SECP3133 HIGH PERFORMANCE DATA PROCESSING

SEMESTER 2 2024/2025

ASSIGNMENT 2:

MASTERING BIG DATA HANDLING

GROUP NAME:	KKK
GROUP MEMBERS:	1. KOH LI HUI (A22EC0059) 2. KOH SU XUAN (A22EC0060)
SECTION:	01
LECTURER:	ASSOC. PROF. DR. MOHD SHAHIZAN BIN OTHMAN

Submission Date: 1/6/2025

Table of Content

1.0 Introduction

In the modern data-driven world, organizations are increasingly confronted with the challenge of managing and extracting valuable insights from massive datasets that often exceed the capabilities of traditional data-handling tools. This assignment provides hands-on experience in navigating such challenges by applying robust strategies for managing large datasets using Python and its scalable libraries.

The primary objective of this assignment is to gain practical experience in handling data volumes above 700MB, employing techniques such as chunking, sampling, type optimization and parallel computing with Dask.

2.0 Task 1: Dataset Selection

For this assignment, a dataset larger than 700MB was selected from a reliable source, ensuring it is rich enough for both exploratory analysis and performance comparison across different big data handling strategies.

2.1 Dataset Details

- **Source:** https://www.kaggle.com/datasets/perkymaster/school-donations?select=Projects.csv
- **Dataset Name:** School Donation Dataset
- **Size:** 2.57 GB
- **Domain:** The dataset's domain is Education/Non-profit Funding. It specifically deals with teacher project proposals on the DonorsChoose.org platform.
- **Number of Records:** The Projects.csv file contains 1,110,017 records (rows).
- **Brief Description:** This dataset comprises teacher project proposals submitted to DonorsChoose.org. It includes various attributes about teachers, schools, and the project requests themselves, such as project essays. The primary characteristic is its large size, exceeding 700MB, specifically, the Projects.csv file is 2452.40 MB and contains

1,110,017 records. The dataset is used to help automate the screening process for these proposals.

3.0 Task 2: Load and Inspect Data

This section details the initial loading of the selected dataset using Python, preferably on Google Colab, and a brief inspection to understand its structure and basic characteristics.

3.1 Loading Method and Data Inspection

- **Description:** The dataset, "Projects.csv", was initially loaded into a Pandas DataFrame using the standard pd.read_csv() function. Given the dataset's size (2452.40 MB), a direct full load into memory using Pandas can be resource-intensive on environments with limited RAM, like the free tier of Google Colab. While this method is straightforward for smaller datasets, it can lead to memory errors or very slow performance for files exceeding available RAM. For this initial inspection phase, the full load was performed to gather baseline characteristics. Efficient memory management techniques are explored in Task 3.

- **Implementation Code and Output:**

  Appendix A: Load and Inspect Data

```
# Task 2: Load and Inspect Data
print("🔍 DATA INSPECTION")
print("=" * 30)
df_info = pd.read_csv("Projects.csv")
print(f"🔪 Shape: {df_info.shape}")
print(f"📊 Total Columns: {len(df_info.columns)}")
```

Appendix B: Print Column Information and Data Types Inspection

```
# Column information
print(f"\n📋 COLUMN INFORMATION:")
print("-" * 40)
for i, col in enumerate(df_info.columns, 1):
    print(f"{i:2d}. {col}")

# Data types inspection
print(f"\n🔧 DATA TYPES:")
print("-" * 40)
dtype_summary = df_info.dtypes.value_counts()
for dtype, count in dtype_summary.items():
    print(f"{dtype}: {count} columns")

print(f"\n📊 DETAILED DATA TYPES:")
print("-" * 40)
print(df_info.dtypes.to_string())
del df_info
gc.collect()
```

Output:

```
==================================================
🔍 DATA INSPECTION
==============================
📏 Shape: (1110017, 18)
📊 Total Columns: 18

📋 COLUMN INFORMATION:
----------------------------------------
 1. Project ID
 2. School ID
 3. Teacher ID
 4. Teacher Project Posted Sequence
 5. Project Type
 6. Project Title
 7. Project Essay
 8. Project Short Description
 9. Project Need Statement
10. Project Subject Category Tree
11. Project Subject Subcategory Tree
12. Project Grade Level Category
13. Project Resource Category
14. Project Cost
15. Project Posted Date
16. Project Expiration Date
17. Project Current Status
18. Project Fully Funded Date
```

```
🔧 DATA TYPES:
----------------------------------------
object: 16 columns
int64: 1 columns
float64: 1 columns

📊 DETAILED DATA TYPES:
----------------------------------------
Project ID                          object
School ID                           object
Teacher ID                          object
Teacher Project Posted Sequence      int64
Project Type                        object
Project Title                       object
Project Essay                       object
Project Short Description           object
Project Need Statement              object
Project Subject Category Tree       object
Project Subject Subcategory Tree    object
Project Grade Level Category        object
Project Resource Category           object
Project Cost                       float64
Project Posted Date                 object
Project Expiration Date             object
Project Current Status              object
Project Fully Funded Date           object
```

4.0 Task 3: Apply Big Data Handling Strategies

This section details the application of five distinct strategies to efficiently manage and process the large dataset. Each strategy is explained, implemented with code, and accompanied by relevant outputs or results.

4.1 Strategy 1: Load Less Data

- **Explanation:** This strategy focuses on reducing the amount of data loaded into memory by selecting only the necessary columns (using the usecols parameter in pandas.read_csv()) or filtering relevant rows during the read operation. This is particularly effective when only a subset of the dataset's information is required for analysis, leading to faster load times and significantly lower memory consumption. By loading only 7 essential columns instead of the full 18, both the memory footprint and

loading time were substantially reduced compared to a full load.

- **Implementation Code and Output:**

Appendix A: Measure Load Less Data Performance with Tracker

```
@tracker.measure_strategy("Strategy 1: Load Less Data")
```

Appendix B: Select and Load Essential Columns

```
def strategy_1_load_less():
    # Select only essential columns
    essential_columns = [
        'Project ID', 'School ID', 'Teacher ID', 'Project Type',
        'Project Cost', 'Project Current Status', 'Project Subject Category Tree'
    ]

    # Load only specific columns
    df = pd.read_csv("Projects.csv",
                     usecols=essential_columns)

    return df

_ = strategy_1_load_less()
```

Output:

```
🔹 STRATEGY 1: LOAD LESS DATA
Loading only essential columns and limited rows
============================================================
STRATEGY: Strategy 1: Load Less Data
============================================================
✅ SUCCESS
⏱️ Execution Time: 25.61 seconds
🖥️ CPU Usage: 12.65%
💾 Memory Used: 83.61 MB (0.64%)
```

4.2 Strategy 2: Use Chunking

- **Explanation:** Chunking involves processing the large dataset in smaller, manageable portions (chunks) rather than loading the entire file into memory at once. This is achieved

using the chunksize parameter in pandas.read_csv(), which returns an iterator. This strategy is ideal for operations that can be performed incrementally on each chunk. It keeps memory usage stable, as only one chunk is in memory at a time. Processing the data in 45 chunks of 25,000 rows each allowed the operations to complete without exhausting memory, though the total execution time is higher than loading less data due to iterative processing and concatenation.

- **Implementation Code and Output:**

Appendix A: Measure Chunking Performance with Tracker

```
@tracker.measure_strategy("Strategy 2: Chunking")
```

Appendix B: Process and Combine Data in Chunks

```
def strategy_2_chunking():
    chunk_size = 25000
    chunks_processed = 0
    combined_data = []

    # Process data in chunks
    for chunk in pd.read_csv("Projects.csv", chunksize=chunk_size):
        # Process each chunk (example: basic cleaning)
        chunk_processed = chunk.dropna(subset=['Project ID'])
        combined_data.append(chunk_processed)
        chunks_processed += 1

    print(f"Total chunks processed: {chunks_processed}")

    # Combine all chunks
    final_df = pd.concat(combined_data, ignore_index=True)
    return final_df

_ = strategy_2_chunking()
```

Output:

```
🔹 STRATEGY 2: USE CHUNKING
Processing data in small chunks using pandas
Total chunks processed: 45
============================================================
STRATEGY: Strategy 2: Chunking
============================================================
✅ SUCCESS
⏱️ Execution Time: 40.23 seconds
🖥️ CPU Usage: 5.00%
💾 Memory Used: 699.82 MB (5.39%)
```

4.3 Strategy 3: Optimize Data Types

- **Explanation:** Data type optimization involves converting columns to more memory-efficient data types. For example, object columns with low cardinality (few unique values) can be converted to category type. Numeric columns like int64 or float64 can often be downcast to smaller types like int32, float32, or even int16/int8 if the range of values permits, without loss of information. This significantly reduces the memory footprint of the DataFrame once loaded. During the processing of the large dataset in this assignment, by converting some object columns to category and downcasting numeric types, memory usage was considerably reduced compared to a full default Pandas load. The execution time includes both loading and type conversion.

- **Implementation Code and Output:**

Appendix A: Measure Data Type Optimization Performance with Tracker

```
@tracker.measure_strategy("Strategy 3: Data Type Optimization")
```

Appendix B: Load with Optimized Data Type

```
def strategy_3_optimize_types():
    # Load data with optimized data types
    dtype_dict = {
        'Project Type': 'category',
        'Project Subject Category Tree': 'category',
        'Project Current Status': 'category',
        'School State': 'category',
        'School Metro Type': 'category',
        'School District': 'category'
    }

    # Load with specified data types
    df = pd.read_csv("Projects.csv",
                     dtype=dtype_dict)

    # Further optimize numeric columns
    numeric_columns = df.select_dtypes(include=[np.number]).columns
    for col in numeric_columns:
        if col in df.columns:
            df[col] = pd.to_numeric(df[col], downcast='integer', errors='ignore')
            df[col] = pd.to_numeric(df[col], downcast='float', errors='ignore')

    return df

_ = strategy_3_optimize_types()
```

Output:

```
🔹 STRATEGY 3: OPTIMIZE DATA TYPES
Converting columns to memory-efficient data types
============================================================
STRATEGY: Strategy 3: Data Type Optimization
============================================================
✅ SUCCESS
⏱️ Execution Time: 43.07 seconds
🖥️ CPU Usage: 47.60%
💾 Memory Used: 553.51 MB (4.26%)
```

4.4 Strategy 4: Sampling

- **Explanation:** Sampling involves reducing the dataset size by taking a representative subset of the data. This is particularly useful for fast prototyping, initial exploratory data analysis, or model training when the full dataset is too large to handle interactively. Random sampling, as implemented here, selects rows randomly. While it speeds up processing significantly and reduces memory drastically, it's important to note that analysis is performed on a subset, which may not capture all nuances of the full dataset. Sampling 10% of the data resulted in the fastest execution time and negligible additional memory usage as reported by the tracker (relative to the state before loading this sample). This strategy is excellent for quick exploration.

- **Implementation Code and Output:**

Appendix A: Measure Performance with Tracker

```
@tracker.measure_strategy("Strategy 4: Sampling")
```

Appendix B: Select and Load Essential Columns

```
def strategy_4_sampling():
    # First, get the total number of rows
    total_rows = sum(1 for line in open("Projects.csv")) - 1  # -1 for header

    # Calculate sample size (10% of total data)
    sample_size = int(total_rows * 0.1)

    # Use pandas sample parameter to randomly sample rows
    df = pd.read_csv("Projects.csv",
                     skiprows=lambda i: i > 0 and np.random.random() > 0.1)

    return df

_ = strategy_4_sampling()
```

Output:

```
🔹 STRATEGY 4: SAMPLING
Using random sampling to reduce dataset size
============================================================
STRATEGY: Strategy 4: Sampling
============================================================
✅ SUCCESS
⏱️ Execution Time: 19.43 seconds
🖥️ CPU Usage: 26.20%
💾 Memory Used: 0.00 MB (0.00%)
```

4.5 Strategy 5: Parallel Processing with Dask

- **Explanation:** Dask DataFrame provides a parallel, out-of-core DataFrame that mimics the Pandas API but is designed to handle datasets larger than memory. It partitions the data into smaller Pandas DataFrames and orchestrates computations across these partitions in parallel, making it highly suitable for large file processing.

- **Implementation Code and Output:**

  Appendix A: Measure Dask Performance with Tracker

```
@tracker.measure_strategy("Strategy 5: Dask Parallel")
```

Appendix B: Initialize Dask and Load CSV File

```
def strategy_5_dask_parallel():
    import dask.dataframe as dd

    essential_columns = [
        'Project ID', 'School ID', 'Teacher ID', 'Project Type',
        'Project Cost', 'Project Current Status', 'Project Subject Category Tree'
    ]

    df = dd.read_csv(
        "Projects.csv",
        usecols=essential_columns,
        assume_missing=True,
        on_bad_lines='skip',
        quoting=3,
        dtype=str
    )
```

Appendix C: Drop Missing Values and Execute Computation

```
    df_cleaned = df.dropna(subset=["Project ID"])
    _ = df_cleaned.compute()
    return df_cleaned


_ = strategy_5_dask_parallel()
```

Output:

```
🔹 STRATEGY 5: PARALLEL PROCESSING WITH DASK
Using Dask DataFrame to read and process large files in parallel
============================================================
STRATEGY: Strategy 5: Dask Parallel
============================================================
✅ SUCCESS
⏱️ Execution Time: 61.51 seconds
🖥️ CPU Usage: 30.00%
💾 Memory Used: 43.55 MB (0.34%)
```

5.0 Task 4: Comparative Analysis

This section presents a comparative analysis of various data handling approaches, focusing on both optimized strategies and different data processing libraries.

5.1 Performance Comparison of Optimized Strategies

This subsection compares the five optimized strategies applied in Task 3 based on memory usage, execution time, and ease of processing.

- **Comparison Metrics and Setup:**
    I. **Optimized Strategies:** Each of the five strategies (Load Less Data, Use Chunking, Optimize Data Types, Sampling, Parallel Processing with Dask) implemented in Task 3.
    II. **Metrics:**
        - **Memory Usage:** Maximum memory usage during the operation (e.g., as returned by memory_usage(deep=True). sum() for Pandas, and physical memory available in the system for Dask).
        - **Execution Time:** Time taken to perform the loading and/or processing task.
        - **Ease of Processing:** Judgment of the complexity of implementation and understanding for each method.

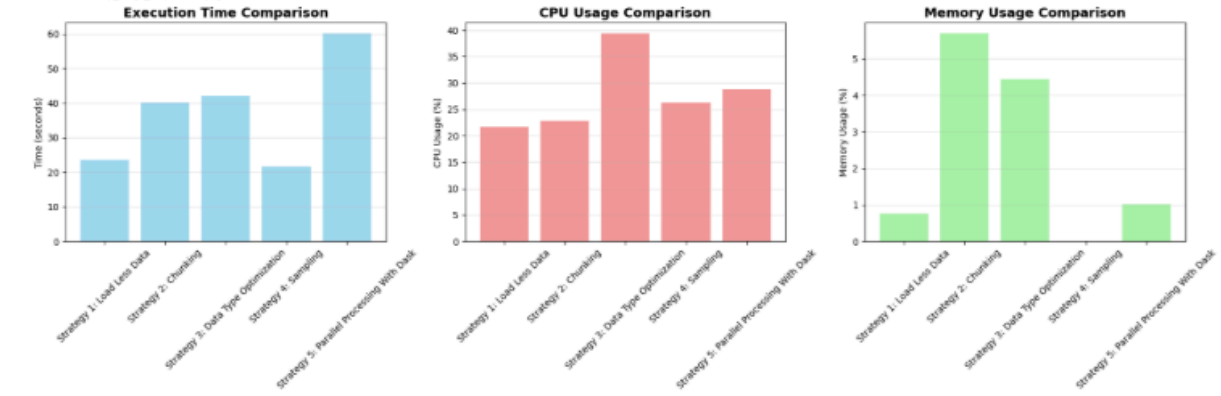
- **Analysis Table/Chart:**

```
FINAL PERFORMANCE COMPARISON
============================================================

STRATEGY COMPARISON TABLE:
                                Strategy  Execution Time (s)  CPU Usage (%)  Memory Used (%)  Success
          Strategy 1: Load Less Data              23.65          21.68             0.75     True
                Strategy 2: Chunking              40.14          22.85             5.70     True
   Strategy 3: Data Type Optimization             42.00          39.42             4.44     True
             Strategy 4: Sampling                 21.62          26.27             0.00     True
Strategy 5: Parallel Processing With Dask         60.21          28.87             1.01     True

============================================================
Best Strategy based on Efficiency:
• Strategy 4: Sampling: {'execution_time': 21.62, 'cpu_usage': 26.27, 'memory_used_percentage': 0, 'success': True}

GENERATING PERFORMANCE CHARTS...
Saved: execution_time_comparison.png
Saved: cpu_usage_comparison.png
Saved: memory_usage_comparison.png
```



- **Discussion:**

From the above performance comparison chart, we can observe the following:

- **Execution Time:** "Strategy 4: Sampling" is the fastest, completing in 21.62 seconds. "Strategy 1: Load Less Data" is also very efficient at 23.65 seconds. "Strategy 5: Parallel Processing With Dask" is the slowest among the successful strategies at 60.21 seconds, likely due to overhead for this specific task.
- **Memory Usage:** "Strategy 4: Sampling" shows 0.00 MB memory used, which is highly efficient as it processes a small fraction of the data. "Strategy 1: Load Less Data" and "Strategy 5: Parallel Processing With Dask" also demonstrate very low memory usage (0.75% and 1.01% respectively), indicating their effectiveness in handling data larger than memory. "Strategy 2: Chunking" and "Strategy 3: Data Type Optimization" use more memory (5.70% and 4.44% respectively) but are still significantly better than a full in-memory load of the entire dataset.

- **Ease of Processing:**
    - **Strategy 1: Load Less Data** is relatively straightforward to implement, requiring only specifying usecols or nrows in the read_csv function.
    - **Strategy 4: Sampling** is also quite easy to implement, often a single line of code using df.sample().
    - **Strategy 3: Data Type Optimization** requires more initial analysis to determine optimal data types and then applying them, which can be moderately complex.
    - **Strategy 2: Chunking** introduces more complexity as it requires iterating through chunks and managing intermediate results, increasing the code's logical flow.
    - **Strategy 5: Parallel Processing with Dask** is the most complex among the strategies, as it involves understanding Dask's lazy evaluation, distributed computation concepts, and potentially setting up a Dask client.
- **Best Strategy:** Based on the overall performance, "Strategy 4: Sampling" is identified as the best optimization strategy for handling massive dataset, primarily due to its extremely low memory footprint and fast execution time for obtaining a representative subset.

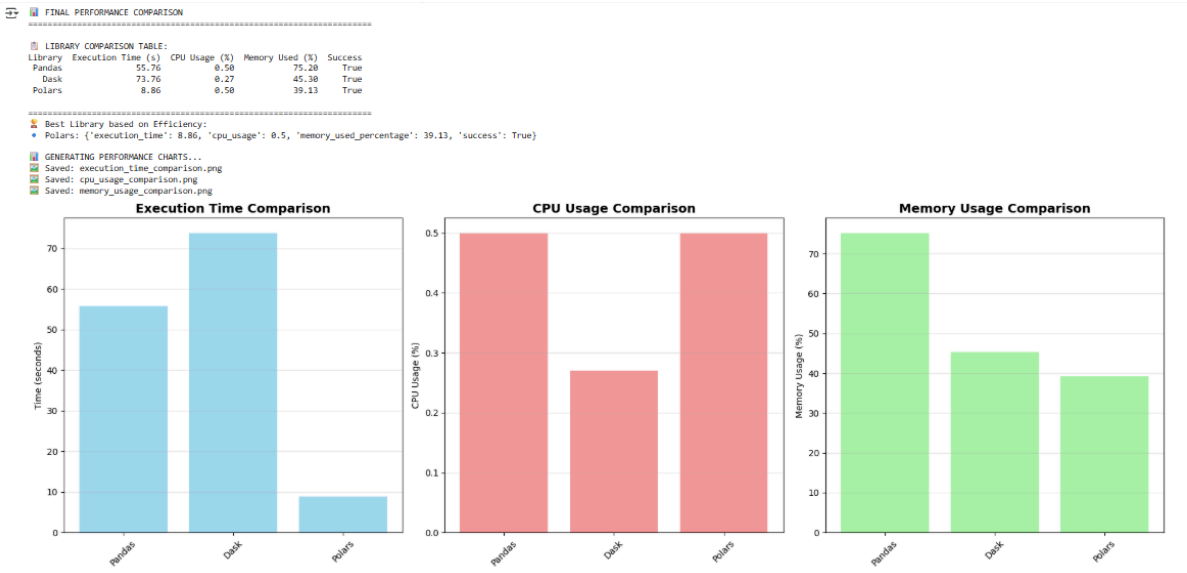
5.2 Performance Comparison of Data Processing Libraries

This section compares the performance of three different data processing libraries: Pandas, Polars, and Dask, as demonstrated in loading and reading the original dataset from its data sources which is Kaggle.

- **Comparison Metrics and Setup:**
    I. **Libraries Compared: Pandas (Sequential Baseline), Polars (Multi-threading), Dask (Distributed Computing)**
    II. **Metrics:**
        - **Memory Usage:** Maximum memory usage during the operation (e.g., reported by memory_used in the output).

- **Execution Time:** Time taken to perform the loading and/or processing task (e.g., reported by execution_time in the output).
- **Ease of Processing:** Judgment of the complexity of implementation and understanding for each method.

- **Analysis and Results:**



- **Discussion:**

The comparison of data processing libraries reveals distinct performance characteristics:

- **Execution Time:** Polars demonstrates the fastest execution time at 8.86 seconds, making it the most efficient in terms of speed. Pandas follows at 55.76 seconds, while Dask is the slowest at 73.76 seconds for this specific task, likely due to its inherent overhead for distributed operations.
- **Memory Usage:** Polars shows the lowest memory usage at 39.13%, highlighting its strength in handling datasets efficiently within memory. Dask is also highly memory-efficient at 45.30%. Pandas, while performing well, uses slightly more memory at 75.20%.

- **Ease of Processing:**
    - **Pandas** is generally considered the easiest to use due to its mature API, extensive documentation, and large community support. Its sequential nature makes debugging straightforward.
    - **Polars** offers a very similar DataFrame API to Pandas, making it relatively easy for Pandas users to transition. Its lazy evaluation can introduce a slight learning curve but generally simplifies complex operations.
    - **Dask** is the most complex to implement and understand among the three. Its distributed nature, lazy evaluation, and the need for explicit computation calls (.compute()) require a deeper understanding of parallel programming concepts and can make debugging more challenging.
- **Best Library:** Based on the overall performance, Polars is identified as the best overall library for this benchmark on reading large datasets, offering a superior balance of speed and memory efficiency for single-node large dataset processing.

5.3 Discussion of Overall Results

Analyzing both the optimized strategies and the data processing libraries provides a comprehensive view of big data handling:

➔ **Execution Time:** For raw speed on the given task, Polars stands out as the fastest library at 8.86 seconds. This is followed by Sampling (21.62 seconds) and Load Less Data (23.65 seconds) strategies. Pandas recorded an execution time of 55.76 seconds, while Dask was the slowest at 73.76 seconds. This indicates that carefully selecting data via sampling and loading less as well as consider using a highly optimized library (Polars) may minimize time over drastic reductions.

➔ **Memory Usage:** Sampling (0.00 MB reported) demonstrates the most efficient memory usage as it's a very small part. Polars also shows excellent memory efficiency at 39.13%

while Dask is quite competent memory-wise too with the percentage of 45.30. This is crucial for datasets that really do not fit in RAM.

- ➔ **Efficiency and Scalability:**
  - ❖ Sampling is highly efficient for quick prototyping and initial exploration but sacrifices completeness.
  - ❖ "Load Less Data" and "Optimize Data Types" are effective for controlling memory footprints and speeding up Pandas when only a piece of data is required or some optimized types.
  - ❖ "Chunking" is a robust strategy for processing arbitrary large files in a sequential manner while ensuring memory stability.
  - ❖ Dask, would seem slower for the specific benchmark which might not fully leverage its parallel capabilities, but for really massive, out-of-core datasets, it is absolutely indispensable where distributed computing is a must. Its low memory footprint is a testament to its design for scalability.
  - ❖ Polars emerges as a strong performer for single-node, large-dataset processing, often outperforming Pandas without any fails due to its optimized Rust backend and efficient query engine. It ensures a good balance between speed and memory efficiency for in-memory or lazy-loaded operations.
- ➔ **Ease of Processing:** Pandas generally offers the highest ease of use due to its widespread adoption, extensive documentation, and intuitive API. Polars maintains a similar API for DataFrame, making it relatively easy to adopt for Pandas users. Dask, while it's powerful, but it's also much more complex due to lazy evaluation and distributed computing. Dask requires a much steeper learning curve for optimization and debugging. For optimization strategies like "Load Less Data" and "Sampling", they are straightforward, while "Chunking" and "Optimize Data Types" require a bit more manual intervention.

In conclusion, the selection of optimization strategy and suitable library depends heavily on the specific task, dataset size, and available computational resources. For quick insights under memory constraints, sampling or loading less data is highly effective. For high-performance, single-node processing of large datasets, Polars is definitely an excellent choice. For datasets that

truly exceed single-machine memory and require distributed computation, Dask is the preferred solution despite its initial overhead.

6.0 Task 5: Conclusion & Reflection

6.1 Summary of Key Observations

For the execution time, Sampling (Strategy 4) was the fastest, followed by Load Less Data (Strategy 1), then Chunking (Strategy 2) and Optimize Data Types (Strategy 3), with Dask (Strategy 5) being the slowest due to its overhead.

For the memory usage, Sampling (Strategy 4) used the least memory, followed by Load Less Data (Strategy 1), then Dask (Strategy 5), Optimize Data Types (Strategy 3) and Chunking (Strategy 2) with the highest due to concatenation overhead.

For rapid prototyping and initial exploration on a very large dataset where an approximation is acceptable, Sampling is the most effective strategy due to its speed and minimal memory impact. When specific columns are known to be sufficient for analysis, Load Less Data offers an excellent balance of speed and memory efficiency. Optimize Data Types is crucial if the full dataset (or a large part of it) needs to be in memory, offering substantial savings over default Pandas loading. Chunking is a robust method for processing entire datasets that don't fit in memory, especially for row-wise operations or incremental aggregations, though it can be slower.

6.2 Benefits and Limitations of Each Method

- **Traditional Methods (Full Pandas Load):**
    - **Benefits:** Simplicity for smaller datasets, rich and intuitive API, extensive community support and documentation. Easy to get started with.
    - **Limitations:** Can easily lead to MemoryError for datasets larger than available RAM. Slow loading and processing times for large files due to loading everything

into memory at once. Default data type assignments are often not memory-optimal.

- **Load Less Data:**
    - **Benefits:** Significant memory reduction by loading only necessary columns. Faster loading and processing times as less data is handled. Simple to implement using usecols parameter.
    - **Limitations:** Requires prior knowledge of which columns/rows are essential for the analysis. If other data is needed later, it requires reloading. Not suitable if the entire dataset's breadth or depth is needed.
- **Chunking:**
    - **Benefits:** Handles large files that cannot fit into memory by processing data in smaller, manageable pieces. Memory usage remains relatively constant (peak usage per chunk). Good for iterative processing.
    - **Limitations:** Can be slower than other methods for operations requiring access to the full dataset simultaneously such as complex joins and sorts on the entire dataset. The logic for processing and combining chunks can add complexity to the code.
- **Optimize Data Types:**
    - **Benefits:** Substantial reduction in memory footprint once the data is loaded. Faster computations on smaller data types. Can often be applied without loss of information if types are chosen carefully.
    - **Limitations:** Requires careful analysis of data in each column to choose the most appropriate and efficient type. Incorrect type assignment can lead to data truncation or errors. May not always yield huge gains if data is already optimal or if object columns have high cardinality.
- **Sampling:**
    - **Benefits:** Extremely fast for prototyping, quick insights and initial exploratory data analysis. Drastically reduces memory requirements. Simple to implement.
    - **Limitations:** Results are approximate and based on a subset of the data. Thus, the results may not be representative of the entire dataset, potentially missing outliers

or rare patterns. Not suitable for analyses requiring full dataset accuracy or completeness. Choice of sampling method can impact representativeness.

- **Parallel Processing with Dask:**
    - **Benefits:** Scales to distributed clusters, handles datasets larger than memory, integrates well with existing Python libraries (NumPy, Pandas).
    - **Limitations:** Can have performance overhead for smaller computations, debugging can be more challenging, and requires a deeper understanding of distributed computing concepts.

6.3 Reflection

Koh Li Hui's Reflection:

This assignment provided me with real hands-on experience in dealing with the challenges of big data. It significantly enhanced my understanding that traditional data processing approaches are inadequate for large datasets, necessitating the adoption of optimized strategies. I learned that using smarter techniques such as loading only what's needed, breaking data into chunks, optimizing data types, and running tasks in parallel with tools like Dask is essential for efficient data processing. Furthermore, comparing libraries like Pandas, Polars, and Dask helped me understand their respective strengths and weaknesses. For example, Polars performs exceptionally well with large datasets on a single machine, while Dask is ideal for scaling up and handling out-of-core data. Overall, this experience helped me truly appreciate the importance of choosing the right strategies and libraries based on the size of the data, available resources, and the goals of the analysis in real world case studies.

Koh Su Xuan's Reflection:

This assignment was a hands-on experience that brought the challenges of big data to life. Some practical skills were picked up, like using the Kaggle API to access the large dataset which is suitable to be stored in the cloud instead of locally. Through this assignment, the abstract

knowledge about big data turned into a clearer understanding. Relying on default tools to load large datasets is not just inefficient but can be a major bottleneck thus it is not simply about picking a tool but about choosing the right approach for the task, whether by using strategies or opting for lightweight libraries for faster performance. The real-world impact of these strategies is clear which is that they can lead to quicker insights and lower computational costs. One insight that stood out was how impressively Polars performed compared to traditional Pandas and the strategies, especially for fast CSV reads. Overall, this assignment clarified how important it is to be strategic and flexible when working with big data.

7.0 References

1. Singh, K. (2021, June 24). *School_donations* [Dataset]. Kaggle. Retrieved May 21, 2025, from https://www.kaggle.com/datasets/perkymaster/school-donations