SECP3133 HIGH PERFORMANCE DATA PROCESSING

SEMESTER 2 2024/2025

ASSIGNMENT 2

Mastering Big Data Handling

SECTION	02
GROUP	DATIFY
GROUP MEMBERS	1. CAMILY TANG JIA LEI (A22EC0039) 2. NG SHU YU (A22EC0228)
LECTURER	DR. ARYATI BINTI BAKRI
SUBMISSION DATE	4 JUNE 2025

Table of Contents

1.0 Introduction

The amount of data created today is accelerating. This poses challenges for classical data processing tools. For instance, libraries such as Pandas may run into issues when working with datasets in the range of several hundred megabytes. These tools can reach memory limits and experience long processing times. Consequently, businesses require more scalable and efficient methods to process and analyse the growing volume of data more effectively.

1.1 Objectives

The objectives of this assignment are to address the challenges of large-scale data extraction and processing by:

- To apply various big data techniques, including chunking, sampling, type optimisation, and parallel computing.
- To evaluate and compare the performance of traditional Pandas methods against optimised data handling strategies, such as Polars and Dask.
- To assess the effectiveness of each technique in terms of memory usage, execution time, and ease of processing.

2.0 Dataset Selection

The dataset selected is the "500K+ Spotify Songs with Lyrics, Emotions & More" dataset, which is publicly available on Kaggle (Devdope, n.d.). This dataset contains information on over 500,000 Spotify songs, including audio features, metadata, and popularity metrics. For the purpose of this analysis, the CSV version of the dataset was utilised, comprising 551,440 entries.

- **Source:** Kaggle (https://www.kaggle.com/datasets/devdope/900k-spotify/data)
- **File Format:** CSV
- **File Size:** Approximately 1,122 MB (uncompressed)
- **Domain:** Music Streaming and Audio Analysis

- **Total Records:** 551,440 entries
- **Key Features:** Song title, artist, genre, release date, popularity, and audio features such as tempo, energy, acousticness, and danceability.

The size and variety of the dataset used to test big data handling methods are considered impressive. Both categorical and continuous data are included, allowing for the application of different optimisation approaches. Moreover, challenges commonly faced by organisations in the music streaming sector are presented in the dataset, effectively mirroring real-world conditions.

3.0 Load and Inspect Data

The selected dataset was loaded and processed using three libraries—Pandas, Polars, and Dask—to demonstrate their performance. The objective was to efficiently read and process a large file while preserving the validity of the data and enabling further analysis, as illustrated in Appendix A (Table A1).

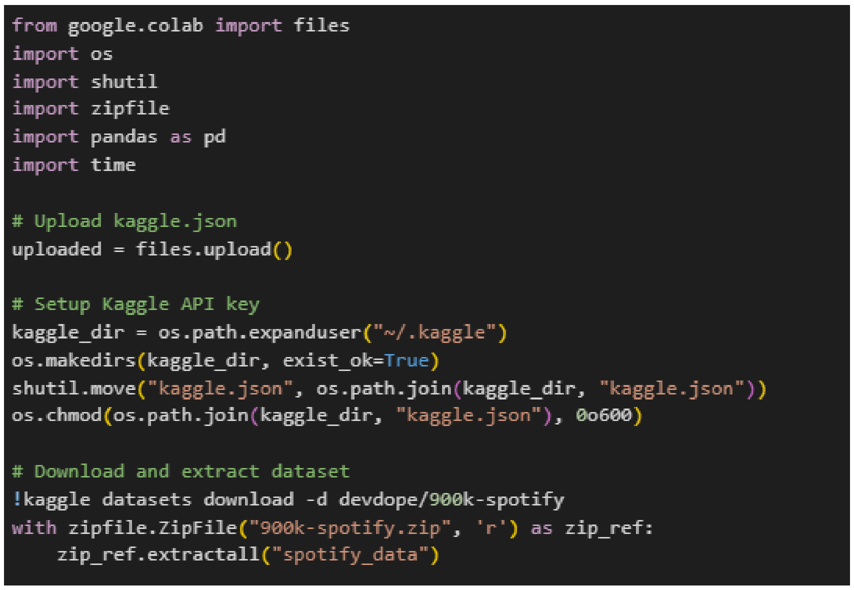
4.0 Big Data Handling Strategies

Efficient handling of large datasets is regarded as a critical aspect of working with data at scale. Traditional data processing methods are often constrained by memory limitations, reduced speed, and limited scalability. To overcome these challenges, various techniques have been developed to optimise memory usage, enhance processing speed, and enable parallel execution of tasks. In this section, several useful strategies—such as loading specific columns, optimising data types, and reading files efficiently—are discussed. These techniques are examined in the context of the capabilities provided by Pandas, Dask, and Polars to highlight how each library addresses big data challenges in resource- and performance-sensitive scenarios.

4.1 Environment Setup and Dataset Acquisition

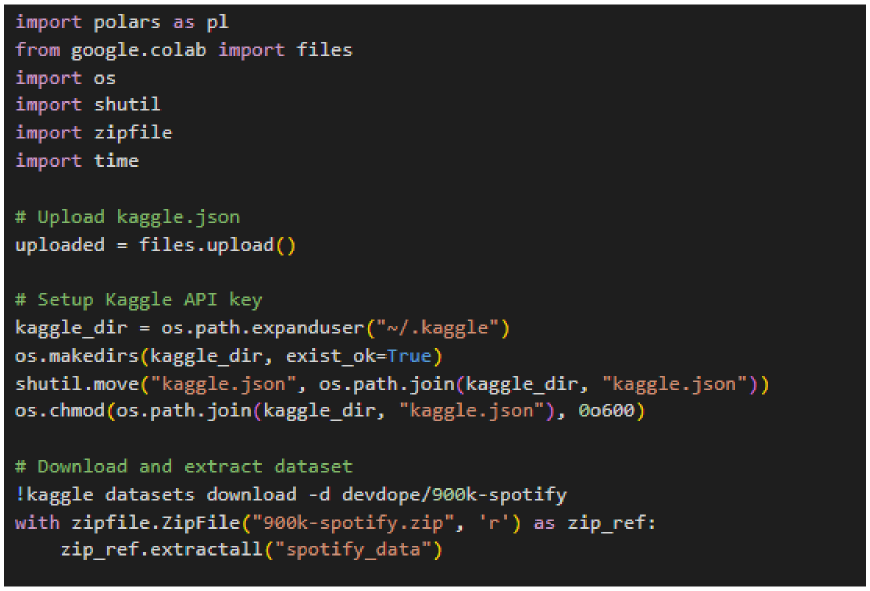
A uniform environment setup was employed across the three data processing libraries—Pandas, Polars, and Dask—to ensure real, comparable, and controlled performance experiments. All experiments were conducted using Google Colab, a platform well-suited for working with large datasets under limited resources, and offering support for live collaboration.

