LARGE-SCALE CSV DATA PROCESSING AND VALIDATION REPORT

ASHWINI B N

ASHWINIBN3@ICLOUD.COM

LISUM 46

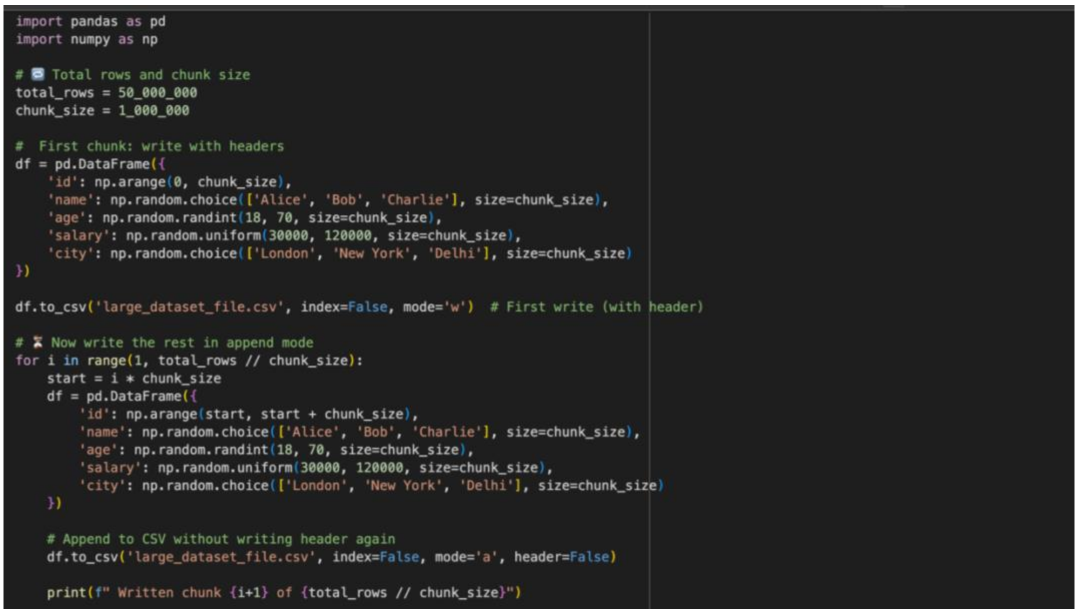
Introduction

The purpose of this task was to simulate a real-world data engineering workflow using a large CSV file. I generated a synthetic dataset with 50 million rows to replicate the challenges of working with big data. The goal was to read, clean, validate, and export the dataset efficiently using various Python tools like Pandas, Dask, Modin, and Ray.

I also created a YAML schema to define the structure of the dataset, which was later used to validate the column names and format. Finally, I saved the cleaned data in a compressed format using pipe (|) as a separator and gzip compression. This project helped me understand how to handle large files, manage memory in cloud environments like Google Colab, and follow a step-by-step process to prepare data for further use.

Step 1: Generating a Large CSV File

To start the task, I created a synthetic dataset with 50 million rows and 5 columns: id, name, age, salary, and city. Since generating all rows at once could crash the session, I wrote the file in chunks of 1 million rows using pandas. The first chunk included headers, and the rest were appended without headers. This helped avoid memory issues while still producing a large file (large_dataset_file.csv) close to 2 GB in size.



