FACULTY OF COMPUTING

SECP3133-02 HIGH PERFORMANCE DATA PROCESSING

ASSIGNMENT 2 - REPORT

TITLE: Mastering Big Data Handling

PREPARED BY: GROUP DEUX

NAME	MATRIC NO.
JASLENE YU	A22EC0171
NICOLE LIM TZE YEE	A22EC0123

PREPARED FOR: DR. ARYATI BINTI BAKRI

DATE: 04/06/2025

Table of Contents

1.0 Introduction

The rapid expansion of data generation across various domains has presented substantial challenges in the field of data science, particularly in the efficient handling, processing, and analysis of large-scale datasets. Traditional data processing tools, such as standard in-memory operations using Pandas, often become inadequate when working with datasets exceeding several hundred megabytes, resulting in performance bottlenecks and resource constraints.

To address these issues, this assignment explores big data handling techniques using Python, with a focus on three modern libraries: Pandas, Polars, and Dask. A dataset larger than 700MB was selected to replicate real-world big data scenarios and evaluate the effectiveness of several optimization strategies, including selective column loading, chunk-wise reading, data type optimization, sampling, and parallel processing.

The primary objectives of this assignment are to:

- Identify the limitations and challenges of traditional big data processing techniques.
- Apply practical strategies for managing and analyzing large datasets efficiently.
- Compare the performance of conventional methods against optimized approaches in terms of memory usage, execution time, and ease of implementation.

Through this process, the assignment aims to highlight the advantages and limitations of each technique and provide a clear understanding of how scalable tools and methods can improve the performance of data-intensive applications.

2.0 Tools and Framework Used

In this assignment, various tools, libraries, and platforms were utilized to effectively handle and analyze a large dataset exceeding 700MB. Table 2.0.1 summarizes the key technologies and their respective roles.

Table 2.0.1 : Tools and Frameworks used

Category	Software/Application/Library/Website
Documentation	Google Doc
Progression Monitoring	Github
Data Source	Kaggle
Initial Dataset Form	Excel file (.csv)
IDE	Google Colab
Coding Language	Python
Data Visualization	Matplotlib Library (Python)
Basic Data Loading	Pandas Library (Python)
Optimization Libraries	Polars, Dask
Optimization Strategies	Columns selection, Chunking, Optimize Data types, Sampling

3.0 Data Set

In this section, we provide an overview of the dataset used for analyzing and detecting fraudulent financial transactions. The dataset is designed to replicate realistic transactional behaviors and patterns across various domains and is suitable for machine learning applications focused on fraud detection.

3.1 Data Set Background

The data set is collected from **Kaggle** with a size of approximately **2.73 GB**. This dataset is a synthetically generated **collection of financial transactions**, designed to simulate real-world purchasing behaviors across various domains such as retail, travel, dining, entertainment, healthcare, and education. It offers a privacy-preserving yet realistic foundation for developing and testing fraud detection models. The data set captures essential aspects of transactional activity, ranging from customer behavior and device usage to geographic details and risk indicators, such as card presence, merchant risk, and transaction velocity. Its scale and richness make it an ideal foundation for advanced analytics in the areas of finance and e-commerce fraud prevention.

3.2 Data Set Description

The dataset contains **7,483,766 rows** and **24 columns**.

Table 3.2.1 : Columns Description

No	Columns Name	Descriptions
1	transaction_id	Unique identifier for each transaction
2	customer_id	Unique identifier for each customer in the dataset
3	card_number	Masked card number associated with the transaction
4	timestamp	Date and time of the transaction
5	merchant_category	General category of the merchant
6	merchant_type	Specific type within the merchant category
7	merchant	Name of the merchant where the transaction took place
8	amount	Transaction amount (currency based on the country)
9	currency	Currency used for the transaction
10	country	Country where the transaction occurred
11	city	City where the transaction took place