As shown in Figures 1, 2, and 3, the respective libraries were imported and initiated with standard preprocessing procedures. The uniform setup began with the upload of the Kaggle API access key (kaggle.json) and its definition in the system environment, which enabled secure access to the dataset. The dataset, titled “500K+ Spotify Songs with Lyrics, Emotions & More”, was accessed via the Kaggle API and extracted using Python’s ‘zipfile module’. Once extracted, the dataset was prepared for loading and processing within each of the libraries.



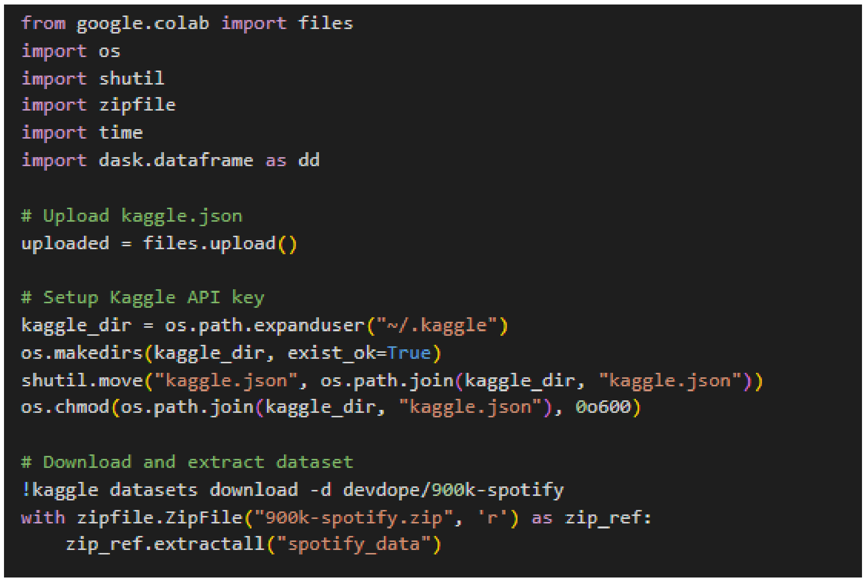
```
from google.colab import files
import os
import shutil
import zipfile
import pandas as pd
import time

# Upload kaggle.json
uploaded = files.upload()

# Setup Kaggle API key
kaggle_dir = os.path.expanduser("~/.kaggle")
os.makedirs(kaggle_dir, exist_ok=True)
shutil.move("kaggle.json", os.path.join(kaggle_dir, "kaggle.json"))
os.chmod(os.path.join(kaggle_dir, "kaggle.json"), 0o600)

# Download and extract dataset
!kaggle datasets download -d devdope/900k-spotify
with zipfile.ZipFile("900k-spotify.zip", 'r') as zip_ref:
    zip_ref.extractall("spotify_data")
```

Figure 1: Pandas Environment Setup and Data Processing Workflow



```python
import polars as pl
from google.colab import files
import os
import shutil
import zipfile
import time

# Upload kaggle.json
uploaded = files.upload()

# Setup Kaggle API key
kaggle_dir = os.path.expanduser("~/.kaggle")
os.makedirs(kaggle_dir, exist_ok=True)
shutil.move("kaggle.json", os.path.join(kaggle_dir, "kaggle.json"))
os.chmod(os.path.join(kaggle_dir, "kaggle.json"), 0o600)

# Download and extract dataset
!kaggle datasets download -d devdope/900k-spotify
with zipfile.ZipFile("900k-spotify.zip", 'r') as zip_ref:
    zip_ref.extractall("spotify_data")
```

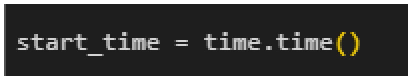
Figure 2: Polars Environment Setup and Data Processing Workflow



```python
from google.colab import files
import os
import shutil
import zipfile
import time
import dask.dataframe as dd

# Upload kaggle.json
uploaded = files.upload()

# Setup Kaggle API key
kaggle_dir = os.path.expanduser("~/.kaggle")
os.makedirs(kaggle_dir, exist_ok=True)
shutil.move("kaggle.json", os.path.join(kaggle_dir, "kaggle.json"))
os.chmod(os.path.join(kaggle_dir, "kaggle.json"), 0o600)

# Download and extract dataset
!kaggle datasets download -d devdope/900k-spotify
with zipfile.ZipFile("900k-spotify.zip", 'r') as zip_ref:
    zip_ref.extractall("spotify_data")
```

Figure 3: Dask Environment Setup and Data Processing Workflow

To facilitate performance analysis, execution time was monitored using the ‘time’ module by marking the beginning and end of the execution interval. This process is illustrated in Figure 4, where the script initiates time tracking.



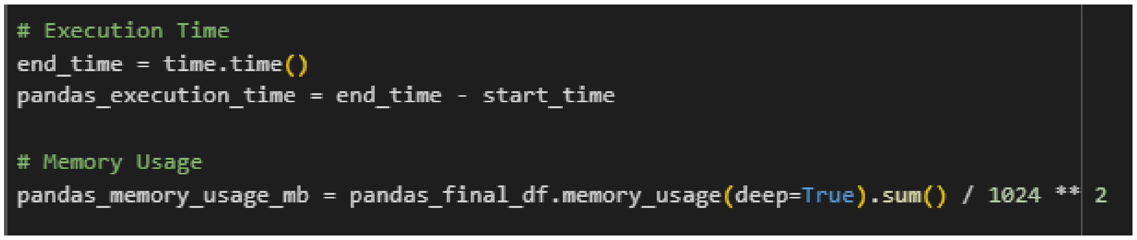
```
start_time = time.time()
```

Figure 4: Start of Execution Time Tracking in All Libraries

Moreover, each library provided its own method for determining execution time and memory usage, as shown in Figures 5, 6, and 7 for Pandas, Polars, and Dask, respectively. The total execution time was calculated by subtracting the start time from the end time, while memory consumption was obtained using library-specific functions.

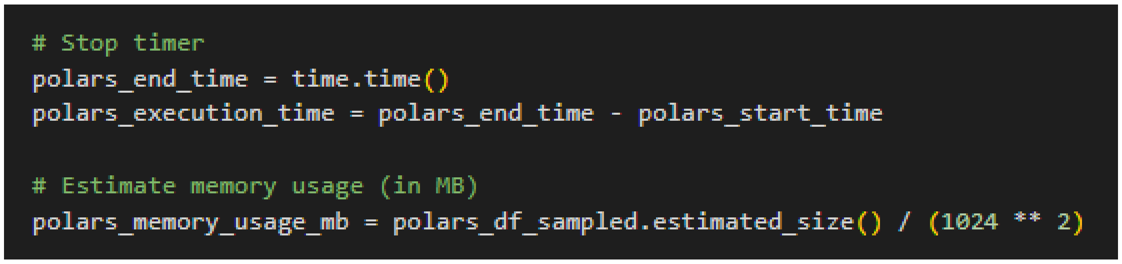
For Pandas, memory usage was calculated using .memory_usage(deep=True).sum(). In Polars, the .estimated_size() method was applied. For Dask, .memory_usage(deep=True) was used, followed by aggregation of the results.

