UTM
UNIVERSITI TEKNOLOGI MALAYSIA
Department of Computer Science
Faculty of Computing

Assignment 2: Mastering Big Data Handling

Programme	:	Bachelor of Computer Science (*Data Engineering*)
Subject Code	:	SECP3133
Subject Name	:	High Performance Data Processing
Session-Sem	:	2024/2025-2

Prepared by : KEK JESSLYN (A22EC0057)
TAN JUN YUAN (A22EC0107)

Section : 01

Lecturer : Dr Mohd Shahizan bin Othman

Date : 31-05-2025

Table of Content

1.0 Introduction

1.1 Background of the Project

Nowadays, with big data becoming more common, many companies rely on their ability to process and analyse huge amounts of data to get useful insights. However, the usual data processing tools often can't cope well with very large datasets, especially when the file size goes beyond a few hundred megabytes due to memory problems and slow processing.

This project looks into how we can handle big data more efficiently using Python. By using a real dataset that's over 2GB in size, students will get hands-on experience with practical techniques like loading only required data, processing data in chunks, reducing memory usage through data type tweaks, sampling and speeding things up with parallel computing.

The goal is to explore how these different methods can improve performance, especially in terms of memory usage, processing time, and how easy the data is to work with. This kind of project is especially helpful for those aiming to work as data engineers or analysts, where managing large-scale data efficiently is a daily task.

1.2 Objectives

- To build hands-on skills in handling large datasets (more than 700MB) using Python.
- To try out and apply different big data techniques like chunking, sampling, data type tuning, and parallel processing.
- To compare how well normal (traditional) methods perform versus more optimized ways of loading and processing data.
- To understand the pros and cons when it comes to memory usage, speed, and coding difficulty while working with large files.
- To come up with useful insights and suggestions that can help improve how we manage big datasets in real work settings.

1.3 Target Website and Data to be Extracted

The dataset used for this assignment was taken from Kaggle. The main file we're working with is **Book_rating.csv**, which comes from a bigger collection of Amazon book reviews. The file is around **2.86GB** in size which is big enough to really test out different big data processing methods.

This dataset contains **millions of book reviews and ratings** made by users, plus extra info like the book title, author name, user ID, and the review text. With all this data, it's great not just for exploring patterns and trends but also for testing how well different data loading and processing techniques work.

2.0 Dataset Overview and Inspection

2.1 Dataset Description

- **Source:** Kaggle (Amazon Books Reviews Dataset) https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews
- **File Used:** Book_rating.csv
- **Size:** 2.86 GB
- **Domain:** E-commerce / Literature / User Reviews
- **Number of Records:** Over 10 million entries (3000000 rows x 10 columns)
- **Features/Columns:**
    - **Id:** The Id of Book
    - **Title:** Book Title
    - **Price:** The price of Book
    - **User_id:** Id of user who rate the book
    - **profileName:** Name of user who rate the book
    - **review/helpfulness:** Helpfulness rating of the review, e.g. 2/3
    - **review/score:** Rating from 0 to 5 for the book
    - **review/time:** Time of given the review
    - **review/summary:** The summary of text review
    - **review/text:** The full text of a review

2.2 Initial Loading

The dataset was initially loaded using **Pandas** in Google Colab.

```
file =('amazon-books-reviews/Books_rating.csv')
pandas_df = pd.read_csv(file)
```

On standard Colab runtimes (≈12GB RAM), it was able to load this large dataset successfully, but it was not optimal and could slow down further processing. Additionally, due to its large size (2.86 GB), direct loading into memory using pd.read_csv() without optimisations typically leads to memory errors or sluggish performance.

2.3 Basic Inspection

Once the file was loaded, a basic inspection of the dataset was conducted to understand its structure and contents.

```
print("==================== Inspect of Data ====================\n")

print(f"Shape: {df_whole.shape}\n")

df_whole.info()
```

Output:

```
==================== Inspect of Data ====================

Shape: (3000000, 10)

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3000000 entries, 0 to 2999999
Data columns (total 10 columns):
 #   Column              Dtype
---  ------              -----
 0   Id                  object
 1   Title               object
 2   Price               float64
 3   User_id             object
 4   profileName         object
 5   review/helpfulness  object
 6   review/score        float64
 7   review/time         int64
 8   review/summary      object
 9   review/text         object
dtypes: float64(2), int64(1), object(7)
memory usage: 228.9+ MB
```

Observations:

- The dataset has about **3 million rows** and **10 columns**, which means there are around **30 million data points** in total.
- Most of the columns (7 out of 10) are **object type** (usually text), which normally take up more memory and might be improved with type optimisation.
- Only **three columns are numeric**:
    - Price and review/score are in **float64**,
      review/time is in **int64**.
- When loaded fully into **Pandas**, the dataset uses around **229MB of memory**. That's still okay for most systems, but optimising it can help improve processing speed and efficiency.
- Columns like review/text, review/summary, and profileName have **text data**, which can vary a lot in length and might be sparse

3.0 Big Data Handling Strategies and Loading Data Using Different Libraries

3.1 Part 1 (5 Big Data Strategies)

In part 1 of handling our dataset, we performed five big data handling strategies by using **Pandas** library to see which strategy has a better performance.

3.1.1 Load Less Data

```
# Load only required column
df_less = pd.read_csv(file, usecols = ["Id", "Title", "Price", "User_id",
                                       "review/helpfulness", "review/score",
                                        "review/time"])

display(df_less.head(10))
print(f"Total rows: {df_less.shape[0]}")
print(f"Total columns: {df_less.shape[1]}\n\n\n")
```

3.1.2 Chunking

```
#Perform chunking
df_chunk = pd.read_csv(file, chunksize=500_000)
total_rows = 0

for chunk in df_chunk:
  total_rows += chunk.shape[0]
  display(chunk.head(10))
  print(f"Total rows: {chunk.shape[0]}")
  print(f"Total columns: {chunk.shape[1]}\n")
```

3.1.3 Optimize Data Type

```
def optimizedDType(df):

    df_optimized = df.copy()

    for col in df_optimized.columns:
        col_dtype = df_optimized[col].dtype

        if pd.api.types.is_integer_dtype(col_dtype):
            df_optimized[col] = pd.to_numeric(df_optimized[col], downcast='integer')

        elif pd.api.types.is_float_dtype(col_dtype):
            df_optimized[col] = pd.to_numeric(df_optimized[col], downcast='float')

    return df_optimized

df_optimized = pd.read_csv(file)
```

3.1.4 Sampling

```
#Random sampling 10 percent of the dataset
df_sampled = pd.read_csv(file).sample(frac=0.1, random_state=42)

display(df_sampled.head(10))
print(f"Total rows: {df_sampled.shape[0]}")
print(f"Total columns: {df_sampled.shape[1]}\n\n\n")
```

3.1.5 Parallel Processing with Dask

```
df_dask= dd.read_csv(file, dtype={'Id':'object'})

display(df_dask.head(10))

df_shape = df_dask.shape
n_rows = df_dask.shape[0].compute()
n_cols = df_dask.shape[1]
print(f"Total rows: {n_rows}")
print(f"Total columns: {n_cols}\n\n\n")
```

3.2 Part 2 (Loading dataset with different libraries)

In part 2 of handling our dataset, we performed the action of loading the dataset by using different libraries such as **Pandas, Polars** and **Dask** libraries to see which strategy has a better performance.

3.2.1 Using Pandas library

```
file =('amazon-books-reviews/Books_rating.csv')
pandas_df = pd.read_csv(file)
```

3.2.2 Using Polars library

```
file =('amazon-books-reviews/Books_rating.csv')
polars_df = pl.read_csv(file)
```

3.2.3 Using Dask library

```
file =('amazon-books-reviews/Books_rating.csv')
dask_df = dd.read_csv(file, dtype={'Id':'object'})
```

4.0 Comparative Analysis

4.1 Metrics Used

- Code execution time (s)
- Peak memory usage (MB)
- CPU usage (%)
- Throughput (rows/s)

4.2 Results Summary

4.2.1 Part 1 (Comparing Between Strategies)

Metrics	Pandas				Parallel Processing with Dask
	Load Less Data	Chunking	Optimize Data Type	Sampling	
Code Execution Time (s)	23.3248	58.7441	58.0145	60.2045	77.7479
Peak Memory Usage (MB)	742.9210	1190.1397	3856.6556	3856.7046	447.7110
CPU Usage (%)	13.6	4.4	30.6	4.0	4.0
Throughput (rows/s)	128618.50	51068.93	51711.24	4983.01	38586.26

4.2.2 Part 2 (Comparing Between Libraries)

Metrics/Libraries	Pandas	Polars	Dask
Code Execution Time (s)	55.8117	13.4898	81.2019
Peak Memory Usage (MB)	3856.6912	0.0861	428.5369
CPU Usage (%)	20.0	40.5	28.0
Throughput (rows/s)	53752.17	222389.96	36944.93

4.3 Discussions and Key Observation

4.3.1 Part 1 (Comparing Between Strategies)

Several data processing techniques from Pandas are compared in the Part 1 section and Dask is introduced for parallel processing. The fastest way to execute code is to "Load Less Data" (23.3248s), then to "Optimize Data Type" (58.0145s), then to "Chunking" (58.7441s) and finally to "Sampling" (60.2045s). In this case, Dask takes the longest at 77.7479 seconds which suggests parallelization might not be worth it for smaller or medium-sized datasets.

