

0312 Y

Final_Report_DataDrillers.pdf

 My Files My Files University

Document Details

Submission ID

trn:oid::17268:96088939

Submission Date

May 15, 2025, 9:31 PM GMT+5:30

Download Date

May 15, 2025, 9:34 PM GMT+5:30

File Name

Final_Report_DataDrillers.pdf

File Size

2.1 MB

39 Pages**5,999 Words****32,226 Characters**

*% detected as AI

AI detection includes the possibility of false positives. Although some text in this submission is likely AI generated, scores below the 20% threshold are not surfaced because they have a higher likelihood of false positives.

Caution: Review required.

It is essential to understand the limitations of AI detection before making decisions about a student's work. We encourage you to learn more about Turnitin's AI detection capabilities before using the tool.

Disclaimer

Our AI writing assessment is designed to help educators identify text that might be prepared by a generative AI tool. Our AI writing assessment may not always be accurate (it may misidentify writing that is likely AI generated as AI generated and AI paraphrased or likely AI generated and AI paraphrased writing as only AI generated) so it should not be used as the sole basis for adverse actions against a student. It takes further scrutiny and human judgment in conjunction with an organization's application of its specific academic policies to determine whether any academic misconduct has occurred.

Frequently Asked Questions

How should I interpret Turnitin's AI writing percentage and false positives?

The percentage shown in the AI writing report is the amount of qualifying text within the submission that Turnitin's AI writing detection model determines was either likely AI-generated text from a large-language model or likely AI-generated text that was likely revised using an AI-paraphrase tool or word spinner.

False positives (incorrectly flagging human-written text as AI-generated) are a possibility in AI models.

AI detection scores under 20%, which we do not surface in new reports, have a higher likelihood of false positives. To reduce the likelihood of misinterpretation, no score or highlights are attributed and are indicated with an asterisk in the report (*%).

The AI writing percentage should not be the sole basis to determine whether misconduct has occurred. The reviewer/instructor should use the percentage as a means to start a formative conversation with their student and/or use it to examine the submitted assignment in accordance with their school's policies.

What does 'qualifying text' mean?

Our model only processes qualifying text in the form of long-form writing. Long-form writing means individual sentences contained in paragraphs that make up a longer piece of written work, such as an essay, a dissertation, or an article, etc. Qualifying text that has been determined to be likely AI-generated will be highlighted in cyan in the submission, and likely AI-generated and then likely AI-paraphrased will be highlighted purple.

Non-qualifying text, such as bullet points, annotated bibliographies, etc., will not be processed and can create disparity between the submission highlights and the percentage shown.





UTM
UNIVERSITI TEKNOLOGI MALAYSIA

SECP3133 - HIGH PERFORMANCE DATA PROCESSING

DATA DRILLERS

PROJECT 1

Optimising High-Performance Data Processing for
Large-Scale Web Crawlers

Lecturer: Prof. Madya. Ts. Dr Shahizan Bin Othman - Section 01

MUHAMMAD ANAS BIN MOHD PIKRI	A21SC0464
ALIATUL IZZAH BINTI JASMAN	A22EC0136
MULYANI BINTI SARIPUDDIN	A22EC0223
THEVAN RAJU A/L JEGANATH	A22EC0286

Table of Contents

1.0 Introduction	2
1.1 Background of the project	2
1.2 Objectives	2
1.3 Target website and data to be extracted	3
2.0 System Design & Architecture	4
2.1 Description of architecture	4
2.2 Tools and frameworks used	6
2.3 Roles of team members	7
3.0 Data Collection	8
3.1 Crawling method (pagination, rate-limiting, sync)	8
3.2 Number of records collected	11
3.3 Ethical considerations	12
4.0 Data Processing	14
4.1 Cleaning methods	14
4.2 Transformation and formatting	15
4.3 Data structure	16
5.0 Optimisation Techniques	17
5.1 Methods used	17
5.2 Code overview or pseudocode of techniques applied	17
5.2.1 Pandas	17
5.2.2 DuckDB	18
5.2.3 Polars	19
6.0 Performance Evaluation	20
6.1 Before vs after optimisation	20
6.2 Comparative Analysis of Performance Metrics	21
6.3 Charts and graphs	22
Summary	24
7.0 Challenges & Limitations	25
7.1 What didn't go as planned	25
7.2 Any limitations of your solution	25
8.0 Conclusion & Future Work	26
8.1 Summary of findings	26
9.1 References List	28
10.0 Appendices	29
10.1 Sample code snippets	29
10.2 Screenshots of output	36
10.3 Links to full code repo or dataset	38

1.0 Introduction

1.1 Background of the project

Due to the emergence of big data, the capacity to handle data to extract important actionable insights out of the masses of information available on the internet is vital. This project will address the application of High-Performance Computing (HPC) technologies to optimise information gathering from websites. The dramatic expansion of the automotive industry in Malaysia has led to the need for improved tools for analysing car listings. Accessing data from such platforms as [Carlist.my](https://www.carlist.my) is easy, but manually processing it is cumbersome and unproductive. In order to do this, we have developed a web scraper that will allow for the auto-extraction of data, which will improve both the reliability and the type of information that is available to users. The main task of this project is to learn and apply practical skills in multithreading, multiprocessing, optimising and distributed computing, using car listings from a Malaysian website, carlists.my, and showing the example of websites that serve car owners with multimodal functionalities.

1.2 Objectives

The primary objectives of this project are to gain practical skills in multithreading, multiprocessing and distributed computing. By applying those methodologies, we are able to improve the performance of the web scraper, which will be able to handle larger datasets and provide results faster. In addition, with this initiative, the real-world application of HPC techniques to overcome real-world hurdles in the extraction of data will be demonstrated. Below are the objectives of our web scraping project:

- **Data Collection:** Develop a web crawler that extracts a minimum of 100,000 data points efficiently from [carlists.my](https://www.carlists.my).
- **Data Processing:** Implement cleaned data to processes to ensure high-quality data for analysis.
- **Performance Optimisation:** Apply HPC techniques such as multithreading, multiprocessing, and distributed computing using tools like Polars, Pandas, and Duckdb to significantly enhance the data processing speed and efficiency.
- **Evaluation and Comparison:** Conduct a comprehensive performance evaluation of the system before and after optimisation, using appropriate metrics to quantify improvements.

- Visualisation and reporting: Produce a final technical report based on visualisation analysis, cleaned dataset, and presentation of the findings.

1.3 Target website and data to be extracted

The project website is focused on carlists.my, a famous Malaysian site that lists cars for sale. Among other points, our web scraper will pick up the information for these parameters in a car listing. The information gathered from this data can help users understand vehicles while further supporting in-depth examinations of market trends and changes in price.

In addition, certain attributes of the dataset will tell us how well our scrape and process steps are adjusted. The evaluation will be done using metrics related to the optimisation stage, total number of rows, total time taken in seconds, CPU percentage, memory usage in megabytes, throughput of the data by records per second, number of processed data rows, and their times. Our assessments of high-performance computing will help us see how well the web scraper copes with a lot of data. Doing so will make it clear what the benefits are after improving techniques for computing and handling data. Having this well-organised data, we are able to look at the Malaysian car market, including used vehicles, and handle the challenges and ethical aspects of getting information online.

2.0 System Design & Architecture

2.1 Description of architecture

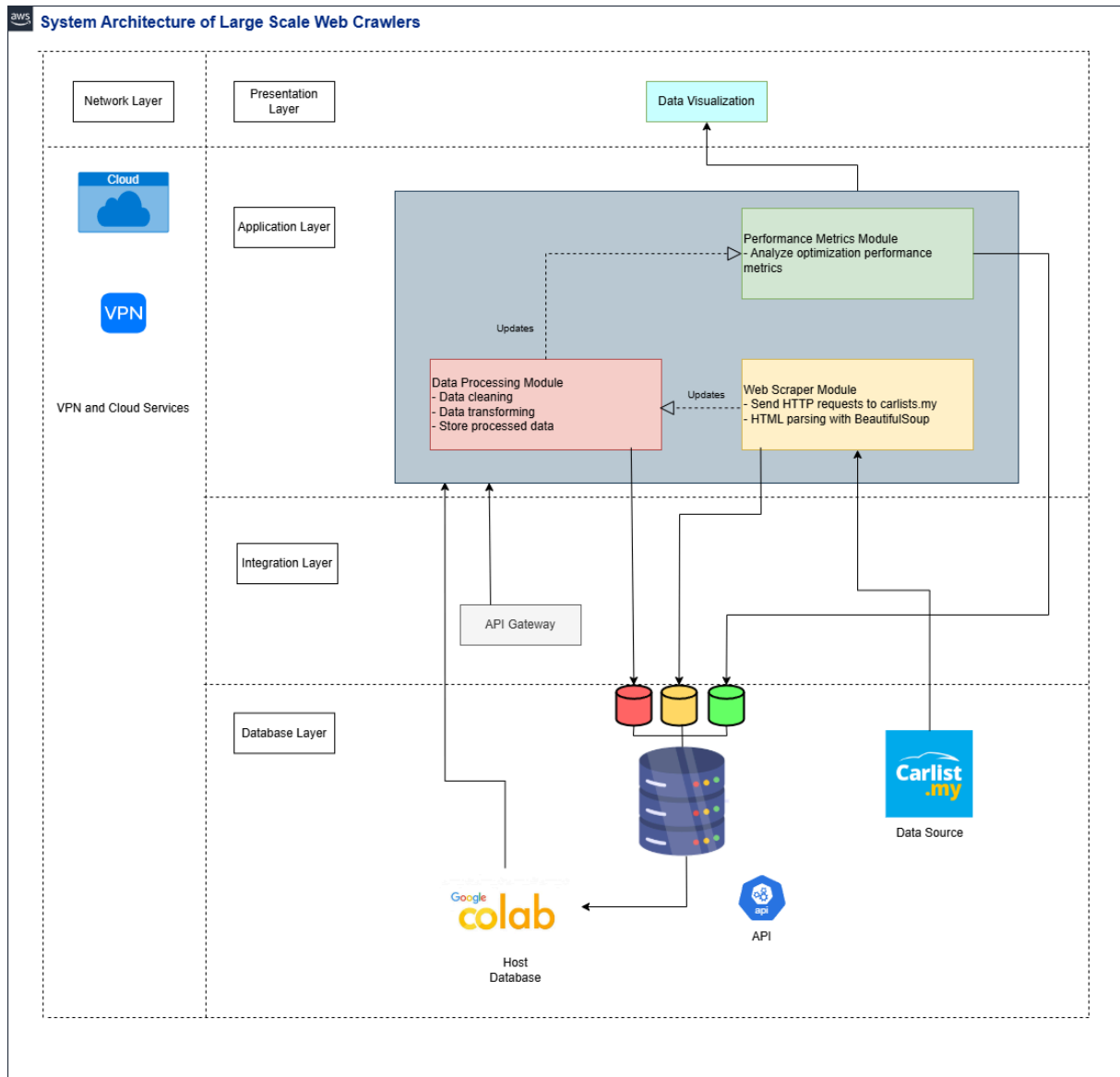


Figure 1 : System Architecture of Large-Scale Web Crawlers

1. Presentation Layer

A simple visualised version of the metrics from optimised data utilising a bar chart to display analysis results to showcase the performance metrics, including processed time, memory usage, CPU usage and throughput of each library.