These measurements allowed for a reasonably fair comparison of memory usage and execution time across the three libraries during the dataset processing.



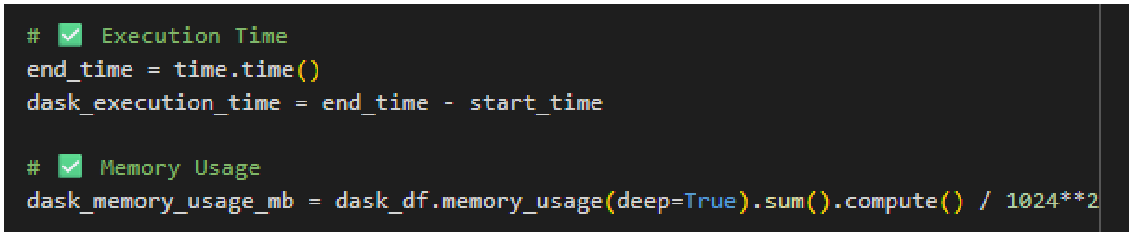
```
# Execution Time
end_time = time.time()
pandas_execution_time = end_time - start_time

# Memory Usage
pandas_memory_usage_mb = pandas_final_df.memory_usage(deep=True).sum() / 1024 ** 2
```

Figure 5: Execution Time and Memory Usage Calculation in Pandas



```
# Stop timer
polars_end_time = time.time()
polars_execution_time = polars_end_time - polars_start_time

# Estimate memory usage (in MB)
polars_memory_usage_mb = polars_df_sampled.estimated_size() / (1024 ** 2)
```

Figure 6: Execution Time and Memory Usage Calculation in Polars



```python
# ✅ Execution Time
end_time = time.time()
dask_execution_time = end_time - start_time

# ✅ Memory Usage
dask_memory_usage_mb = dask_df.memory_usage(deep=True).sum().compute() / 1024**2
```

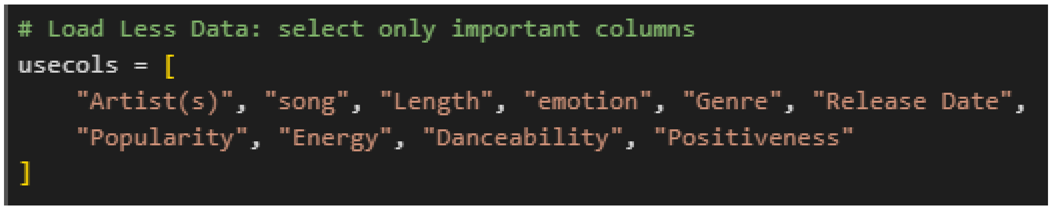
Figure 7: Execution Time and Memory Usage Calculation in Dask

4.2 Pandas

Pandas is a widely used Python library for data manipulation, analysis, and cleaning (Johari, 2018). While Pandas is capable of handling large datasets, its performance is constrained by the available system RAM. This limitation arises because Pandas loads the entire dataset into memory; therefore, if the dataset exceeds the available memory, it cannot be processed and execution is terminated.

4.2.1 Column Filtering

As shown in Figure 8, ten essential columns were selected using the ‘usecols’ parameter. By loading only the required columns, memory usage was reduced and the dataset was efficiently prepared for analysis. The selected columns included: "Artist(s)", "song", "Length", "emotion", "Genre", "Release Date", "Popularity", "Energy", "Danceability", and "Positiveness".

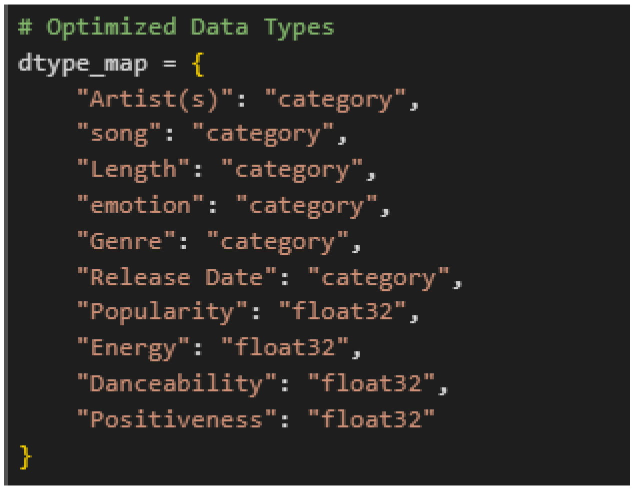


```python
# Load Less Data: select only important columns
usecols = [
    "Artist(s)", "song", "Length", "emotion", "Genre", "Release Date",
    "Popularity", "Energy", "Danceability", "Positiveness"
]
```

Figure 8: Column Filtering Using Pandas

4.2.2 Optimise Data Types

To further reduce memory consumption, data types were optimised using a predefined dictionary ‘dtype_map’, as shown in Figure 9. String-based columns were converted to ‘category’ type, while numerical columns were stored as ‘float32’, which is more memory-efficient than the default ‘float64’.

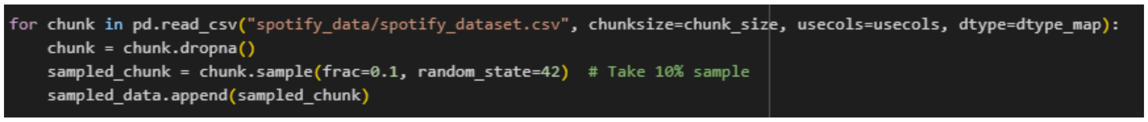


```
# Optimized Data Types
dtype_map = {
    "Artist(s)": "category",
    "song": "category",
    "Length": "category",
    "emotion": "category",
    "Genre": "category",
    "Release Date": "category",
    "Popularity": "float32",
    "Energy": "float32",
    "Danceability": "float32",
    "Positiveness": "float32"
}
```

Figure 9: Data Type Optimisation Using Pandas

4.2.3 Chunking and Data Cleaning, and Sampling

Figure 10 illustrates how chunking, data cleaning, and sampling were applied simultaneously. The dataset was read in chunks of 5,000 rows using the chunksize parameter in pd.read_csv() to avoid memory issues. Within each chunk, missing values were removed using .dropna(). Then, a 10% random sample was extracted using the .sample(frac=0.1, random_state=42) method. The random_state=42 ensures reproducibility by returning the same random selection each time the code is executed—an essential practice for consistency in experiments. This reduced the data volume while preserving representativeness. The sampled subsets were stored and later combined into a final dataset.