The strategy of "Load Less Data" uses the least memory (742.9210 MB) compared to the other three Pandas-based approaches which each use about 3856 MB. Dask uses less memory (447.7110 MB) than Pandas which proves how well its chunked computation model works.

The CPU is used the most by the "Optimize Data Type" option (30.6%) and the rest of the options, including "Chunking", "Sampling" and Dask, use a very small amount (roughly 4% each). Therefore, changing data types can use a lot of CPU power because of the effort needed to convert them.

Among the strategies, "Load Less Data" performs best, handling 128,618.50 rows per second which is more than double the others. Both "Chunking" and "Optimize Data Type" have about the same speed (~51,000 rows/s) and Dask reaches 38,586.26 rows/s which is faster than "Sampling" but slower than most Pandas-based strategies. This suggests Dask is more suitable when memory is more important than speed.

4.3.2 Part 2 (Comparing Between Libraries)

This section looks at how Pandas, Polars and Dask libraries perform compared to each other. Polars is the top performer in most of the key metrics. It finishes the most quickly (13.4898s), uses the least amount of memory (0.0861 MB) and processes the most rows per second (222,389.96). It demonstrates that Polars is faster because its backend is written in Rust and it uses efficient multi-threading.

Pandas is a well-known tool, but it takes the most time (55.8117s) and uses the largest amount of memory (3856.6912 MB). It also uses a moderate amount of CPU (20%) and has low throughput (53752.17 rows/s) which means it struggles with large data or when speed is important.

Dask, which is made for parallel processing, demonstrates a mix of outcomes. Although it uses less memory than Pandas (428.5369 MB), it takes the longest to run (81.2019s) and has the lowest number of rows processed per second (36944.93). The fact that it uses 28% of the CPU means it takes up a lot of resources for handling distributed tasks.

5.0 Conclusion and Reflection

5.1 Summary of Findings

It is clear from the evaluation that using the “Load Less Data” method in Pandas provides the best performance and is therefore suitable for tasks that require high speed. Even though “Optimize Data Type” and “Chunking” are not very fast, they use a lot of memory and “Optimize Data Type” also uses the most CPU. The “Sampling” strategy uses less CPU power, but it is not efficient when processing a lot of data because it delivers the weakest throughput. Dask’s parallel approach saves memory, but it generally takes longer and offers less throughput than Pandas’ best techniques which means parallelization may not be necessary for small datasets.

In every important performance metric, Polars is much faster than Pandas and Dask. It has the shortest execution time, requires the least memory and provides the greatest throughput which is perfect for large and fast data workloads. While pandas are useful and used by many, they are not as fast or efficient with memory as other libraries. Dask is designed for parallel and distributed computing and uses less memory than Pandas, but it is slower for most tasks which means its benefits are best seen in very large-scale or distributed environments. In general, Polars is the best library for handling data processing that requires speed, low memory usage and high throughput.

5.2 Benefits and Limitations of Each Strategies

	Benefits	Limitations
Load Less Data	● Among all strategies, this one took the least time to run. ● The highest throughput makes handling data very efficient. ● Because it uses very little memory, it is suitable for places with limited memory. ● Useful when there is only need to look at a small part of the data.	● May cause insights to be lost if important details are found in the excluded data. ● This approach is not suitable when processing the whole dataset which is necessary (for example, in training models or combining all the data).
Chunking	● Processes big files in sections, so the memory does not get overloaded.	● The program executes longer because it has to read and process the data more than once.