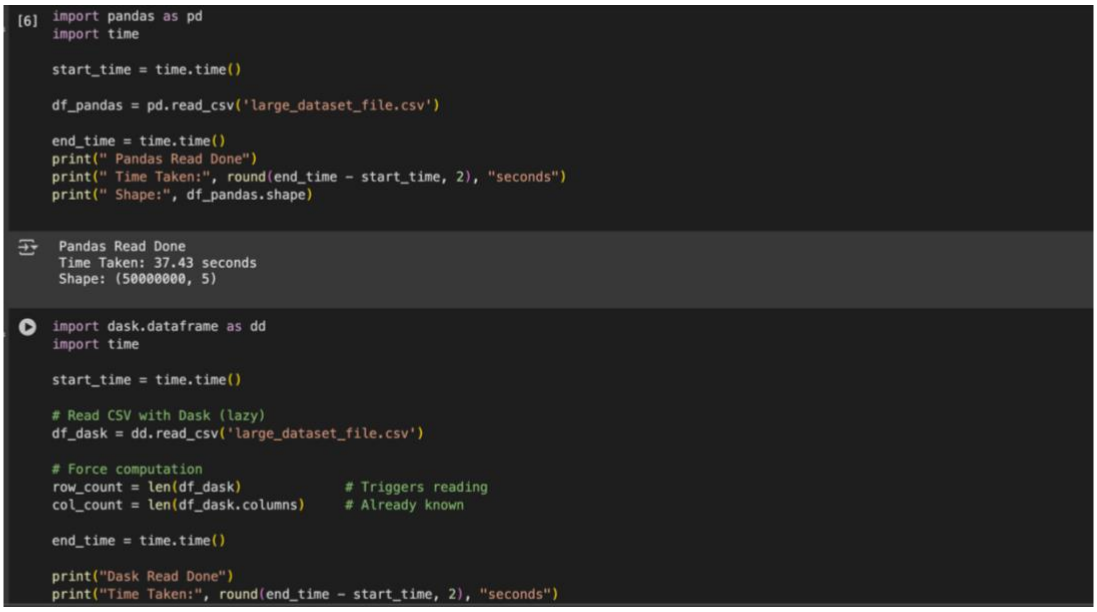
```python
import pandas as pd
import numpy as np

# 🔢 Total rows and chunk size
total_rows = 50_000_000
chunk_size = 1_000_000

#  First chunk: write with headers
df = pd.DataFrame({
    'id': np.arange(0, chunk_size),
    'name': np.random.choice(['Alice', 'Bob', 'Charlie'], size=chunk_size),
    'age': np.random.randint(18, 70, size=chunk_size),
    'salary': np.random.uniform(30000, 120000, size=chunk_size),
    'city': np.random.choice(['London', 'New York', 'Delhi'], size=chunk_size)
})

df.to_csv('large_dataset_file.csv', index=False, mode='w')  # First write (with header)

# ⏳ Now write the rest in append mode
for i in range(1, total_rows // chunk_size):
    start = i * chunk_size
    df = pd.DataFrame({
        'id': np.arange(start, start + chunk_size),
        'name': np.random.choice(['Alice', 'Bob', 'Charlie'], size=chunk_size),
        'age': np.random.randint(18, 70, size=chunk_size),
        'salary': np.random.uniform(30000, 120000, size=chunk_size),
        'city': np.random.choice(['London', 'New York', 'Delhi'], size=chunk_size)
    })

    # Append to CSV without writing header again
    df.to_csv('large_dataset_file.csv', index=False, mode='a', header=False)

    print(f" Written chunk {i+1} of {total_rows // chunk_size}")
```

Step 2: Reading the File Using Different Methods

I tested four different Python libraries to read the 2GB CSV file and compared how long each method took:

- **Pandas**: Standard single-threaded read
- **Dask**: Reads in parallel and handles large files efficiently
- **Modin (Ray backend)**: Parallelized pandas using Ray for speed
- **Ray (Direct)**: Used Ray to run the read task as a remote process