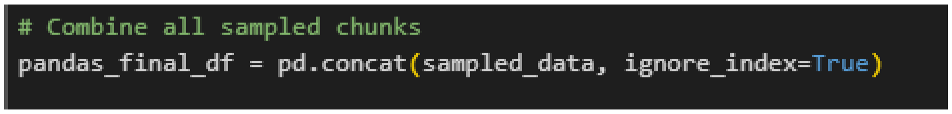


```
for chunk in pd.read_csv("spotify_data/spotify_dataset.csv", chunksize=chunk_size, usecols=usecols, dtype=dtype_map):
    chunk = chunk.dropna()
    sampled_chunk = chunk.sample(frac=0.1, random_state=42)  # Take 10% sample
    sampled_data.append(sampled_chunk)
```

Figure 10: Chunking, Cleaning, and Sampling with Pandas

4.2.4 Combining Sampling Data

As depicted in Figure 11, the list of sampled chunks was concatenated using pd.concat() to create the final DataFrame. This resulting dataset was optimised for further processing and performance testing, offering a balance between accuracy and efficiency.

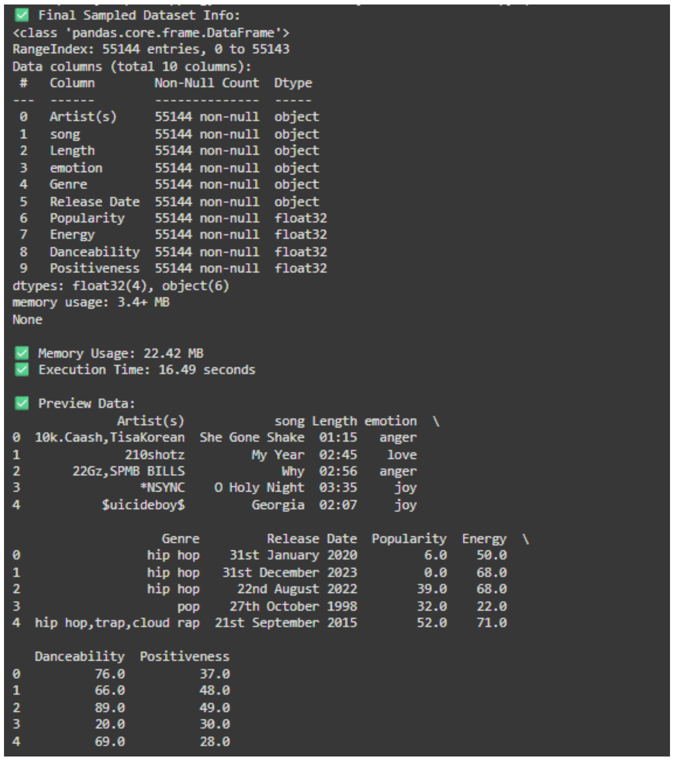


```
# Combine all sampled chunks
pandas_final_df = pd.concat(sampled_data, ignore_index=True)
```

Figure 11: Combining Sampled Chunks in Pandas

4.2.5 Output

Figure 12 shows the output of loading and inspecting the sampled Spotify dataset using Pandas. The DataFrame contains 55,144 records and 10 columns, including artist names, song titles, length, emotion, genre, release date, and several audio features such as popularity, energy, danceability, and positiveness. The data types are a mix of ‘strings’ (object) for categorical fields and ‘float32’ for numeric features, resulting in a memory usage of approximately 22.42 MB. The figure also highlights the execution time for loading and processing the data, which is about 16.49 seconds. The preview of the data in the figure illustrates the typical structure and content, showing detailed values in each column.



```
✅ Final Sampled Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 55144 entries, 0 to 55143
Data columns (total 10 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   Artist(s)     55144 non-null  object 
 1   song          55144 non-null  object 
 2   Length        55144 non-null  object 
 3   emotion       55144 non-null  object 
 4   Genre         55144 non-null  object 
 5   Release Date  55144 non-null  object 
 6   Popularity    55144 non-null  float32
 7   Energy        55144 non-null  float32
 8   Danceability  55144 non-null  float32
 9   Positiveness  55144 non-null  float32
dtypes: float32(4), object(6)
memory usage: 3.4+ MB
None

✅ Memory Usage: 22.42 MB
✅ Execution Time: 16.49 seconds

✅ Preview Data:
              Artist(s)            song Length emotion  \
0  10k.Caash,TisaKorean  She Gone Shake  01:15   anger   
1              210shotz         My Year  02:45    love   
2       22Gz,SPMB BILLS             Why  02:56   anger   
3                *NSYNC    O Holy Night  03:35     joy   
4           $uicideboy$         Georgia  02:07     joy   

                    Genre         Release Date  Popularity  Energy  \
0                 hip hop    31st January 2020         6.0    50.0   
1                 hip hop   31st December 2023         0.0    68.0   
2                 hip hop     22nd August 2022        39.0    68.0   
3                     pop    27th October 1998        32.0    22.0   
4  hip hop,trap,cloud rap  21st September 2015        52.0    71.0   

   Danceability  Positiveness  
0          76.0          37.0  
1          66.0          48.0  
2          89.0          49.0  
3          20.0          30.0  
4          69.0          28.0  
```

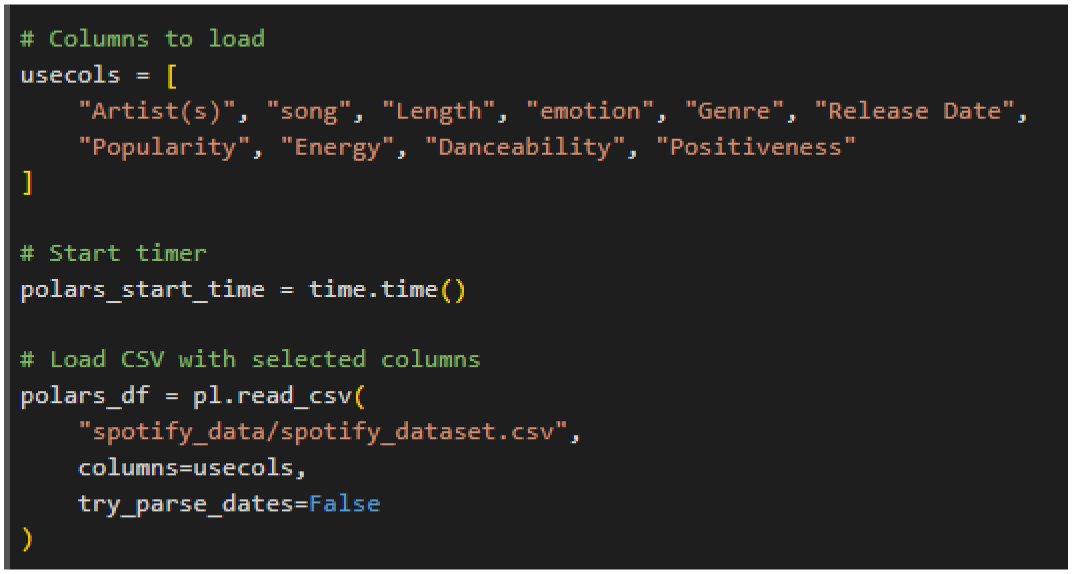
Figure 12: Output of Pandas Library

4.3 Polars

Polars is a high-performance DataFrame library written in Rust, enabling extremely fast data manipulation and thus representing a highly appealing alternative to Pandas. One of its key features is automatic multithreading, whereby parallel operations are executed transparently across multiple cores without requiring any user configuration. This parallel query engine optimises performance, especially on large datasets, significantly reducing execution time compared to Pandas (Fisher, 2024).

