SECP3133

HIGH PERFORMANCE DATA PROCESSING

Project 1:

Optimizing High-Performance Data Processing for Large-Scale Web scrapers

GROUP NAME:	CrawlOps
GROUP MEMBERS:	1. GOH JIALE (A22EA0043) 2. KOH LI HUI (A22EC0059) 3. MAISARAH BINTI RIZAL (A22EC0192) 4. YONG WERN JIE (A22EC0121)
SECTION:	01
LECTURER:	ASSOC. PROF. DR. MOHD SHAHIZAN BIN OTHMAN
SUBMISSION DATE:	16/5/2025

Table of Contents

1. Introduction 4

1.1. Background of the project 4

1.2. Objectives 4

1.3. Target website and data to be extracted 5

2. System Design & Architecture 6

2.1. Description of architecture 6

2.2. Tools and frameworks used 9

2.3. Roles of team members 11

3. Data Collection 13

3.1. Scraping method (pagination, rate-limiting, async) 13

3.2. Number of records collected 14

4. Data Processing 16

4.1. Data Access from Supabase 16

4.2. Cleaning methods 16

4.3. Data structure (CSV/JSON/database) 16

4.4. Transformation and formatting 17

5. Optimization Techniques 18

5.1. Methods used: multithreading, multiprocessing, distributed computing, etc. 18

5.2. Code overview or pseudocode of techniques applied 19

a. Sequential Processing (Baseline) 19

b. Multithreading (using concurrent.features) 20

c. Multiprocessing (using joblib) 20

d. Distributed Computing (Apache Spark) 21

6. Performance Evaluation 23

6.1. Before vs after optimization 23

6.2. Time, memory, CPU usage, throughput 25

6.3. Charts and graphs 28

7. Challenges & Limitations 31

7.1. What didn't go as planned 31

7.2. Any limitations of your solution 31

8. Conclusion & Future Work 33
8.1. Summary of findings 33
8.2. What could be improved 34
9. References 35
10. Appendices 36
10.1. Sample code snippets 36
10.2. Screenshots of output 66
10.3. Links to full code repo or dataset 74

1. Introduction

1.1. Background of the project

In today's data-driven world, the ability to efficiently gather, process, and analyze large volumes of information sourced from the internet is a critical capability, especially in fields like data science and analytics. This project, titled "Optimizing High-Performance Data Processing for Large-Scale Web scrapers," places us directly in this context. As students of High Performance Data Processing, we undertake the challenge of developing a comprehensive system to collect a substantial dataset—specifically, a minimum of 100,000 structured records—from a prominent Malaysian website. The project emphasizes not only the initial data acquisition through web scraping but also the subsequent application and comparison of advanced computing techniques such as multithreading, multiprocessing, and distributed computing (Spark) to enhance the speed, efficiency, and scalability of processing this data. This hands-on approach allows us to explore practical solutions to common performance bottlenecks encountered in data-intensive applications.

1.2. Objectives

This project aims to equip us with practical skills and a deeper understanding of high-performance data processing. The specific objectives are:

1. To design and implement a web scraper using **Selenium and BeautifulSoup** to systematically extract a large volume of car listing data (over 100,000 records achieved: **121,520 records**) from the selected Malaysian website, Mudah.my.
2. To apply effective data cleaning and preprocessing techniques to transform the raw, scraped web data into a structured, high-quality dataset suitable for reliable analysis and storage.
3. To store the cleaned and structured dataset in a Supabase cloud database, ensuring data integrity and accessibility.
4. To develop and execute five distinct analytical queries to derive meaningful insights from the collected car data.
5. To implement, apply, and rigorously evaluate at least three high-performance computing optimization techniques:
    - **Multithreading** (using Python's **concurrent.futures**).
    - **Multiprocessing** (using the **joblib** library).
    - **Distributed Computing** (using Apache Spark via **pyspark**).

6. To conduct a thorough performance evaluation, comparing these optimization methods against a baseline sequential processing approach, using metrics such as total execution time, CPU and memory utilization, and data processing throughput.
7. To cultivate effective teamwork and collaborative problem-solving skills within a diverse student group.
8. To professionally document our project's methodology, system design, implementation details, findings, and analyses in a comprehensive technical report and present our work.

1.3. Target website and data to be extracted

Our project focused on **Mudah.my**, a major online marketplace in Malaysia, as the source for data collection. We specifically targeted the "sell car" category due to the high volume and richness of available listings, which enabled us to meet the project's requirement of collecting over 100,000 records (our team successfully collected **121,520 records**).

The key data fields extracted for each car listing included:

- **Car Name (c_name):** The make, model, and variant of the vehicle.
- **Price (c_price):** The listed selling price in MYR, cleaned and converted to an integer.
- **Location (c_location):** The geographical location (state/city) where the car is listed.
- **Condition (c_condition):** The car's state (e.g., "New," "Used," "Recon").
- **Mileage:** The vehicle's mileage, processed from various string formats into standardized numeric minimum (**c_mileage_min**) and maximum (**c_mileage_max**) values.
- **Manufactured Year (c_year):** The year the car was made.
- **Engine Capacity (c_engine):** The engine's size in cubic centimeters (cc), which was extracted and converted from a string format (e.g., "1499cc") to an integer.

Ethical considerations were paramount during data collection. We ensured our web scraping process respected Mudah.my by implementing rate-limiting (1.5-second delay between page navigations), using appropriate user-agent information, and only collecting publicly accessible data in adherence to the website's implicit guidelines.

2. System Design & Architecture

2.1. Description of architecture

The system developed for this project follows a multi-stage data pipeline, designed to handle the journey of data from web extraction to optimized analysis. This architecture ensures a methodical approach to handling and processing the large dataset.

1. **Data Acquisition (Web scraping):**

    - This initial stage was performed using a Python script leveraging **Selenium** for browser automation and dynamic page interactions, and **BeautifulSoup** for parsing the HTML content from Mudah.my. The development and execution of the scraper were done using Visual Studio Code.
    - **URL Management:** A list of base URLs for different car categories was used. The scraper dynamically generated page URLs (e.g., using **?o=page_number**) to navigate through listings. For categories exceeding 10,000 listings, URL filtering (e.g., by price range) was applied to manage data collection within website display limits.
    - **Pagination & Rate-Limiting:** The scraper iterated up to 1,000 pages per base URL, stopping if no results were found. A 1.5-second delay between page navigations was implemented to ensure responsible server interaction.
    - **Error Handling:** The scraper included mechanisms to handle **WebDriverException, TimeoutException**, and network errors, with up to 3 retries per failed navigation before skipping a problematic URL.
    - **Initial Data Extraction and Cleaning:** As data (car name, price, location, condition, mileage, year, engine capacity) was extracted using BeautifulSoup, some initial cleaning (e.g., converting price to integer) was performed by the scraping script itself. This initially scraped and somewhat cleaned data was then inserted into a Supabase table, referred to as **cars_before_clean**, serving as a staging area.

2. **Detailed Data Cleaning and Preparation:**

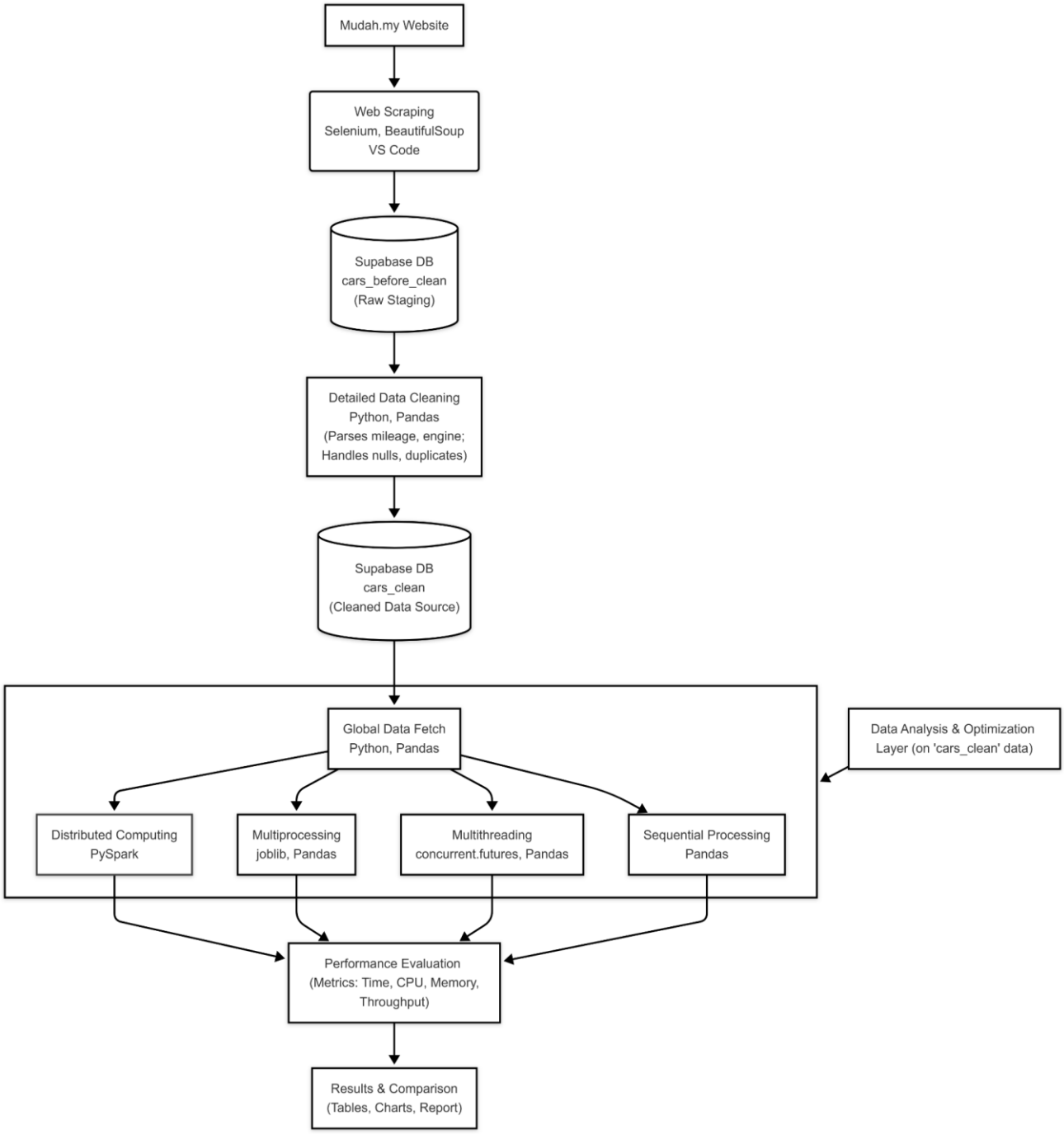
    - Following the initial scrap, a dedicated Python script read data from the **cars_before_clean** Supabase table.

- This script, primarily using the Pandas library, performed more intensive cleaning tasks:
  - Advanced parsing of **c_mileage** strings into numeric **c_mileage_min** and **c_mileage_max** columns.
  - Conversion of the **c_engine** field from a string format (e.g., "1998cc") to a numerical integer.
  - Systematic identification and removal of records with missing values in critical fields.
  - Detection and removal of duplicate car listings to ensure dataset integrity.

2. **Cleaned Data Storage:**
   - After this detailed cleaning, the processed and validated dataset was loaded into a final Supabase table named **cars_clean**. This table became the definitive, high-quality data source for all analytical queries and performance optimization experiments.
3. **Data Analysis & Optimization Layer:**
   - The core analytical part of the project involved executing five predefined queries on the **cars_clean** dataset. To evaluate the impact of different high-performance computing strategies, these queries were implemented and run using four distinct approaches:
     - **Sequential Processing (Baseline):** All data required for analysis was fetched once from **cars_clean** into a Pandas DataFrame. The five analytical queries, implemented as Pandas operations, were then executed one after another.
     - **Multithreading:** Similar to the sequential approach, all data was fetched globally. The five Pandas-based analytical tasks were then executed concurrently using Python's **concurrent.futures.ThreadPoolExecutor**.
     - **Multiprocessing:** Data was fetched globally. The Pandas operations for each of the five analytical tasks were then parallelized using the **joblib** library, which distributes computational work across multiple CPU cores.
     - **Distributed Computing (Apache Spark):** The globally fetched dataset was transformed into an Apache Spark DataFrame. The five analytical queries were then re-implemented using Spark's DataFrame API and executed leveraging its distributed processing capabilities, run in a local mode for this project.

4. **Performance Evaluation:**

   - Each of these four processing approaches was benchmarked. Key performance indicators include total execution time, CPU utilization, memory usage, and data processing throughput were recorded. These metrics facilitated a quantitative comparison of the different optimization techniques.



2.2. Tools and frameworks used

The successful execution of this project was made possible by a combination of various tools, programming languages, and frameworks:

- **Python:** The foundational programming language used for all aspects of the project, including web scraping, data manipulation, implementing optimization techniques, and interacting with external services.
- **Selenium:** A browser automation framework used in the data acquisition phase to navigate dynamic web pages on Mudah.my and enable content extraction.
- **BeautifulSoup4:** A Python library for parsing HTML and XML documents, utilized to extract specific data elements from the web pages fetched by Selenium.
- **Visual Studio Code:** Mentioned as the development environment for the web scraping component.
- **Supabase (with supabase-py client):** A cloud-based service providing a PostgreSQL database. It was used for:
    - Storing the initially scraped data (in a staging table named **cars_before_clean**).
    - Storing the final, cleaned dataset (**cars_clean**).
    - Serving as the data source for the analytical processing stage.
- **Pandas:** An essential Python library for data analysis and manipulation. It was used for:
    - Data structuring (creating and managing DataFrames).
    - Extensive data cleaning and transformation operations.
    - Implementing the core logic for the five analytical queries in the Sequential, Multithreading, and Multiprocessing approaches.
- **NumPy:** A fundamental package for numerical computation in Python. In this project, it was utilized to assist in the data preparation stage for multiprocessing, specifically to help create more balanced data chunks for parallel processing, aiming for a more even workload distribution across processes.
- **concurrent.futures.ThreadPoolExecutor:** Part of Python's standard library, employed to implement the multithreading optimization strategy.
- **joblib:** A Python library leveraged for the multiprocessing optimization, facilitating easy parallel execution of tasks across CPU cores.

- **Apache Spark (with pyspark):** The chosen framework for demonstrating distributed computing principles. **pyspark** enabled the development of Spark applications in Python for processing the dataset in a local cluster mode.
- **psutil:** A cross-platform Python library used to retrieve system utilization details like CPU and memory usage, crucial for performance evaluation.
- **memory_profiler:** Utilized in the Spark context for more detailed memory usage analysis.
- **PrettyTable:** Employed in some of the processing scripts to format console output of query results into readable tables.

2.3. Roles of team members

Team Member	RoleKey	Contributions
MAISARAH BINTI RIZAL	Distributed Computing Optimization Lead, Performance Analysis Lead, & Project Collaborator	- Led the implementation and testing of distributed computing strategies. - Collected and analyzed key performance metrics such as query time, memory usage, CPU usage and throughput. - Created visualizations and comparative reports to support data-driven decision making. - Ran experiments multiple times to ensure accuracy and consistent results. - Collaborated on interpreting findings and identifying the most effective optimization techniques.
KOH LI HUI	Multiprocessing Optimization Lead & Technical Evaluator & Project Collaborator	- Led the design and implementation of multiprocessing techniques to enhance computational efficiency through parallel task execution. - Utilized Python's multiprocessing library to create scalable solutions for simultaneous data processing across multiple CPU cores. - Observed and summarized the challenges and limitations encountered during the entire project, including issues related to web scraping reliability, optimization implementation, data processing bottlenecks, and integration complexity between components. - Actively participated in group discussions and project planning sessions, by sharing personal insights and opinions to facilitate decision-making throughout the project.

GOH JIALE	Data Scraper & Data Processor & Sequential Processing Lead & Project Lead	- Prepare data scraping code using selenium and beautiful soup libraries - Scrape data from website for 7 hours (Redo once) - After scraping, process data by removing redundancy, convert variables from string to int and remove empty data. - To compare performance, sequential processing code is done and used as before optimization version. - Documentation of web scraping and data processing.
YONG WERN JIE	Data Processing Optimization Lead (Multithreading) & Project Contributor	- Designed & implemented the multithreaded data processing solution. - Developed Python script for global data fetch and concurrent Pandas-based query execution using ThreadPoolExecutor. - Integrated performance metric collection for multithreaded tasks. - Actively participated in group discussions, planning, and report preparation.

3. Data Collection

3.1. Scraping method (pagination, rate-limiting, async)

The data collection for this project was conducted using a web scraping approach with Selenium and BeautifulSoup libraries. Selenium was used to automate browser interactions, allowing dynamic page navigation and data extraction, while BeautifulSoup facilitated parsing the HTML content. The scraping process focused on collecting car listing information from the target website. All of the process is done by Visual Studio Code since it needs to simulate a real website in Selenium.

URL Management

- A list of base URLs was defined, each representing a different category of cars (e.g., sedans, SUVs, hatchbacks).
- The script supports dynamic URL generation, using a page parameter (?o=page_number) to navigate through multiple pages of each base URL.
- If a URL needs to start from a specific page (e.g., page 17), this is defined within the script.
- For categories with more than 10,000 listings (e.g., some sedan categories), the URLs were further filtered (e.g., by price range) to ensure all data could be collected without exceeding the website's 10,000 record display limit.

Pagination

- For each base URL, the scraper iterates through pages, starting from page 1 (or a specified start page).
- The script dynamically constructs URLs for each page, using either &o=page_number or ?o=page_number depending on the base URL format.
- It is set to scrape up to a maximum of 1,000 pages per base URL, but stops earlier if a 'No Results Found' message is detected.

Rate-Limiting

- A wait time of 1.5 seconds was set between page navigations (wait_time_per_page), ensuring responsible scraping that avoids overloading the target server.
- A retry mechanism was implemented for network errors, with a maximum of 3 retries per failed navigation.
- Retryable errors include WebDriverException (e.g., connection reset) and TimeoutException.
- If a URL consistently fails after 3 retries, the scraper skips to the next URL.

Error Handling

- WebDriverException, TimeoutException, and network-related errors (like internet disconnection) are explicitly handled.
- The scraper skips to the next URL if persistent errors occur for a specific URL.
- SSL errors and handshake failures are detected, with an option for manual retry by the user.
- Explicit error messages like 'net::ERR_INTERNET_DISCONNECTED' inferred for retry with delay.

Data Extraction

- Scraper does HTML parsing with the help of BeautifulSoup and identifies car listing items by assigned CSS classes.
- The following information is available for each listing:
    - Car name, price, location, condition, mileage, manufactured year, and engine capacity.
- Data is cleaned (e.g., price is converted to an integer) before being inserted into the Supabase database.

Data Storage

- After data extraction, the cleaned and structured car listing details are inserted into a Supabase database.
- The database schema includes tables for storing car information such as:
    - car_name, price, location, condition, mileage, manufactured_year, and engine_capacity.
- The insertion process ensures that all relevant fields are populated, and any missing or invalid data is filtered out.
- Real-time tracking of inserted data is performed to ensure the process runs smoothly, with the total_items_inserted count dynamically updated.
- Supabase's APIs are used to interact with the database, and proper authentication tokens are managed to ensure secure data insertion.

3.2. Number of records collected

A total of 'total_items_inserted' records were successfully collected and stored in the Supabase database. This number is dynamically tracked during the scraping process. As result, 121,520 records had been collected from the website mudah.my in the sell car category.

3.3. Ethical considerations

Below are the ethical considerations that we follow throughout the web scraping process:

- The web scraping process was designed to respect the target website's server load using rate-limiting and retry mechanisms.
- Only publicly accessible data was collected, and no automated actions were performed beyond data extraction.The scraper did not bypass any security mechanisms, authentication, or access control systems.
- User-Agent information was explicitly set to simulate a regular browser session, avoiding bot detection triggers.The target website's terms of service were reviewed, and the scraping process adheres to their guidelines wherever applicable.

4. Data Processing

4.1. Data Access from Supabase

For Data Processing, it is done using Google Colab. Data access was performed using the **Supabase Python SDK** (supabase-py), allowing seamless communication with the Supabase backend.

- **Client Initialization:** A create_client() method was used with the project's Supabase URL and API key to authenticate and establish a connection.
- **Pagination:** To handle large datasets, data was fetched in chunks of 1,000 rows using the .range() method in a loop, ensuring all records were retrieved without hitting size limits.
- **Execution:** All table operations (select, insert) were executed using the .execute() method after constructing the desired queries.
- **Efficiency:** This method ensured fast, reliable, and secure access to both raw and processed datasets stored in Supabase.

4.2. Cleaning methods

The raw dataset was initially stored in the cars_before_clean table in Supabase. To ensure high data quality, we implemented several cleaning steps:

- **Mileage normalization:** Convert text-based mileage (e.g., "30000 or more" and "20000 - 40000") into two numeric fields: c_mileage_min and c_mileage_max. Records with unrecognized formats were marked as None.
- **Engine size conversion:** Parsed the c_engine field by removing the "cc" suffix and converting the value into an integer.
- **Field validation:** Removed rows with missing or empty essential fields, including c_name, c_price, c_location, c_condition, c_mileage_min, c_year, or c_engine.
- **Duplicate removal:** Identified and excluded duplicate entries based on a combination of key features such as c_name, c_price, c_location, c_condition, c_mileage_min, c_mileage_max, c_year, and c_engine.

4.3. Data structure (CSV/JSON/database)

The dataset was managed using a **cloud-based Supabase PostgreSQL database**. During processing:

- The unprocessed data was extracted from the table cars_before_clean.
- The cleaned and correct data was inserted in a new table, which was named as cars_clean.

- Python was used for extraction with the supabase-py client providing some additional parsing.
- Intermediate and final data format was processed using a JSON-like Python dictionaries that were easy to convert to CSV/inject directly into the database.

4.4. Transformation and formatting

The data was transformed as follows:

- **Range extraction:** Processed string ranges (e.g., “20000 - 40000”) into two new numerical fields.
- **Field restructuring:** Dropped outdated fields(c_mileage) after transformation and converted to a structured number (c_mileage_min, c_mileage_max) formal.
- **Type casting:** Made sure all number fields such as mileage, engine size and year were stored as integers for consistency and compatibility downstream.
- **Batch insertion:** For performance and minimized network load reasons, clean records were inserted in batches of 1,000 records to the cars_clean table.

4.5. Data Insert into Supabase

To enter data in to the clean_cars table, the processed dataset was loaded into the clean_cars:

- **Batching:** The cleaned data was chunked (1,000 rows per request) to prevent overloading the API and to write optimally.
- **Insertion Loop:** Each batch was inserted using the .insert() method followed by .execute() from the Supabase client.
- **Progress Monitoring:** A number of inserted rows with a counter is used to identify the number of rows that have been listed and get the real-time push notifications of the insertion of them.
- **Final Output:** After all steps, all cleaned data were successfully stored inside cars_clean table, and is ready for use in applications, analytics, or modeling.

5. Optimization Techniques

To address the challenges of processing a large dataset (121,520 car listings) efficiently, our project explored and implemented several optimization techniques. The goal was to compare their effectiveness in speeding up the execution of five standard analytical queries run on our cleaned dataset, which was initially fetched from the Supabase database and loaded into memory as a Pandas DataFrame for most approaches.

5.1. Methods used: multithreading, multiprocessing, distributed computing, etc.

Our team investigated the following data processing methods to evaluate their impact on performance:

1. **Sequential Processing (Baseline):** This is the standard, non-optimized approach where each of the five analytical queries is executed one after the other, in a single computational stream. It serves as our baseline for measuring the improvements gained from other techniques.

2. **Multithreading:** This technique aims to improve performance by allowing multiple tasks (our five analytical queries) to run concurrently within the same process. We implemented this by having each analytical query (which performs Pandas operations on the pre-fetched global dataset) executed in a separate thread managed by Python's **concurrent.futures.ThreadPoolExecutor**.

3. **Multiprocessing:** To leverage multi-core CPU architectures and overcome Python's Global Interpreter Lock (GIL) for CPU-bound tasks, we used multiprocessing. For each of the five analytical queries, the primary dataset was divided into smaller chunks, and the processing of these chunks was distributed across multiple processes using the **joblib** library. The results from each processed chunk were then combined.

4. **Distributed Computing (Apache Spark):** To explore a technique designed for even larger datasets and more complex computations, we utilized Apache Spark. The globally fetched dataset was converted into a Spark DataFrame, and the five analytical queries were re-implemented using Spark's DataFrame API, allowing its engine to manage distributed execution (run in local mode for this project).

5.2. Code overview or pseudocode of techniques applied

This section provides a high-level overview of how each processing method was implemented for our five analytical queries. A common strategy across the optimized approaches (Multithreading, Multiprocessing, and Spark) was to first **load the entire cleaned dataset from Supabase into memory once**. The optimization techniques were then applied to the execution of the five analytical queries on this in-memory dataset.

a. Sequential Processing (Baseline)

- **Overall Approach:** After an initial global data load from the **cars_clean** table into a list of dictionaries (**response.data**), each of the five analytical queries was executed sequentially. The main data structure (**response.data**) was passed to each query function, which then typically converted it to a Pandas DataFrame for manipulation.
- Execution Flow (Conceptual):
    1. Load entire **cars_clean** dataset into **response.data**.
    2. **result_q1**, ... = **query_most_expensive_car_by_location(response.data)**
    3. Display result_q1 and its metrics.
    4. **result_q2**, ... = **query_total_cars_per_year(response.data)**
    5. Display result_q2 and its metrics (using **print_car_counts_table**).
    6. This pattern was repeated for:
        - **query_average_price_by_engine_size(response.data)** (using **print_average_prices_table**)
        - **query_total_cars_by_location(response.data)** (using **print_total_cars_by_location**)
        - **query_avg_min_mileage_by_condition(response.data)** (using **print_avg_min_mileage_by_condition**)
    7. The script records and prints total time and performance for each query individually.
- **Data Handling:** Each query function receives the full list of dictionaries and typically converts it to a Pandas DataFrame internally for processing.

b. Multithreading (using concurrent.features)

- **Overall Approach:** This method also starts with a global data load via **fetch_all_data_from_supabase(supabase_client)** into a single Pandas DataFrame (**complete_df**). The five analytical processing functions (which perform Pandas operations) are then assigned to different threads to run concurrently.
- Execution Flow (Conceptual using ThreadPoolExecutor in run_multithreaded(complete_df)):
    1. Call **fetch_all_data_from_supabase()** to get **complete_df**.
    2. A list, **processing_functions_list**, defines pairs of (query title, function reference).
    3. The **run_multithreaded** function uses a **ThreadPoolExecutor** to submit each analytical processing function (e.g., **process_most_expensive_car_per_location**, **process_total_cars_per_year**, etc.) with **complete_df** as an argument.
    4. Results (including data and metrics from the **@execute_with_metrics** decorator) are collected from each thread as it completes.
    5. The **display_query_output** function is used to show results and individual metrics for each query, ensuring they are displayed in a predefined order.
    6. The total time taken for all threads to complete their processing is recorded.
- **Data Handling:** Each thread receives a reference to the same **complete_df** and performs its specific Pandas operations.

c. Multiprocessing (using joblib)

- **Overall Approach:** After a global data load into a list of dictionaries (**data** via **fetch_car_data()**), this technique parallelizes the work *within each main analytical query function* by dividing the input **data** into chunks and processing these chunks across different CPU cores.
- Execution Flow (Conceptual, managed by run_all_queries(data) which calls each of the five main query functions):
    1. Call **fetch_car_data()** to get **data**.
    2. The **run_all_queries** function iterates through a list of main query functions:
        - query_most_expensive_car_by_location(data, n_jobs)
        - query_total_cars_per_year(data, n_jobs)
        - query_average_price_by_engine_size(data, n_jobs)
        - query_total_cars_by_location(data, n_jobs)
        - query_avg_min_mileage_by_condition(data, n_jobs)

3. Inside each of these main query functions:
    - The input **data** is divided into several smaller **chunks**.
    - A specific helper function, **process_chunk(chunk)**, is defined locally to perform the core logic for that query (e.g., finding relevant data within that chunk, often using Pandas or direct iteration on the list of dictionaries within the chunk).
    - **joblib.Parallel** is used to execute this **process_chunk** function on all **chunks** concurrently across multiple processes.
    - The partial results from all processed chunks are collected and aggregated to form the final result for that main query.
4. Results (often a **PrettyTable** object) and individual metrics are displayed for each main query.

- **Data Handling:** Data chunks are passed to worker processes, involving serialization. Each worker process operates on its independent chunk.

d. Distributed Computing (Apache Spark)

- **Overall Approach:** This method involves fetching all data from Supabase into a Python list (**all_rows**), then converting it to a Pandas DataFrame, and finally into Spark's native distributed data structure (a Spark DataFrame, **df**). The analytical queries are then expressed using Spark's DataFrame API.
- Execution Flow (Conceptual):
    1. Fetch all data from **cars_clean** into **all_rows**, convert to **pandas_df**, then to Spark DataFrame **df**.
    2. For each of the five analytical queries, a specific function is called, wrapped by **run_query_with_metrics**:
        - **run_query_with_metrics(query1)** where **query1()** uses Spark **df** operations for "Most Expensive Car in Each Location".
        - **run_query_with_metrics(query2)** where **query2()** uses Spark **df** operations for "Total Cars by Year".
        - And similarly for **query3(), query4()**, and **query5()**.
    3. Each query function (**query1** to **query5**) applies a series of Spark DataFrame transformations (e.g., **groupBy()**, **agg()**, **orderBy()**, **filter()**, window functions) and an

action (e.g., **.show()** called within the function, or **.count()** used for throughput in the wrapper).

4. Spark's engine executes these operations in parallel across available cores (simulating distributed execution in local mode).
5. Results are displayed using **.show()** within each query function, and metrics are printed by the **run_query_with_metrics** wrapper.

- **Data Handling:** Data resides in Spark's distributed memory structures, and operations are performed in a parallel, fault-tolerant manner by Spark.

6. Performance Evaluation

The following section is a comparative analysis of the execution parameters under the four execution paradigms: sequential, multiprocessing, multithreading and distributed computing. The analysis is based on significant parameters such as query time, CPU utilization, memory usage and throughput in determining the effectiveness of the optimization methods. Comparing these parameters with and without optimization informs us the significance of these execution methods towards the efficiency of the system along with the utilization of the resources.

6.1. Before vs after optimization

The comparison of the baseline with optimized methods indicates significant improvement of query execution times and overall throughput data. Most specifically, the most efficient optimization method turned out to be that of **multithreading** with the lowest query time and the highest throughput for all queries. For instance, in query 1 execution time decreased from 0.64 seconds all the way down to 0.08 seconds, more than eight times improvement. Throughput was significantly enhanced with up to 1.5 million records per second for multithreading versus the baseline 332, 123 per second throughput in query 5. These findings affirm that multithreading is highly appropriate for such a workload, most of all with handling of concurrent tasks that benefit from shared access.

On the other hand, **distributed computing** failed to yield beneficial results in this context. Despite its potential for scalability, it showed the highest query latencies as well as the lowest throughput, presumably due to the coordination delays and communication overheads inherent in distributed systems. Also, while **multiprocessing** showed minimal improvements over the baseline, its effectiveness was hampered by the Global Interpreter Lock (GIL) of Python, which prevents parallel execution. As such, in many instances, multiprocessing was inferior compared to multithreading, and at times even fell behind the baseline approach. Interestingly, the **baseline** itself performed quite well in several scenarios, highlighting the importance of targeted application of optimization with regard to the peculiarities of the workload.

The findings suggest that, while optimization techniques dramatically enhance performance, their effectiveness is also highly dependent on both the nature of the workload

and the underlying system architecture. It appears that multithreading is particularly useful for I/O bound operations with lightweight computational requirements, whereas more advanced techniques like multiprocessing and distributed computing will require greater scales or more computationally intensive tasks to justify their inherent overhead.

6.2. Time, memory, CPU usage, throughput

To evaluate the impact of different execution methods, we measured four key performance metrics across five queries. These metrics include query time, CPU usage, memory usage and throughput. Each execution method was **run three times for every query** and strategy, and the **average values** were recorded to ensure reliable and consistent performance. This approach helps minimize variability due to system fluctuations and provides a more accurate representation of each method's performance.

Query Time (seconds)

Query time measures how fast a system can process and return results for a given query. A shorter execution time indicates better responsiveness and overall system performance. The following table presents the query times for all five queries under each execution strategy.

Query	Sequential	Multithreading	Multiprocessing	Distributed Computing
q1	0.43	0.08	1.01	6.73
q2	0.35	0.06	0.39	1.65
q3	0.43	0.09	0.41	1.56
q4	0.43	0.08	0.62	1.40
q5	0.35	0.08	0.62	1.44

Table 1: Query Time (seconds)

From this data, it is evident that **multithreading** consistently achieved the shortest query times across all queries. For example, in query 1, multithreading completed the query in 0.08 seconds, compared to 0.43 for sequential execution. On average, multithreading reduced query time by up to ten times compared to other methods. In contrast, **distributed computing** had the longest execution time in most cases, likely due to communication overhead between nodes.

Memory usage (%)

Memory usage measures how much system memory was consumed during query execution. Lower memory usage is preferred, especially in environments with limited resources or when

running multiple tasks concurrently. The table below summarizes the average memory usage percentages for each strategy.

Query	Sequential	Multithreading	Multiprocessing	Distributed Computing
q1	9.63	10.25	10.83	3.12
q2	9.63	10.25	10.83	3.12
q3	9.73	10.27	10.87	3.12
q4	9.77	10.27	10.90	3.12
q5	9.63	10.27	10.97	3.12

Table 2: Memory Usage (%)

Among all strategies, **distributed computing** used the least amount of memory, averaging only 3.12% across all queries. This makes it a strong candidate for memory constrained environments. On the other hand, **multiprocessing** had the highest usage, likely due to overhead associated with creating and managing multiple processes. **Multithreading** and **sequential** maintained similar memory footprints, indicating efficient use of shared memory.

CPU usage (%)

CPU usage reflects how much of the processor's capacity was utilized during query execution. An optimal scenario involves moderate CPU usage, neither underutilizing nor overloading the processor. The table below shows the CPU usage percentages for each query and execution method.

Query	Sequential	Multithreading	Multiprocessing	Distributed Computing
q1	4.00	60.50	2.50	17.17
q2	3.67	63.98	3.50	48.92
q3	17.47	58.33	26.30	42.50
q4	22.50	50.00	53.27	32.73
q5	3.67	46.67	21.10	55.35

Table 3: CPU Usage (%)

Multithreading used significantly more CPU than any other method, with usage up to 50% in some queries. This high CPU utilization is expected, as multithreading leverages concurrent execution to achieve faster performance. While **multiprocessing** and **distributed computing** showed varied CPU usage, they did not deliver comparable improvements in query time. Therefore, while multithreading uses more CPU resources, this is justified by the significant gains in speed.

Throughput (records per second)

Throughput refers to the number of records processed per second. Higher throughput means the system is handling more data efficiently, which is crucial for applications requiring high performance data processing. The table below shows the throughput values across each query and method.

Query	Sequential	Multithreading	Multiprocessing	Distributed Computing
q1	265644.60	1459580.94	153514.51	7.61
q2	330018.67	2016209.76	298418.99	21.61
q3	279144.96	1215580.05	282946.66	10.45
q4	284087.32	1514009.44	184890.36	11.97
q5	332123.54	1511455.57	187832.48	2.37

Table 4: Throughput (records per second)

The results show that **multithreading** achieved the highest throughput across all queries, often exceeding baseline performance by over five times. For example, in query 5, multithreading processed 1.5 million records per second compared to only 332, 123 records per second using sequential. Conversely, **distributed computing** performed poorly in terms of throughput, with extremely low values in most queries, suggesting inefficiencies in task distribution and coordination.

6.3. Charts and graphs

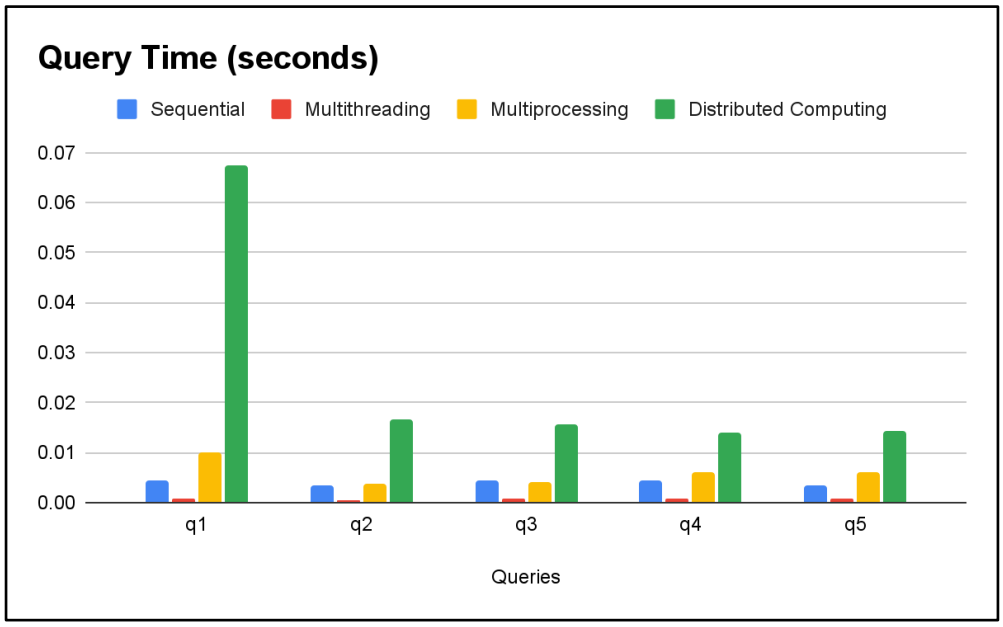


Figure 1: Query Time (seconds)

The chart above shows how the execution time varies across all queries and methods. It clearly emphasizes the consistent speed advantage of multithreading, with steep drops in query duration compared to other methods. The graph also highlights outliers, such as notably high execution time for distributed computing, suggesting potential inefficiencies in task distribution.

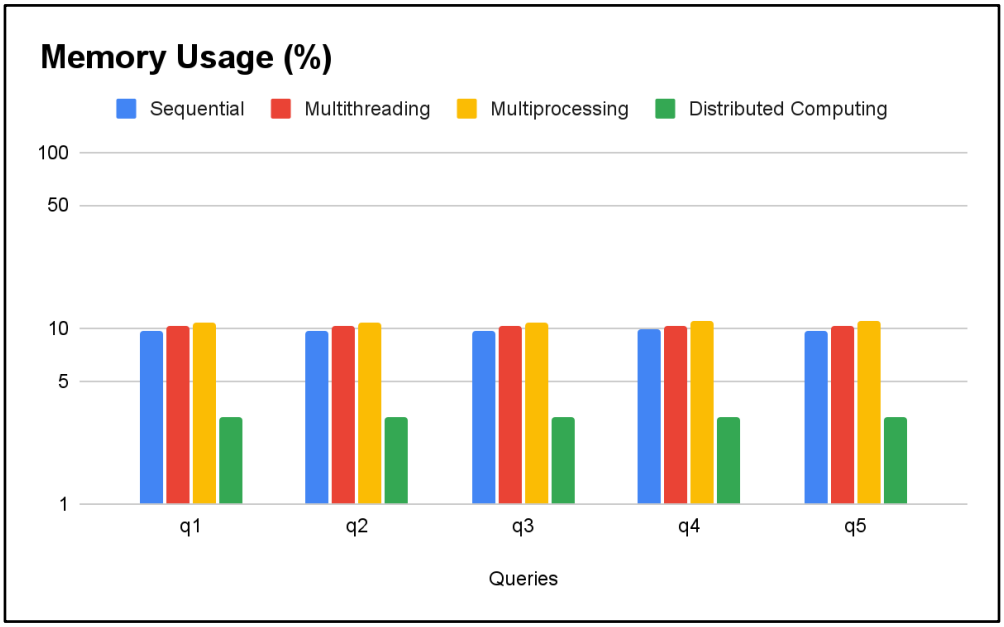


Figure 2: Memory Usage (%)

The memory usage chart provides a side-by-side comparison of how different methods can affect system memory during query execution. It shows that distributed computing maintains a consistently low memory footprint across all queries, while the other methods remain closely grouped, showing minimal variation between them.

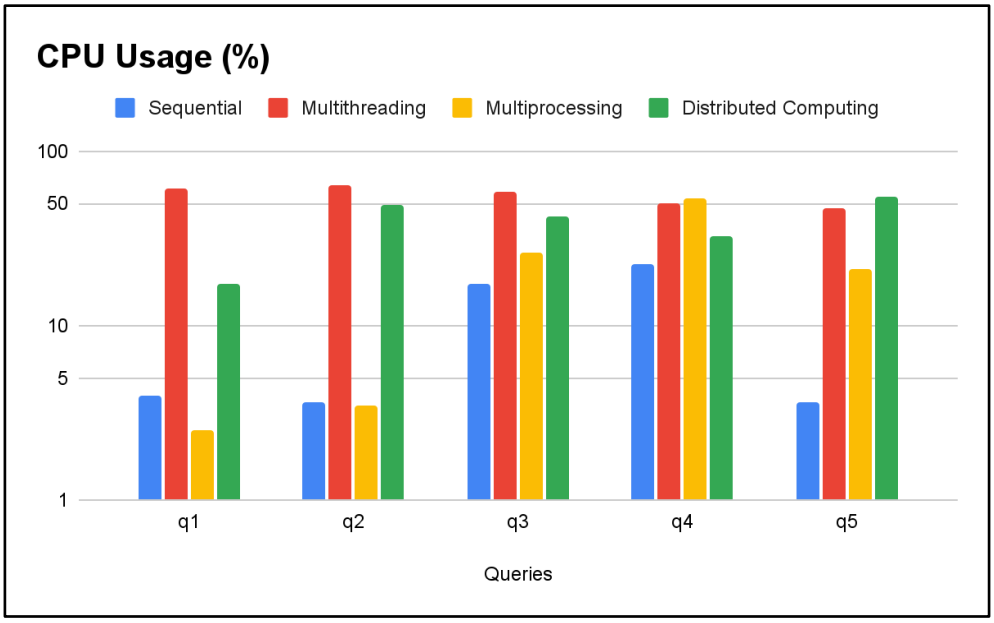


Figure 3: CPU Usage (%)

This chart illustrates the variability in CPU utilization across queries and strategies. It shows that **multithreading**'s CPU demand fluctuates more across queries, indicating dynamic resource allocation based on workload intensity. Sequential and multiprocessing show steadier

patterns, making them potentially more predictable in production environments where CPU stability is desired.

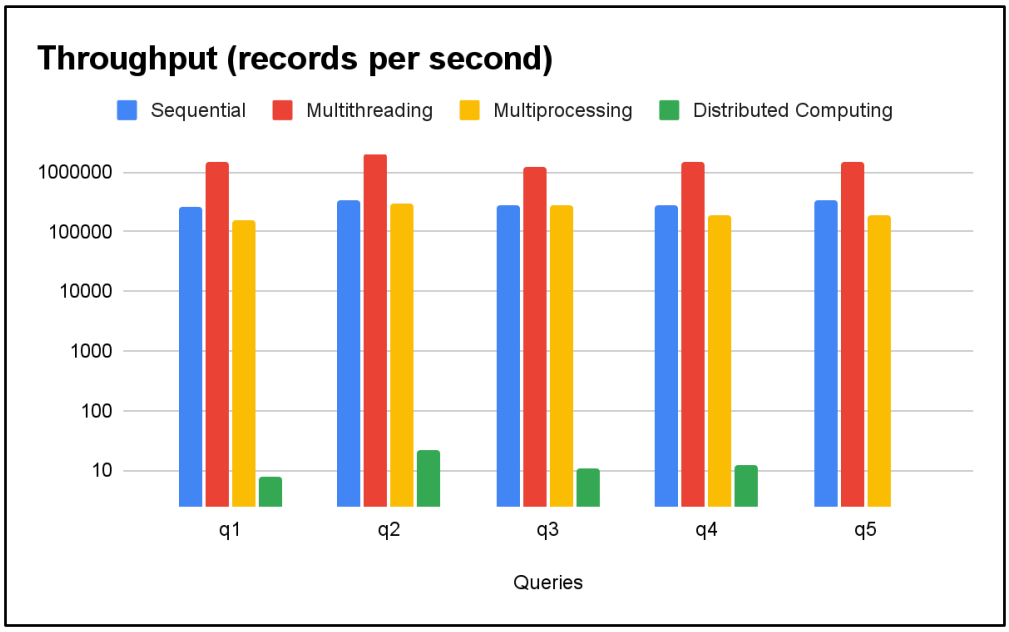


Figure 4: Throughput (records per second)

The throughput chart highlights performance trends by displaying how many records each strategy can process per second. Visually, it confirms **multithreading** dominance, especially in heavier queries like query 2 and query 5. It also reveals that distributed computing underperforms consistently, reinforcing coordination overhead outweighs its theoretical scalability benefits in this context.

7. Challenges & Limitations

7.1. What didn’t go as planned

- **Web Scraping Reliability:** The scraping script crashed because of unforeseen alterations in the website's layout during the data extraction phase. These modifications disrupted previously functional selectors and required urgent code changes, which caused delays in progress. Additionally, while trying to gather over one million records in a limited time, we faced some hardware limitations such as inadequate memory and overheating issues during long scraping sessions which disrupted the web scraping process and necessitated manual restarts.
- **Optimization Inconsistencies:** Despite applying various optimization strategies such as multithreading, multiprocessing, and distributed computing, we did not consistently achieve the expected performance improvements in some cases. This was especially evident while handling CPU-intensive tasks like retrieve a few queries from a large datasets because of Python’s Global Interpreter Lock (GIL).
- **Data Storage and Retrieval Challenges:** Managing and accessing extensive amounts of scraped data in Supabase resulted in considerable latency, particularly when dealing with more than a million records. As the dataset expanded, the performance of query operations diminished significantly. Furthermore, the API rate limits imposed by Supabase and the delays associated with cold starts affected the consistency and reliability of data retrieval in critical processing periods.
- **Difficulty in Benchmarking Optimization:** Comparing the performance improvements across different optimization techniques (multithreading, multiprocessing, distributed computing) was challenging. The results varied significantly based on the specific nature of each query and the structure of the code used. This variability made it difficult to draw clear and consistent conclusions regarding which optimization technique was the most efficient overall.

7.2. Any limitations of your solution

Despite the achievements and insights gained throughout the project, our solution comes with several inherent limitations that impacted its overall scalability, consistency, and adaptability in real-world scenarios:

- **Scalability Constraints in Supabase:** While Supabase served as a convenient and flexible backend for storing scraped data, it is not inherently optimized for high-throughput analytical queries on datasets that surpass one million records. As a result,

operations like joins, filters, and aggregate functions experienced delays, especially when processing multiple concurrent queries. The performance limits of Supabase hindered the efficiency of our data processing pipeline in high-load scenarios.

- **Hardware Dependency and Resource Limitations:** Much of the web scraping and preprocessing was conducted on personal devices or in restricted cloud environments. These setups lacked sufficient memory and processing power to maintain performance during extensive, large-scale scraping and transformation operations. The system's performance, especially in multithreading and multiprocessing scenarios, was potentially influenced by hardware availability, which restricts its applicability across various environments.
- **Manual Intervention for Error Recovery:** Frequent layout changes on the target website introduced instability that caused the scraping script to sometimes fail or gather corrupted data. Due to our web scraper's absence of automated failover and error management features, we had to rely on manual monitoring, data verification, and code modifications adjustments to restore functionality. This dependence on human intervention limited the robustness and autonomy of our system, making it unsuitable for fully automated, large-scale data scraping tasks.
- **Time and Scope Constraints:** Due to academic constraints and time restrictions of the project, we were unable to fully develop or refine several crucial components of the system. We had to simplify key areas such as data cleaning logic, parallelism configuration, and detailed performance profiling to meet the project deadlines. These simplifications limited the depth of our analysis and optimization. For a real-world or production-level implementation, substantial additional work would be necessary for testing, tuning, and scaling the pipeline to ensure its long-term resilience, fault tolerance, and efficiency.

8. Conclusion & Future Work

8.1. Summary of findings

This project involved both web scraping and the performance evaluation of various Python optimization strategies, including **baseline sequential processing**, **multithreading** (using concurrent.futures), **multiprocessing** (via joblib), and **distributed computing** (with PySpark), based on a dataset of over 100,000 records. The data was collected through web scraping techniques using libraries like **Selenium**, targeting the website **Mudah.my**, which served as the foundation for further query processing and analysis.

Each method demonstrated unique characteristics regarding execution speed, memory consumption, and processing efficiency:

- **Multithreading (concurrent.features)** proved to be the fastest and most effective choice for data processing. It leverages thread-level parallelism and efficient scheduling, making it particularly suitable for lightweight and I/O-bound tasks, as well as effective for medium-sized datasets.
- **Sequential baseline** is a popular method that employs pandas as the key tool for in-memory data manipulation. While Pandas is very effective for moderately sized datasets and features an intuitive syntax, its absence of integrated parallelism limits its effectiveness for high-throughput operations.
- **Multiprocessing (joblib)** is a technique that aims to speed up computation by executing tasks in parallel across multiple processes, with each process utilizing Pandas for data manipulation. While joblib improves performance compared to single-threaded pandas, it may introduce overhead due to the creation of processes and data transfer between them.
- **Distributed computing (PySpark)**, despite being designed for large-scale distributed computing, exhibited the slowest performance in this analysis. This is likely due to the overhead involved in setting up distributed tasks and data serialization, which outweighs its advantages when processing smaller datasets like the one used in this project.

In addition, web scraping played a crucial role in building the dataset used in this project. It highlighted the significance of efficient data gathering as a preliminary step before data processing. Well-structured scraping scripts minimized preprocessing time and guaranteed

consistent, usable data for the benchmarking stage. In conclusion, the **performance ranking** observed was:

concurrent.futures (Multithreading) > Sequential (Baseline) > joblib (Multiprocessing) > PySpark (Distributed)

This finding verifies that more complex or distributed libraries do not always guarantee better performance, particularly when dealing with medium-sized workloads.

8.2. What could be improved

There are several aspects of this project that could be improved:

- Expand the focus of both web scraping and query processing to include millions of records, enabling a more precise performance comparison for distributed systems like PySpark.
- Examine optimization options and configurations for each library to maximize their individual performance.
- Include more detailed metrics, such as disk I/O, overhead from threads/processes, and memory usage during sustained workloads.
- Incorporate profiling tools such as cProfile or memory_profiler to to detect performance issues and optimize both the execution time and memory usage of functions more efficiently.
- Explore hybrid approaches that combine scraping, storage, and processing workflows.
- Assess the effectiveness of each technique across various hardware configurations to gain insights into their scalability and performance under different system loads and setups.

With these improvements, future research can yield deeper insights into both data acquisition and processing efficiency, leading to more scalable and effective data pipelines.

9. References

1. GeeksforGeeks. (2017, July 13). Multithreading in Python | Set 1. GeeksforGeeks. https://www.geeksforgeeks.org/multithreading-python-set-1/
2. Python Software Foundation. (n.d.). threading — Thread-based parallelism — Python 3.9.0 documentation. Docs.python.org. https://docs.python.org/3/library/threading.html
3. Python - Multithreaded Programming - Tutorialspoint. (2009). Tutorialspoint.com. https://www.tutorialspoint.com/python/python_multithreading.html
4. concurrent.futures — Launching parallel tasks — Python 3.9.5 documentation. (n.d.). Docs.python.org. https://docs.python.org/3/library/concurrent.futures.html
5. Pandas. (2014). User Guide — pandas 1.0.1 documentation. Pydata.org. https://pandas.pydata.org/docs/user_guide/index.html
6. GeeksforGeeks. (2021, April). How to use ThreadPoolExecutor in Python3 ? GeeksforGeeks. https://www.geeksforgeeks.org/how-to-use-threadpoolexecutor-in-python3/
7. Supabase. (2023). Handling Routing in Functions | Supabase Docs. Supabase.com; Supabase Docs. https://supabase.com/docs/guides/functions/routing
8. PySpark Overview — PySpark 3.5.5 documentation. (n.d.). https://spark.apache.org/docs/latest/api/python/index.html
9. TutorialsPoint. (2025, March 25). PySpark Tutorial. https://www.tutorialspoint.com/pyspark/index.htm

10. Appendices

10.1. Sample code snippets

10.1.1. Web Scraping

Appendix Snippet A: Library Import

```
import time
import pandas as pd
from bs4 import BeautifulSoup
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.common.exceptions import WebDriverException
from supabase import create_client, Client
import socket
```

Appendix Snippet B: Supabase Configuration

```
# --- Supabase Configuration ---
SUPABASE_URL = "https://ugjwigpcopmtjgylopwf.supabase.co"
SUPABASE_KEY =
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzdXBhYmFzZSIsInJlZiI6I
nVnandpZ3Bjb3BtdGpneWxvcHdmIiwicm9sZSI6ImFub24iLCJpYXQiOjE3NDU4MjgxMjIs
ImV4cCI6MjA2MTQwNDEyMn0.oFcP1wCt1upByqTU8NgD4FpJUdv9I8sG1ECWMX1wz8I"
supabase: Client = create_client(SUPABASE_URL, SUPABASE_KEY)
```

Appendix Snippet C: Chrome Options

```
chrome_options = Options()
chrome_options.add_argument("--start-maximized")
chrome_options.add_argument('user-agent=Mozilla/5.0 (Windows NT 10.0;
Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124
Safari/537.36')

driver = webdriver.Chrome(options=chrome_options)
```

Appendix Snippet D: Setting before Scrape

```
base_urls = [
    'https://www.mudah.my/malaysia/cars-for-sale/4-wheels',
    'https://www.mudah.my/malaysia/cars-for-sale/coupe',
    'https://www.mudah.my/malaysia/cars-for-sale/hatchback?price=-
40000',
```

```
    'https://www.mudah.my/malaysia/cars-for-sale/hatchback?price=40000-
',
    'https://www.mudah.my/malaysia/cars-for-sale/mpvs?price=-90000',
    'https://www.mudah.my/malaysia/cars-for-sale/mpvs?price=90000-',
    'https://www.mudah.my/malaysia/cars-for-sale/pick-up',
    'https://www.mudah.my/malaysia/cars-for-sale/sedan?price=-20000',
    'https://www.mudah.my/malaysia/cars-for-sale/price-from-20000-
below-30000-sedan',
    'https://www.mudah.my/malaysia/cars-for-sale/sedan?price=30000-
60000',
    'https://www.mudah.my/malaysia/cars-for-sale/sedan?price=60000-
180000',
    'https://www.mudah.my/malaysia/cars-for-sale/sedan?price=180000-',
    'https://www.mudah.my/malaysia/cars-for-sale/sports',
    'https://www.mudah.my/malaysia/cars-for-sale/suvs?price=-70000',
    'https://www.mudah.my/malaysia/cars-for-sale/suvs?price=70000-
210000',
    'https://www.mudah.my/malaysia/cars-for-sale/suvs?price=210000-',
    'https://www.mudah.my/malaysia/cars-for-sale/others'
]
max_pages = 1000
wait_time_per_page = 1.5
max_retries = 3 # Max retries for network operations
retry_delay = 5 # Seconds to wait before retrying

total_items_inserted = 0
```

Appendix Snippet E: Scrape Process

```
print(f"Starting scraping process for {len(base_urls)} base URLs, up to
{max_pages} pages each...")

# --- Loop through each base URL ---
for base_url in base_urls:
    print(f"\n--- Starting scrape for base URL: {base_url} ---")
    skip_current_base_url = False # Flag to skip to next base_url on
persistent error

    # Determine the starting page number for this base_url
```

```
    start_page = 1
    if base_url == 'https://www.mudah.my/malaysia/cars-for-
sale/sedan?price=30000-60000':
        start_page = 17
        print(f"Starting from page {start_page} for this specific
URL.")

    for page_num in range(start_page, max_pages + 1):
        if skip_current_base_url:
            break

        # --- URL Generation ---
        if page_num == 1 and start_page == 1: # Only use base_url
directly if starting from page 1
             current_url = base_url
        # elif page_num == start_page and start_page != 1: # If
starting from a later page, construct the URL immediately
        #     if '?' in base_url:
        #         current_url = f"{base_url}&o={page_num}"
        #     else:
        #         current_url = f"{base_url}?o={page_num}"
        else: # For all subsequent pages (page > 1 or page >
start_page)
            if '?' in base_url:
                current_url = f"{base_url}&o={page_num}"
            else:
                current_url = f"{base_url}?o={page_num}"


        # --- Retry logic for driver.get() ---
        retries = 0
        while retries < max_retries:
            try:
                # Check if it's the very first navigation attempt for a
non-page-1 start
                if page_num == start_page and start_page != 1:
                    if '?' in base_url:
```

```
                        current_url = f"{base_url}&o={page_num}"
                    else:
                        current_url = f"{base_url}?o={page_num}"

                print(f"Navigating to page {page_num} for
{current_url}")
                driver.get(current_url)
                time.sleep(wait_time_per_page)
                break
            except WebDriverException as e:
                retries += 1
                print(f"Error navigating to {current_url}: {e}")
                if "net::ERR_INTERNET_DISCONNECTED" in str(e) or
"net::ERR_CONNECTION_RESET" in str(e):
                    if retries < max_retries:
                        print(f"Internet connection error detected.
Retrying in {retry_delay} seconds... (Attempt
{retries}/{max_retries})")
                        time.sleep(retry_delay)
                    else:
                        print(f"Max retries reached for navigating to
{current_url}. Skipping this base URL.")
                        skip_current_base_url = True # Set flag to skip
remaining pages for this base_url
                        break # Exit retry loop
                else:
                    print("Unhandled WebDriverException occurred.
Skipping this base URL.")
                    skip_current_base_url = True # Skip on other
critical WebDriver errors
                    break # Exit retry loop
            except Exception as e:
                retries += 1
                print(f"An unexpected error occurred during
navigation: {e}")
                if retries < max_retries:
```

```
                    print(f"Retrying in {retry_delay} seconds...
(Attempt {retries}/{max_retries})")
                    time.sleep(retry_delay)
                else:
                    print(f"Max retries reached for unexpected
navigation error. Skipping this base URL.")
                    skip_current_base_url = True
                    break # Exit retry loop

        if skip_current_base_url:
            continue # Go to the next iteration of the page loop (which
will then break)

        # --- Error Handling (Keep as is) ---
        while True:
            page_source = driver.page_source
            if "handshake failed" in page_source or "SSL error code" in
page_source:
                # ("SSL haprintndshake error detected. Please manually
refresh the page in the browser, then press Enter to continue...")
                input()
                driver.get(current_url)
                time.sleep(wait_time_per_page)
            else:
                break

        # print(f"Getting page source for page {page_num}...")
        page_source = driver.page_source

        # --- Check for "No Results Found" text ---
        if "Sorry, No Results Found!" in page_source:
            print(f"Found 'Sorry, No Results Found!' on page {page_num}
for {base_url}. Moving to next base URL or stopping scrape.")
            break # Stop scraping for the current base_url

        # print(f"Parsing page source for page {page_num} with
BeautifulSoup...")
```

```
        soup = BeautifulSoup(page_source, 'html.parser')

        listing_items = soup.find_all('div', attrs={'data-testid':
lambda x: x and x.startswith('listing-ad-item-')})

        # print(f"Found {len(listing_items)} potential listing items on
page {page_num} for {base_url}.")

        # --- Check if listing_items is empty ---
        if not listing_items:
            print(f"No listing items found on page {page_num} for
{base_url} (after checking for 'No Results' text). Continuing to next
page.")

        # --- Item Processing Loop (Keep as is, inside the page loop) -
--
        for item in listing_items:
            title_element = item.find(class_='flex flex-col flex-1 gap-
2 self-center')

            title = None
            price_str = None
            price_int = None
            car_name = None
            condition = None
            mileage = None
            manufactured_year_str = None
            engine_capacity = None
            location = None
            manufactured_year_int = None

            if title_element:
                title = title_element.text.strip()

                price_element = title_element.find(class_='text-sm
text-[#E21E30] font-bold')
```

```
                price_str = price_element.text.strip() if price_element
else None

                # --- Clean and convert price to integer ---
                if price_str:
                    try:
                        cleaned_price = price_str.replace('RM',
'').replace(',', '').strip()
                        price_int = int(cleaned_price)
                    except (ValueError, TypeError):
                        print(f"Warning: Could not convert price
'{price_str}' to integer for item: {title}") # Use title if name isn't
available yet
                        price_int = None

                # --- Adjust selectors based on what you are scraping -
--
                item_name_link = title_element.find('a')
                item_name = item_name_link.text.strip() if
item_name_link else title

                details_container = title_element.find('div',
class_='gap-2 grid grid-cols-2 mt-2')

                if details_container:
                    def extract_detail(detail_title):
                        detail_div = details_container.find('div',
attrs={'title': detail_title})
                        if detail_div:
                            value_div = detail_div.find('div',
class_='text-[11px]')
                            if value_div:
                                return value_div.text.strip()

                        if detail_title == 'Manufactured Year':
```

```
                            year_badge_div =
details_container.find('div', attrs={'data-testid': 'year-verified-
badge'})
                            if year_badge_div:
                                year_text_div =
year_badge_div.find('div', class_=lambda x: x and 'text-black' in x and
('text-xs' in x or 'text-[11px]' in x))
                                if year_text_div:
                                    return year_text_div.text.strip()

                        return None



                    condition = extract_detail('Condition')
                    mileage = extract_detail('Mileage')
                    manufactured_year_str =
extract_detail('Manufactured Year')
                    engine_capacity = extract_detail('Engine capacity')

            region_span_parent = item.find('span', attrs={'title':
'Region'})
            if region_span_parent:
                img_tag = region_span_parent.find('img')
                if img_tag:
                    location_span = img_tag.find_next_sibling('span')
                    if location_span:
                        location = location_span.text.strip()

            # Attempt to convert year to integer if needed by your
Supabase schema
            if manufactured_year_str:
                try:
                    manufactured_year_int = int(manufactured_year_str)
                except (ValueError, TypeError):
                    manufactured_year_int = None # Set to None if
conversion fails
```

```
            # --- Prepare data for Supabase (Adjust field names if
needed) ---
            # Ensure these field names match your Supabase table
columns
            item_data = {
                'c_name': item_name,
                'c_price': price_int,
                'c_location': location,
                'c_condition': condition,
                'c_mileage': mileage,
                'c_year': manufactured_year_int,
                'c_engine': engine_capacity
            }

            # --- Insert data into Supabase with Retry ---
            retries = 0
            while retries < max_retries:
                try:
                    data, count =
supabase.table('Jiale_cars').insert(item_data).execute()
                    # Check response structure carefully based on
actual Supabase client behavior
                    success = False
                    if isinstance(data, tuple) and len(data) > 1 and
hasattr(data[1], '__len__') and len(data[1]) > 0: # Check if count is
meaningful
                        success = True
                        total_items_inserted += 1
                    elif hasattr(data, 'data') and data.data:
                        success = True
                        total_items_inserted += 1

                    if success:
                        break # Success, exit retry loop
                    else:
                        print(f"Insertion did not return expected
success data for: {item_name}. Response: {data}")
```

```
                        break

                except socket.gaierror as e: # Specifically catch DNS
errors
                    retries += 1
                    print(f"DNS Error (getaddrinfo failed) inserting
data for {item_name}: {e}")
                    if retries < max_retries:
                        print(f"Retrying Supabase insert in
{retry_delay} seconds... (Attempt {retries}/{max_retries})")
                        time.sleep(retry_delay)
                    else:
                        print(f"Max retries reached for Supabase insert
(DNS error) for {item_name}. Skipping item.")
                        break # Exit retry loop
                except ConnectionError as e: # Catch broader connection
errors
                    retries += 1
                    print(f"Connection Error inserting data for
{item_name}: {e}")
                    if retries < max_retries:
                        print(f"Retrying Supabase insert in
{retry_delay} seconds... (Attempt {retries}/{max_retries})")
                        time.sleep(retry_delay)
                    else:
                        print(f"Max retries reached for Supabase insert
(Connection error) for {item_name}. Skipping item.")
                        break # Exit retry loop
                except Exception as e:
                    # Catch other Supabase client errors or unexpected
issues
                    print(f"Unexpected error inserting data for
{item_name} into Supabase: {e}")
                    break

    print(f"--- Finished scraping for base URL: {base_url} ---")
```

```
print("\nFinished scraping all base URLs. Closing browser...")
driver.quit()

print(f"\nTotal items successfully inserted into Supabase across all
URLs: {total_items_inserted}")
```

10.1.2. Data Cleaning

Appendix Snippet A: Fetching All Data from Supabase

```
response = supabase.table("cars_before_clean").select("*").range(0,
999).execute()

# Set up pagination loop
batch_size = 1000
offset = 1000  # Start from after the first 1000 rows

# Continue fetching in chunks until no more data is left
while True:
    batch_response =
supabase.table("cars_before_clean").select("*").range(offset, offset
+ batch_size - 1).execute()
    rows = batch_response.data

    if not rows:
        break

    # Add fetched rows to the response.data
    response.data.extend(rows)

    # Update offset for the next batch
    offset += batch_size

    print(f"Fetched {len(response.data)} rows so far...")

# Final result stored in response.data
print(f"☑ Done. Total rows fetched: {len(response.data)}")
```

Appendix Snippet B: Convert data from string to proper variable

```
for car in response.data:
    mileage_str = car.get("c_mileage", "")
```

```
    # Ensure mileage_str is a valid string before checking
    if mileage_str:
        # Check for 'or more' pattern in mileage string
        if "or more" in mileage_str:
            min_mileage = mileage_str.split(" ")[0]  # Get the
number before "or more"
            car["c_mileage_min"] = int(min_mileage)
            car["c_mileage_max"] = None  # No maximum mileage
        elif " - " in mileage_str:
            # Handle the range case as before
            min_mileage, max_mileage = mileage_str.split(" - ")
            car["c_mileage_min"] = int(min_mileage.strip())
            car["c_mileage_max"] = int(max_mileage.strip())
        else:
            # Handle cases where the mileage format is unrecognized
            car["c_mileage_min"] = None
            car["c_mileage_max"] = None
    else:
        # Handle the case when mileage_str is None or empty
        car["c_mileage_min"] = None
        car["c_mileage_max"] = None

    # Optionally remove the original c_mileage
    if "c_mileage" in car:
        del car["c_mileage"]

    # Convert c_engine from "1299cc" to 1299
    engine_str = car.get("c_engine", "")
    # Check if engine_str is not None and not empty before
proceeding
    if engine_str:
        engine_str = engine_str.lower().replace("cc", "").strip()
        try:
            car["c_engine"] = int(engine_str)
        except ValueError:
            car["c_engine"] = None
    else:
        car["c_engine"] = None

# Show result
from pprint import pprint
pprint(response.data)
```

Appendix Snippet C: Remove redundancy and empty data

```
count = 0
filtered_data = []
seen_cars = set()

for car in response.data:

    has_issue = False   # Check for issues and skip if any are found

    # Check if 'c_name' is empty or None
    if not car.get("c_name"):
        print('c_name' + str(car))
        count += 1
        has_issue = True

    # Check if 'c_price' is NULL
    if car.get("c_price") is None:
        # print('c_price' + str(car))
        count += 1
        has_issue = True

    # Check if 'c_location' is empty or None
    if not car.get("c_location"):
        print('c_location' + str(car))
        count += 1
        has_issue = True

    # Check if 'c_condition' is empty or None
    if not car.get("c_condition"):
        # print('c_condition' + str(car))
        count += 1
        has_issue = True

    # Check if 'c_mileage_min' is NULL
    if car.get("c_mileage_min") is None:
        print('c_mileage_min' + str(car))
        count += 1
        has_issue = True

    # Check if 'c_year' is NULL
    if car.get("c_year") is None:
        print('c_year' + str(car))
```

```
            count += 1
            has_issue = True

        # Check if 'c_engine' is NULL
        if car.get("c_engine") is None:
            print('c_engine' + str(car))
            count += 1
            has_issue = True

        # Check for duplicates
        car_tuple = (car.get("c_name"), car.get("c_price"),
car.get("c_location"),
                     car.get("c_condition"), car.get("c_mileage_min"),
                     car.get("c_mileage_max"), car.get("c_year"),
car.get("c_engine"))

        if car_tuple in seen_cars:
            print(f"Duplicate car found: {car}")
            count += 1
            has_issue = True
        else:
            seen_cars.add(car_tuple)

        # If no issues, add the car to the filtered list
        if not has_issue:
            filtered_data.append(car)

print(f"☑ Before. Total rows: {len(response.data)}")
print(f"☑ After Done. Total valid rows: {len(filtered_data)}")
print(f"☑ Removed Data: {count}")
```

Appendix Snippet D: Insert Data back to Supabase

```
# Bulk insert all rows into the "cars" table in batches
batch_size = 1000  # Adjust batch size as needed
total_rows = len(filtered_data)
inserted_rows = 0

for i in range(0, total_rows, batch_size):
    batch_data = filtered_data[i:i + batch_size]  # Get a chunk of
data
    response2 =
supabase.table("cars_clean").insert(batch_data).execute()
```

```
    inserted_rows += len(batch_data)  # Update the count of inserted
rows
    print(f"Inserted {inserted_rows} of {total_rows} rows so
far...")

print(f"☑ Done. Total rows inserted: {inserted_rows}")
```

10.1.3. Sequential Processing (Pandas)

Appendix Snippet A: Import data from supabase

```
response = supabase.table("cars_clean").select("*").range(0,
999).execute()

# Set up pagination loop
batch_size = 1000
offset = 1000  # Start from after the first 1000 rows

# Continue fetching in chunks until no more data is left
while True:
    batch_response =
supabase.table("cars_clean").select("*").range(offset, offset +
batch_size - 1).execute()
    rows = batch_response.data

    if not rows:
        break

    # Add fetched rows to the response.data
    response.data.extend(rows)

    # Update offset for the next batch
    offset += batch_size

    print(f"Fetched {len(response.data)} rows so far...")

# Final result stored in response.data
print(f"☑ Done. Total rows fetched: {len(response.data)}")
```

Appendix Snippet B: Example of Query - Most Expensive Car By Location (include performance calculation)

```
def query_most_expensive_car_by_location(data):
    start_time = time.time()

    # Create a pandas DataFrame
    df = pd.DataFrame(data)

    # Filter out rows with missing or invalid prices
    df = df[df['c_price'].notna() & df['c_price'].apply(lambda x:
isinstance(x, (int, float)))]
```

```
    # Group by location and get the most expensive car
    most_expensive_cars =
df.loc[df.groupby('c_location')['c_price'].idxmax()]

    end_time = time.time()
    query_time = end_time - start_time

    cpu_percent = psutil.cpu_percent(interval=1)
    memory_info = psutil.virtual_memory()
    throughput = len(data) / query_time if query_time > 0 else 0

    # Create PrettyTable from DataFrame
    table = PrettyTable()
    table.field_names = ["ID", "Location", "Car Name", "Price"]
    for _, row in most_expensive_cars.iterrows():
        table.add_row([row["id"], row["c_location"], row["c_name"],
row["c_price"]])

    print(f"Total records processed: {len(df)}")
    return table, query_time, cpu_percent, memory_info.percent,
throughput  # Memory in MB
```

Appendix Snippet C: Example of Query - Get total cars per year (include performance calculation)

```
def query_total_cars_per_year(data):
    start_time = time.time()

    df = pd.DataFrame(data)
    cars_per_year = df['c_year'].value_counts().to_dict()

    end_time = time.time()
    query_time = end_time - start_time

    cpu_percent = psutil.cpu_percent(interval=1)
    memory_info = psutil.virtual_memory()
    throughput = len(data) / query_time if query_time > 0 else 0

    return cars_per_year, query_time, cpu_percent,
memory_info.percent, throughput
```

Appendix Snippet D: Performance Calculation Display:

```
def print_car_counts_table(car_counts, query_time, cpu_percent,
memory_percent, throughput):
```

```
    sorted_counts = sorted(car_counts.items(), key=lambda x:
x[0])[:5]

    print("\nQuery 2: Total Cars per Year (Top 5)")
    print("-" * 30)
    print("{:<10} {:<10}".format("Year", "Count"))
    print("-" * 30)
    for year, count in sorted_counts:
      print("{:<10} {:<10}".format(year, count))
    print("-" * 30)

    print("\nQuery Performance:")
    print(f"  Query Time: {query_time:.4f} seconds")
    print(f"  Average CPU Usage: {cpu_percent:.2f}%")
    print(f"  Average Memory Usage: {memory_percent:.2f}%")
    print(f"  Throughput: {throughput:.2f} records/second")
```

10.1.4. Multithreading

Appendix Snippet A: Function for Fetching All Data from Supabase

```
def fetch_all_data_from_supabase(client: Client) ->
        Optional[pd.DataFrame]:
    """
    Fetches all data from the cars_clean table in Supabase using
        pagination.

    Args:
        client: The Supabase client instance

    Returns:
        A pandas DataFrame containing all fetched data, or None if
        an error occurs
    """
    try:
        # Parameters for pagination
        page_size = 1000
        start_range = 0

        # List to store all fetched rows
        all_data = []
        cumulative_row_count = 0

        # Continue fetching until no more data is returned
        while True:
            # Fetch a batch of data using range pagination
            response = client.table("cars_clean") \
                .select("*") \
                .range(start_range, start_range + page_size - 1) \
                .execute()

            # Get the current batch of data
            batch_data = response.data

            # If no data is returned, we've reached the end
            if not batch_data:
                break

            # Add the batch to our collected data
            all_data.extend(batch_data)
```

```
            # Update the count and display progress
            cumulative_row_count += len(batch_data)
            print(f"Fetched {cumulative_row_count} rows so far...")

            # Move to the next page
            start_range += page_size

        # Create a DataFrame from all collected data
        df = pd.DataFrame(all_data)

        # Print final summary
        print(f"☑ Done. Total rows fetched: {len(df)}")

        return df

    except Exception as e:
        print(f"Error fetching data from Supabase: {e}")
        return None
```

Appendix Snippet B: Decorator for Performance Metric Collection

```
def execute_with_metrics(func):
    """Decorator to add performance metrics to processing
        functions"""
    def wrapper(*args, **kwargs):  # Modified to accept any
        arguments
        # Record start metrics
        start_time = time.time()
        start_cpu = psutil.cpu_percent(interval=None)
        start_memory = psutil.virtual_memory().percent

        # Execute function
        result_df = func(*args, **kwargs)

        # Record end metrics
        end_time = time.time()
        end_cpu = psutil.cpu_percent(interval=None)
        end_memory = psutil.virtual_memory().percent

        # Calculate performance metrics
        duration = end_time - start_time
        avg_cpu = (start_cpu + end_cpu) / 2
        avg_memory = (start_memory + end_memory) / 2
```

```
        # Modified Throughput Calculation (using input DataFrame
        length)
        input_all_data_df = args[0]  # Access the first positional
        argument
        throughput = len(input_all_data_df) / duration if duration >
        0 else 0

        return {
            "data": result_df,
            "metrics": {
                "duration": duration,
                "cpu_percent": avg_cpu,
                "memory_percent": avg_memory,
                "throughput": throughput,  # Using the modified
        throughput
                "result_count": len(result_df)
            }
        }
    return wrapper
```

Appendix Snippet C: Implementing Multithreaded Execution of Analytical Tasks

```
def run_multithreaded(all_data_df: pd.DataFrame):
    """
    Execute all data processing functions using multithreading

    Args:
        all_data_df: The pre-fetched DataFrame containing all data
    """
    print("\n===== MULTITHREADED DATA PROCESSING =====\n")

    # Define processing functions and their titles in a fixed,
        ordered list
    processing_functions_list = [
        ("Query 1: Most Expensive by Location",
        process_most_expensive_car_per_location),
        ("Query 2: Total Cars Per Year",
        process_total_cars_per_year),
        ("Query 3: Average Price By Engine Group",
        process_average_price_by_engine_group),
        ("Query 4: Total Cars By Location",
        process_total_cars_by_location),
        ("Query 5: Average Mileage By Condition",
        process_average_mileage_by_condition)
    ]

    # Convert to dictionary for ThreadPoolExecutor
    processing_functions = dict(processing_functions_list)

    # Record start time for multithreaded execution
    multithreaded_start = time.time()

    # Execute processing functions concurrently using
        ThreadPoolExecutor
    results = {}
    with concurrent.futures.ThreadPoolExecutor() as executor:
        # Submit all processing functions to the executor with the
        pre-fetched data
        future_to_query = {executor.submit(func, all_data_df): title
                           for title, func in
        processing_functions.items()}

        # Collect results as they complete
```

```
        for future in
        concurrent.futures.as_completed(future_to_query):
            query_title = future_to_query[future]
            try:
                results[query_title] = future.result()
            except Exception as e:
                print(f"{query_title} generated an exception: {e}")

    # Record end time and calculate total duration
    multithreaded_end = time.time()
    multithreaded_duration = multithreaded_end - multithreaded_start

    # Display results in the predefined order
    for query_title, _ in processing_functions_list:
        if query_title in results:
            display_query_output(query_title, results[query_title])
        else:
            print(f"\n[{query_title}]")
            print("Error: This query did not complete
        successfully.")
            print("----------------------------------------------
        -")

    print(f"\n===== MULTITHREADED PROCESSING COMPLETED =====")
    print(f"Total Execution Time: {multithreaded_duration:.4f}
        seconds")
```

Appendix Snippet D: Example Analytical Task - Average Price by Engine Group

```
@execute_with_metrics
def process_average_price_by_engine_group(all_data_df:
        pd.DataFrame):
    """
    Process 3: Calculate the average c_price grouped by c_engine
        size in 500cc intervals.
    Returns DataFrame with columns: engine_group_start_cc,
        average_price (rounded to 2 decimal places),
    limited to top 5 engine groups with highest average prices.
    """
    # Filter to relevant columns
    df = all_data_df[['c_engine', 'c_price']]

    # Create engine group intervals (500cc each)
    df['engine_group_start_cc'] = (df['c_engine'] // 500) * 500
```

```
    # Calculate average price per engine group
    result_df =
        df.groupby('engine_group_start_cc')['c_price'].mean().reset_
        index()

    # Round average_price to 2 decimal places
    result_df = result_df.rename(columns={'c_price':
        'average_price'})
    result_df['average_price'] = result_df['average_price'].round(2)

    # Sort by average_price in descending order and limit to top 5
    result_df = result_df.sort_values('average_price',
        ascending=False).head(5)

    return result_df
```

Appendix Snippet E: Function for Displaying Query Outputs and Metrics

```
def display_query_output(query_title, results):
    """
    Display query output in a formatted way

    Args:
        query_title: The title of the query
        results: Dictionary containing data and metrics returned by
        the execute_with_metrics decorator
    """
    metrics = results['metrics']
    data = results['data']

    print(f"\n[{query_title}]")
    print(data.to_string(index=False))

    print("\nQuery Performance:")
    print(f"  Query Time: {metrics['duration']:.4f} seconds")
    print(f"  Average CPU Usage: {metrics['cpu_percent']:.2f}%")
    print(f"  Average Memory Usage:
        {metrics['memory_percent']:.2f}%")
    print(f"  Throughput: {metrics['throughput']:.2f}
        records/second")
    print("-------------------------------------------------")
```

10.1.5. Multiprocessing

Appendix Snippet A: Batched Data Fetching from Supabase

```python
def fetch_car_data():
    response = supabase.table("cars_clean").select("*").range(0, 999).execute()
    batch_size = 1000
    offset = 1000

    while True:
        batch_response = supabase.table("cars_clean").select("*").range(offset, offset + batch_size - 1).execute()
        rows = batch_response.data
        if not rows:
            break
        response.data.extend(rows)
        offset += batch_size
        print(f"Fetched {len(response.data)} rows so far...")

    print(f"✅ Done. Total rows fetched: {len(response.data)}")
    return response.data
```

Appendix Snippet B: Real-time Performance Monitoring

```python
end_time = time.time()
query_time = end_time - start_time
cpu_percent = psutil.cpu_percent(interval=1)
memory_info = psutil.virtual_memory()
throughput = len(data) / query_time if query_time > 0 else 0
```

Appendix Snippet C: Parallel Data Processing with Joblib

```python
# Use numpy for balanced chunking
chunks = np.array_split(data, n_jobs)

def process_chunk(chunk):
    most_expensive_cars = {}
    for car in chunk:
        location = car.get("c_location")
        price = car.get("c_price")

        # Skip invalid or noisy location values
        if not location or location in ["Used", "c_location"]:
            continue

        if price is None or not isinstance(price, (int, float)):
            continue

        if location not in most_expensive_cars or price > most_expensive_cars[location]["price"]:
            most_expensive_cars[location] = {
                "price": price,
                "c_name": car.get("c_name"),
                "id": car.get("id")
            }
    return most_expensive_cars

results = Parallel(n_jobs=n_jobs)(delayed(process_chunk)(chunk) for chunk in chunks)
```

Appendix Snippet D: Parallelized Price Analysis by Engine Group

```python
# Parallel processing
results = Parallel(n_jobs=n_jobs)(delayed(process_chunk)(chunk) for chunk in chunks)

# Combine results
combined = defaultdict(list)
for result in results:
    for engine_group, prices in result.items():
        combined[engine_group].extend(prices)

# Compute average prices
average_prices = {
    group: sum(prices) / len(prices)
    for group, prices in combined.items()
    if prices
}

# Sort by average price descending and keep top 5
top5_avg_prices = sorted(average_prices.items(), key=lambda x: x[1], reverse=True)[:5]
```

Appendix Snippet E: Unified Query Execution with Performance Benchmarking

```python
def run_all_queries(data, n_jobs=4):
    queries = [
        ("Query 1: Most Expensive Car by Location", query_most_expensive_car_by_location),
        ("Query 2: Total Cars per Year", query_total_cars_per_year),
        ("Query 3: Avg Price by Engine Size", query_average_price_by_engine_size),
        ("Query 4: Total Cars by Location", query_total_cars_by_location),
        ("Query 5: Avg Min Mileage by Condition", query_avg_min_mileage_by_condition)
    ]

    for title, query_func in queries:
        print(f"\n{title}")
        table, time_taken, cpu, mem, throughput = query_func(data, n_jobs)
        print(table)
        print("\nQuery Performance: ")
        print(f"Time: {time_taken:.2f}s | CPU: {cpu}% | Memory: {mem}% | Throughput: {throughput:.2f} records/s\n")
```

10.1.6. Distributed Computing

Appendix Snippet A: Supabase Connection and Data Fetching

```
from supabase import create_client, Client
import pandas as pd

# Supabase connection info
url = "https://ugjwigpcopmtjgylopwf.supabase.co ".strip()
key =
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzdXBhYmFzZSIsInJlZi
I6InVnandpZ3Bjb3BtdGpneWxvcHdmIiwicm9sZSI6ImFub24iLCJpYXQiOjE3NDU4Mj
gxMjIsImV4cCI6MjA2MTQwNDEyMn0.oFcP1wCt1upByqTU8NgD4FpJUdv9I8sG1ECWMX
1wz8I"

supabase = create_client(url, key)

# Fetch data with pagination
batch_size = 1000
offset = 0
all_rows = []

while True:
    response =
supabase.table("cars_clean").select("*").range(offset, offset +
batch_size - 1).execute()
    rows = response.data
    if not rows:
        break
    all_rows.extend(rows)
    offset += batch_size
    print(f"Fetched {len(all_rows)} rows so far...")

print(f"☑ Done. Total rows fetched: {len(all_rows)}")
```

Appendix Snippet B: Start Spark Session

```
# Install required packages
!apt-get update -qq > /dev/null # Ensure apt-get is up to date
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!pip install pyspark
!pip install supabase
!pip install memory_profiler

# Verify Java installation and path
```

```
print("Verifying Java installation...")
!which java
!ls /usr/lib/jvm/java-8-openjdk-amd64/

# Set up environment variables
import os
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Set JAVA_HOME
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"

# Verify JAVA_HOME is set
print(f"JAVA_HOME set to: {os.environ.get('JAVA_HOME')}")

# Create Spark Session
spark = SparkSession.builder \
    .appName("CarAnalytics") \
    .config("spark.sql.shuffle.partitions", "8") \
    .getOrCreate()

print("☑ Spark Session created successfully.")
```

Appendix Snippet C: Convert to Spark DataFrame

```
# Convert to Pandas then Spark DataFrame
if all_rows:
    pandas_df = pd.DataFrame(all_rows)
    df = spark.createDataFrame(pandas_df)

    # Cast numeric columns
    numeric_cols = ["c_price", "c_mileage_min", "c_mileage_max",
"c_year", "c_engine"]
    for col_name in numeric_cols:
        df = df.withColumn(col_name, col(col_name).cast("int"))
else:
    raise ValueError("No data fetched from Supabase.")
```

Appendix Snippet D: Performance Measure Function

```
import time
import psutil
from memory_profiler import memory_usage

def run_query_with_metrics(query_func):
```

```
    start_time = time.time()
    cpu_start = psutil.cpu_percent(interval=0.1)

    # Measure memory usage during function execution
    mem_usage, retval = memory_usage((query_func, (), {}),
max_iterations=1, retval=True)

    end_time = time.time()
    duration = end_time - start_time
    cpu_end = psutil.cpu_percent(interval=0.1)

    avg_cpu = (cpu_start + cpu_end) / 2
    total_memory_mb = psutil.virtual_memory().total / (1024 * 1024)
    peak_mem_mb = (max(mem_usage)) * 1.048576  # in MiB
    peak_mem_percent = (peak_mem_mb / total_memory_mb) * 100
    record_count = retval.count()  # assuming query_func returns a
DataFrame
    throughput = record_count / duration if duration > 0 else 0

    print(f"☑ Duration: {duration:.2f}s")
    print(f"🔄 Avg CPU Usage: {avg_cpu:.2f}%")
    print(f"💾 Peak Memory Usage: {peak_mem_percent:.2f} %")
    print(f"📈 Throughput: {throughput:.2f} records/sec\n")

    return retval
```

Appendix Snippet E: Example Query Execution Using PySpark for Average Price By Engine Size

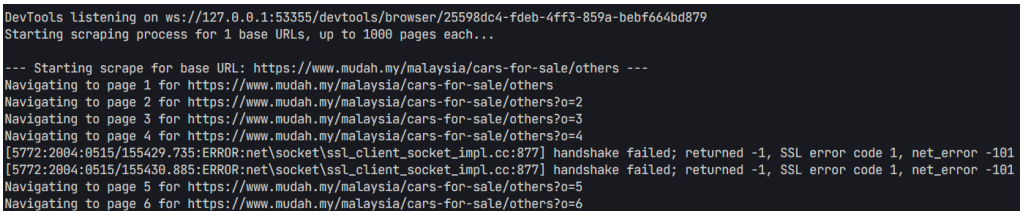
```
from pyspark.sql.functions import col, floor, round, avg

def query3():
    interval = 500

    filtered_df = df.filter(col("c_engine").isNotNull())

    # Create engine size groups in 500cc intervals
    df_with_group = filtered_df.withColumn(
        "engine_group",
        floor(col("c_engine") / interval) * interval
    )

    # Compute average price per engine group
```

```
    avg_price_by_engine = df_with_group.groupBy("engine_group") \
        .agg(round(avg("c_price"), 2).alias("avg_price"))

    # Sort by avg_price descending and take top 5
    result = avg_price_by_engine.orderBy(col("avg_price").desc())

    # Show only the top 5 results
    print("Average Price By Engine Size (Top 5 Highest):")
    result.limit(5).show(truncate=False)

    return result

run_query_with_metrics(query3)
```

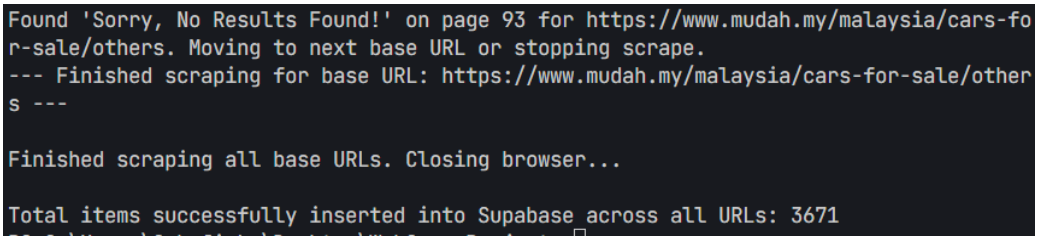
10.2. Screenshots of output

10.2.1. Web Scraping

Appendix A: Web Scrape Initialize and start scrape process

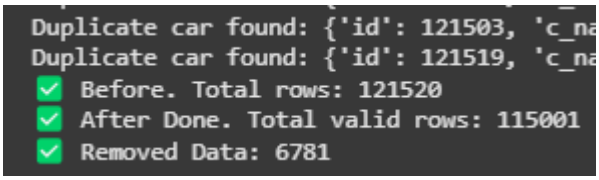


Appendix B: Stop Web Scrape (Example of 3671 data)

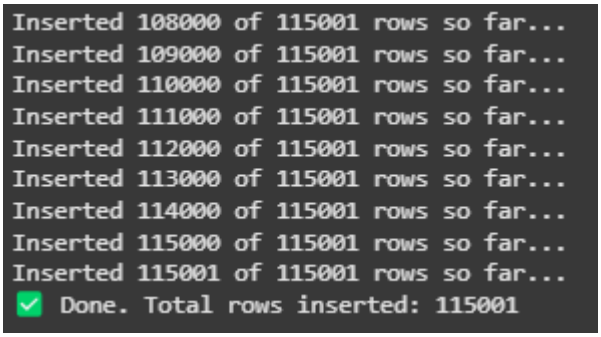


10.2.2. Data Cleaning

Appendix A: Remove Duplicate data



Appendix B: Insert data back to supabase



10.2.3. Multithreading

Appendix Snippet A: Most Expensive by Location

```
[Query 1: Most Expensive by Location]
     c_location                                          c_name  c_price
          Johor Ferrari SF90 STRADALE 3.9 *1000hp Ready Stock  4180000
          Kedah Mclaren 720S 765LT SPIDER 1 OF 765 READY UNIT  1880000
       Kelantan  Land Rover RANGE ROVER 3.0 Big Spec P400 HST   548000
   Kuala Lumpur             Rolls Royce PHANTOM  6.75 V12 EWB  4900000
         Labuan  Hyundai GRAND STAREX EXECUTIVE PLUS 2.5L (A)   125000
         Melaka             Porsche 911 3.8 CARRERA S 991 (A)   489000
Negeri Sembilan               Ferrari F12 BERLINETTA 6.3 (A)  1010000
         Pahang  Mercedes Benz S63 4.0 AMG COUPE V8 BiTurbo S   707000
         Penang Rolls Royce CULLINAN 6.75 V12 (A) BLACK BADGE  3498888
          Perak                 Porsche 992 3.0 CARRERA S (A)   828000
         Perlis    Bmw 340i xDRIVE M SPORT PRO M 3.0 MY19 G20   343800
      Putrajaya Mercedes Benz G350 D 3.0 AMG (A) S/ROOF UNREG   568000
          Sabah                  Lexus LX500d 3.3 F SPORT (A)   889000
        Sarawak    Ferrari 488 Pista 3.9 V8 Twin-turbocharged  2660000
       Selangor            Ferrari 812 Competizione Rosso TRS  8988000
     Terengganu                  Honda CIVIC 1.8 S i-VTEC (A)  2700000

Query Performance:
  Query Time: 0.0796 seconds
  Average CPU Usage: 56.10%
  Average Memory Usage: 10.40%
  Throughput: 1444293.20 records/second
------------------------------------------------
```

Appendix Snippet B: Total Cars per Year

```
[Query 2: Total Cars Per Year]
 c_year  total_cars
   1995        1349
   1996         374
   1997         493
   1998         188
   1999         318

Query Performance:
  Query Time: 0.0524 seconds
  Average CPU Usage: 83.35%
  Average Memory Usage: 10.40%
  Throughput: 2196769.87 records/second
------------------------------------------------
```

Appendix Snippet C: Average Price by Engine Group

```
[Query 3: Average Price By Engine Group]
 engine_group_start_cc  average_price
                  6500     2350916.15
                  6000     2230549.17
                 99500     1184000.00
                  3500      924417.12
                  5000      825514.12

Query Performance:
  Query Time: 0.0896 seconds
  Average CPU Usage: 75.00%
  Average Memory Usage: 10.40%
  Throughput: 1284182.32 records/second
------------------------------------------------
```

Appendix Snippet D: Total Cars by Location

```
[Query 4: Total Cars By Location]
     c_location  total_cars
          Johor       18904
          Kedah        3245
       Kelantan        1441
   Kuala Lumpur       31232
         Labuan          33
         Melaka        1647
Negeri Sembilan        1552
         Pahang        1509
         Penang        6312
          Perak        5107
         Perlis         143
      Putrajaya         197
          Sabah        3440
        Sarawak        2268
       Selangor       36908
     Terengganu        1063

Query Performance:
  Query Time: 0.0743 seconds
  Average CPU Usage: 50.00%
  Average Memory Usage: 10.40%
  Throughput: 1546898.19 records/second
------------------------------------------------
```

Appendix Snippet E: Average Min Mileage by Condition

```
[Query 5: Average Mileage By Condition]
c_condition  average_min_mileage
        New          3906.976744
      Recon         21719.844845
       Used         99596.602852

Query Performance:
  Query Time: 0.0817 seconds
  Average CPU Usage: 25.00%
  Average Memory Usage: 10.40%
  Throughput: 1406942.57 records/second
--------------------------------------------------
```

10.2.4. Multiprocessing

Appendix Snippet A: Most Expensive by Location

```
Query 1: Most Expensive Car by Location
+--------+-----------------+-----------------------------------------------+---------+
|   ID   |     Location    |                    Car Name                   |  Price  |
+--------+-----------------+-----------------------------------------------+---------+
| 114269 |      Sabah      |          Lexus LX500d 3.3 F SPORT (A)         |  889000 |
|  4697  |      Kedah      | Mclaren 720S 765LT SPIDER 1 OF 765 READY UNIT | 1880000 |
| 88400  |      Johor      | Ferrari SF90 STRADALE 3.9 *1000hp Ready Stock | 4180000 |
| 86014  |   Kuala Lumpur  |       Rolls Royce PHANTOM  6.75 V12 EWB       | 4900000 |
|  6984  |     Selangor    |       Ferrari 812 Competizione Rosso TRS      | 8988000 |
|  2327  |    Terengganu   |          Honda CIVIC 1.8 S i-VTEC (A)         | 2700000 |
| 90583  |      Perak      |         Porsche 992 3.0 CARRERA S (A)         |  828000 |
| 87864  |      Pahang     |  Mercedes Benz S63 4.0 AMG COUPE V8 BiTurbo S |  707000 |
| 88258  | Negeri Sembilan |         Ferrari F12 BERLINETTA 6.3 (A)        | 1010000 |
| 111306 |      Penang     | Rolls Royce CULLINAN 6.75 V12 (A) BLACK BADGE | 3498888 |
| 117166 |     Kelantan    |  Land Rover RANGE ROVER 3.0 Big Spec P400 HST |  548000 |
| 90503  |     Sarawak     |   Ferrari 488 Pista 3.9 V8 Twin-turbocharged  | 2660000 |
| 92133  |      Melaka     |       Porsche 911 3.8 CARRERA S 991 (A)       |  489000 |
| 111030 |    Putrajaya    | Mercedes Benz G350 D 3.0 AMG (A) S/ROOF UNREG |  568000 |
| 118043 |      Labuan     |  Hyundai GRAND STAREX EXECUTIVE PLUS 2.5L (A) |  125000 |
| 86503  |      Perlis     |   Bmw 340i xDRIVE M SPORT PRO M 3.0 MY19 G20  |  343800 |
+--------+-----------------+-----------------------------------------------+---------+

Query Performance:
Time: 1.34s | CPU: 2.5% | Memory: 10.9% | Throughput: 85671.76 records/s
```

Appendix Snippet B: Total Cars per Year

```
Query 2: Total Cars per Year
+------+-------+
| Year | Count |
+------+-------+
| 1995 |  1349 |
| 1996 |  374  |
| 1997 |  493  |
| 1998 |  188  |
| 1999 |  318  |
+------+-------+

Query Performance:
Time: 0.40s | CPU: 3.5% | Memory: 10.9% | Throughput: 287575.11 records/s
```

Appendix Snippet C: Average Price by Engine Group

```
Query 3: Avg Price by Engine Size
+-----------------------+---------------+
| Engine Size Group (cc) | Average Price |
+-----------------------+---------------+
|         6500cc        | $2,350,916.15 |
|         6000cc        | $2,230,549.17 |
|        99500cc        | $1,184,000.00 |
|         3500cc        |  $924,417.12  |
|         5000cc        |  $825,514.12  |
+-----------------------+---------------+

Query Performance:
Time: 0.44s | CPU: 14.6% | Memory: 11.3% | Throughput: 264270.50 records/s
```

Appendix Snippet D: Total Cars by Location

```
Query 4: Total Cars by Location
+-----------------+-------+
|     Location    | Count |
+-----------------+-------+
|      Johor      | 18904 |
|      Kedah      |  3245 |
|     Kelantan    |  1441 |
|   Kuala Lumpur  | 31232 |
|      Labuan     |   33  |
|      Melaka     |  1647 |
| Negeri Sembilan |  1552 |
|      Pahang     |  1509 |
|      Penang     |  6312 |
|      Perak      |  5107 |
|      Perlis     |  143  |
|    Putrajaya    |  197  |
|      Sabah      |  3440 |
|     Sarawak     |  2268 |
|     Selangor    | 36908 |
|    Terengganu   |  1063 |
+-----------------+-------+

Query Performance:
Time: 0.65s | CPU: 57.8% | Memory: 11.1% | Throughput: 177178.63 records/s
```

Appendix Snippet E: Average Min Mileage by Condition

```
Query 5: Avg Min Mileage by Condition
+-----------+-----------------+
| Condition | Avg Min Mileage |
+-----------+-----------------+
|    Used   |     99596.60    |
|    New    |     3906.98     |
|   Recon   |     21719.84    |
+-----------+-----------------+

Query Performance:
Time: 0.56s | CPU: 3.0% | Memory: 11.2% | Throughput: 206451.23 records/s
```

10.2.5. Distributed Computing

Appendix Snippet A: Most Expensive by Location

```
Most Expensive Car in Each Location:
+------+---------------+---------------------------------------------+-------+
|id    |c_location     |c_name                                       |c_price|
+------+---------------+---------------------------------------------+-------+
|88400 |Johor          |Ferrari SF90 STRADALE 3.9 *1000hp Ready Stock|4180000|
|4697  |Kedah          |Mclaren 720S 765LT SPIDER 1 OF 765 READY UNIT|1880000|
|117166|Kelantan       |Land Rover RANGE ROVER 3.0 Big Spec P400 HST |548000 |
|86014 |Kuala Lumpur   |Rolls Royce PHANTOM  6.75 V12 EWB            |4900000|
|118043|Labuan         |Hyundai GRAND STAREX EXECUTIVE PLUS 2.5L (A) |125000 |
|92133 |Melaka         |Porsche 911 3.8 CARRERA S 991 (A)            |489000 |
|88258 |Negeri Sembilan|Ferrari F12 BERLINETTA 6.3 (A)               |1010000|
|87864 |Pahang         |Mercedes Benz S63 4.0 AMG COUPE V8 BiTurbo S |707000 |
|111306|Penang         |Rolls Royce CULLINAN 6.75 V12 (A) BLACK BADGE|3498888|
|90583 |Perak          |Porsche 992 3.0 CARRERA S (A)                |828000 |
|86503 |Perlis         |Bmw 340i xDRIVE M SPORT PRO M 3.0 MY19 G20   |343800 |
|111030|Putrajaya      |Mercedes Benz G350 D 3.0 AMG (A) S/ROOF UNREG|568000 |
|114269|Sabah          |Lexus LX500d 3.3 F SPORT (A)                 |889000 |
|90503 |Sarawak        |Ferrari 488 Pista 3.9 V8 Twin-turbocharged   |2660000|
|6984  |Selangor       |Ferrari 812 Competizione Rosso TRS           |8988000|
|2327  |Terengganu     |Honda CIVIC 1.8 S i-VTEC (A)                 |2700000|
+------+---------------+---------------------------------------------+-------+

✅ Duration: 1.25s
🧠 Avg CPU Usage: 9.75%
💾 Peak Memory Usage: 3.37 %
📈 Throughput: 12.79 records/sec
```

Appendix Snippet B: Total Cars per Year

```
Total Cars by Year:
+------+-----+
|c_year|count|
+------+-----+
|  1995| 1349|
|  1996|  374|
|  1997|  493|
|  1998|  188|
|  1999|  318|
+------+-----+

✅ Duration: 1.33s
🧠 Avg CPU Usage: 32.50%
💾 Peak Memory Usage: 3.37 %
📈 Throughput: 23.25 records/sec
```

Appendix Snippet C: Average Price by Engine Group

```
Average Price By Engine Size (Top 5 Highest):
+------------+----------+
|engine_group|avg_price |
+------------+----------+
|6500        |2350916.15|
|6000        |2230549.17|
|99500       |1184000.0 |
|3500        |924417.12 |
|5000        |825514.12 |
+------------+----------+

✅ Duration: 1.61s
🧠 Avg CPU Usage: 76.20%
💾 Peak Memory Usage: 3.37 %
📈 Throughput: 9.93 records/sec
```

Appendix Snippet D: Total Cars by Location

```
Total Cars by Location:
+---------------+-----+
|     c_location|count|
+---------------+-----+
|          Johor|18904|
|          Kedah| 3245|
|       Kelantan| 1441|
|   Kuala Lumpur|31232|
|         Labuan|   33|
|         Melaka| 1647|
|Negeri Sembilan| 1552|
|         Pahang| 1509|
|         Penang| 6312|
|          Perak| 5107|
|         Perlis|  143|
|      Putrajaya|  197|
|          Sabah| 3440|
|        Sarawak| 2268|
|       Selangor|36908|
|     Terengganu| 1063|
+---------------+-----+

✅ Duration: 1.02s
🧠 Avg CPU Usage: 37.65%
💾 Peak Memory Usage: 3.37 %
📈 Throughput: 15.68 records/sec
```

Appendix Snippet E: Average Min Mileage by Condition

```
Average Min Mileage by Condition:
+-----------+---------------+
|c_condition|avg_min_mileage|
+-----------+---------------+
|       Used|        99596.6|
|      Recon|       21719.84|
|        New|        3906.98|
+-----------+---------------+

✅ Duration: 0.94s
🧠 Avg CPU Usage: 20.80%
💾 Peak Memory Usage: 3.37 %
📈 Throughput: 3.19 records/sec
```

10.3. Links to full code repo or dataset

10.3.1. Web Scraping

https://github.com/drshahizan/HPDP/blob/main/2425/project/p1/CrawlOps/webScrape.py

10.3.2. Supabase Dataset

10.3.2.1. Scraped Data (in csv):

https://github.com/drshahizan/HPDP/blob/main/2425/project/p1/CrawlOps/data/raw_data.csv

10.3.2.2. Cleaned Data (in csv):

https://github.com/drshahizan/HPDP/blob/main/2425/project/p1/CrawlOps/data/cleaned_data.csv

10.3.3. Data Cleaning

https://colab.research.google.com/drive/1kNjBb4VWe5vKj5ANYFEMUS3tICHTOx8c?usp=sharing

10.3.4. Sequential

https://colab.research.google.com/drive/1IQT9O1j2DK5ezOY2_JkD_mDsHJKKaxgp?usp=sharing

10.3.5. Multithreading

https://colab.research.google.com/drive/10oxpSehCc8aFohLEZT44IhxqEsBUt0cI?usp=sharing

10.3.6. Multiprocessing

https://colab.research.google.com/drive/18W75VT1qJsMPc5XMvJfHlYC_X-6_plI5?usp=sharing

10.3.7. Distributed Computing

https://colab.research.google.com/drive/1CKgZKrX0ZxMpJX_Ja2CKo_ot-P8slgfX?usp=sharing