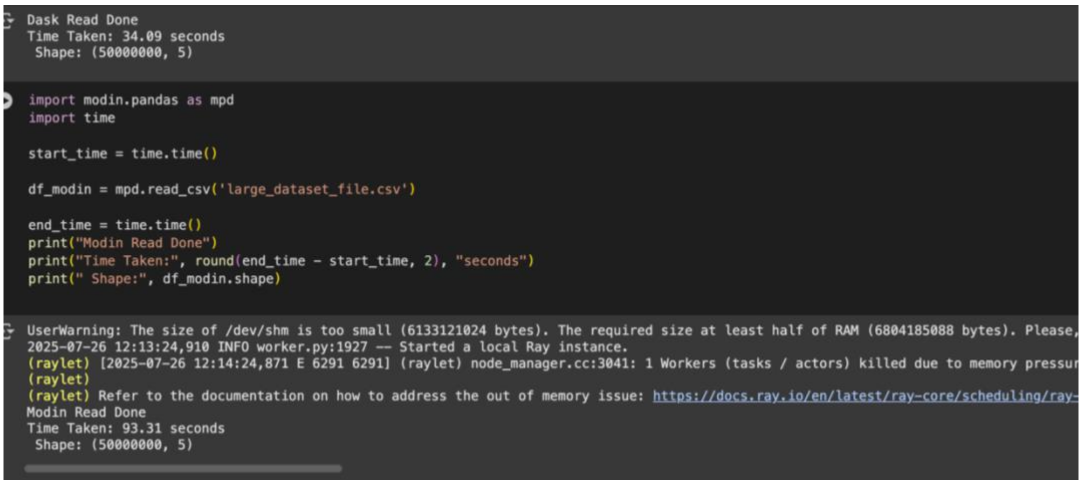
For each method, I recorded the time taken and shape of the dataset to confirm it read correctly. Then I created a summary table comparing their performance.



```
[6] import pandas as pd
    import time

    start_time = time.time()

    df_pandas = pd.read_csv('large_dataset_file.csv')

    end_time = time.time()
    print(" Pandas Read Done")
    print(" Time Taken:", round(end_time - start_time, 2), "seconds")
    print(" Shape:", df_pandas.shape)
```

```
 Pandas Read Done
 Time Taken: 37.43 seconds
 Shape: (50000000, 5)
```

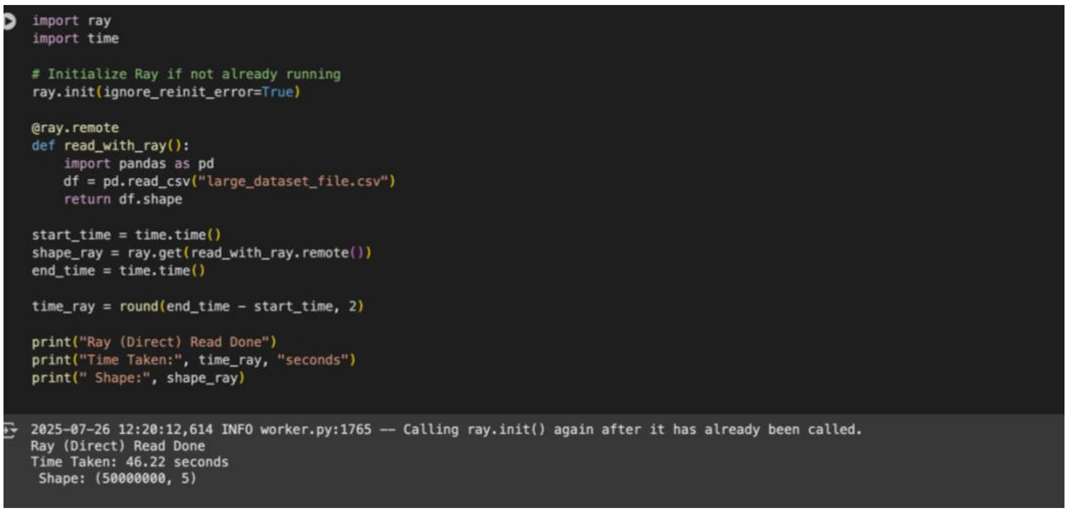
```
import dask.dataframe as dd
import time

start_time = time.time()

# Read CSV with Dask (lazy)
df_dask = dd.read_csv('large_dataset_file.csv')

# Force computation
row_count = len(df_dask)             # Triggers reading
col_count = len(df_dask.columns)     # Already known

end_time = time.time()

print("Dask Read Done")
print("Time Taken:", round(end_time - start_time, 2), "seconds")
```



```
Dask Read Done
Time Taken: 34.09 seconds
 Shape: (50000000, 5)
```