4.3.1 Column Filtering

Figure 13 demonstrates the Load Less Data strategy, where only ten relevant columns were selected using the ‘columns’ parameter in pl.read_csv(). This selective loading reduces memory usage and speeds up the initial data import process, which is especially beneficial when working in constrained environments like Google Colab. Additionally, Polars automatically leverages multithreading during the CSV parsing process, further improving efficiency by parallelising the workload across multiple CPU cores.

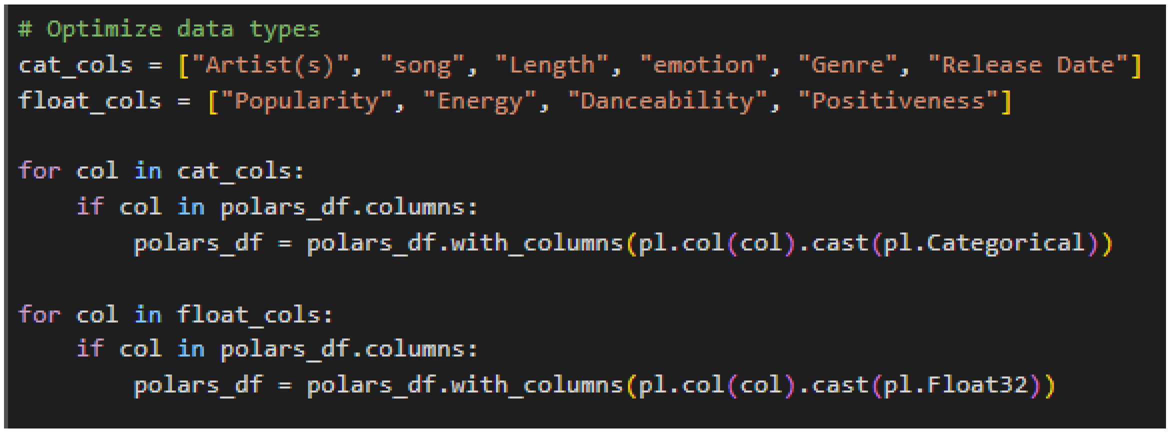


```python
# Columns to load
usecols = [
    "Artist(s)", "song", "Length", "emotion", "Genre", "Release Date",
    "Popularity", "Energy", "Danceability", "Positiveness"
]

# Start timer
polars_start_time = time.time()

# Load CSV with selected columns
polars_df = pl.read_csv(
    "spotify_data/spotify_dataset.csv",
    columns=usecols,
    try_parse_dates=False
)
```

Figure 13: Column Filtering Using Polars

4.3.2 Optimise Data Types

Following this, Figure 14 shows how the optimise data types strategy was implemented in Polars. Columns containing string or categorical data—such as "Artist(s)", "song", "Length", "emotion", "Genre", and "Release Date"—were cast to the 'category' type, which is efficient for repeated string values. Similarly, numerical columns like "Popularity", "Energy", "Danceability", and "Positiveness" were converted from default floating-point precision to 'float32', reducing memory usage without losing significant numerical detail. These operations are also automatically parallelised by Polars' multithreaded engine, which contributes to reduced processing time.

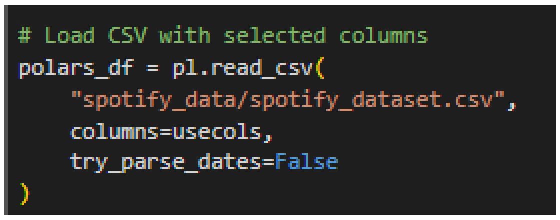


```
# Optimize data types
cat_cols = ["Artist(s)", "song", "Length", "emotion", "Genre", "Release Date"]
float_cols = ["Popularity", "Energy", "Danceability", "Positiveness"]

for col in cat_cols:
    if col in polars_df.columns:
        polars_df = polars_df.with_columns(pl.col(col).cast(pl.Categorical))

for col in float_cols:
    if col in polars_df.columns:
        polars_df = polars_df.with_columns(pl.col(col).cast(pl.Float32))
```

Figure 14: Data Type Optimisation Using Polars

4.3.3 File Loading

Figure 15 shows the column-wise file loading using Polars. Only the required columns are loaded using the 'columns' parameter. This reduces memory usage and improves efficiency by avoiding unnecessary data.

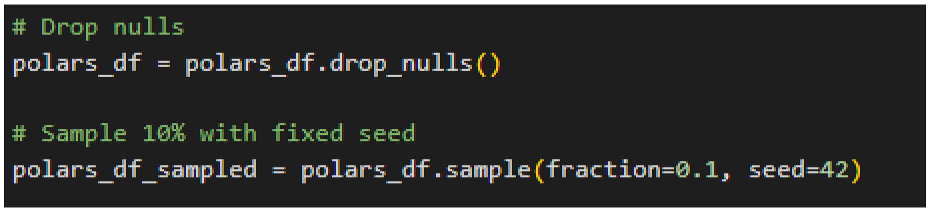


```
# Load CSV with selected columns
polars_df = pl.read_csv(
    "spotify_data/spotify_dataset.csv",
    columns=usecols,
    try_parse_dates=False
)
```

Figure 15: File Loading using Polars

4.3.4 Data Cleaning and Sampling

Figure 16 illustrates how missing values were handled using the .drop_nulls() method. At this stage, a clean dataset was obtained, making it ready for sampling. Subsequently, 10% of the cleaned dataset was sampled using .sample(fraction=0.1, seed=42). The use of seed=42 ensured that the sampling process produced consistent results across multiple runs, which is essential for reproducibility and fair performance comparisons. Both the data cleaning and sampling processes benefitted from Polars' built-in automatic multithreading, which enabled parallel data processing and resulted in significantly faster execution times.

