_counter()
mem_end_dask = process_dask.memory_info().rss / (1024 * 1024)
cpu_usage_dask = psutil.cpu_percent(interval=1)

elapsed_time_dask = end_time_dask - start_time_dask
mem_used_dask = mem_end_dask - mem_start_dask
throughput_dask = initial_count_dask / elapsed_time_dask if elapsed_time_dask >
0 else 0

# ========== Metrics ==========
print("\n==============    Data    Processing    Performance    Summary    (Dask)
==============")
```

```
print(f"Wall Time (Elapsed)     : {elapsed_time_dask:.4f} seconds")
print(f"Memory Used (MB)        : {mem_used_dask:.2f} MB")
print(f"CPU Usage               : {cpu_usage_dask:.2f}%")
print(f"Throughput              : {throughput_dask:,.2f} records/second")
print("=========================================================================
==")

print(f"Final row count: {len(df_cleaned_dask)}")
# Dask: Show top 3 rows
df_cleaned_dask.head(3)
```

10.5 Appendix F

Modin Data Processing

```
import os
import time
import psutil
import tracemalloc
import modin.pandas as mpd  # Modin's pandas-like API
import pandas as pd  # Regular pandas for to_numeric
import numpy as np

# Convert pandas DataFrame to Modin DataFrame
df_modin = mpd.DataFrame(df)  # Assuming df is already your pandas DataFrame

# ========== Performance Tracking ==========
process_md = psutil.Process(os.getpid())
tracemalloc.start()
start_time_md = time.perf_counter()
mem_start_md = process_md.memory_info().rss / (1024 * 1024)  # in MB

# ========== Data Processing ==========

# 1. Replace empty values with None (null)
df_modin.replace(["", "NaN", "N/A", "Not Sure"], np.nan, inplace=True)

# 2. Drop rows where all columns (except 'Pet ID') are NULL
df_modin.dropna(subset=[col for col in df_modin.columns if col != 'Pet ID'],
how='all', inplace=True)

# 3. Strip whitespace from string columns
str_cols = df_modin.select_dtypes(include='object').columns
df_modin[str_cols] = df_modin[str_cols].apply(lambda x: x.str.strip())

# 4. Remove duplicates
df_modin.drop_duplicates(inplace=True)

# 5. Process 'Profile' column: Split into 'Gender' and 'Age'
if 'Profile' in df_modin.columns:
     df_modin[['Gender', 'Age']] = df_modin['Profile'].str.extract(r'(?P[^,]+),
(?P.+)')
    df_modin.drop("Profile", axis=1, inplace=True)

# 6. Process 'Body' column: Split into 'Body Size' and 'Fur Length'
if 'Body' in df_modin.columns:
                      df_modin[['Body Size', 'Fur Length']] =
df_modin['Body'].str.extract(r'(?P[^,]+), (?P.+)')
    df_modin.drop("Body", axis=1, inplace=True)
```

```
# 7. Process 'Posted' column: Extract dates
if 'Posted' in df_modin.columns:
                                df_modin[['Original',        'Updated']]        =
df_modin['Posted'].str.extract(r'(?P[^()]+)(?:
Updated(?P[)]+)
)?')

    # Clean and parse dates
          df_modin['Original   Date']   =   mpd.to_datetime(df_modin['Original'],
errors='coerce')
           df_modin['Updated   Date']   =   mpd.to_datetime(df_modin['Updated'],
errors='coerce')
    df_modin['Updated Date'].fillna(df_modin['Original Date'], inplace=True)
    df_modin.drop(['Posted', 'Original', 'Updated'], axis=1, inplace=True)

# 8. Process boolean columns
bool_cols = ['Vaccinated', 'Dewormed', 'Spayed']
for col_name in bool_cols:
    if col_name in df_modin.columns:
                         df_modin[col_name]   =   df_modin[col_name].map({'Yes':
1}).fillna(0).astype(int)

# 9. Process 'Amount' column
if 'Amount' in df_modin.columns:
                                                 df_modin['Amount']            =
df_modin['Amount'].astype(str).str.extract(r'(\d+)').astype(float)
    df_modin['Amount'].fillna(1, inplace=True)
    df_modin['Amount'] = df_modin['Amount'].astype(int)

# 10. Process 'Price' column
if 'Price' in df_modin.columns:
    df_modin['Price'] = df_modin['Price'].astype(str).str.strip().str.upper()
    df_modin['Price'].replace('FREE', '0', inplace=True)
    df_modin['Price'] = df_modin['Price'].str.replace("^RM", "", regex=True)
     # Use Modin's to_numeric if available, otherwise convert to pandas Series
temporarily
    try:
                       df_modin['Price']   =   mpd.to_numeric(df_modin['Price'],
errors='coerce').fillna('Enquire')
    except AttributeError:
        # Fallback to pandas if Modin doesn't have to_numeric
               df_modin['Price']   =   mpd.Series(pd.to_numeric(df_modin['Price'],
errors='coerce')).fillna('Enquire')

# ========== Performance Tracking ==========
end_time_md = time.perf_counter()
mem_end_md = process_md.memory_info().rss / (1024 * 1024)  # in MB
cpu_usage_md = psutil.cpu_percent(interval=1)
```

```
elapsed_time_md = end_time_md - start_time_md
mem_used_md  =  mem_end_md  -  mem_start_md   #  Actual  memory  used  during  the
operation
throughput_md = len(df_modin) / elapsed_time_md

# ========== Metrics ==========
print("\n==============    Data    Processing    Performance    Summary    (Modin)
==============")
print(f"Wall Time (Elapsed)      : {elapsed_time_md:.4f} seconds")
print(f"Memory Used (MB)         : {mem_used_md:.2f} MB")
print(f"CPU Usage                : {cpu_usage_md:.2f}%")
print(f"Throughput               : {throughput_md:,.2f} records/second")
print("==================================================")

print(f"Final row count: {len(df_modin)}")
# Display top 3 rows
df_modin.head(3)
```

10.5 Appendix G

Project GitHub Repo

https://github.com/Jingyong14/HPDP02/tree/962dd7bee60acf1c5bfe31e2e6d783499c8e148c/2425/project/p1/Group%205

MongoDB Atlas Connector

mongodb+srv://neoweng:<db_password>@hpdp-p1.uya0htc.mongodb.net/?retryWrites=true&w=majority&appName=hpdp-p1