12	city_size	Size of the city
13	card_type	Type of card used
14	card_present	Indicates if the card was physically present during the transaction
15	device	Device used for the transaction
16	channel	Type of channel used for the transaction
17	device_fingerprint	Unique fingerprint for the device used in the transaction
18	ip_address	IP address associated with the transaction
19	distance_from_home	Binary indicator showing if the transaction occurred outside the customer's home country
20	high_risk_merchant	Indicates if the merchant category is known for higher fraud risk
21	transaction_hour	Hour of the day when the transaction was made
22	weekend_transaction	Boolean indicating if the transaction took place on a weekend
23	velocity_last_hour	Dictionary containing metrics on the transaction velocity, including num_transactions, total_amount, unique_merchants, unique_countries, max_single_amount
24	is_fraud	Binary indicator showing if the transaction is fraudulent

3.3 Data Loading and Inspecting

The dataset is imported from Kaggle using the Kaggle API and loaded into a Pandas DataFrame for initial exploration.

The use of the pandas library, which is optimized for performance, allowed the dataset to be loaded smoothly into memory. Additionally, using Google Colab provides access to cloud-based computational resources, including extended RAM when required, which ensures the environment can handle large datasets without crashing. This setup is both practical and efficient for conducting exploratory data analysis and model development.

Code Overview

To access the dataset, the Kaggle API key (kaggle.json) is first uploaded and configured. Figure 3.3.1 shows the dataset titled "transactions" is then downloaded and extracted for use.

```
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json
!kaggle datasets download -d ismetsemedov/transactions
```

Figure 3.3.1 : Access dataset in Kaggle

As the data set in Kaggle is stored in a ZIP file, the data set is extracted from the ZIP file after downloading, as shown in Figure 3.3.2.

```
[ ]  import zipfile

     with zipfile.ZipFile("transactions.zip", "r") as zip_ref:
         zip_ref.extractall("transactions")
```

Figure 3.3.2 : Extract zip file

By using the Pandas library, the CSV file is loaded successfully for further inspection. Figure 3.3.3 shows the data is loaded into a Pandas DataFrame.

```
[1]  import pandas as pd

     # Adjust filename based on the dataset contents
     df = pd.read_csv("transactions/synthetic_fraud_data.csv")
     df.head()
```

Figure 3.3.3 : Loading Data into data frame

At this stage, the data has been successfully loaded and previewed. The first five rows of the data set are shown in Figure 3.3.4.

	transaction_id	customer_id	card_number	timestamp	merchant_category	merchant_type	merchant	amount	currency	country
0	TX_a0ad2a2a	CUST_72886	6646734767813109	2024-09-30 00:00:01.034820+00:00	Restaurant	fast_food	Taco Bell	294.87	GBP	UK
1	TX_3599c101	CUST_70474	376800864692727	2024-09-30 00:00:01.764464+00:00	Entertainment	gaming	Steam	3368.97	BRL	Brazil
2	TX_a9461c6d	CUST_10715	5251909460951913	2024-09-30 00:00:02.273762+00:00	Grocery	physical	Whole Foods	102582.38	JPY	Japan
3	TX_7be21fc4	CUST_16193	376079286931183	2024-09-30 00:00:02.297466+00:00	Gas	major	Exxon	630.60	AUD	Australia
4	TX_150f490b	CUST_87572	6172948052178810	2024-09-30 00:00:02.544063+00:00	Healthcare	medical	Medical Center	724949.27	NGN	Nigeria

Figure 3.3.4 : Data Set Preview

The data set shape is configured and shown in Figure 3.3.5 by using df.shape() syntax. From the figure, the output shows (7483766, 24) where 7,483,766 represents the number of transaction records (rows) and 24 represents the number of features or attributes (columns) associated with each transaction. This confirms the large-scale nature of the dataset, making it rich enough for exploratory and performance comparison in this assignment.

```
df.shape

(7483766, 24)
```

Figure 3.3.5 : Data Shape

The syntax df.info() provides a quick overview of the dataset's structure, including the number of entries, column names, and data types. The output shown in Figure 3.3.6 reveals that the dataset contains **7,483,766 entries** across **24 columns**. Most of the features are of type **object** or **bool**, with a few **int64** and **float64** types as well.

```
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7483766 entries, 0 to 7483765
Data columns (total 24 columns):
 #   Column               Dtype
---  ------               -----
 0   transaction_id       object
 1   customer_id          object
 2   card_number          int64
 3   timestamp            object
 4   merchant_category    object
 5   merchant_type        object
 6   merchant             object
 7   amount               float64
 8   currency             object
 9   country              object
 10  city                 object
 11  city_size            object
 12  card_type            object
 13  card_present         bool
 14  device               object
 15  channel              object
 16  device_fingerprint   object
 17  ip_address           object
 18  distance_from_home   int64
 19  high_risk_merchant   bool
 20  transaction_hour     int64
 21  weekend_transaction  bool
 22  velocity_last_hour   object
 23  is_fraud             bool
dtypes: bool(4), float64(1), int64(3), object(16)
memory usage: 1.1+ GB
```

Figure 3.3.6 : Data Set Inspection

Figure 3.3.7 shows that df contains no duplicated rows, as indicated by df.duplicated().sum() returning np.int64(0).

```
df.duplicated().sum()
```

```
np.int64(0)
```

Figure 3.3.7 : Duplicated Values

The dataset is also confirmed to have no null values, as shown in Figure 3.3.8, where df.isnull().any() returned "False" for all rows.

```
[17] df.isnull().any()
```

	0
transaction_id	False
customer_id	False
card_number	False
timestamp	False
merchant_category	False
merchant_type	False
merchant	False
amount	False
currency	False
country	False
city	False
city_size	False
card_type	False
card_present	False
device	False
channel	False
device_fingerprint	False
ip_address	False
distance_from_home	False
high_risk_merchant	False
transaction_hour	False
weekend_transaction	False
velocity_last_hour	False
is_fraud	False

dtype: bool

Figure 3.3.8 : Null Values

An inspection of the target column, 'is_fraud', using df['is_fraud'].unique(), confirmed it holds only True and False values. Figure 3.3.9 highlights the ratio between the two values, with ‘False’ values making up roughly 80% of the dataset and ‘True’ values about 20%.

```
df['is_fraud'].unique()
```

```
array([False,  True])
```

```
fraud_ratio = df['is_fraud'].value_counts(normalize=True)
fraud_ratio
```

	proportion
is_fraud	
False	0.800272
True	0.199728

dtype: float64

Figure 3.3.9 : Target Column Inspection

4.0 Big Data Handling Strategies

To optimize the loading of the dataset, aiming for reduced memory consumption and execution time, several big data handling strategies were applied. These strategies included careful column selection, processing data through chunking, optimizing data types for memory efficiency, employing sampling techniques, and leveraging parallel processing capabilities with Dask and Polars for enhanced performance. For each of these applied strategies, the time taken and memory usage were meticulously recorded to assess their impact.

As shown in Figure 4.0.1, loading the dataset directly into a Pandas DataFrame without applying any optimization strategies resulted in a loading time of 82.33 seconds and consumed 9062.15MB of memory.

No applied strategies

```
start_time = time.time()

# Load the CSV file into a pandas DataFrame
df = pd.read_csv(file_path)

end_time = time.time()

# Calculate time taken
time_taken = end_time - start_time

# Calculate memory usage in MB
memory_used = df.memory_usage(deep=True).sum() / (1024 * 1024)

print("No Applied Strategies")
print("Time:", end_time - start_time, "seconds")
print("Memory:", memory_usage(df))
```

```
No Applied Strategies
Time: 82.33181047439575 seconds
Memory: 9062.15 MB
```

Figure 4.0.1: Data Set Loading (No Applied Strategies)

4.1 Columns Selection

Column selection is a strategy employed to optimize data loading and processing by only retrieving and retaining the columns essential for analysis, thereby reducing the overall data volume. By reducing the DataFrame from its original 24 columns to a more focused selection of 7 (specifically 'transaction_id', 'customer_id', 'timestamp', 'amount', 'merchant_category', 'high_risk_merchant', and 'is_fraud'), significant improvements in loading performance were observed. As detailed in Figure 4.1.1, this strategy resulted in a loading time of 48.43 seconds and a memory usage of 2133.99MB, representing a substantial reduction compared to the baseline with no strategies applied.

Columns Selection

```
use_cols = [
    'transaction_id', 'customer_id', 'timestamp',
    'amount', 'merchant_category', 'high_risk_merchant', 'is_fraud'
]

start_time = time.time()

# Load only specific columns from the CSV
df_less_data = pd.read_csv(file_path, usecols=use_cols)


end_time = time.time()

print("Load Less Data")
print("Time:", end_time - start_time, "seconds")
print("Memory:", memory_usage(df_less_data))
df_less_data.head()
```

```
Load Less Data
Time: 48.43134427070618 seconds
Memory: 2133.99 MB
```

Figure 4.1.1: Columns Selection

4.2 Chunking

Chunking is a strategy that involves processing a dataset in smaller, manageable segments instead of loading it entirely into memory. By specifying a chunk_size (e.g., 50,000 or 100,000 rows), data is processed incrementally. This means each chunk is handled sequentially, and the preceding chunk is released from memory. This method significantly reduces both memory consumption and execution time. Therefore, the time taken and memory usage were calculated only for the first chunk to demonstrate the immediate impact of this memory management. As Figure 4.2.1 shows, employing chunking resulted in a dramatically reduced loading time of 0.59 seconds and a memory usage of 60.63MB.

Chunking

```
chunk_size = 50000  # Number of rows per chunk

start_time = time.time()

chunks = []

for chunk in pd.read_csv(file_path, chunksize=chunk_size):
    chunks.append(chunk)
    break  # Just read the first chunk

end_time = time.time()

print("Chunking (First Chunk Only)")
print("Time:", end_time - start_time, "seconds")
print("Memory of first chunk:", memory_usage(chunks[0]))
print("First chunk size:", len(chunks[0]))
```

```
Chunking (First Chunk Only)
Time: 0.5931453704833984 seconds
Memory of first chunk: 60.63 MB
First chunk size: 50000
```

Figure 4.2.1: Chunking

4.3 Optimize Data Types

Optimizing data types allows for lesser memory consumption and potentially faster processing by ensuring that each column uses the most memory-efficient data type suitable for its content. This often involves "downcasting" numeric types (e.g., from int64 to int32 or float64 to float32) and using categorical types for columns with a limited number of unique values. As shown in Figure 4.3.1, specific columns were selected for downcasting: 'amount' to 'float32', 'distance_from_home' to 'int32', 'transaction_hour' to 'int8', and 'is_fraud' to 'category'.

Optimize Data Types

```
start_time = time.time()

df_opt_pandas = pd.read_csv(
    file_path,
    dtype={
        'amount': 'float32',
        'distance_from_home': 'int32',
        'transaction_hour': 'int8',
        'is_fraud': 'category'
    }
)

end_time = time.time()

print("Pandas - Data Type Optimization")
print("Time:", end_time - start_time, "seconds")
print("Memory:", df_opt_pandas.memory_usage(deep=True).sum() / (1024 ** 2), "MB")
```

```
Pandas - Data Type Optimization
Time: 84.25377464294434 seconds
Memory: 8955.09714794159 MB
```

Figure 4.3.1: Chunking

Interestingly, the results for this data type optimization strategy alone showed a loading time of 84.25 seconds and memory usage of 8955.10MB. This is slightly longer than the baseline loading time of 82.33 seconds without any strategies applied, though it does result in a marginal memory reduction. This slight increase in loading time, despite the memory reduction, can be attributed to the overhead involved when Pandas' read_csv function actively performs type conversions during the loading process. When data type is specified directly, Pandas must parse the string value for each cell and then explicitly convert it to the designated, smaller data type. This conversion step can introduce a minor computational cost, particularly for very large datasets where the primary benefits of memory reduction often become apparent in subsequent data operations rather than during the initial load itself. For example, converting strings to specific numeric types or to a category can be more complex than simply allowing Pandas to infer a default int64 or float64.

4.4 Sampling

Sampling is a strategy used to reduce the dataset size by selecting a representative subset of the data. In our case, stratified sampling was deemed more appropriate given the imbalanced nature of our target variable, 'is_fraud' (approximately 80% False and 20% True). Stratified sampling ensures the proportion of these classes is maintained in the sample, thereby preserving the original distribution. As shown by the code snippet in Figure 4.4.1, 10% of the total rows were selected using sklearn.model_selection.train_test_split with the stratify parameter set to df_sample['is_fraud'].

Stratified Sampling

```
df_sample = pd.read_csv(file_path)

from sklearn.model_selection import train_test_split

df_sample['is_fraud'] = df_sample['is_fraud'].astype('category')

start_time = time.time()

# Split the data while preserving the fraud distribution
_, stratified_sample = train_test_split(
    df_sample,
    test_size=0.1,
    stratify=df_sample['is_fraud'],
    random_state=42
)


end_time = time.time()

print("Stratified Sampling")
print("Time:", end_time - start_time, "seconds")
print("Sample Size:", len(stratified_sample))
print("Fraud Rate:\n", stratified_sample['is_fraud'].value_counts(normalize=True))
print("Memory:", memory_usage(stratified_sample))
```

Figure 4.4.1: Stratified Sampling

This process, including the initial loading of the full dataset to perform the sampling, took 37.04 seconds. The results, presented in Figure 4.4.2, show the resulting stratified_sample contained 748,377 rows, with the 'is_fraud' distribution accurately maintained at approximately 80% False and 20% True. The memory usage for this sampled DataFrame was 911.93MB.

```
Stratified Sampling
Time: 37.037365436553955 seconds
Sample Size: 748377
Fraud Rate:
 is_fraud
False    0.800272
True     0.199728
Name: proportion, dtype: float64
Memory: 911.93 MB
```

Figure 4.4.2: Output for Stratified Sampling

4.5 Parallel Processing with Dask

Dask offers a powerful framework for parallel and out-of-core computation, allowing for the handling of datasets that are larger than available RAM by coordinating computations across multiple cores or machines. A key characteristic of Dask DataFrames is their lazy evaluation: data is not loaded into memory, nor are computations performed, until an explicit action, such as .compute(), is called to trigger the final result. This means that operations like aggregation, filtering, and cleaning can be defined and chained without consuming significant memory upfront, as Dask builds a task graph to execute efficiently when computation is required.

As demonstrated by the code in Figure 4.5.1, simply creating a Dask DataFrame by reading the CSV file (using dd.read_csv(file_path)) was remarkably fast, taking only 0.07 seconds. This incredibly short time reflects Dask's lazy loading nature; at this stage, Dask merely constructs the plan for how to read the data, rather than loading the entire dataset into memory.

Dask Parallel Processing

```
[8] import dask.dataframe as dd
    import time
    file_path = "transactions/synthetic_fraud_data.csv"
```

```
[9] # Load with Dask
    start_time_dask = time.time()
    df_dask = dd.read_csv(file_path)
    #df_dask_computed = df_dask.compute() # Trigger computation to bring data into memory
    end_time_dask = time.time()

    print("Load with Dask:")
    print("Time:", end_time_dask - start_time_dask, "seconds")
    #print("Memory:", memory_usage(df_dask))
```

```
Load with Dask:
Time: 0.0668489933013916 seconds
```

Figure 4.5.1: Dask Parallel Processing

Consequently, direct memory measurement of df_dask at this point would not reflect the full dataset's size. Even when calling df_dask.head(), as illustrated conceptually by Figure 4.6.2, Dask efficiently computes only the necessary small portion of the data to display the head, without loading the entire DataFrame into memory. This highlights Dask's capability to manage very large datasets by executing only what is strictly necessary.

```
df_dask.head()
```

	transaction_id	customer_id	card_number	timestamp	merchant_category	merchant_type	merchant	amount	currency	country	...
0	TX_a0ad2a2a	CUST_72886	6646734767813109	2024-09-30 00:00:01.034820+00:00	Restaurant	fast_food	Taco Bell	294.87	GBP	UK	...
1	TX_3599c101	CUST_70474	376800864692727	2024-09-30 00:00:01.764464+00:00	Entertainment	gaming	Steam	3368.97	BRL	Brazil	...
2	TX_a9461c6d	CUST_10715	5251909460951913	2024-09-30 00:00:02.273762+00:00	Grocery	physical	Whole Foods	102582.38	JPY	Japan	...
3	TX_7be21fc4	CUST_16193	376079286931183	2024-09-30 00:00:02.297466+00:00	Gas	major	Exxon	630.60	AUD	Australia	...
4	TX_150f490b	CUST_87572	6172948052178810	2024-09-30 00:00:02.544063+00:00	Healthcare	medical	Medical Center	724949.27	NGN	Nigeria	...

5 rows × 24 columns

Figure 4.5.2: Output for df_dask.head()

4.6 Polars Optimization

Polars offers a highly efficient and fast DataFrame library written in Rust, designed for high-performance data manipulation and analysis, making it an excellent choice for handling large datasets. Its column-oriented architecture and lazy evaluation capabilities contribute to its speed and memory efficiency. As demonstrated by the code in Figure 4.6.1, directly loading the CSV file into a Polars DataFrame yielded significantly improved performance.

```
pip install polars
```

```
Requirement already satisfied: polars in /usr/local/lib/python3.11/dist-packages (1.21.0)
```

```
[6] import polars as pl
    import time

    # Replace with your actual file path if using Google Drive
    file_path = "transactions/synthetic_fraud_data.csv"
```

```
[9] start_time = time.time()

    # Load the CSV file into a Polars DataFrame
    df_polars = pl.read_csv(file_path)

    end_time = time.time()

    print("Polars")
    print("Time:", end_time - start_time, "seconds")
    # Polars has a convenient estimated_size method
    print("Memory:", df_polars.estimated_size('mb'), "MB")

    print(df_polars.head())
```

Figure 4.6.1: Polars Optimization

The results, shown in Figure 4.6.2, indicate that loading the entire dataset with Polars took only 11.98 seconds and consumed 2528.16MB of memory, showcasing its substantial optimization capabilities compared to the standard Pandas approach.

```
Polars
Time: 11.984588861465454 seconds
Memory: 2528.1607761383057 MB
shape: (5, 24)
```

transaction_id --- str	customer_id --- str	card_number --- i64	timestamp --- str	…	transaction_hour --- i64	weekend_transaction --- bool	velocity_last_hour --- str	is_fraud --- bool
TX_a0ad2a2a	CUST_72886	6646734767813109	2024-09-30 00:00:01.034820+00:…	…	0	false	{'num_transactions': 1197, 'to…	false
TX_3599c101	CUST_70474	376800864692727	2024-09-30 00:00:01.764464+00:…	…	0	false	{'num_transactions': 509, 'tot…	true
TX_a9461c6d	CUST_10715	5251909460951913	2024-09-30 00:00:02.273762+00:…	…	0	false	{'num_transactions': 332, 'tot…	false
TX_7be21fc4	CUST_16193	376079286931183	2024-09-30 00:00:02.297466+00:…	…	0	false	{'num_transactions': 764, 'tot…	false
TX_150f490b	CUST_87572	6172948052178810	2024-09-30 00:00:02.544063+00:…	…	0	false	{'num_transactions': 218, 'tot…	true

Figure 4.6.2: Polars Optimization Output

5.0 Comparative Analysis

This section compares the performance of traditional data loading methods against various big data optimization strategies. The aim is to highlight the efficiency improvements in terms of execution time and memory usage when processing a dataset larger than 700MB.

Data loading was tested using:

- Traditional Pandas loading
- Optimized techniques (Column Selection, Chunking, Data Type Optimization, Stratified Sampling)
- Polars library
- Dask library

5.1 Traditional vs Optimized Methods

In this section, the efficiency and ease of processing between traditional data handling methods and optimized strategies are compared when working with a large dataset. The goal is to evaluate how each method performs in terms of **execution time, memory usage**, and **ease of processing**.

Table 5.1.1 : Result Comparison

Method	Execution Time (s)	Memory Usage (MB)
Pandas (No Optimization)	82.33	9062.15
Pandas – Column Selection	48.43	2133.99
Pandas – Chunking	0.59	60.63
Pandas – Data Type Optimization	84.25	8955.10
Pandas – Stratified Sampling	37.04	911.93
Dask	0.07	0
Polars	11.98	2528.16

From Table 5.1.1, we can observe significant differences across the methods used. The traditional Pandas (without optimization) approach took the longest time to load (82.33 seconds) and consumed the highest amount of memory (9062.15 MB), making it inefficient for big data handling. In contrast, optimized methods significantly improved performance.

Column Selection reduced memory usage to 2133.99 MB and lowered execution time to 48.43 seconds by only loading relevant columns. Chunking was the most efficient strategy within Pandas, requiring only 0.59 seconds and using just 60.63 MB of memory. It is especially useful for sequential processing of large files. Data Type Optimization helped reduce memory to 8955.10 MB but had minimal impact on execution time (84.25 seconds), showing that while memory footprint improved, type conversions added overhead. Stratified Sampling allowed for faster prototyping by reducing the data volume, achieving 37.04 seconds and 911.93 MB in memory. Apart from that, the modern alternatives like Dask demonstrated the best overall performance, completing the load in 0.07 seconds with negligible memory usage, making it ideal for parallel and distributed data processing. At the same time, Polars offered a good balance of performance, completing in 11.98 seconds with moderate memory usage (2528.16 MB), outperforming Pandas in both aspects.

In terms of ease of processing, Pandas is the most simple and familiar for beginners but less efficient without optimization. Chunking and sampling are easy to apply and offer quick benefits for performance. Dask offers excellent scalability and speed, but it requires a deeper understanding of parallel computing concepts. Polars is combining ease of use with better performance on large datasets compared to traditional Pandas.

The effectiveness of the libraries and methods can be arranged based on their performance and scalability as follows. Dask demonstrated the highest efficiency for both execution time and memory usage, making it the most suitable for large-scale, distributed processing. Pandas with chunking provided an excellent lightweight alternative within the Pandas ecosystem, ideal for systems with limited resources. Polars offered a strong balance between speed and usability, making it a robust option for large datasets. Meanwhile, stratified sampling and column selection enhanced performance for specific use cases where reduced data volume is acceptable. Data type optimization, while beneficial for memory, showed limited impact on processing time. Lastly, the traditional Pandas method, though straightforward and beginner-friendly, proved to be the least efficient for big data handling.

5.2 Visualization

Visualization has been done to generate graphs showing the comparison of various data processing methods in terms of execution time and memory usage.

Figure 5.2.1 shows the visualization of the comparison of execution time using different data processing methods. It highlights how quickly each method completes the task.

Figure 5.2.1 : Graph of Comparison Different Strategies (Execution Time)

Figure 5.2.2 shows the visualization of the comparison of memory usage using different data processing methods. It provides insight into memory efficiency for different methods.

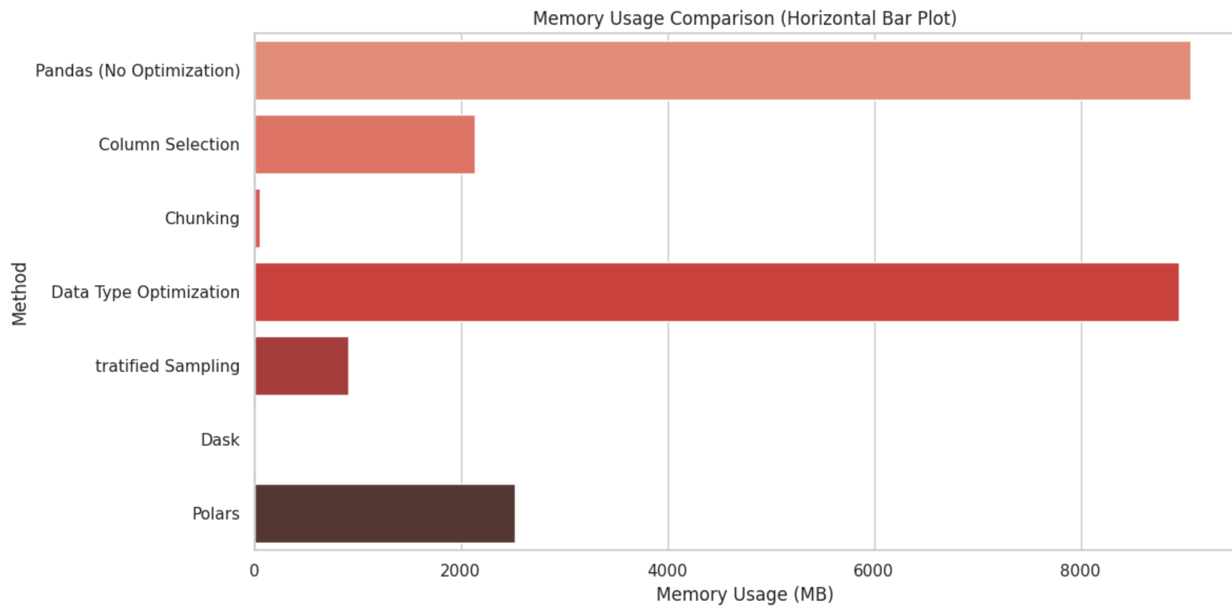

Figure 5.2.2 : Graph of Comparison Different Strategies (Memory Usage)

6.0 Conclusion and Reflection

Based on the comparative analysis, we proved that handling large data sets efficiently requires more than just a standard library like Pandas. Traditional methods lead to slow execution time and high memory usage, as seen in the result from Pandas load. In contrast, the execution time and memory usage drastically improve performance by applying appropriate optimized strategies and modern libraries like Dask and Polars.

Although the optimization strategies and libraries significantly enhanced data processing performance, each method comes with its own set of benefits and limitations. The traditional Pandas approach is straightforward and widely used, making it ideal for small to medium-sized datasets. However, it has low scalability and becomes slow when handling large volumes of data, as we can observe from the result. Column selection and data type optimization help reduce memory usage but may require manual effort from us to identify and convert appropriate columns. The increase in execution time for data type conversion is likely due to the overhead of explicitly parsing and casting each value to the specified, smaller data types. While Pandas' default read_csv can quickly infer common types, forcing specific conversions (eg: from inferred int64 to int32) adds a computational step for each cell, slowing down the initial load despite potential memory savings. Furthermore, chunking stands out for its ability to process large files in manageable portions, which is especially useful when system memory is limited. However, it restricts the ability to perform operations that require access to the entire dataset at once. Stratified sampling supports faster experimentation and prototyping by reducing the dataset size while maintaining representative distributions. Yet, it may result in biased insights due to an unbalanced representation of the original data. On the other hand, Dask offers powerful parallel processing capabilities, enabling rapid computation on large datasets. At the same time, it requires additional understanding of distributed computing concepts. Thus, beginners in Dask may need to invest time in learning its execution model and debugging parallel tasks. Polars provides high performance, making it a promising tool for big data tasks. As a relatively newer library, it may lack certain advanced features and a large support community compared to more established libraries like Pandas. In conclusion, the choice of method depends on the specific data requirements, computational resources, and the user's familiarity with each method. While each approach has its strengths, their limitations must be carefully considered to ensure efficient and effective data processing.

This assignment offered practical exposure to the challenges of handling large-scale datasets and the importance of big data processing. Effective data processing is not just about speed, it requires a balance between memory efficiency and ease of use. Through the implementation of both traditional and modern techniques, valuable insights were gained into memory optimization strategies such as column selection and data type casting, as well as scalable methods like chunking and stratified sampling. Modern libraries including Dask and Polars, demonstrated significant advantages in processing performance and efficiency. Overall, the assignment deepened understanding of data handling strategies, which are essential for managing big data in contemporary data science practices.

7.0 Appendix

1. Dataset: https://www.kaggle.com/datasets/ismetsemedov/transactions

2. GitHub Repository link: https://github.com/Jingyong14/HPDP02/tree/main/2425/assignment/asgn2/submission/Group_Deux

3. Google Colab files:
   - Data Inspection: Data_Inspection.ipynb
   - Pandas: Pandas.ipynb
   - Polars: Polars.ipynb
   - Dask: Dask.ipynb
   - Visualization: HPDP_Assignment2_Visualization.ipynb