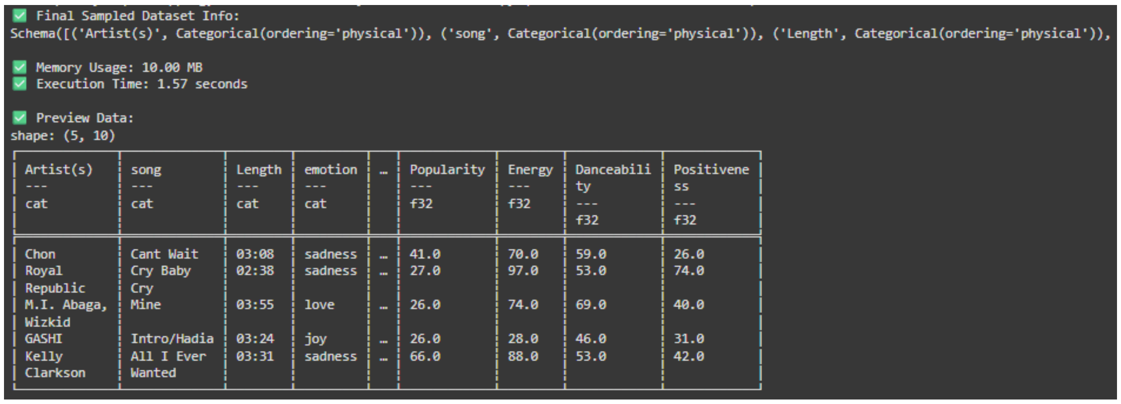


```
# Drop nulls
polars_df = polars_df.drop_nulls()

# Sample 10% with fixed seed
polars_df_sampled = polars_df.sample(fraction=0.1, seed=42)
```

Figure 16: Cleaning and Sampling with Polars

4.3.5 Output

Figure 17 displays the corresponding output when the same Spotify dataset is loaded using Polars. This figure demonstrates Polars' more efficient handling of categorical data by using optimised categorical types, leading to a reduced memory footprint of only 10 MB. The schema presented in the figure explicitly shows categorical columns with physical ordering, which enhances performance. Furthermore, the execution time for loading and processing the data is recorded at 1.57 seconds. The data preview in Figure 17 shows a compact, well-structured view of the dataset, reflecting polars' performance-orientated design.



```
✅ Final Sampled Dataset Info:
Schema([('Artist(s)', Categorical(ordering='physical')), ('song', Categorical(ordering='physical')), ('Length', Categorical(ordering='physical')),

✅ Memory Usage: 10.00 MB
✅ Execution Time: 1.57 seconds

✅ Preview Data:
shape: (5, 10)
┌────────────┬─────────────┬────────┬─────────┬───┬────────────┬────────┬────────────┬────────────┐
│ Artist(s)  ┆ song        ┆ Length ┆ emotion ┆ … ┆ Popularity ┆ Energy ┆ Danceabili ┆ Positivene │
│ ---        ┆ ---         ┆ ---    ┆ ---     ┆   ┆ ---        ┆ ---    ┆ ty         ┆ ss         │
│ cat        ┆ cat         ┆ cat    ┆ cat     ┆   ┆ f32        ┆ f32    ┆ ---        ┆ ---        │
│            ┆             ┆        ┆         ┆   ┆            ┆        ┆ f32        ┆ f32        │
╞════════════╪═════════════╪════════╪═════════╪═══╪════════════╪════════╪════════════╪════════════╡
│ Chon       ┆ Cant Wait   ┆ 03:08  ┆ sadness ┆ … ┆ 41.0       ┆ 70.0   ┆ 59.0       ┆ 26.0       │
│ Royal      ┆ Cry Baby    ┆ 02:38  ┆ sadness ┆ … ┆ 27.0       ┆ 97.0   ┆ 53.0       ┆ 74.0       │
│ Republic   ┆ Cry         ┆        ┆         ┆   ┆            ┆        ┆            ┆            │
│ M.I. Abaga,┆ Mine        ┆ 03:55  ┆ love    ┆ … ┆ 26.0       ┆ 74.0   ┆ 69.0       ┆ 40.0       │
│ Wizkid     ┆             ┆        ┆         ┆   ┆            ┆        ┆            ┆            │
│ GASHI      ┆ Intro/Hadia ┆ 03:24  ┆ joy     ┆ … ┆ 26.0       ┆ 28.0   ┆ 46.0       ┆ 31.0       │
│ Kelly      ┆ All I Ever  ┆ 03:31  ┆ sadness ┆ … ┆ 66.0       ┆ 88.0   ┆ 53.0       ┆ 42.0       │
│ Clarkson   ┆ Wanted      ┆        ┆         ┆   ┆            ┆        ┆            ┆            │
└────────────┴─────────────┴────────┴─────────┴───┴────────────┴────────┴────────────┴────────────┘
```

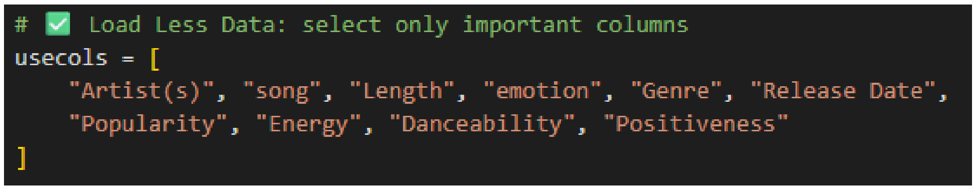
Figure 17: Output of Polars Library

4.4 Dask

Dask, a scalable parallel computing Python library is used to operate on large datasets in an efficient manner (Edwin, 2025). Dask is a library specifically for big data and provides the same interface as Pandas but with parallelism and out-of-core computation (Wijaya, 2025).

4.4.1 Column Filtering

Figure 18 illustrates the memory optimisation achieved through column filtering. To reduce memory usage and improve processing efficiency, only the necessary columns are loaded by specifying them with the ‘usecols’ parameter.

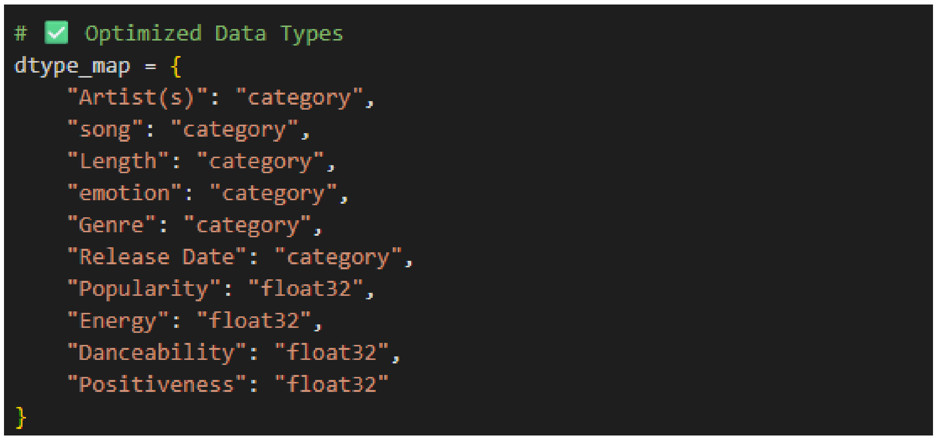


```
# ✅ Load Less Data: select only important columns
usecols = [
    "Artist(s)", "song", "Length", "emotion", "Genre", "Release Date",
    "Popularity", "Energy", "Danceability", "Positiveness"
]
```

Figure 18: Column Filtering Using Dask

4.4.2 Optimise Data Types

Figure 19 demonstrates memory optimisation through data type conversion. String-based columns are converted to the ‘category’ type, while numerical columns use ‘float32’ instead of the default ‘float64’. These optimisations help reduce memory consumption when working with large datasets.