	• Less RAM is used when loading the small dataset than when loading the full dataset. • Prevents crashes that happen because of memory errors.	• The data is processed at a slow rate. • Makes the code more complicated because it needs to manage and merge the results from different chunks.
Optimize Data Type	• Reduces the amount of memory used for future calculations when used properly. • Improves how well the system works after data has been optimized and less RAM is needed.	• During optimization, the program used a lot of memory which might be because of temporary conversions or copying. • The high CPU usage indicates that the application uses a lot of system resources. • May result in losing data or encountering errors when the wrong data type is used (for example, changing float to int).
Sampling	• Analyzing is allowed and testing quickly using a small sample of the data. • The CPU is being used very little and memory is used less than it is when processing all the data. • Can be used to create models and test ideas.	• Because the throughput is low and the execution time is long, the performance is not efficient. • If the way samples are chosen is biased, the findings may not reflect the whole population. • Should not be used for final production or analysis that needs all the data.
Parallel Processing with Dask	• Since the model uses very little memory, it is perfect for working with large datasets that RAM cannot handle. • Can be used in programs that use multiple cores or distributed systems. • Good for situations where data does not fit in memory and for real-time streaming.	• The longest execution time suggests that there is some overhead involved in setting up the parallelization. • Because of the low throughput, the system is not efficient for handling small tasks.

Pandas	● Many people use and support it and there is a lot of information available. ● Many functions: Allows users to handle, clean, transform and study data in many ways. ● Easy to pick up and work with, especially for those who are just starting to code. ● Easy to use with other Python libraries, including NumPy, Matplotlib and Scikit-learn.	● Too much memory is used: Not recommended for datasets that are larger than the system's RAM. ● It takes more time to execute queries than newer libraries like Polars. ● Default is single-threaded: Does not fully use multi-core processors which can make large-scale operations run slower. ● A lower throughput rate means the system is processing data more slowly.
Polars	● Due to its Rust backend, this engine offers the fastest performance: quickest code execution and most rows per second. ● The program uses very little memory: It is designed to use memory efficiently. ● It is multi-threaded by design: All CPU cores are used automatically for better speed. ● Lazy evaluation is available: It helps by carrying out computations only when needed.	● Not as mature: There are fewer people in the community and not as many tutorials or tools available as there are in Pandas. ● Limited features (at the moment): It lacks some of the specialized functions found in Pandas. ● API differences: Users who know Pandas will have to learn a bit of the Polars way.
Dask	● Can handle both parallel and distributed computing and can be used on a laptop or a cluster. ● The low memory usage makes it good for working with data that is larger than the available RAM.	● The benchmark shows that it takes the longest time to execute. ● The lower throughput is caused by the extra work needed for task scheduling and lazy execution.

	● Pandas-compatible API: Helps users who already use Pandas make the switch. ● Perfect for streaming and handling big data, as it can also work with cloud storage and distributed file systems.	● More difficult to learn: People need to know about delayed execution and task graphs. ● Not as helpful for small to medium data sets, as the overhead can be greater than the advantages unless you are working with huge or complex workloads.

5.3 Personal Reflection

5.3.1 Kek Jesslyn

By studying these data processing strategies and libraries, I have learned how different ways of working and tools can affect how fast tasks are completed, how much memory is used, how efficient the CPU is and the overall throughput. It was very clear to me that using “Load Less Data” can boost Pandas’ performance, so I learned that being careful with data in the beginning is very important. I found out that Pandas and similar tools help a lot, but they may not be suitable for working with huge datasets. Looking into Polars was very interesting, I realized how much better modern libraries are at handling large datasets. In addition, Dask showed me how parallel and distributed computing can be useful for working with large amounts of data in the future. By doing this assignment, I now see more clearly the trade-offs among usability, speed and scalability which makes me more careful when choosing tools and strategies for data processing. Now, I feel more assured about picking the right solution for different project requirements.

5.2.2 Tan Jun Yuan

This assignment gave us a solid hands-on experience in managing large datasets efficiently using Python. By working with a 2.86GB file containing about 3 million rows, we faced real-world challenges like memory limits and slow processing. We explored techniques such as chunking, sampling, data type optimization, and parallel processing with Dask, and saw how even small changes like converting object types to categories or loading only specific columns can greatly reduce memory use and improve speed. Using Google Colab also highlighted the importance of building scalable solutions within limited resources. Overall, this project improved our problem-solving skills and deepened our understanding of big data handling in real-world scenarios.

6.0 Reference

- McKinney, W. (2022). *Python for data analysis: Data wrangling with pandas, NumPy, and Jupyter* (3rd ed.). O'Reilly Media. https://wesmckinney.com/book/

- Dask Development Team. (2024). *Dask documentation*. Dask. https://docs.dask.org/en/stable/

- MohamedBakhet. (2022). *Amazon books reviews* [Data set]. Kaggle. https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews