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	16 MAY 2025

Table of Contents

1.0 Introduction..........1

1.1 Background of the Project..........1

1.2 Objectives..........2

1.3 Target Website and Data to be Extracted..........3

2.0 System Design and Architecture..........5

2.1 Description of Architecture..........5

2.2 Tools and Frameworks Used..........6

2.3 Roles of Team Members..........8

3.0 Data Collection..........9

3.1 Crawling Method..........9

3.2 Number of Records Collected..........9

3.3 Ethical Considerations..........10

4.0 Data Processing..........11

4.1 Cleaning Methods..........11

4.2 Data Structure..........12

4.3 Transforming and Formatting..........14

5.0 Optimization Techniques..........16

5.1 Methods Used..........16

5.2 Code Overview or Pseudocode of Techniques Applied..........17

5.2.1 Fast API (Async)..........17

5.2.2 PySpark..........18

5.2.3 Dask..........23

5.2.4 Modin..........25

6.0 Performance Evaluation..........29

6.1 Before vs After Optimization..........29

6.2 Time, Memory, CPU Usage, Throughput.......... 31

6.3 Charts and Graphs..........32

7.0 Challenges and Limitations..........34

7.1 What Didn’t Go as Planned.......... 34

7.2 Limitations of Solution.......... 34

8.0 Conclusion and Future Work.......... 36

8.1 Summary of Findings..........36

8.2 What Could be Improved..........36

9.0 References..........38

10.0 Appendices..........39

1.0 Introduction

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 PetFinder Malaysia using various web scraping tools.
- To clean and transform the extracted datasets using different Python libraries.
- To implement optimization techniques like multithreading, multiprocessing, and distributed computing to improve data processing efficiency.
- 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 will be included in the data that needs to be extracted:

Table 1. List of Data Fields to be Extracted

No	Data Field	Description
1	Pet ID	Unique identifier for each pet listing
2	Name	Name of the pet
3	Type	Type of pet (e.g., dog, cat, rabbit)
4	Species	Specific species or breed
5	Profile	Brief description of the pet's characteristics
6	Amount	Adoption fee (if available)
7	Vaccinated	Whether the pet is vaccinated
8	Dewormed	Whether the pet is dewormed
9	Spayed	Whether the pet is spayed or neutered
10	Condition	Health condition or special needs (if any)
11	Body	Size or physical attributes of the pet
12	Color	Color of the pet
13	Location	Location of the pet (e.g., city, state)
14	Posted	Date when the listing was posted
15	Price	Adoption fee or any costs involved
16	Uploader Type	Type of user uploading the pet (e.g., shelter, individual)

17	Uploader Name	Name of the uploader (shelter or individual)
18	Status	Status of the pet (e.g., available, adopted, pending)

This data will be 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

2.1 Description of Architecture

The diagram below shows the architecture for web scraping pet details from PetFinder.my:

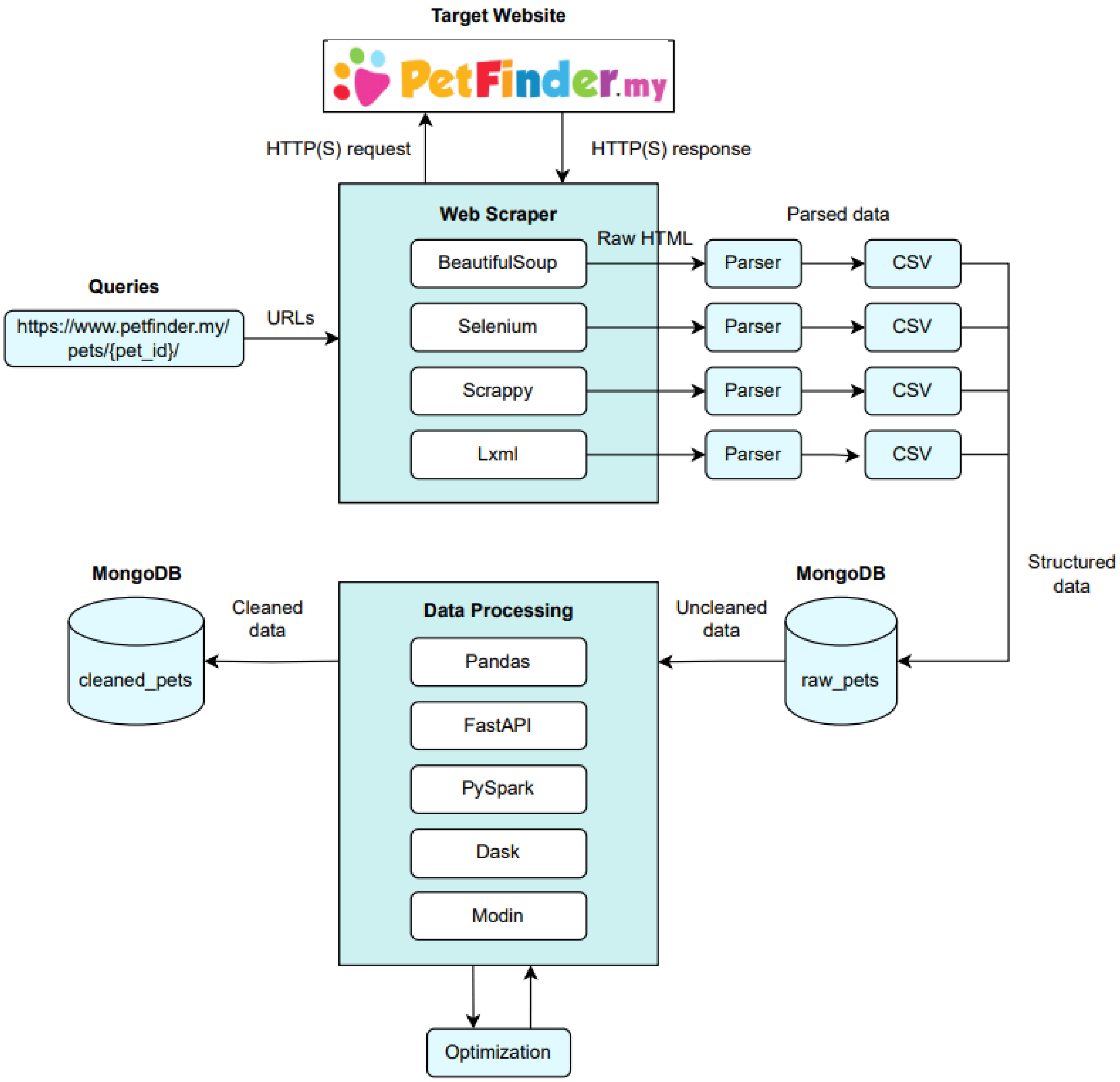


Figure 2.1. Web Scraping Architecture for PetFinder.my

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. 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 like names of pets, adoption fees, and so on 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.

2.2 Tools and Frameworks Used

The table below shows the tools and frameworks used in the project:

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.

2.3 Roles of Team Members

The table below outlines the roles of the team members involved in the web scraping project:

Table 3. List of Roles of Team Members

Roles	**Team Members**
Group Leader	Ng Shu Yu
HPC Specialist	Neo Zheng Weng
Architect	Navasarathy A/L S.Ganeswaran
Evaluator	Muhammad Safwan Bin Mohd Azmi

3.0 Data Collection

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

4.1 Cleaning Methods

The cleaning process was a crucial step in preparing the raw scraped data for downstream processing and analysis. It was performed by all group members using different tools (Pandas, PySpark, Dask, Modin, and FastAPI), but followed a consistent set of logical steps to ensure standardization across all implementations. The goal was to convert inconsistent, incomplete, and messy raw data into a structured, usable format.

Table 5. List of Cleaning Methods Used

Step	Description
Handling Missing Values	- Removed rows with all fields marked as 'N/A' or missing. - Replaced placeholder values like 'N/A' and 'Not Sure' with NaN for consistency.
Removing Duplicates	- Remove duplicate records based on unique identifiers (e.g., Pet ID). - Only the first occurrence of each duplicate was retained.
Whitespace & Formatting	- Stripped leading/trailing spaces from all string fields. - Optionally normalized text casing (e.g., title case) to support clean grouping.
Preparing for Transformation	- Retained raw but cleaned fields for later processing. - Final cleaned dataset was exported for transformation.

4.2 Data Structure

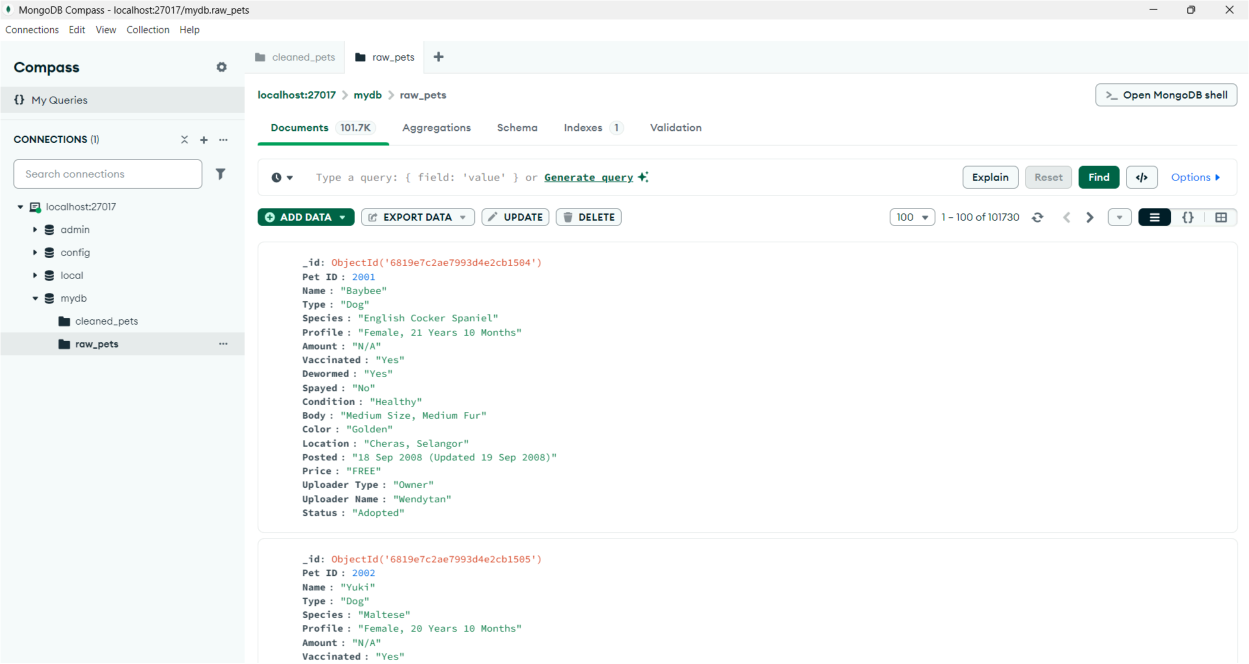
In this project, the data scraping and processing workflow is organized using a structured combination of CSV files and MongoDB storage. Initially, the output from the web scraping process is saved in CSV format, ensuring simplicity, compatibility, and ease of sharing among team members. To maintain consistency and facilitate seamless integration, a standardized file naming convention is implemented, allowing efficient tracking and merging of individual contributions. The individual CSV files (**pets1.csv**, **pets2.csv**, **pets3.csv**, and **pets4.csv**) are then systematically merged and consolidated into a single master file, **pets.csv**, which serves as the finalized, cleaned dataset for subsequent processing and storage in the MongoDB database. This approach provides a scalable and flexible solution for efficient querying and future application development.

Table 6. List of CSV Files

File Name	Contributor	Description
pets1.csv	Shu Yu	Scraped pet data from PetFinder (Part 1)
pets2.csv	Safwan	Scraped pet data from PetFinder (Part 2)
pets3.csv	Nava	Scraped pet data from PetFinder (Part 3)
pets4.csv	Neo	Scraped pet data from PetFinder (Part 4)

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.

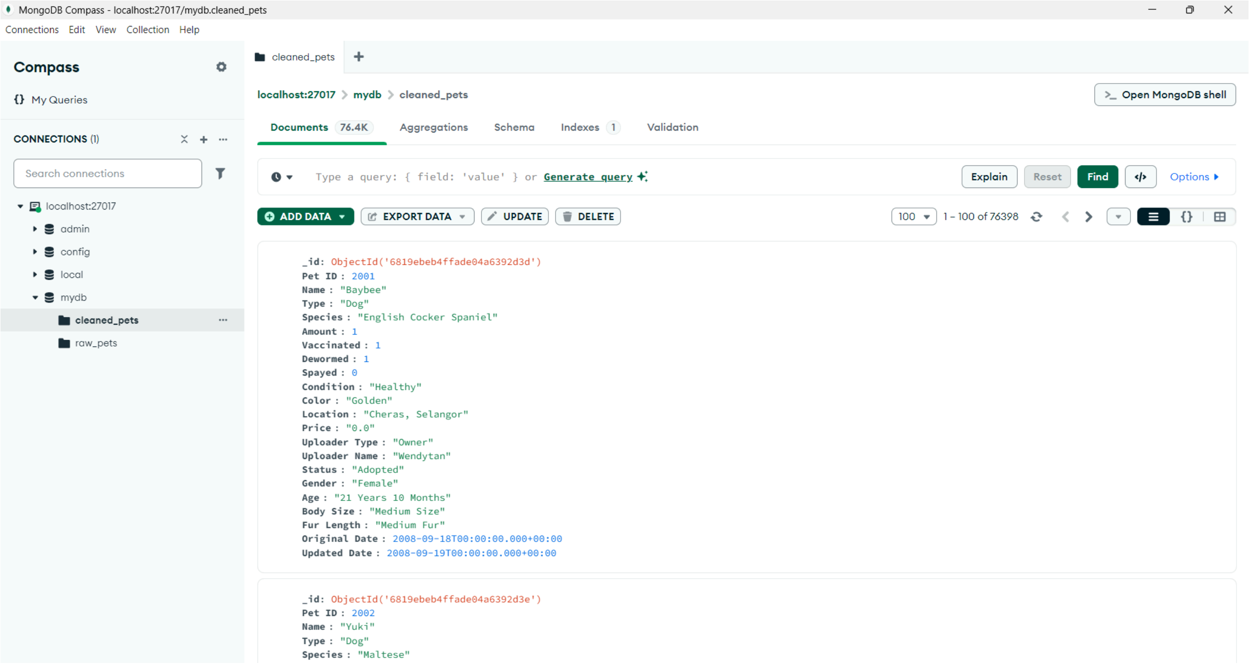
The figure below illustrates the data loaded into MongoDB Compass, showing the structure and sample records within the database :



```
_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.

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, 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, 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.

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

5.1 Methods Used

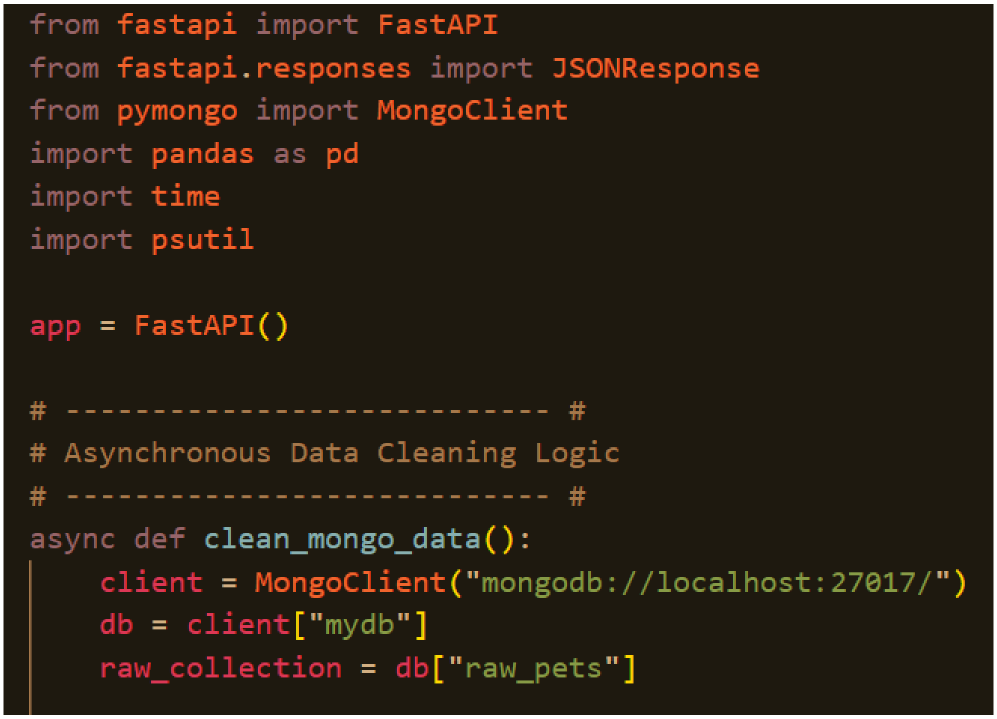
Table 7 below shows the high-performance optimization techniques implemented with various data processing libraries, including FastAPI (Async), PySpark, Dask, and Modin.

Table 7. List of Optimization Techniques Used with Each Library

Libraries	Optimization Techniques	Explanation
FastAPI (Async) FastAPI	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 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 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 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.

5.2 Code Overview or Pseudocode of Techniques Applied

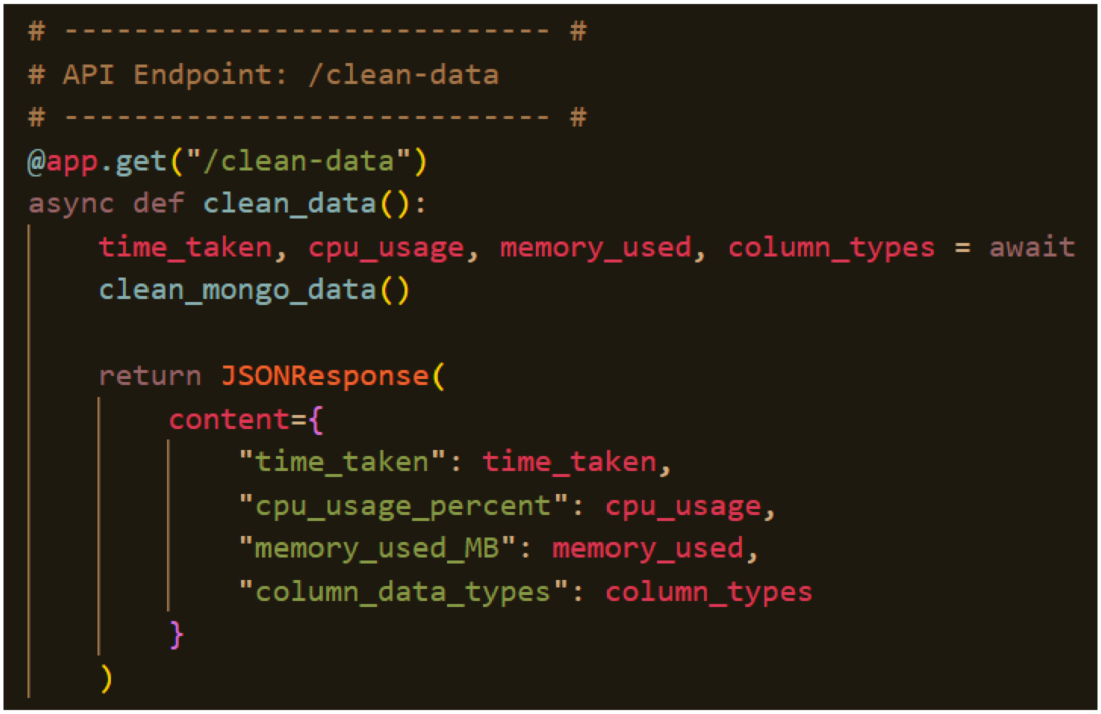
5.2.1 Fast API (Async)



```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

This 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.



```
# ---------------------------- #
# 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

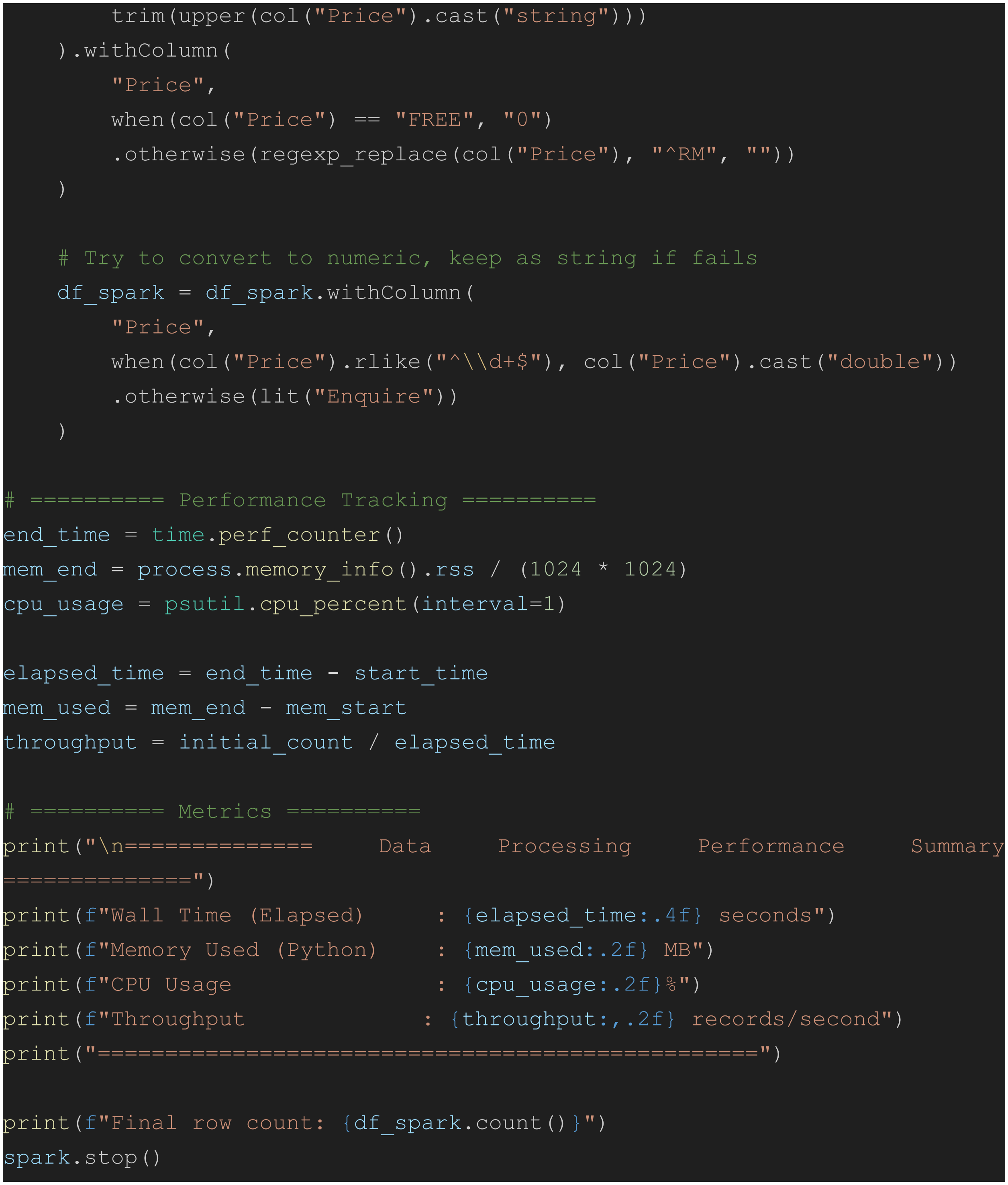
This is an asynchronous endpoint handler in FastAPI. Declaring the endpoint with 'async def' tells it that it will run in an asynchronous event loop. This enables the server to process multiple requests concurrently without having to wait for one request to complete before serving another. It keeps the 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.

5.2.2 PySpark

```
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",
```



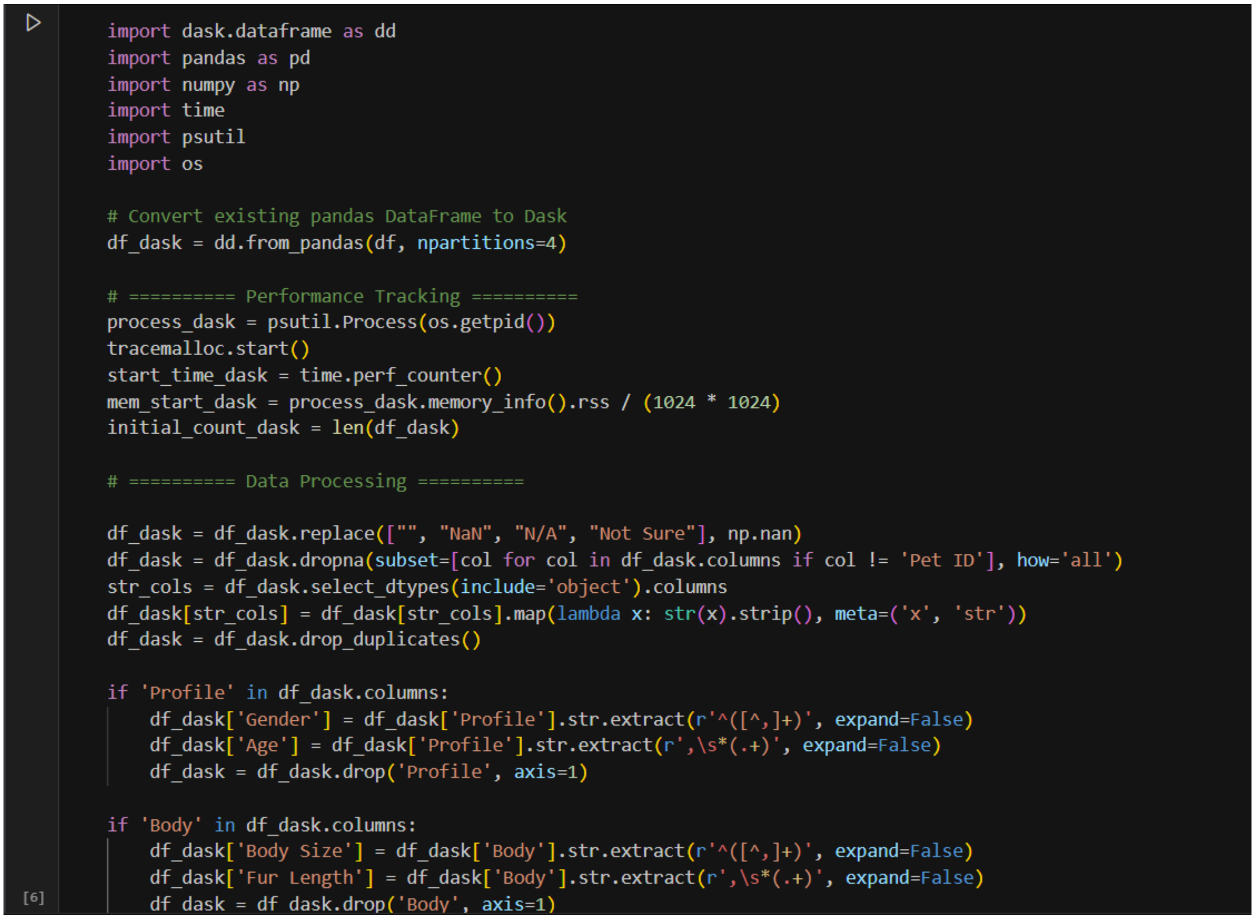
```
        trim(upper(col("Price").cast("string")))
    ).withColumn(
        "Price",
        when(col("Price") == "FREE", "0")
        .otherwise(regexp_replace(col("Price"), "^RM", ""))
    )

    # 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()
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
==============")
print(f"Wall Time (Elapsed)      : {elapsed_time:.4f} seconds")
print(f"Memory Used (Python)     : {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()}")
spark.stop()
```

Figure 5.2.2 PySpark Data Processing

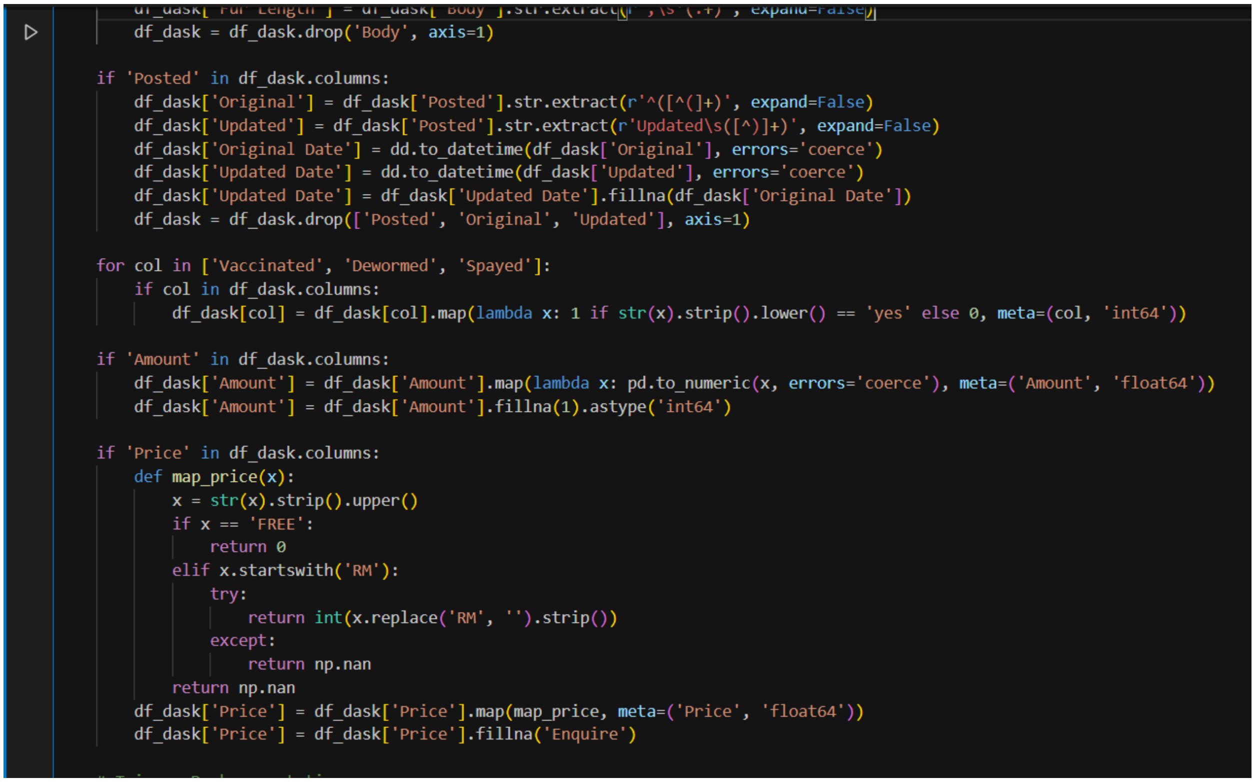
The script coded in Figure 5.2.2 maximizes the processing of a big pet adoption dataset with PySpark. It carries out data cleaning and transformation operations like replacing empty strings with nulls, deleting rows with missing required data, removing leading and trailing whitespace, eliminating duplicates, 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.

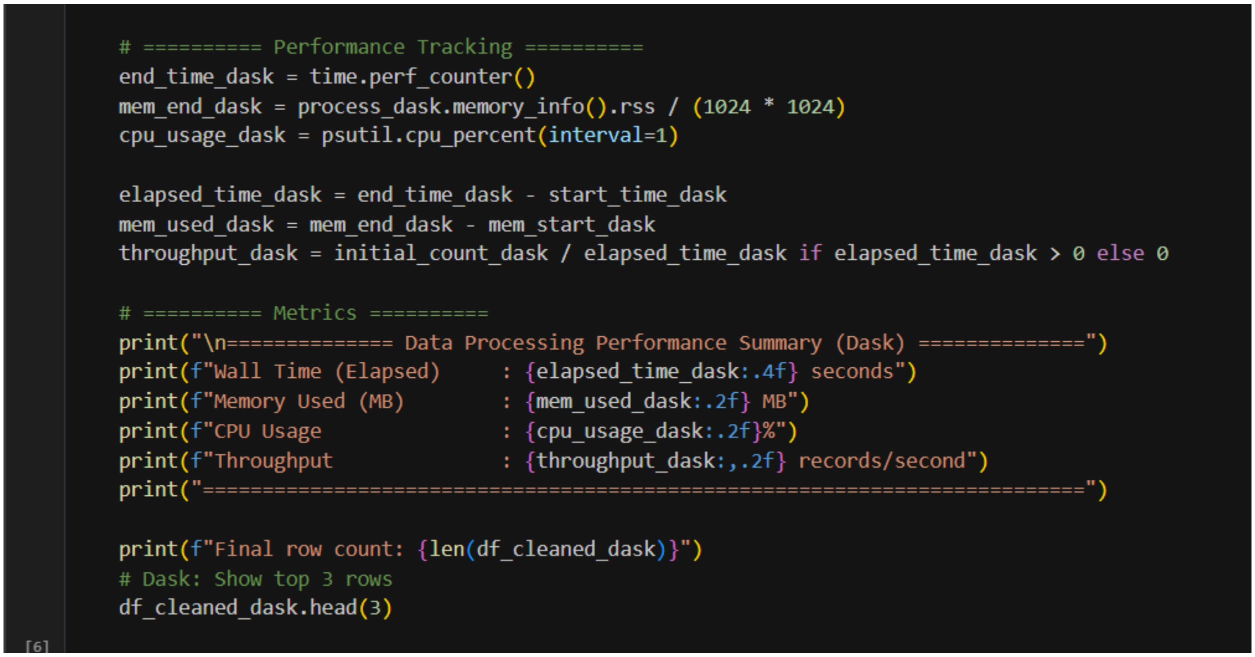
5.2.3 Dask



```python
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)
```



```python
    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')
```



```python
# ========== 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)
```

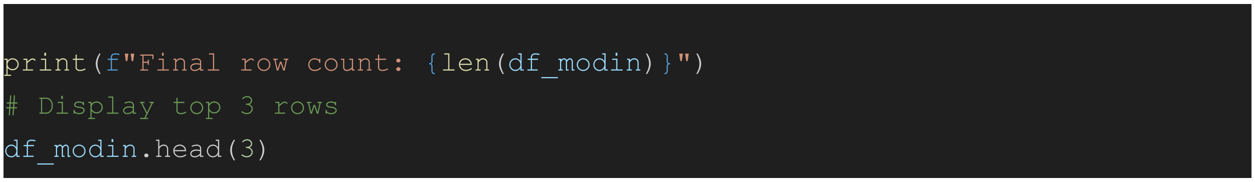
Figure 5.2.3 Dask Data Processing

This Dask pipeline 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

```
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<Gender>[^,]+), (?P<Age>.+)')
    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<Body_Size>[^,]+), (?P<Fur_Length>.+)')
    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<Original>[^()]+)(?:\(Updated
(?P<Updated>[^)]+)\))?')

    # 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)
```

Figure 5.2.4 Modin Data Processing

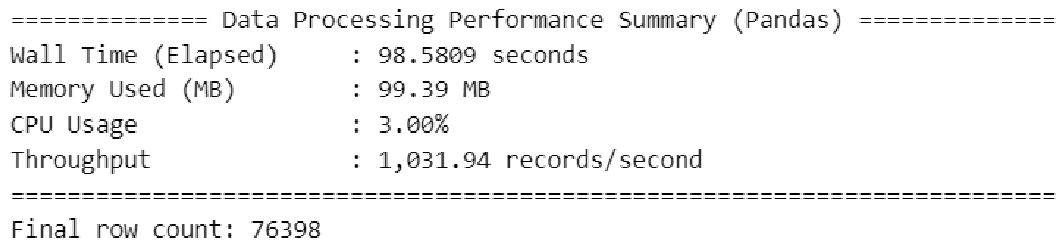
This script shown in Figure 5.2.4 utilizes Modin, a parallelized implementation of Pandas, to accelerate data processing by using multiple CPU cores to do things in parallel and quicker. It is refactored to process large data efficiently by doing operations like replacing missing values with `NaN`, stripping 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

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.

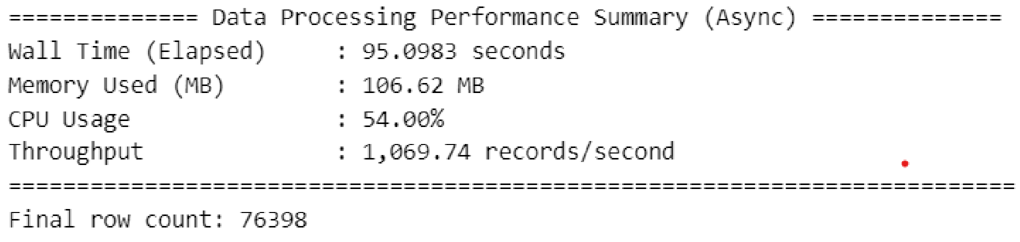
Result of Panda Data Processing Performance:



```
============== 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
```

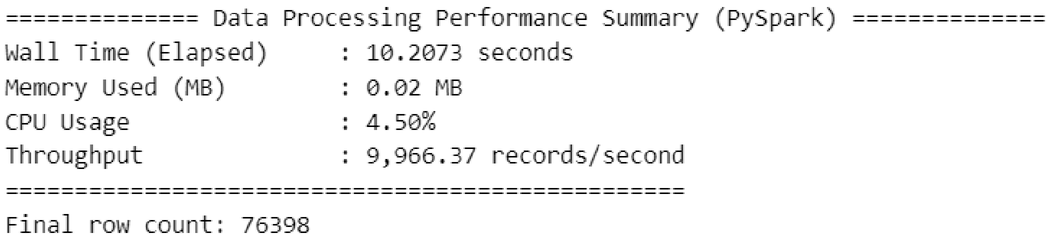
To optimize this process, the same logic and dataset were used with the following high-performance frameworks:

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:



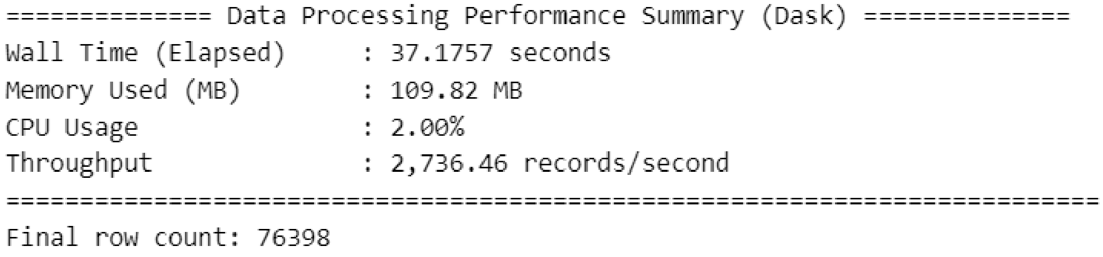
```
============== Data Processing Performance Summary (Async) ==============
Wall Time (Elapsed)      : 95.0983 seconds
Memory Used (MB)         : 106.62 MB
CPU Usage                : 54.00%
Throughput               : 1,069.74 records/second
==========================================================================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

B. **PySpark:** Used distributed processing across multiple cores or clusters. It transformed and cleaned large datasets efficiently with built-in parallelism and fault tolerance:



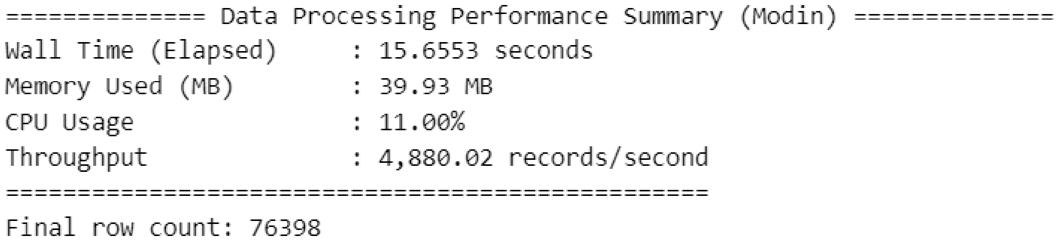
```
============= Data Processing Performance Summary (PySpark) =============
Wall Time (Elapsed)     : 10.2073 seconds
Memory Used (MB)        : 0.02 MB
CPU Usage               : 4.50%
Throughput              : 9,966.37 records/second
=================================================
Final row count: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

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:



```
============== 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
```

D. **Modin:** Served as a drop-in replacement for Pandas, it accelerated Pandas operations by utilizing all available CPU cores through parallelization:



```
============= 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
```

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.

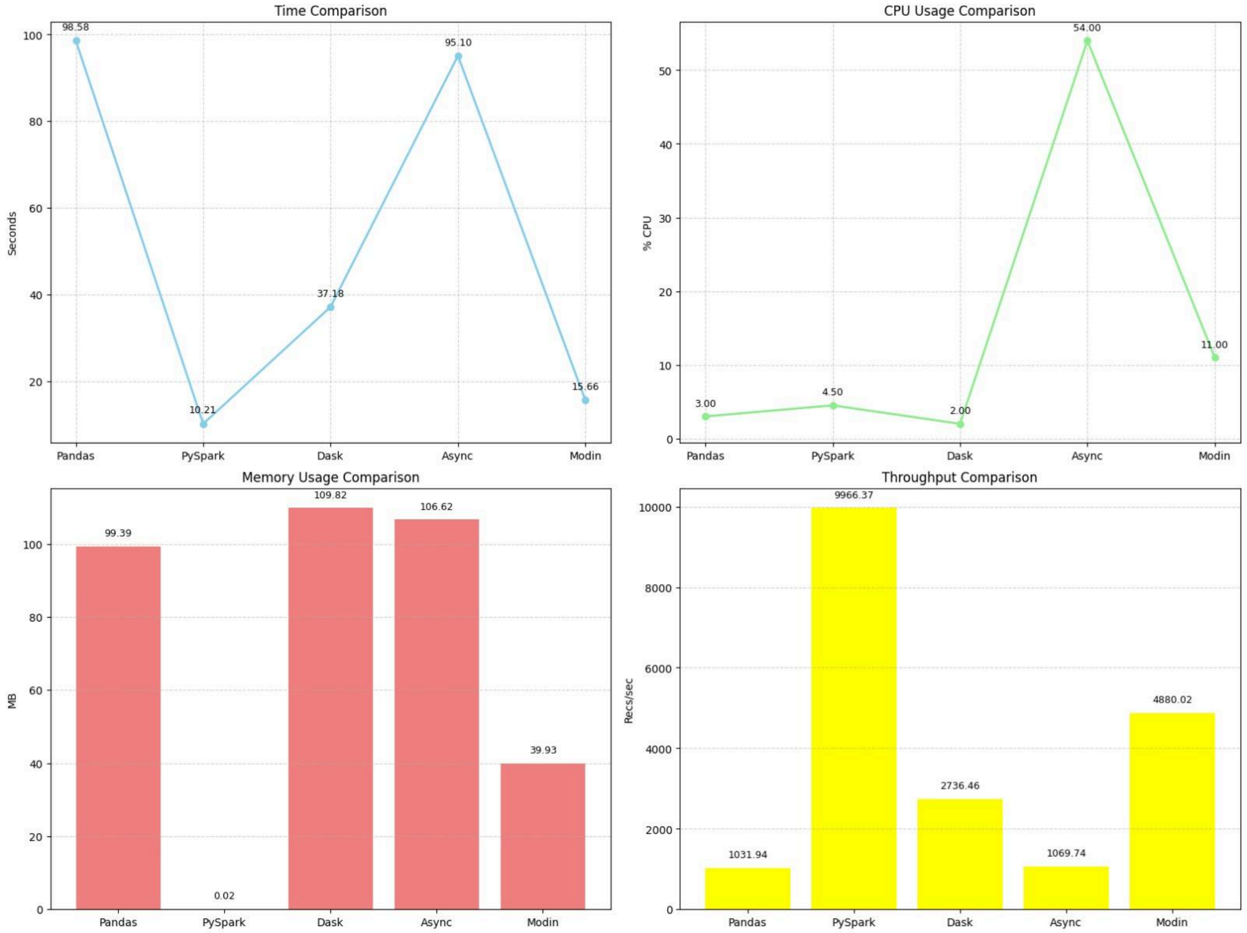
6.2 Time, Memory, CPU Usage, Throughput

The table below shows the performance metrics in data processing for each technique, including Pandas, PySpark, Dask, FastAPI, and Modin:

Table 8. 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	9,966.37	4880.02	1069.74	4880.02

6.3 Charts and Graphs



Pandas is a popular Python data manipulation package that is well-known for its comprehensive capability and user-friendly API. However, Pandas used 99.39 MB of memory and processed 1,031.94 records/sec in 98.58 seconds to finish this benchmark. Due to its single-threaded processing, its CPU utilization stayed low at 3%. It is ineffective for large-scale or time-sensitive applications, although it works well for small to medium datasets.

The performance of PySpark, the Python API for Apache Spark, was exceptional. It used just 0.02 MB of memory, finished the work in 10.21 seconds, and achieved the greatest throughput of 9,966.37 records/sec. PySpark made great use of Spark's distributed computing design, which made it perfect for large-scale, compute-intensive tasks, even though it only used 4.5% CPU.

With a throughput of 2,736.46 records/sec and a memory use of 109.82 MB, Dask, which allows parallel computation on a single computer or cluster, processed the data in 37.18 seconds. With a CPU use of 2%, there may be space for improvement. Dask balances scalability and flexibility, even if it is slower than PySpark, particularly for datasets that are too big to fit in memory.

With a throughput of 1,069.74 records/sec, async processing—which is often more efficient for I/O-bound operations—finished the task in 95.10 seconds while using 106.62 MB of memory. Its extremely high CPU utilization of 54%, however, indicates that it is less effective for CPU-bound operations like data transformations. Although it performed noticeably worse than Dask and PySpark, it exceeded Pandas by a little margin in throughput.

Modin, which uses multi-core parallelization under the hood while maintaining Pandas compatibility, showed promising results. It completed the task in 15.66 seconds, using 39.93 MB of memory and achieving 4,880.02 records/sec throughput, with 11% CPU usage. These results suggest that Modin is a strong alternative for users seeking Pandas-like syntax with better scalability.

To sum up, 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.

7.0 Challenges and Limitations

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 and MongoDB Atlas restrictions 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

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 theseBy 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, 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

- 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