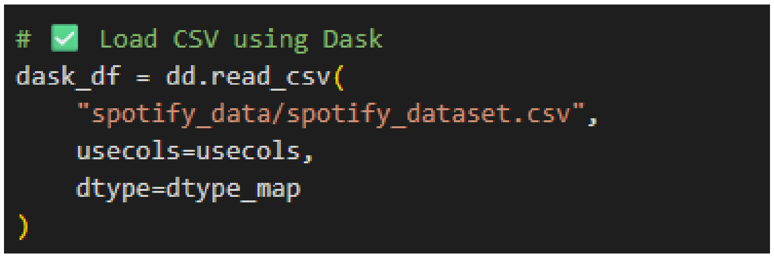


```
# ✅ Optimized Data Types
dtype_map = {
    "Artist(s)": "category",
    "song": "category",
    "Length": "category",
    "emotion": "category",
    "Genre": "category",
    "Release Date": "category",
    "Popularity": "float32",
    "Energy": "float32",
    "Danceability": "float32",
    "Positiveness": "float32"
}
```

Figure 19: Data Type Optimisation Using Dask

4.4.3 File Loading

Figure 20 shows the parallel file loading and lazy evaluation. The dataset was read using dd.read_csv(), which is different from how Pandas works. Dask reads the file in smaller chunks and processes them in parallel. This makes it possible to work with larger files without running into memory issues. Dask also does not run everything immediately; it waits until the result is needed, which helps speed up the process and avoid wasting memory.

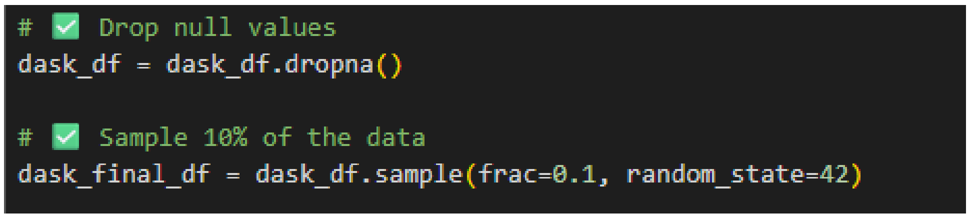


```python
# ✅ Load CSV using Dask
dask_df = dd.read_csv(
    "spotify_data/spotify_dataset.csv",
    usecols=usecols,
    dtype=dtype_map
)
```

Figure 20: Parallel File Loading and Lazy Evaluation using Dask

4.4.4 Data Cleaning and Sampling

Figure 21 shows the data cleaning and sampling without fully loading into memory. The dropna() function was used to remove rows with missing values, and a 10% sample of the data was taken using sample(). Both of these steps were done in parallel across the chunks of data, which makes them faster and more efficient, especially when dealing with big datasets.

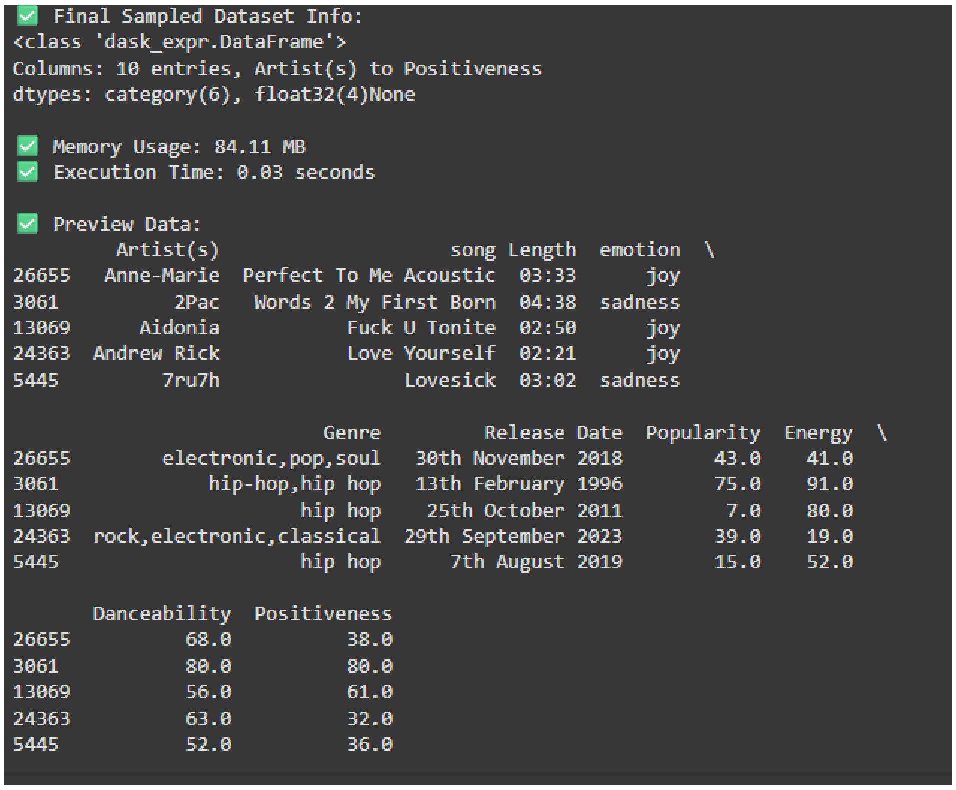


```python
# ✅ Drop null values
dask_df = dask_df.dropna()

# ✅ Sample 10% of the data
dask_final_df = dask_df.sample(frac=0.1, random_state=42)
```

Figure 21: Cleaning and Sampling with Dask

4.4.5 Output

Figure 22 shows the output and how the dataset was successfully loaded and processed using the Dask library. The info section lists the number of entries of columns and the data types. The memory usage tells how much space the dataset takes, and the execution time shows how long the whole process took. The first few rows of data are also displayed, giving a quick view that everything looks in order.



```
✅ Final Sampled Dataset Info:
<class 'dask_expr.DataFrame'>
Columns: 10 entries, Artist(s) to Positiveness
dtypes: category(6), float32(4)None

✅ Memory Usage: 84.11 MB
✅ Execution Time: 0.03 seconds

✅ Preview Data:
         Artist(s)                    song Length  emotion  \
26655   Anne-Marie  Perfect To Me Acoustic  03:33      joy
3061          2Pac   Words 2 My First Born  04:38  sadness
13069      Aidonia           Fuck U Tonite  02:50      joy
24363  Andrew Rick           Love Yourself  02:21      joy
5445         7ru7h                Lovesick  03:02  sadness

                           Genre        Release Date  Popularity  Energy  \
26655        electronic,pop,soul   30th November 2018        43.0    41.0
3061             hip-hop,hip hop   13th February 1996        75.0    91.0
13069                    hip hop    25th October 2011         7.0    80.0
24363  rock,electronic,classical  29th September 2023        39.0    19.0
5445                     hip hop      7th August 2019        15.0    52.0

       Danceability  Positiveness
26655          68.0          38.0
3061           80.0          80.0
13069          56.0          61.0
24363          63.0          32.0
5445           52.0          36.0
```

Figure 22: Output of Dask Library

5.0 Comparative Analysis

In this comparative analysis, three data processing libraries—Pandas, Polars and Dask—are compared based on the time taken for execution, memory, and the ease of processing. Figure 23 shows the comparison of execution time and memory usage in bar charts.

In terms of execution time, Dask far outshone Pandas and Polars in that aspect. Dask completed the task in just 0.03 seconds, a testament to the power of its lazy and parallelised computation. Polars also performed well with an execution time of 1.57 seconds due to the capability of its Rust-based design. Pandas took 16.49 seconds and was the slowest among the three.

In regard to memory usage, Pandas used 22.42MB, Polars much less at 10MB, and Dask the most at 84.11MB. This is due to the fact that Dask has the distributed task overhead, which compromises memory for speed and scalability. Polars, which use the columnar memory model, use the most efficient memory usage.

By ease of processing, Pandas is the most straightforward to use, most appropriate for small and medium-sized datasets due to its easy syntax and rich ecosystem. Polars is slightly less familiar to beginners but nevertheless finds a good balance between speed and simplicity. Dask brings complications with its delayed computation, which may require a learning curve, but it can be powerful at handling large-scale data.

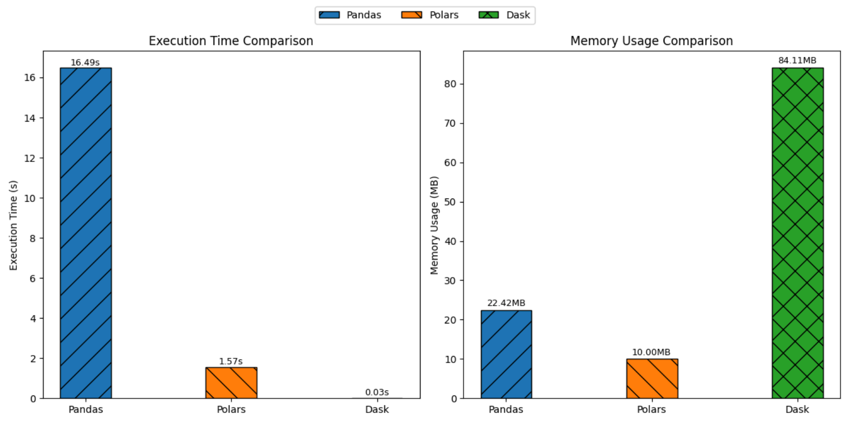


Figure 23: Comparison of Execution Time and Memory Usage between Libraries

6.0 Conclusion and Reflection

Two important observations were made regarding the execution time and resource usage of three data processing libraries (Pandas, Polars, and Dask). Vastly different execution times and memory usages were recorded, reflecting fundamentally different approaches to processing. First, Pandas, a heavyweight, widely used, and popular library, recorded the highest processing time among the three (16.49 seconds) and moderate memory consumption (22.42 MB). It is designed to operate in memory using a single-threaded approach. Polars, which utilizes Rust-based multi-threading, demonstrated the best execution time (1.57 seconds) and consumed approximately half the memory of Pandas (10.00 MB). Dask outperformed both libraries in execution time by completing the task in 0.03 seconds. Designed with a scalable parallel architecture and lazy execution, Dask minimised the risk of high memory usage from loading all data at once. However, it did record the highest memory consumption at 84.11 MB.

Each method possesses distinct advantages and disadvantages. Although Pandas offers extensive features and enjoys the highest level of user acceptance, it is primarily suitable for small to medium-sized datasets and exploratory data analysis due to its heavy memory usage, which limits scalability in terms of computing efficiency and speed. Even when sufficient memory is available, its performance is relatively slower compared to other options when handling large datasets. Polars enables fast, multithreaded data processing and requires less memory, making it well-suited for larger datasets, provided that adequate hardware is available. Dask is most effective when working with extremely large datasets that exceed RAM capacity and is particularly suitable for parallel and lazy computation. However, Dask also introduces significant complexity to programming, and its debugging process can be notably challenging.

This study emphasises the importance of selecting appropriate data processing tools and optimisation strategies tailored to dataset size and resource availability. It was demonstrated that traditional single-threaded approaches may be insufficient for big data applications without techniques such as chunking or sampling. While modern parallelised libraries are capable of delivering considerable performance gains, their effective use requires a deeper technical understanding and careful attention to detail. Ultimately, the empirical findings presented offer valuable insights into the trade-offs among usability, scalability, and efficiency in data processing—factors that are essential for timely and effective analysis in real-world data science applications.

7.0 References

Devdope. (n.d.). *500K+ Spotify songs with lyrics, emotions & more* [Data set]. Kaggle. https://www.kaggle.com/datasets/devdope/900k-spotify

Edwin. (2025, February 27). *Dask: Detailed guide for scalable computing*. Python Central. https://www.pythoncentral.io/dask-detailed-guide-for-scalabale-computing/

Fischer, B. (2024, December 23). *Python advanced: 10 things you can do with Polars (and didn't know about it)*. Medium. https://captain-solaris.medium.com/python-advanced-10-things-you-can-do-with-polars-and-didnt-know-about-it-cb8c071227ba

Johari, A. (2018, April 5). *Python Pandas guide - Learn Pandas for data analysis*. Medium. https://medium.com/edureka/python-pandas-tutorial-c5055c61d12e

Shahizan, D. (n.d.). *drshahizan* [GitHub repository]. GitHub. https://github.com/drshahizan/drshahizan

Wijaya, C. Y. (2025, May 5). *Building end-to-end data pipelines with Dask*. KDnuggets. https://www.kdnuggets.com/building-end-to-end-data-pipelines-with-dask

8.0 Appendices

8.1 Appendix A

Table A1 outlines the key steps and criteria used to compare Pandas, Polars, and Dask in handling the selected dataset. It covers environment setup, data loading, processing, performance measurement, and inspection to ensure a fair and consistent evaluation.

Table A1: Summary of Dataset Handling and Evaluation Procedures Across Libraries

Environment Setup	The Kaggle API key was uploaded and configured in Google Colab to download and extract the dataset for all methods.
Data Loading	Each library loaded the same subset of relevant columns: artist, song, length, emotion, genre, release date, popularity, energy, danceability, and positiveness. Data types were optimised by converting strings to categorical types and numerical features to 32-bit floats to reduce memory usage.
Data Processing	Missing values were removed, and a 10% random sample was taken to ensure fair comparison across libraries.
Performance Measurement	Execution time for loading, cleaning, and sampling was recorded. Memory usage of the sampled dataset was also calculated to evaluate efficiency.
Inspection	Each sampled dataset was inspected by displaying basic info such as column names, data types, number of entries, execution time, memory usage, and a preview of the first few rows.