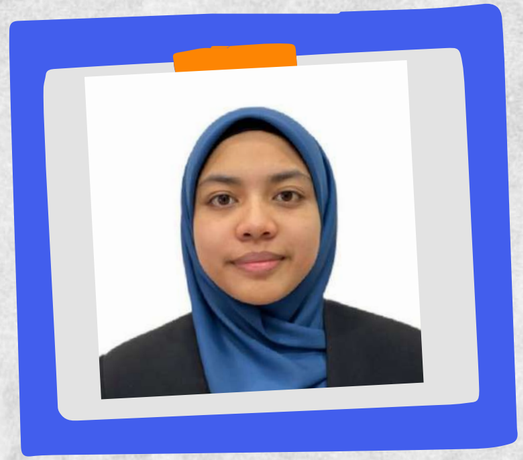
Optimizing High-Performance Data Processing for Large-Scale Web Crawlers

Presented by CrawlOps

Our Team Member

GOH JIALE

KOH LI HUI



MAISARAH RIZAL

YONG WERN JIE

Table of Content

Introduction

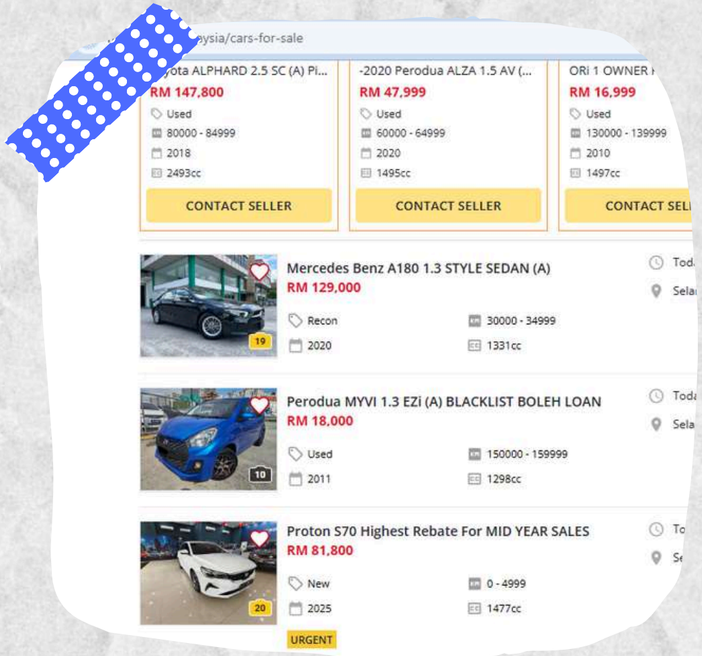
The Challenge:
In today's digital world, handling massive amounts of web data efficiently is key. Our project tackled this by building a system to scrape and process over 100,000 car listings from a real Malaysian website.

Why This Matters: This project provides hands-on experience with real-world data challenges and the impact of optimization on performance - crucial skills for data professionals.

Our Goal: We aimed to:

- Successfully scrape a large dataset (121,520 records from Mudah.my).
- Clean and store this data effectively in Supabase.
- Analyze the data using 5 key queries.
- Optimize & Compare: Test and evaluate different high-performance techniques:
  - Sequential (Baseline)
  - Multithreading
  - Multiprocessing (joblib)
  - Distributed Computing (Spark)

Web Scraping



The project used Selenium and BeautifulSoup to scrape 121,520 car listings from mudah.my, collecting data on car name, price, location, condition, mileage, year, and engine capacity. Data was cleaned and stored in a Supabase database. Ethical practices included rate-limiting, error handling, and adherence to the website's terms.

Data Processing

1

Data from Supabase was accessed in Google Colab using the Supabase Python SDK, fetched in 1,000-row batches, and stored in the cars_before_clean table. It was cleaned by normalizing mileage, converting engine size, validating fields, and removing duplicates, then saved in cars_clean.

2

Cleaned data was transformed for consistency, with mileage split into numeric fields and type-casting for integers. It was inserted in batches of 1,000 rows into cars_clean, with real-time progress tracking.

Data Before Processing

id int8	c_name text	c_price int4	c_location text	c_condition text	c_mileage text	c_year int4	c_engine text
1	Proton SATRIA 1.3 GL (M)	6200	Sabah	Used	5000 - 9999	1995	1299cc
2	Perodua KEMBARA 1.3 GX (M)	2500	Kedah	Used	200000 - 249999	1998	1296cc
3	KENARI 1.0 L9 auto 2006	6500	Johor	Used	25000 - 29999	2006	989cc
4	Mercedes Benz GLA45 AMG 4MATIC (CE	168000	Kuala Lumpur	Used	70000 - 74999	2015	1991cc
5	Toyota HILUX 2.4 V (A) CAR KING WARRA	97700	Selangor	Used	65000 - 69999	2021	2393cc
6	Isuzu D-MAX 3.0 Ddi iTEQ 4×4 (A)	14000	Selangor	Used	85000 - 89999	2008	2999cc
7	Perodua KEMBARA 1.3 CT EX (M)	4800	Kedah	Used	150000 - 159999	1999	1296cc

Multiprocessing

Goal: Speed up query execution using parallel processes.

Library used: jobib

How it Works:

- Data split into chunks.
- Each chunk processed by a separate process.
- Results merged after processing.

Key Takeways:

- Parallel execution - faster than sequential for some tasks.
- Fault-tolerant - each process runs independently.
- Overhead in data chunking and process synchronization.

Multithreading

Core Approach:

1. **Central Data:** Loaded all 115,001 cleaned records from Supabase into one main Pandas DataFrame first.
2. **Concurrent Processing:** Executed our five analytical queries (as Pandas-based functions) at the same time using Python's concurrent.futures.ThreadPoolExecutor.

Why It Worked for Us

- Efficiently processed the pre-fetched in-memory DataFrame by overlapping the execution of our five distinct Pandas analysis tasks.
- Lower overhead compared to heavier parallelization methods for our dataset size.

Performance Snapshot

- **Measured:** Execution Time, CPU & Memory (psutil), Throughput (input records/sec).
- **Result:** Consistently the fastest method in our project for overall query completion.

Distributed Computing

PySpark was used as the distributed computing library for scalable and fast processing. It enables parallel computation and in-memory operations, making it ideal for big data. With PySpark, we efficiently answered complex queries like finding the most expensive car per location and average prices by engine size using distributed transformations.

Performance Comparison

Method	Query Time ⏱️	Throughput 📈	CPU Usage 💻	Memory Usage 🧠
Sequential	Moderate	Moderate	Low	Moderate
Multithreading	✅ Best	✅ Best	High	Moderate
Multiprocessing	Mixed	Moderate	Moderate	❌ Highest
Distributed Computing	❌ Worst	❌ Worst	Moderate	✅ Lowest

Conclusion

🟢 Project Highlists:

- Scraped 121,520 car listings from Mudah.my.
- Cleaned and processed 115,001 valid records stored in Supabase.
- Applied and evaluated 4 data processing techniques across 5 analytical queries.

🔑 Key Insight:

Optimization must match workload size and system context. For medium-scale, I/O-bound tasks, simpler concurrency models like multithreading with Pandas outperform heavier frameworks like Spark. More complex solutions don't always yield better results.

🏁 Performance Ranking:

1. **Multithreading (conccurent.features):** Best performance; fastest execution and highest throughput.
2. **Sequential (Baseline):** Strong baseline, efficient for moderate-sized datasets.
3. **Multiprocessing (joblib):** Some improvements; limited by overhead and data splitting.
4. **Distributed Computing (PySpark):** Lowest performance due to coordination overhead on medium-sized data.

Thank you