2. Application Layer

Web Scraper Module

A module that is responsible for sending HTTP requests to the carlists.my via requests library to extract relevant data such as car name, car type, to do analysis.

Data Processing Module

A module that is responsible for cleaning and transforming data, such as removing duplicated rows, handling null and checking the datatype of each

column before conducting optimisation and analysis.

Performance Metrics Module

A module that is responsible for calculating performance metrics of different libraries, including optimisation libraries such as time processed, CPU usage, memory usage and throughput of number of records per second.

3. Integration Layer

The integration layer serves as an important go-between, coordinating the involvement of all components in the web scraping system. Within this layer, an API Gateway is used to regulate both incoming and outgoing requests to guarantee the web scraper communicates well with other services, especially due to the large data scraping. In addition, the integration layer makes it possible to apply HPC by easily connecting data processing and analysis tools and libraries.

4. Database Layer

[Carlists.my](#) acted as a data resource for web scraping over 170k rows of a dataset. This layer is organised to store raw, processed, and clean data in a simple format that is convenient for analysis, which is in csv file. It also supports the storage of a lot of data via Google Colab, which is a cloud data storage that can monitor performance to judge the quality of the scraping and processing. With properly organised storage, the web crawler can easily manage large quantities of data and give accurate information on market trends.

5. Network Layer

The web crawler architecture relies heavily on the network layer to help the web scraper and target website communicate properly. The layer uses secure VPN and cloud services to deliver requests safely and with reduced latency while maintaining a high throughput of data from [Carlists.my](#). Handling significant traffic is the purpose of the network layer, as web scraping large sites requires extracting a large amount of data at once. With this solid communication system, the scraper can get and use data in real time.

2.2 Tools and frameworks used

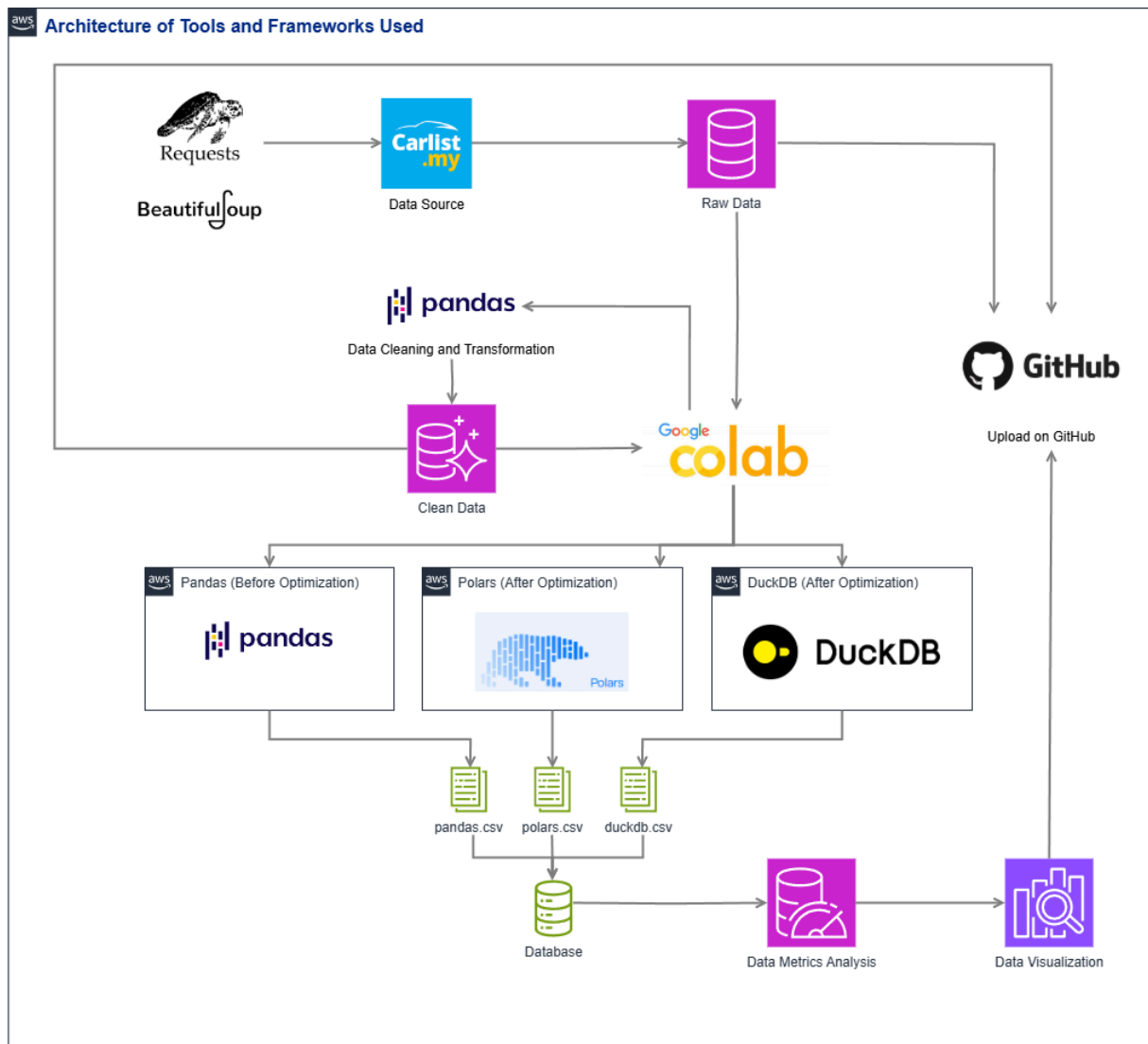


Figure 2 : Architecture of Tools and Frameworks Used

Table 1 : Descriptions of the Tools and Frameworks Used

No.	Tools / Frameworks	Description
1.	Carlists.my	The data source of cars available to be purchase in Malaysia.
2.	Google Colab	An online platform that allows writing and executing Python code in a Jupyter notebook environment and allows collaborative work.
3.	GitHub	Upload results and source code for version control and collaboration purposes.

4.	Requests & BeautifulSoup	These Python libraries are used to scrape data from Carlist.my , which allows sending HTTP requests and parsing HTML content to extract relevant information.
5.	Pandas	Used during the data cleaning and transformation phase, and acts as an initial analysis of perforation metrics. Also utilised for data processing, visualisation for visual analysis.
6.	Polars	Used during optimisation for data processing, which is a faster alternative to Pandas.
7.	DuckDB	Used during optimisation for data processing, which is a faster alternative to Pandas

2.3 Roles of team members

Table 2 : Distributions of Tasks

Member Name	Task
Thevan Raju A/L Jeganath	Data Cleaning, Exploring possible optimisation library, Data Visualisation. Documentation of data processing, Performance evaluation explanation.
Mulyani Binti Saripuddin	Web crawling, Data visualisation, Exploring the possible optimisation libraries. Documentation of optimisation techniques. Summary of findings and conclusion
Aliatul Izzah Binti Jasman	Pandas Before Optimisation, DuckDB Optimisation, Introduction, System design and architecture, Tools and frameworks used, challenges and limitations.
Muhammad Anas Bin Mohd Pikri	Polar's Optimisation, Exploring the possible optimisation library. Data collection, Challenges and limitations, Appendices, Future work and conclusion.

3.0 Data Collection

3.1 Crawling method (pagination, rate-limiting, sync)

A synchronous web crawler was designed with the use of Python to scrape car listings from Carlist.my. The crawler carries out a straightforward page-by-page data extraction process with the help of usual libraries such as requests, BeautifulSoup, time, and pandas. The crawler does not make use of any asynchronous processing or parallel requests.

3.1.1 Synchronous Architecture

Other than using concurrency, the crawler also uses a synchronous approach in crawling the pages, where pages are processed line by line. It promotes reliability and simplicity, which means that it will be easier to handle request retries, potential errors, and retain clear execution logic.

During the synchronous process, there are a few steps that have been used:

- The crawler starts off with the first page, generates the appropriate **request** URL, and issues an HTTP request to get the content.
- It waits for the HTTP response before going to the next operation. If the request fails, then a retry mechanism provides up to three times for trying before proceeding without the page.
- When successful, the crawler then picks up the HTML content, parses, extracts data fields of relevance (like car name, price, mileage, location, and other specifications) and writes out the output to a CSV file.
- The crawler only proceeds to the **next page** after completing the current one.

This sequential technique is in place to make sure that processing of every page is complete before the next request is made. Although not as fast as concurrent crawling, this approach provides better predictability and control, something that comes in handy, especially in websites that have the possibility of having request rate limits or are sensitive to high-frequency access.

3.1.2 Pagination

Pagination is done by iteration of page numbers starting from 1 up to a given maximum (max_page). Each of the page numbers is used to create a URL that leads to the particular results page. This occurs within the concurrent crawling function, whereby a ThreadPoolExecutor is implemented for sending concurrent requests for several pages.

```
def fetch_page_data(page):  
    url =  
    f"https://www.carlist.my/cars-for-sale/malaysia?page={page}"  
    ...
```

In this loop, the pages to crawl are generated, submitting tasks to fetch each of the page numbers.

```
with ThreadPoolExecutor(max_workers=8) as executor:  
    futures = [executor.submit(fetch_page_data, page) for  
               page in range(1, max_page + 1)]
```

3.1.3 Rate Limiting and Retry Logic

There is synchronous processing of the page requests within the fetch_page_data function. It employs a retry mechanism to deal with intermittent failures in requests, like connection errors or HTTP errors. The retry logic is the logic that tries the HTTP GET request 3 times before abandoning that page. Each failed attempt is logged.

```
retries = 0  
max_retries = 3  
  
while retries < max_retries:  
    try:  
        response = requests.get(url, headers=headers)  
        response.raise_for_status()  
        break
```

```
except requests.exceptions.RequestException as e:
    retries += 1
    print(f"Retry {retries}/{max_retries} for page
{page}: {e}")
    if retries == max_retries:
        print(f"Failed to fetch page {page} after
{max_retries} retries.")
```

3.1.4 Synchronous Data Fetch and Parse in Each Thread

Although other pages are also fetched in parallel, each page received is processed completely and synchronously by each thread.

The thread starts by forming the URL of the page assigned. It then makes an HTTP request to retrieve the content of the page, which employs a retry mechanism which will retry the request up to three times if failures occur. When a positive reply is received, a parsing of the HTML content is conducted through the use of BeautifulSoup. Inside the HTML, the crawler finds a `

```

row = {
    "Brand": car["brand"]["name"],
    "Model": car["model"]["name"],
    "Price": car["price"]["amount"],
    "Mileage": car["mileage"]["value"],
    "Transmission": car["transmission"]["name"],
    "Location": car["dealer"]["location"]["state"],
    # other fields as present
}
writer.writerow(row)
total_rows += 1

```

3.2 Number of records collected

After running the full crawl job on all paginated car listings on [Carlist.my](https://www.carlist.my), the crawler was able to scrape 175,545 individual car listings and approximately more than 2000 pages crawled. Each record is formed by the structured data attributes that are critical for downstream analysis and optimisation:

Table 3: Attributes collected from web scraping

Data	Description
car_name	The car's title or name on the listing is defensible
price	Asking price for the car
location	City/ locality where the car is sold
region	Malaysian state or territory
brand	Car manufacturer
model	Car model type
year	Manufacturing year
mileage	Driven distance specifically in kilometres (km)
fuel_type	Fuel type, such as petrol or diesel
color	The body colour of the car
body_type	Car type, such as hatchback, sedan, etc
seating_capacity	The seating number in the car

condition	Vehicle condition, such as used, new, etc
image	Vehicle image link to identify the car
description	Seller's description of the car
url	Full list link.

All listings were saved in the form of a .csv file (raw_carlist_data.csv) and were cleaned and transformed during the preprocessing stage.

3.3 Ethical considerations

Compliance with ethics was essential to the process of scraping. These were the measures which were taken to ensure responsible and respectful data collection

3.3.1 Respecting Server Load

- The scraper implements request throttling and connection limitation to avoid overstraining the server.
- Bulk data fetching or flooding behaviour was not implemented, while scraping was done purposefully in small batches.

3.3.2 Data Scope and Privacy

- Only public listings were used, such as no login, authentication tokens, or hidden APIs were used.
- No scraping of sensitive or personal information like users' emails, phone numbers, and IP addresses.
- Listing data can be used freely in the academic context by everyone visiting the site and is deemed appropriate.

3.3.3 Legal Review and Transparency

The crawler was set up with a custom `User-Agent` header, which indicated clearly what the purpose of the crawler was, which is an academic research project, so communicate transparency and not be commercial. On the onset of data collection, the [Carlist.my](#)'s Terms of Use were checked to ensure that they are compliant, and no data extracted from the sources was used other than for educational purposes in the course. Moreover, the dataset was not transferred beyond the project group of the project, nor was it provided to the public community to maintain confidentiality and stick to ethical data usage rules, as evidenced by

the approval of the university.

The custom User-Agent header:

```
headers = {  
    "User-Agent": "Mozilla/5.0 (compatible;  
AcademicCrawler/1.0; +https://github.com/datadrillers)"  
}
```

The header is passed in every HTTP request via the request library:

```
response = requests.get(url, headers=headers)
```

Before starting the crawl, the team studied the Terms of Use and policies of the [Carlist.my](https://carlist.my) in order to make sure that scraping activity does not violate any clear regulations. Since the data was publicly available and no barriers towards non-commercial, academic research were discovered, the project followed the ethical data collection standards. Moreover, any use of the data kept extending to the university project group only, and was staying within the limits of the academic integrity guidelines determined by the institution.

4.0 Data Processing

A systematic [data cleaning](#) and transformation procedure was used to make sure the Malaysian car listings dataset was appropriate for using optimisation techniques. It was important to carry out this step in order to ensure data quality, consistency, and dependability because optimisation approaches require clean, well-organised input. The next section explains how formatting issues, errors, and inconsistencies were resolved in order to get the dataset ready for optimisation-based methods.

4.1 Cleaning methods

The raw dataset included 175,545 rows and 17 columns at first; however, following methodical cleaning, several types of problems were found and fixed:

4.1.1 Handling missing values

Three columns had missing values:

- location: 143 missing entries
- fuel_type: 39 missing entries
- body_type: 31 missing entries

All missing values that we encountered in our raw data were imputed with the placeholder "unknown" instead of dropping these rows to maintain the integrity of the data while clearly highlighting uncertain data. After this replacement, no missing values remained in the dataset.

4.1.2 Filtering invalid URLs

Each car listing has a unique URL column that identifies it. Null-valued rows and entries that did not begin with "http" were eliminated. There was only one such row discovered and removed. This made sure there was a valid link in each listing.

4.1.3 Duplicate check

Duplicate listings were checked using a combination of columns (car_name, price, mileage, and url). No duplicate entries were found in the dataset.

4.1.4 Fixing encoding issues

Mojibake—text corruption existed in the description field as a result of the encoding errors. A custom function was used to re-encode these values from Latin-1 to UTF-8 in order to restore back readable text.

4.1.5 Outlier removal

Listings with irrational values, such as negative mileage or prices, were discarded to preserve data quality. These rows were considered invalid and removed during processing.

4.1.6 Removing redundant columns

The currency column was dropped since all entries contained "MYR", making it irrelevant for analysis.

4.1.7 Safe editing practices

All cleaning steps were conducted on a copy of the original set using .copy to prevent unintended changes or side effects during processing.

4.2 Transformation and formatting

After the cleaning of the data process, transformation steps were applied to enhance consistency and prepare the dataset for the optimisation process.

4.2.1 Text normalisation

All string-type columns were converted to lowercase, and their leading or trailing whitespaces were trimmed. So, inconsistencies were avoided by treating "Used " and "used" as different values.

4.2.2 Standardising condition labels

Under the “condition” column, inconsistent labels were found, such as “usedcondition” or “newcondition”. These were mapped to standardised values: "used", "refurbished", and "new".

4.2.3 Column renaming and reordering

Columns were renamed for clarity and consistency. For example, "fuel type" became "fuel_type", and "price (myr)" became "price". Columns were also rearranged to group related features such as car specs, location, and condition.

4.2.4 Data type conversion

In the dataset, fields which are numeric in nature, i.e. price, year, mileage, seating_capacity were explicitly cast to proper numeric types. Based on their severity, invalid or non-numeric values were either eliminated or replaced.

4.2.5 Resetting index

After all transformations and filters, the dataset index was reset to maintain continuity and support accurate referencing.

4.3 Data structure

The cleaned dataset has 16 columns and 175,545 rows after processing. A unique car listing is represented by each row, which has standardised, analytically ready fields:

Column Name	Description
car_name	Name of the listed vehicle
price	Price in Malaysian Ringgit (MYR)
location	City or area where the car is listed
region	State or region in Malaysia
brand	Car manufacturer (e.g., Toyota, Honda)
model	Specific model name (e.g., City, Vios)
year	Manufacturing year of the car
mileage	Total distance driven (km)
fuel_type	Type of fuel used (e.g., petrol, diesel)
color	Exterior color of the car
body_type	Type of vehicle body (e.g., sedan, hatchback)
seating_capacity	Number of seats in the vehicle
condition	Vehicle condition (e.g., new, used, refurbished)
image	URL or file path to the car's image
description	Seller's description of the vehicle
url	Web link to the listing

Table 4 : Metadata of cleaned dataset

The cleaned dataset was saved as [cleaned_data.csv](#) and suitable for the optimisation process.

5.0 Optimisation Techniques

5.1 Methods used

In this project, we explored three different data processing strategies that are **Pandas**, **Polars** and **DuckDB** to compare and evaluate performance for analysing a big dataset of automobile listings. We sought to measure how each of these options performs in terms of speed, CPU/memory usage and data throughput and determine the most efficient option for big data workloads.



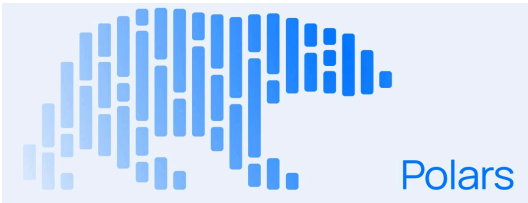
No	Tools	Description
1.	Pandas 	Pandas is a powerful data manipulation library, but it is time-consuming and memory-intensive for large datasets. Optimisation techniques aimed at making use of efficient data loading and processing by making use of vectorised operations and reducing the utilisation of memory.
2.	DuckDB 	DuckDB is an in-process OLAP database SQL with high performance that renders queries of large sets of data efficiently. We used in-memory database processing to reduce I/O and speed up query execution.
3.	Polars 	Polars is a parallelised and performance-focused DataFrame library designed for fast processing of large datasets. It leverages the use of a Rust-based backend to speed up operations.

Table 5 : Library methods used for performance analysis

5.2 Code overview or pseudocode of techniques applied

Optimisation techniques used with Pandas, DuckDB and Polars for processing the data are discussed here. These techniques are used to speed up the computation and make it memory-efficient.

5.2.1 Pandas

Pseudocode for pandas library:

```

START

INPUT:
  1. cleaned_data.csv file

PROCESS:
  1. Read the CSV file using Pandas
  2. Compute:
    a) Total number of rows
    b) Mean price
    c) Mean mileage
    d) Most common region
  3. Track CPU, memory, and processing time
    a) Save metrics to pandas_optimized.csv

OUTPUT:
  1. pandas_optimized.csv with performance metrics
  2. metrics summary

END

```

5.2.2 DuckDB

Pseudocode for DuckDB library:

```

START

INPUT:
  1. cleaned_data.csv file

PROCESS:
  1. Start tracking time, CPU and memory usage
  2. Connect to DuckDB (in memory)
  3. Use a single query to:
    a) Compute total rows, average price, average
       mileage
    b) Find most common region using window functions
  4. End performance tracking
  5. Save metrics to duckdb_optimized.csv

OUTPUT:
  1. duckdb_optimized.csv with performance metrics
  2. metrics summary

END

```

5.2.3 Polars

Pseudocode for polars library:

```
START

INPUT:
  1. cleaned_data.csv file

PROCESS:
  1. Track time, CPU and memory usage
  2. Read CSV using Polars with error handling
  3. Compute:
    a) Total number of rows
    b) Mean price
    c) Mean mileage
    d) Most common region
  4. End performance tracking
  5. Save metrics to polars_optimized.csv

OUTPUT:
  1. polars_optimized.csv with cleaned data
  2. metrics summary

END
```

6.0 Performance Evaluation

6.1 Before vs after optimisation

We initially chose [Pandas](#) for our data processing pipeline since it is popular for its user-friendly nature and flexibility. However, Pandas only runs in a single thread, which results in low performance when handling large-scale datasets.

We included two more optimisation techniques to improve performance:

- [DuckDB](#): A lightweight in-process SQL engine that has been optimised for analytical queries and columnar data processing is lightweight.
- [Polars](#): A high-performance DataFrame library written in Rust which has multi-threaded execution and lazy evaluation capabilities.

6.1.1 Controlled Testing Environment

In order to have a reliable and unbiased comparison, the following variables were kept constant throughout all test runs:

- Hardware: All the tests were run on the same machine with an Intel i7 CPU (8 cores, 16 threads) and 16GB of RAM.
- Dataset: All the stages of optimisation used a structured CSV file, having 16 columns and 175,545 rows.
- Codebase: Same query logic, cleaning operations, and processing steps were used for the Pandas, DuckDB, and Polars, to ensure functional consistency.
- Execution Consistency: Each of the methods was tested three times. The average values were calculated to minimise variance and increase reliability in results.

6.1.2 Logging & Metrics Collection

We used a few Python libraries to measure performance throughout all the optimisation stages:

- time: to track duration for total execution
- psutil: to monitor CPU and memory usage at the process level
- os.getpid(): to isolate the current process
- json: to export the captured performance metrics

Generally, the system has significantly improved in terms of performance after applying the optimisation techniques, with a considerable reduction in the processing time and resource usage.

6.2 Comparative Analysis of Performance Metrics

We performed three test runs for each stage, consisting of Pandas, DuckDB, and Polars and recorded four key performance metrics like total processing time, CPU usage, memory usage, and throughput (records processed per second) in order to better understand the impacts of each optimisation technique. The results for each run are shown in the tables below with the computed averages. These measurements make it easier to compare how well each method worked in similar conditions.

Table 6: Total Processing Time (seconds)

Optimization Stage	Run 1	Run 2	Run 3	Average
Pandas Optimization	5.35	3.49	3.44	4.09
DuckDB Optimization	0.45	0.89	0.43	0.59
Polars Optimization	0.43	0.26	0.61	0.43

Table 7: CPU Usage (%)

Optimization Stage	Run 1	Run 2	Run 3	Average
Pandas Optimization	14.67	16.67	3.6	11.65
DuckDB Optimization	19.8	50	46.2	38.67
Polars Optimization	83.1	36.2	14.8	44.7

Table 8: Memory Usage (MB)

Optimization Stage	Run 1	Run 2	Run 3	Average
Pandas Optimization	102.34	119.98	125.8	116.04
DuckDB Optimization	65.79	98.83	81.64	82.09
Polars Optimization	28.63	3.81	129.63	54.02

Table 9: Throughput (records/second)

Optimization Stage	Run 1	Run 2	Run 3	Average
Pandas Optimization	32827.06	50356.60	51003.49	44729.0
DuckDB Optimization	390337.6	196502.90	406446.43	331095.6
Polars Optimization	412467.7	676460.87	289688.39	459539.0

6.3 Charts and graphs

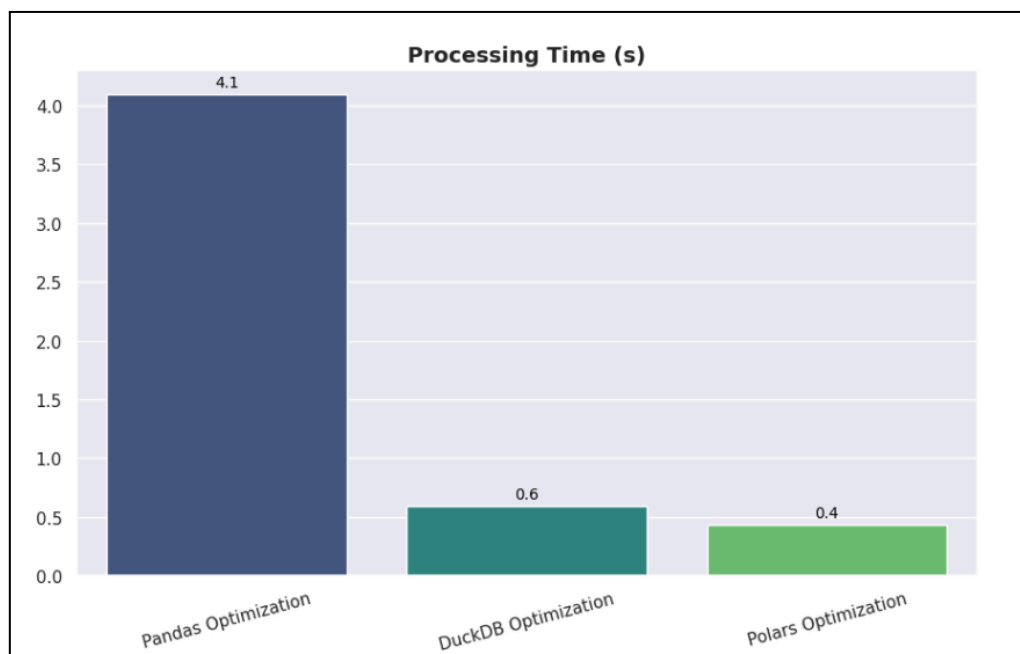


Figure 3 : Bar chart for Processing Time (s)

This graph shows a significant reduction in processing time after optimisation. While Pandas took the longest to complete the task, both DuckDB and Polars drastically reduced execution time to under 1 second. Polars had a minimum average time, which denotes the best data handling capability among the rest of the three.

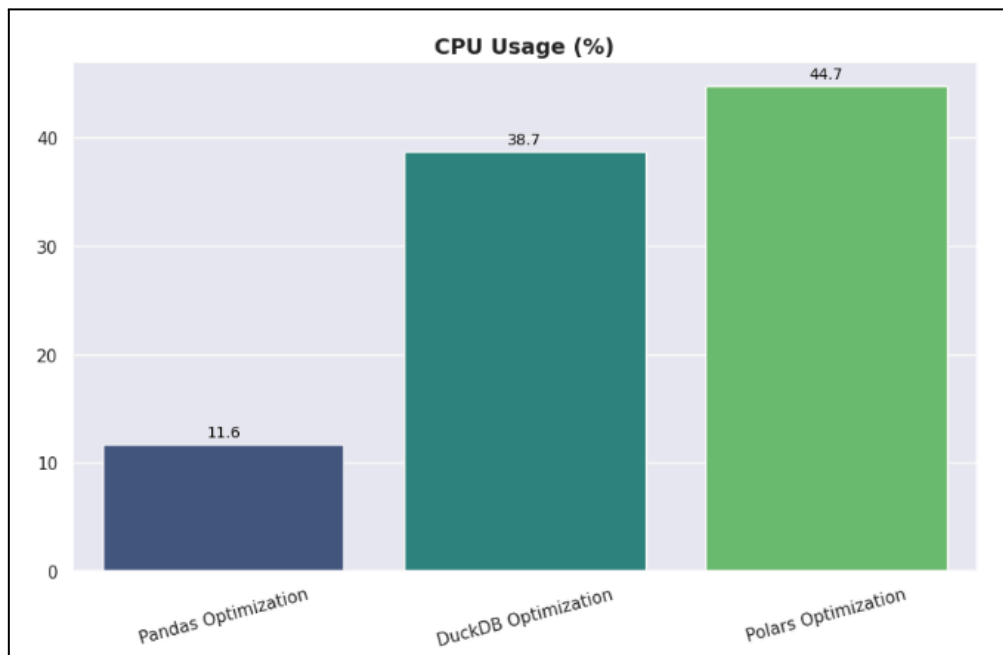


Figure 4 : Bar chart for CPU Usage (%)

Based on the bar chart above, Pandas had the lowest CPU utilisation, suggesting that it did not make full use of available cores. On the other hand, DuckDB and particularly Polars demonstrated increased usage of the CPU, which indicated the ability to apply multicore processing. Polars had the highest average CPU usage, whereas this metric is consistent with its multi-threaded architecture.

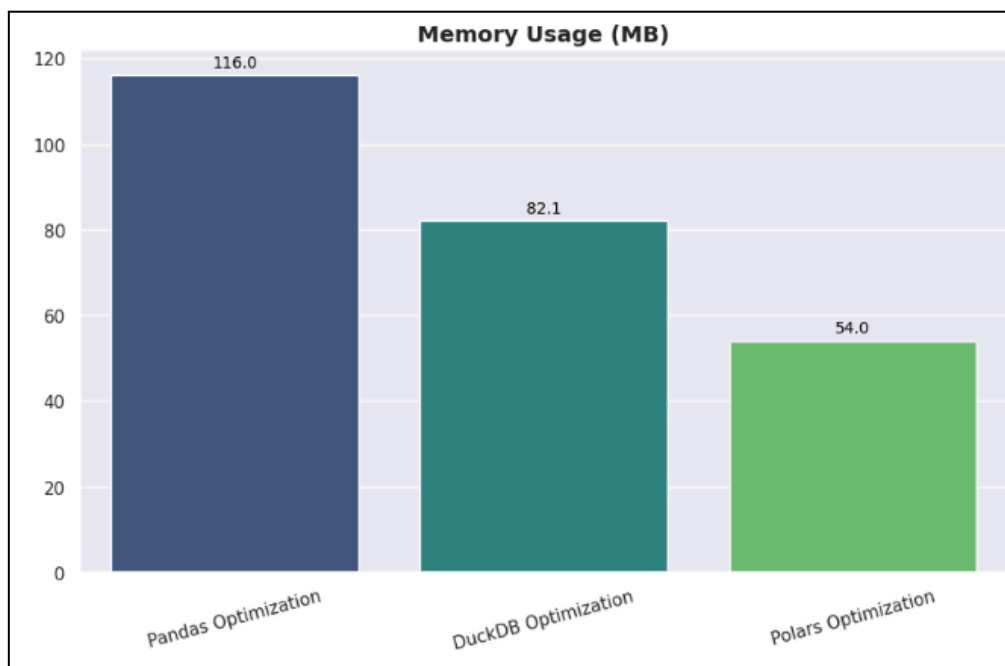


Figure 5 : Bar chart for Memory Usage (MB)

in-memory, non-optimised architecture. Both DuckDB and Polars utilised memory with more efficiency, whereby Polars averaged the lowest utilisation rate, even though it had the highest processing speed. This balance of speed and memory efficiency is the performance trump card of Polars.

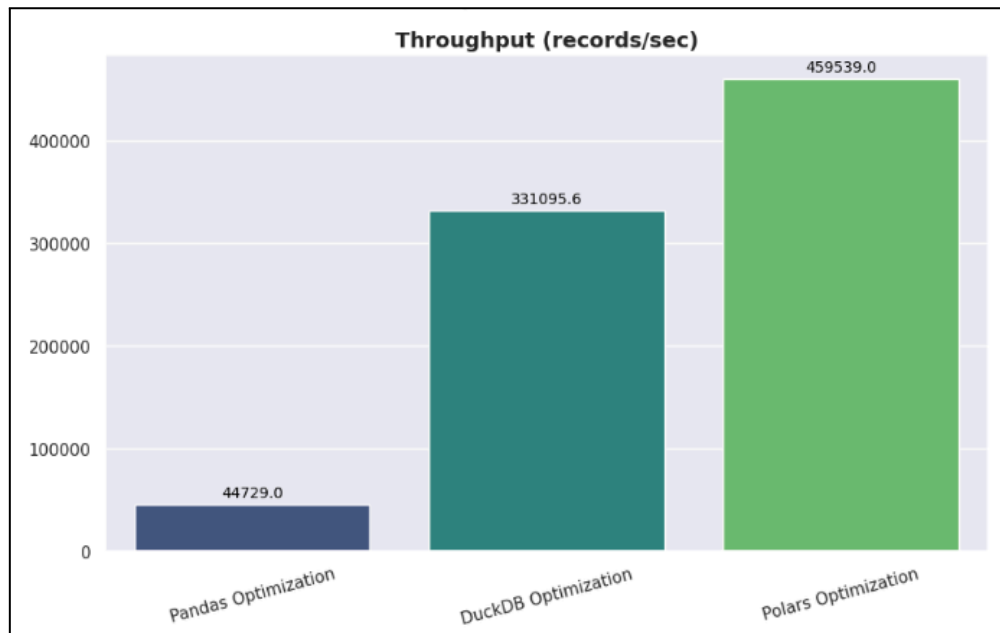


Figure 6 : Bar chart for Throughput (records/sec)

According to the bar chart above, throughput increased dramatically with optimisation. Pandas ran through about 44,700 records per second, whereas DuckDB reached more than 330,000, and Polars reached over 450,000. Polars had the maximum throughput, showing greater efficiency for working with massive datasets.

Summary

Overall, the graphs depict clear evidence that the adoption of such high-performance libraries like DuckDB and Polars enhanced the processing capabilities of the system to a greater extent. Although both techniques outperformed the baseline Pandas approach, Polars showed the best balance between the time, CPU consumption and memory efficiency.

7.0 Challenges & Limitations

7.1 What didn't go as planned

1. Long data scraping duration during the early phase

Initially, we ran into big issues that influenced how much time our project would require. Because we were new to web scraping and had a weak internet signal, scraping the needed data for two days was necessary in the early phases. Collecting all the records in the beginning was a slow process, and it took approximately three hours to complete.

Luckily, the experiences we gained enabled us to make progress that eases our scraping. Optimising our programme and running it with multiple threads allowed us to speed up the process and finish scraping in less than one hour. As a result, I understood that regularly updating skills and approaches helps deal with surprises, which resulted in an improved system for data collection.

2. Sudden changes and restrictions by carlists.my

After data optimisations were done, a week passed, and all of our scraping code suddenly failed to work. The website's new settings made it so that our previous approaches were useless. We were really upset by this, since we had already achieved a lot. We discussed this problem with Dr. Shahizan, who gave us useful advice to address this issue.

7.2 Any limitations of your solution

One significant limitation we encountered was that we were not able to extract the 175,545 records again using the same source code. Because of the sudden steps enforced by carlist.my, our current approaches did not work, so we could not retrieve the same dataset again. Because of this problem, we were not able to verify our findings and could not use the data for additional analysis.

Furthermore, the rapid changes in website restrictions revealed the weaknesses of using data corporations. It highlighted the need to develop scraping tools that are able to adjust to any upcoming changes in web pages. Because of this, we have planned to try new data collection techniques to avoid similar obstacles on other assignments.

8.0 Conclusion & Future Work

8.1 Summary of findings

This project successfully implemented the application of high-performance computing (HPC) techniques to optimise large-scale web crawling, such as data extraction and processing. Aiming for the car listings on Carlist.my, a web crawler was built with a strong, morally acceptable web scraper, which was functional in scraping more than 175,000 unique records of vehicles. The project workflow included:

- Synchronised multithreaded crawling, allowing reliability of data and adherence to the rate limits.
- In-depth data cleaning and transformation procedures to create a high-quality structured dataset.
- The use of three separate data processing libraries, which are Pandas, DuckDB, and Polars, in evaluating their performance in processing huge datasets.

The data collection process was to be ethically compliant. Publicly viewable pages only were scraped without any use of authentication tokens or access rights to private information. Rate limiting and request throttling mechanisms had been implemented, as well as a custom User-Agent header, which could clearly convey that it was an academic research initiative. Before its execution, we had checked the terms of use of Carlist.my to make sure that our scraping process did not break any legal or ethical rules, and we did not use the collected data for anything outside the academic limits of our project.

The comparative performance analysis showed:

- Polars was faster when it comes to processing speed, throughput, and memory efficiency, being multi-threaded and a Rust-based backend.
- Similarly, DuckDB exhibited great performance improvement with its in-memory SQL processing engine, provides powerful querying abilities and a moderate consumption of resources.
- Although pandas are easy to use, they turned out to be the least efficient when under large data loads because of their poor single-threaded design.

These results serve as indications of the major advantages of employing high-performance data frameworks within the large-scale processing of data. Both DuckDB and Polars showed undeniable superiority over Pandas, especially related to speed and resource efficiency.

Among them, Polars became the most efficient solution, with fast processing, low memory usage, and high CPU usage. This assessment further emphasises the significance of choosing proper tools and optimisation tactics for working with big data workloads, particularly in conditions where performance, scalability, and resources are a concern.

8.2 What could be improved

Although the project managed to reach its key objectives, certain limitations and future improvement areas emerged as well:

1. Website Dependency & Fragility

Once scraping was done, Carlist.my added stricter scraping protections, making the crawler ineffective. Future work should include:

- Creating adaptive crawler strategies using headless browsers or such services as Selenium or Scrapy with proxy rotation.
- Adding dynamic User Agent spoofing and throttling requests according to the real-time server response.

2. Scalability Testing on Distributed Systems

- The present solution was benchmarked on one machine. Testing on cloud environments or in distributed environments (for example, if using Spark or Dask) would offer insight into how the system can scale to higher degrees.

3. There is no real-time or incremental scraping.

The existing system aims at a one-time collection of data. A future version could include:

- Incremental scraping mechanisms that periodically change listings.
- Change detection logic to return new and updated data only, which saves load and redundancy.

9.0 References

9.1 References List

Asplin, M. (2025). Vertical scaling for big data analytics and processing - A case study (Dissertation). Retrieved from <https://urn.kb.se/resolve?urn=urn:nbn:se:mdh:diva-70150>

Khazanchi, A. (2023). Faster Reading with DuckDB and Arrow Flight on Hopsworks : Benchmark and Performance Evaluation of Offline Feature Stores (Dissertation, KTH Royal Institute of Technology). Retrieved from <https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-337297>

Mozzillo, A., Zecchini, L., Gagliardelli, L., Aslam, A., Bergamaschi, S., & Simonini, G. (2023). Evaluation of Dataframe Libraries for Data Preparation on a Single Machine. *ArXiv (Cornell University)*, (337-349). Proceedings 28th International Conference on Extending Database Technology. <https://doi.org/10.48550/arxiv.2312.11122>

10.0 Appendices

10.1 Sample code snippets

Data Scraping:

```
[ ] import requests
import time
import os
from bs4 import BeautifulSoup
import json
import csv
import concurrent.futures
from bs4 import BeautifulSoup

[ ] base_url = 'https://www.carlist.my/cars-for-sale/malaysia'

[ ] page = requests.get(base_url)

[ ] BeautifulSoup(page.text, 'html')

[ ] page_size = 50 # Number of cars per page
max_retries = 3 # Retry limit for failed requests
max_workers = 8 # Max concurrent workers to scrape pages

total_rows = 0
csv_file = 'carlists_data.csv'

# Create CSV with headers
with open(csv_file, mode='w', newline='', encoding='utf-8') as file:
    writer = csv.writer(file)
    writer.writerow([
        'Car Name', 'Price (MYR)', 'Currency', 'Location', 'Region',
        'Brand', 'Model', 'Year', 'Mileage', 'Fuel Type', 'Color',
        'Body Type', 'Seating Capacity', 'Condition',
        'Image', 'Description', 'URL',
    ])

# Define function to fetch a page and handle retries
def fetch_page(url, retries=0):
    try:
        response = requests.get(url, timeout=10)
        response.raise_for_status() # Raise exception for 4xx/5xx HTTP status codes
        return response.text
    except requests.exceptions.RequestException as e:
        if retries < max_retries:
            print(f"Retrying {url}... Attempt {retries + 1}")
            return fetch_page(url, retries + 1)
        else:
            print(f"Failed to fetch {url}: {e}")
            return None

# Define function to parse car data from the page
def parse_car_data(listings):
    car_data = []
    for listing in listings:
        try:
            car_info = json.loads(listing.string.strip())
            if isinstance(car_info, list):
                for item in car_info:
                    if 'itemListElement' in item:
                        for car in item['itemListElement']:
                            car_name = f"{car['item']['brand']} {car['item']['name']} {car['item']['model']}"
                            car_price = car['item']['offers']['price']
                            car_url = car['item']['url']
                            car_location = car['item']['offers']['seller']['homeLocation']['address']['addressLocality']
                            car_region = car['item']['offers']['seller']['homeLocation']['address']['addressRegion']
                            car_brand = car['item']['brand']
                            car_model = car['item']['model']
                            car_year = car['item']['vehicleModelDate']
                            car_mileage = car['item']['mileageFromOdometer']['value']
                            car_fuel_type = car['item']['fuelType']
                            car_color = car['item']['color']
                            car_body_type = car['item']['bodyType']
                            car_seating_capacity = car['item']['seatingCapacity']
                            condition = car['item']['itemCondition'].split('/')[0]
                            image = car['item']['image'][0] if isinstance(car['item']['image'], list) else car['item']['image']
                            description = car['item']['description'].strip()
                            price_currency = car['item']['offers']['priceCurrency']
```



```

        price_currency = car['item']['offers']['priceCurrency']

        car_data.append([
            car_name, car_price, price_currency, car_location, car_region,
            car_brand, car_model, car_year, car_mileage, car_fuel_type,
            car_color, car_body_type, car_seating_capacity,
            condition, image, description, car_url,
        ])
    except Exception as e:
        print(f"Error parsing car data: {e}")
    return car_data

# Define function to scrape a page
def scrape_page(page):
    url = f'{base_url}?page_size={page_size}&page_number={page}'
    print(f"Scraping page {page}...")
    html = fetch_page(url)
    if html:
        soup = BeautifulSoup(html, 'html.parser')
        listings = soup.find_all('script', type='application/ld+json')
        if listings:
            car_data = parse_car_data(listings)
            return car_data
    return []

# Function to save car data to CSV
def save_to_csv(car_data):
    global total_rows
    with open(csv_file, mode='a', newline='', encoding='utf-8') as file:
        writer = csv.writer(file)

        for row in car_data:
            writer.writerow(row)
            total_rows += 1

# Function to detect the total number of pages
def get_total_pages():
    url = f'{base_url}?page_size={page_size}&page_number=1'
    response = requests.get(url)
    soup = BeautifulSoup(response.text, 'html.parser')
    last_page_tag = soup.select_one("ul.pagination li:last-child a")
    if last_page_tag and last_page_tag.get("href"):
        try:
            last_page = int(last_page_tag.get("href").split("page_number=")[-1])
            return last_page
        except Exception:
            pass
    return 100 # fallback if not detected

# Main function to manage the scraping process
def main():
    start_time = time.time()

    total_pages = get_total_pages() # Dynamically fetch the total number of pages

    # Loop through pages in parallel using ThreadPoolExecutor
    with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as executor:
        future_to_page = {executor.submit(scrape_page, page): page for page in range(1, total_pages + 1)}
        for future in concurrent.futures.as_completed(future_to_page):
            page = future_to_page[future]
            try:
                car_data = future.result()

                if car_data:
                    save_to_csv(car_data)
                    print(f"Page {page} scraped and saved, total rows: {total_rows}")
            except Exception as e:
                print(f"Error scraping page {page}: {e}")

    end_time = time.time()
    execution_time = end_time - start_time
    file_size_kb = os.path.getsize(csv_file) / 1024

    print(f"\nDone scraping.")
    print(f"Total rows: {total_rows}")
    print(f"Time taken: {execution_time:.2f} seconds")
    print(f"File size: {file_size_kb:.2f} KB")

# Ensure the main function runs when executed directly
if __name__ == "__main__":
    main()

```

Data Cleaning:

```

[ ] import pandas as pd

# Load CSV with headers (since your file already has headers)
df_raw = pd.read_csv("raw_data.csv", encoding='utf-8', low_memory=False)

# Optionally, you can standardize the column names to lowercase to ensure consistency
df_raw.columns = df_raw.columns.str.lower()

# --- Initial Data Inspection ---
# Print initial number of rows and columns
print(f"[INITIAL DATA SHAPE] -> Rows: {df_raw.shape[0]}, Columns: {df_raw.shape[1]}")

# List columns before cleaning
print("\n[INITIAL COLUMN NAMES] ->")
print(df_raw.columns.values)

[ ] # Attempt to fix mojibake in description
def fix_encoding(text):
    try:
        return text.encode('latin1').decode('utf-8')
    except:
        return text # fallback to original if error

df_raw['description'] = df_raw['description'].apply(fix_encoding)

```

```
[ ] # Drop the 'currency' column
df_raw = df_raw.drop(columns=['currency'])

# Make a working copy
df = df_raw.copy()

[ ] # Check for duplicates
dups = df.duplicated(subset=["car name", "price (myr)", "location", "year", "mileage", "url"])
print(f"\n[NUMBER OF DUPLICATES] -> {dups.sum()}")

# Check missing values
print("\n[MISSING VALUES PER COLUMN]:")
print(df.isna().sum())

[ ] # --- Cleaning ---
# Standardize string columns: lowercase + strip
for col in df.select_dtypes(include='object').columns:
    df[col] = df[col].astype(str).str.strip().str.lower()

# Fix 'condition' values
df['condition'] = df['condition'].replace([
    'usedcondition': 'used',
    'refurbishedcondition': 'refurbished',
    'newcondition': 'new'
])

# Fill missing text columns with "unknown"
text_cols = df.select_dtypes(include='object').columns
df[text_cols] = df[text_cols].fillna('unknown')

# Re-check missing values
print("\n[MISSING VALUES PER COLUMN AFTER REPLACE WITH UNKNOWN]:")
print(df.isna().sum())

[ ] # Filter out invalid or missing URLs
df = df[df['url'].notna() & df['url'].str.startswith("http")]
print(f"\n[VALID URL ROWS ONLY] -> {df.shape}")

[ ] # Drop duplicates
df = df.drop_duplicates()

# Reset index for clean CSV output
df = df.reset_index(drop=True)

[ ] # Convert 'price (myr)', 'year', 'mileage' to numeric values (if possible)
df['price (myr)'] = pd.to_numeric(df['price (myr)'], errors='coerce')
df['year'] = pd.to_numeric(df['year'], errors='coerce')
df['mileage'] = pd.to_numeric(df['mileage'], errors='coerce')

# Handle missing numeric columns by filling with median or dropping
df['price (myr)'] = df['price (myr)'].fillna(df['price (myr)'].median())
df['mileage'] = df['mileage'].fillna(df['mileage'].median())
df['year'] = df['year'].fillna(df['year'].mode()[0])

# Handle outliers: Remove rows where price or mileage is unreasonable (e.g., negative or extreme values)
df = df[(df['price (myr)'] >= 0) & (df['mileage'] >= 0)]

# Ensure valid year (e.g., no future years)
current_year = pd.to_datetime('today').year
df = df[df['year'] <= current_year]

[ ] # Standardize and handle the "condition" column more robustly (already done in previous steps)
# Filter out rows with invalid 'condition' values if necessary
valid_conditions = ['used', 'new', 'refurbished']
df = df[df['condition'].isin(valid_conditions)]

# --- Final Inspection ---
# Re-check the data shape after cleaning
print("\n[CLEANED DATA SHAPE] ->", df.shape)

print("\n[INITIAL COLUMN NAMES] ->")
print(df.columns.values)

[ ] # --- Save Cleaned File ---
df.to_csv("cleaned_data.csv", index=False)
print("\n✅ Cleaned dataset saved as 'cleaned_data.csv'")
```

Before Optimization (Pandas):

```
import time
import os
import psutil
import pandas as pd

# ===== Start Performance Tracking =====
start_time = time.time()
process = psutil.Process(os.getpid())
cpu_start = psutil.cpu_percent(interval=None)
memory_start = process.memory_info().rss / (1024 * 1024) # in MB

# ===== Read CSV with Pandas =====
df = pd.read_csv('cleaned_data.csv', encoding='utf-8')

# Drop rows with missing or invalid data
df.dropna(subset=['price (myr)', 'mileage', 'region'], inplace=True)

# Convert columns to appropriate numeric types
df['price (myr)'] = pd.to_numeric(df['price (myr)'], errors='coerce')
df['mileage'] = pd.to_numeric(df['mileage'], errors='coerce')

# Filter out rows where conversion failed
df = df.dropna(subset=['price (myr)', 'mileage'])

# ===== Compute Metrics =====
total_rows = len(df)
mean_price = df['price (myr)'].mean()
mean_mileage = df['mileage'].mean()
most_common_region = df['region'].mode()[0] if not df['region'].empty else "N/A"

# ===== End Performance Tracking =====
end_time = time.time()
cpu_end = psutil.cpu_percent(interval=None)
memory_end = process.memory_info().rss / (1024 * 1024) # in MB
execution_time = end_time - start_time

# ===== Save Processed DataFrame as CSV =====
df.to_csv('pandas_optimized.csv', index=False)

# ===== Prepare Metrics for CSV =====
metrics = {
    "Optimization Stage": "Pandas Optimization",
    "Total Rows": total_rows,
    "Total Processing Time (seconds)": execution_time,
    "CPU Usage (%)": cpu_end - cpu_start,
    "Memory Usage (MB)": memory_end - memory_start,
    "Throughput (records/second)": total_rows / execution_time,
    "Data Processed (rows)": total_rows,
    "Time Taken (seconds)": execution_time,
    "Records per second": total_rows / execution_time
}

# Convert metrics to DataFrame and save as CSV
metrics_df = pd.DataFrame([metrics])
metrics_df.to_csv('pandas_optimized.csv', index=False)

# ===== Print Metrics =====
print("\n🏁 Pandas Optimization Complete")
print(f"📊 Total Rows: {total_rows}")
print(f"⌚ Total Processing Time: {execution_time:.2f} seconds")
print(f"💻 CPU Usage: {cpu_end - cpu_start:.2f}%")
print(f"📀 Memory Usage: {memory_end - memory_start:.2f} MB")
print(f"🚀 Throughput: {total_rows / execution_time:.2f} records/second")
print(f"💰 Mean Price (MYR): RM {mean_price:.2f}")
print(f"📏 Mean Mileage: {mean_mileage:.2f} km")
print(f"📍 Most Common Region: {most_common_region}")

print("\n💾 Metrics saved to pandas_optimized.csv")
```

After Optimization (Polars and DuckDB):

```
[ ] import polars as pl
import time
import psutil
import os
import json

def run_polars_optimization():
    start_time = time.time()
    process = psutil.Process(os.getpid())
    cpu_start = psutil.cpu_percent(interval=None)
    memory_start = process.memory_info().rss / (1024 * 1024) # MB

    # Read CSV with error handling for problematic columns
    df = pl.read_csv(
        "cleaned_data.csv",
        try_parse_dates=True,
        ignore_errors=True, # Skip rows with parsing errors
        null_values=["-", "NA", "N/A", ""] # Treat these as null values
    )

    # Alternatively, specify dtypes for problematic columns
    # df = pl.read_csv(
    #     "cleaned_data.csv",
    #     dtypes={
    #         "seating capacity": pl.Utf8 # Read as string first
    #     }
    # )

    # If needed, you can then clean the problematic column
    # df = df.with_columns(
```

```
[ ] row_count = df.shape[0]
mean_price = df["price (myr)"].mean()
mean_mileage = df["mileage"].mean()

# Handle mode safely in case all values are null
region_mode = df["region"].mode()
most_common_region = region_mode[0] if len(region_mode) > 0 else "N/A"

end_time = time.time()
execution_time = end_time - start_time
cpu_end = psutil.cpu_percent(interval=None)
memory_end = process.memory_info().rss / (1024 * 1024) # MB

metrics = {
    "Optimization Stage": "Polars Optimization",
    "Total Rows": row_count,
    "Total Processing Time (seconds)": execution_time,
    "CPU Usage (%)": cpu_end - cpu_start,
    "Memory Usage (MB)": memory_end - memory_start,
    "Throughput (records/second)": row_count / execution_time,
    "Data Processed (rows)": row_count,
    "Time Taken (seconds)": execution_time,
    "Records per second": row_count / execution_time
}

print("\n✅ Polars Optimization Complete")
print(f"📊 Total Rows: {row_count}")
print(f"⌚ Total Processing Time: {execution_time:.2f} seconds")
print(f"💻 CPU Usage: {cpu_end - cpu_start:.2f}%")
print(f"📈 Memory Usage: {memory_end - memory_start:.2f} MB")

print(f"⚡ Throughput: {row_count / execution_time:.2f} records/second")
print(f"💰 Mean Price (MYR): RM {mean_price:.2f}")
print(f"🚗 Mean Mileage: {mean_mileage:.2f} km")
print(f"📍 Most Common Region: {most_common_region}")

return metrics

if __name__ == "__main__":
    metrics = run_polars_optimization()
    # Save metrics to a temporary JSON file
    with open("polars_metrics.json", "w") as f:
        json.dump(metrics, f)
```

```
[ ] import time
import os
import psutil
import pandas as pd
import json

def run_pandas_optimization():
    # ===== Warm-up Run =====
    _ = pd.read_csv('cleaned_data.csv').head(100) # Initialize libraries

    # ===== Start Performance Tracking =====
    start_time = time.time()
    process = psutil.Process(os.getpid())

    # Stable CPU measurement
    cpu_readings = [psutil.cpu_percent(interval=0.1) for _ in range(3)]
    cpu_start = sum(cpu_readings) / len(cpu_readings)

    memory_start = process.memory_info().rss / (1024 * 1024) # in MB

    # ===== Read and Process Data =====
    df = pd.read_csv('cleaned_data.csv', encoding='utf-8')

    # Data cleaning
    df.dropna(subset=['price (myr)', 'mileage', 'region'], inplace=True)
    df['price (myr)'] = pd.to_numeric(df['price (myr)'], errors='coerce')
    df['mileage'] = pd.to_numeric(df['mileage'], errors='coerce')
    df = df.dropna(subset=['price (myr)', 'mileage'])

    # ===== Compute Metrics =====
    total_rows = len(df)

    mean_price = df['price (myr)'].mean()
    mean_mileage = df['mileage'].mean()
    most_common_region = df['region'].mode()[0] if not df['region'].empty else "N/A"

    # ===== End Performance Tracking =====
    end_time = time.time()
    cpu_end = psutil.cpu_percent(interval=0.1)
    memory_end = process.memory_info().rss / (1024 * 1024)
    execution_time = end_time - start_time

    # ===== Prepare Results =====
    metrics = {
        "Optimization Stage": "Pandas Optimization",
        "Total Rows": total_rows,
        "Total Processing Time (seconds)": execution_time,
        "CPU Usage (%)": cpu_end - cpu_start,
        "Memory Usage (MB)": memory_end - memory_start,
        "Throughput (records/second)": total_rows / execution_time,
        "Data Processed (rows)": total_rows,
        "Time Taken (seconds)": execution_time,
        "Records per second": total_rows / execution_time
    }

    # ===== Print Results =====
    print("\n✅ Pandas Optimization Complete")
    print(f"📊 Total Rows: {total_rows}")

    print(f"⌚ Total Processing Time: {execution_time:.2f} seconds")
    print(f"💻 CPU Usage: {cpu_end - cpu_start:.2f}%")
    print(f"📈 Memory Usage: {memory_end - memory_start:.2f} MB")
    print(f"⚡ Throughput: {total_rows / execution_time:.2f} records/second")
    print(f"💰 Mean Price (MYR): RM {mean_price:.2f}")
    print(f"🚗 Mean Mileage: {mean_mileage:.2f} km")
    print(f"📍 Most Common Region: {most_common_region}")

    return metrics

if __name__ == "__main__":
    metrics = run_pandas_optimization()
    with open("pandas_metrics.json", "w") as f:
        json.dump(metrics, f)
    print("\n✅ Metrics saved to pandas_metrics.json")
```

```

import duckdb
import time
import os
import psutil
import json

def run_duckdb_optimization():
    # ===== Start Performance Tracking =====
    start_time = time.time()
    process = psutil.Process(os.getpid())
    cpu_start = psutil.cpu_percent(interval=None)
    memory_start = process.memory_info().rss / (1024 * 1024) # in MB

    # ===== Connect to DuckDB =====
    conn = duckdb.connect(database=':memory:')

    try:
        # ===== Execute Optimized Query =====
        result = conn.execute("""
            WITH car_data AS (
                SELECT
                    TRY_CAST("price (myr)" AS DOUBLE) AS price,
                    TRY_CAST(mileage AS DOUBLE) AS mileage,
                    TRIM(region) AS region
                FROM read_csv('cleaned_data.csv',
                    header=true,
                    auto_detect=true,
                    ignore_errors=true)
                WHERE "price (myr)" IS NOT NULL
            )
            SELECT
                COUNT(*) AS total_rows,
                AVG(price) AS mean_price,
                AVG(mileage) AS mean_mileage,
                FIRST(region ORDER BY region_count DESC) AS most_common_region
            FROM (
                SELECT
                    *,
                    COUNT(*) OVER (PARTITION BY region) AS region_count
                FROM car_data
            )
        """).fetchone()

        total_rows = result[0]
        mean_price = result[1]
        mean_mileage = result[2]
        most_common_region = result[3]

    # ===== End Performance Tracking =====
    end_time = time.time()
    cpu_end = psutil.cpu_percent(interval=None)
    memory_end = process.memory_info().rss / (1024 * 1024) # in MB
    execution_time = end_time - start_time

    # Prepare metrics dictionary
    metrics = {
        "Optimization Stage": "DuckDB Optimization",
        "Total Rows": total_rows,
        "Total Processing Time (seconds)": execution_time,
        "CPU Usage (%)": cpu_end - cpu_start,
        "Memory Usage (MB)": memory_end - memory_start,
        "Throughput (records/second)": total_rows / execution_time,
        "Data Processed (rows)": total_rows,
        "Time Taken (seconds)": execution_time,
        "Records per second": total_rows / execution_time
    }

    # ===== Print Metrics =====
    print("\n🦆 DuckDB (FireDucks) - After Optimization")
    print(f"📊 Total Rows: {total_rows}")
    print(f"⌚ Total Processing Time: {execution_time:.2f} seconds")
    print(f"🔥 CPU Usage: {cpu_end - cpu_start:.2f}%")
    print(f"💾 Memory Usage: {memory_end - memory_start:.2f} MB")
    print(f"🚀 Throughput: {total_rows / execution_time:.2f} records/second")
    print(f"💰 Mean Price (MYR): RM {mean_price:.2f}")
    print(f"📏 Mean Mileage: {mean_mileage:.2f} km")
    print(f"🏠 Most Common Region: {most_common_region}")

    return metrics

```

Evaluation:

```

[ ] import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import warnings
from collections import Counter

[ ] warnings.filterwarnings("ignore", category=FutureWarning)

[ ] df_pandas = pd.read_csv("pandas_optimized.csv")
df_duckdb = pd.read_csv("duckdb_optimized.csv")
df_polars = pd.read_csv("polars_optimized.csv")

```

```
[ ] df_pandas["Method"] = "Pandas Optimization"
df_duckdb["Method"] = "DuckDB Optimization"
df_polars["Method"] = "Polars Optimization"

df_combined = pd.concat([ df_pandas, df_duckdb, df_polars], ignore_index=True)

# Use consistent order
method_order = [ "Pandas Optimization", "DuckDB Optimization", "Polars Optimization"]
df_combined["Method"] = pd.Categorical(df_combined["Method"], categories=method_order, ordered=True)

[ ] sns.set(style="darkgrid", context="notebook")
colors = sns.color_palette("viridis", len(method_order))

[ ] metrics = {
    "Total Processing Time (seconds)": ("Processing Time (s)", "lower is better"),
    "CPU Usage (%)": ("CPU Usage (%)", "lower is better"),
    "Memory Usage (MB)": ("Memory Usage (MB)", "lower is better"),
    "Throughput (records/second)": ("Throughput (records/sec)", "higher is better")
}

[ ] fig, axes = plt.subplots(2, 2, figsize=(18, 12))
axes = axes.flatten()
best_optimization = {}

for i, (column, (title, preference)) in enumerate(metrics.items()):
    ax = axes[i]
    sns.barplot(
        data=df_combined,
        x="Method",
        y=column,
        hue="Method", # Add this line
        palette=colors,
        ax=ax,
        legend=False # Remove legend as we don't need it in this case
    )

    # Add value labels
    for container in ax.containers:
        ax.bar_label(container, fmt="%1f", padding=3, fontsize=10, color='black')

    ax.set_title(title, fontsize=14, fontweight='bold')
    ax.set_xlabel("")
    ax.set_ylabel("")
    ax.tick_params(axis='x', rotation=15)

    # Determine the best optimization for the current metric
    grouped = df_combined.groupby("Method", observed=False)[column].mean()
    if preference == "lower is better":
        best_method = grouped.idxmin()
    elif preference == "higher is better":
        best_method = grouped.idxmax()

    best_method = grouped.idxmax()
    best_optimization[title] = best_method

fig.suptitle("Performance Comparison Before vs After Optimization", fontsize=18, fontweight='bold', y=1.02)
plt.tight_layout()
plt.show()

[ ] for metric, best_method in best_optimization.items():
    print(f"For {metric}: {best_method}")

[ ] overall_best = Counter(best_optimization.values()).most_common(1)[0][0]
print(f"\n--- Overall Recommended Optimization (Based on Simple Majority) ---")
print(f"The most frequently identified best optimization method across all metrics is: {overall_best}")
```

10.2 Screenshots of output

Data Scraping:

```

Scraping page 3507...
Page 3500 scraped and saved, total rows: 174845
Page 3499 scraped and saved, total rows: 174895
Scraping page 3508...
Page 3501 scraped and saved, total rows: 174945
Scraping page 3509...
Page 3502 scraped and saved, total rows: 174995
Scraping page 3510...
Page 3503 scraped and saved, total rows: 175045
Scraping page 3511...
Page 3507 scraped and saved, total rows: 175095
Retrying https://www.carlist.my/cars-for-sale/malaysia?page_size=50&page_number=3498... Attempt 1
Scraping page 3512...
Page 3504 scraped and saved, total rows: 175145
Page 3506 scraped and saved, total rows: 175195
Page 3510 scraped and saved, total rows: 175245
Page 3498 scraped and saved, total rows: 175295
Page 3505 scraped and saved, total rows: 175345
Page 3509 scraped and saved, total rows: 175395
Page 3508 scraped and saved, total rows: 175445
Page 3511 scraped and saved, total rows: 175495
Retrying https://www.carlist.my/cars-for-sale/malaysia?page_size=50&page_number=3512... Attempt 1
Page 3512 scraped and saved, total rows: 175545

Done scraping.
Total rows: 175545
Time taken: 3491.63 seconds
File size: 83864.67 KB

```

Data Cleaning:

```

[INITIAL DATA SHAPE] -> Rows: 175545, Columns: 17

[INITIAL COLUMN NAMES] ->
['car name' 'price (myr)' 'currency' 'location' 'region' 'brand' 'model'
'year' 'mileage' 'fuel type' 'color' 'body type' 'seating capacity'
'condition' 'image' 'description' 'url']

[MISSING VALUES PER COLUMN]:
car name      0
price (myr)   0
location      143
region        0
brand         0
model         0
year          0
mileage       0
fuel type     39
color         0
body type     31
seating capacity 0
condition     0
image         0
description   0
url           0
dtype: int64

[MISSING VALUES PER COLUMN AFTER REPLACE WITH UNKNOWN]:
car name      0
price (myr)   0
location      0
region        0
brand         0
model         0
year          0
mileage       0
fuel type     0
color         0
body type     0
seating capacity 0
condition     0
image         0
description   0
url           0
dtype: int64

[VALID URL ROWS ONLY] -> (175545, 16)

[CLEANED DATA SHAPE] -> (175545, 16)

[INITIAL COLUMN NAMES] ->
['car name' 'price (myr)' 'location' 'region' 'brand' 'model' 'year'
'mileage' 'fuel type' 'color' 'body type' 'seating capacity' 'condition'
'image' 'description' 'url']

✓ Cleaned dataset saved as 'cleaned_data.csv'

```

Before Optimization (Pandas):

- Pandas Optimization Complete
- Total Rows: 175545
- Total Processing Time: 2.16 seconds
- CPU Usage: 70.60%
- Memory Usage: 86.04 MB
- Throughput: 81291.59 records/second
- Mean Price (MYR): RM 208327.84
- Mean Mileage: 56206.03 km
- Most Common Region: selangor
- Metrics saved to pandas_optimized.csv

After Optimization (Polars and DuckDB):

- Pandas Optimization Complete
- Total Rows: 175545
- Total Processing Time: 2.16 seconds
- CPU Usage: 70.60%
- Memory Usage: 86.04 MB
- Throughput: 81291.59 records/second
- Mean Price (MYR): RM 208327.84
- Mean Mileage: 56206.03 km
- Most Common Region: selangor
- Metrics saved to pandas_optimized.csv

- Polars Optimization Complete
- Total Rows: 175545
- Total Processing Time: 0.61 seconds
- CPU Usage: 14.80%
- Memory Usage: 129.63 MB
- Throughput: 289688.39 records/second
- Mean Price (MYR): RM 208327.84
- Mean Mileage: 56206.03 km
- Most Common Region: selangor

- Pandas Optimization Complete
- Total Rows: 175545
- Total Processing Time: 3.44 seconds
- CPU Usage: 3.60%
- Memory Usage: 125.80 MB
- Throughput: 51003.49 records/second
- Mean Price (MYR): RM 208327.84
- Mean Mileage: 56206.03 km
- Most Common Region: selangor
- Metrics saved to pandas_metrics.json

- DuckDB (FireDucks) - After Optimization
- Total Rows: 175545
- Total Processing Time: 0.43 seconds
- CPU Usage: 46.20%
- Memory Usage: 81.64 MB
- Throughput: 406446.43 records/second
- Mean Price (MYR): RM 208327.84
- Mean Mileage: 56206.03 km
- Most Common Region: selangor

Successfully saved cleaned metrics to performance_after.csv

Cleaned Performance Metrics:

Optimization Stage	Total Rows	Total Processing Time (seconds)
0 Polars Optimization	175545	0.61
1 Pandas Optimization	175545	3.44
2 DuckDB Optimization	175545	0.43

	CPU Usage (%)	Memory Usage (MB)	Throughput (records/second)
0	14.8	129.63	289688.39
1	3.6	125.80	51003.49
2	46.2	81.64	406446.43

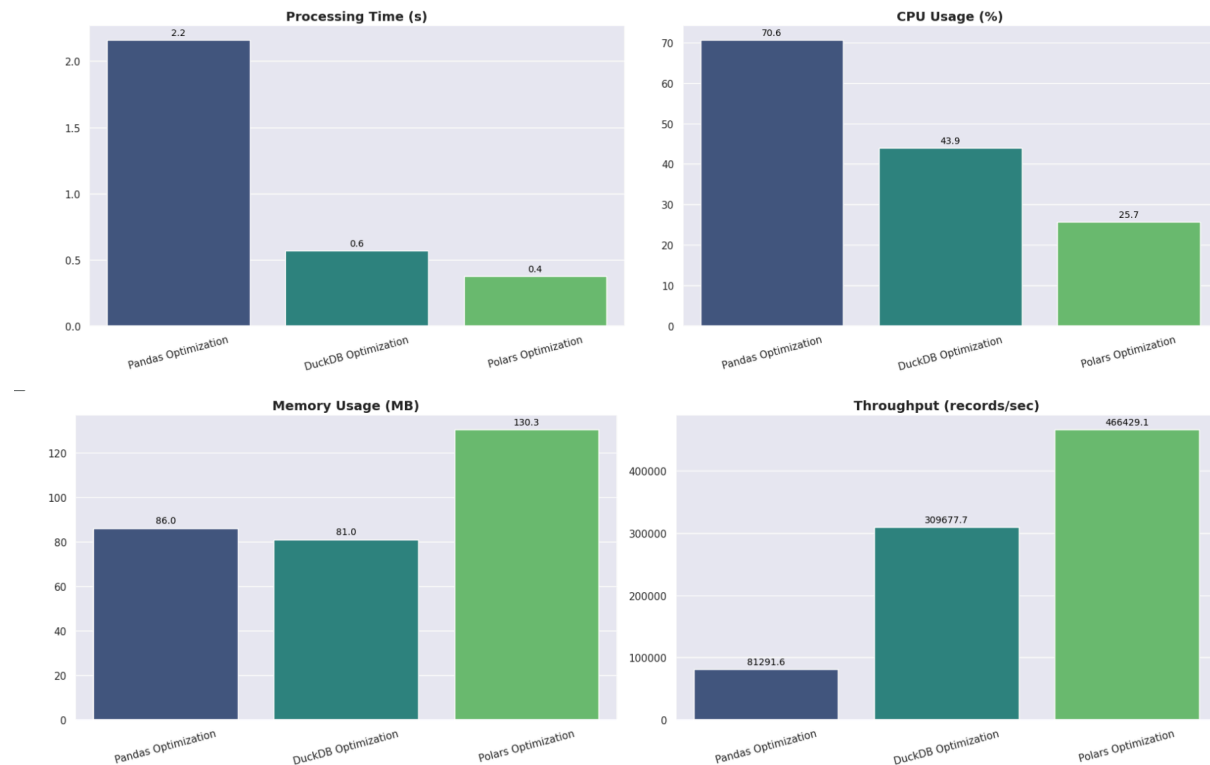
	Data Processed (rows)	Time Taken (seconds)	Records per second
0	175545	0.61	289688.39
1	175545	3.44	51003.49
2	175545	0.43	406446.43

<ipython-input-7-f6e9277017df>:33: FutureWarning: The behavior of DataFrame concatenation with empty or all-NA entries is deprecated. In a future version, this will no longer exclude
df = pd.concat([df, temp_df], ignore_index=True)

Evaluation:



Performance Comparison Before vs After Optimization



For Processing Time (s): Polars Optimization
 For CPU Usage (%): Polars Optimization
 For Memory Usage (MB): DuckDB Optimization
 For Throughput (records/sec): Polars Optimization

--- Overall Recommended Optimization (Based on Simple Majority) ---
 The most frequently identified best optimization method across all metrics is: Polars Optimization

10.3 Links to full code repo or dataset

Group Github:

<https://github.com/drshahizan/HPDP/tree/main/2425/project/p1/GroupB>

Dataset (raw and clean dataset):

<https://github.com/drshahizan/HPDP/tree/main/2425/project/p1/GroupB/data>

Main crawler and data cleaning:

<https://github.com/drshahizan/HPDP/tree/main/2425/project/p1/GroupB/p1>

Optimization:

<https://github.com/drshahizan/HPDP/tree/main/2425/project/p1/GroupB/p1/optimization>

After Optimization:

<https://github.com/drshahizan/HPDP/tree/main/2425/project/p1/GroupB/p2/after%20optimization>

Evaluation and performance:

<https://github.com/drshahizan/HPDP/tree/main/2425/project/p1/GroupB/p2>