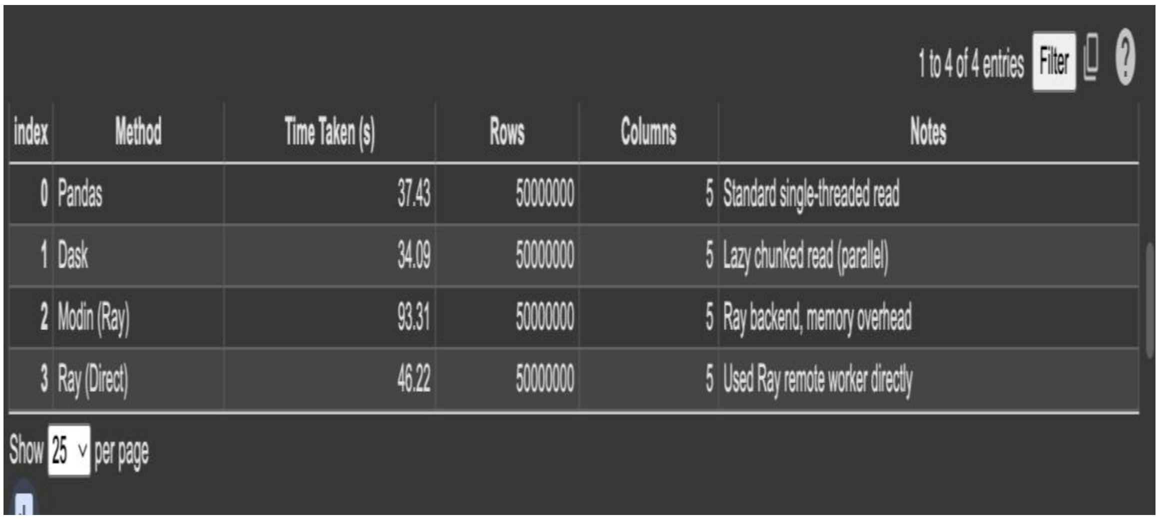
```
import modin.pandas as mpd
import time

start_time = time.time()

df_modin = mpd.read_csv('large_dataset_file.csv')

end_time = time.time()
print("Modin Read Done")
print("Time Taken:", round(end_time - start_time, 2), "seconds")
print(" Shape:", df_modin.shape)
```

```
UserWarning: The size of /dev/shm is too small (6133121024 bytes). The required size at least half of RAM (6804185088 bytes). Please,
2025-07-26 12:13:24,910 INFO worker.py:1927 -- Started a local Ray instance.
(raylet) [2025-07-26 12:14:24,871 E 6291 6291] (raylet) node_manager.cc:3041: 1 Workers (tasks / actors) killed due to memory pressur
(raylet)
(raylet) Refer to the documentation on how to address the out of memory issue: https://docs.ray.io/en/latest/ray-core/scheduling/ray-
Modin Read Done
Time Taken: 93.31 seconds
 Shape: (50000000, 5)
```



```
import ray
import time

# Initialize Ray if not already running
ray.init(ignore_reinit_error=True)

@ray.remote
def read_with_ray():
    import pandas as pd
    df = pd.read_csv("large_dataset_file.csv")
    return df.shape

start_time = time.time()
shape_ray = ray.get(read_with_ray.remote())
end_time = time.time()

time_ray = round(end_time - start_time, 2)

print("Ray (Direct) Read Done")
print("Time Taken:", time_ray, "seconds")
print(" Shape:", shape_ray)
```

```
2025-07-26 12:20:12,614 INFO worker.py:1765 -- Calling ray.init() again after it has already been called.
Ray (Direct) Read Done
Time Taken: 46.22 seconds
 Shape: (50000000, 5)
```



1 to 4 of 4 entries

index	Method	Time Taken (s)	Rows	Columns	Notes
0	Pandas	37.43	50000000	5	Standard single-threaded read
1	Dask	34.09	50000000	5	Lazy chunked read (parallel)
2	Modin (Ray)	93.31	50000000	5	Ray backend, memory overhead
3	Ray (Direct)	46.22	50000000	5	Used Ray remote worker directly

Show 25 per page

Step 3: Cleaning Column Names

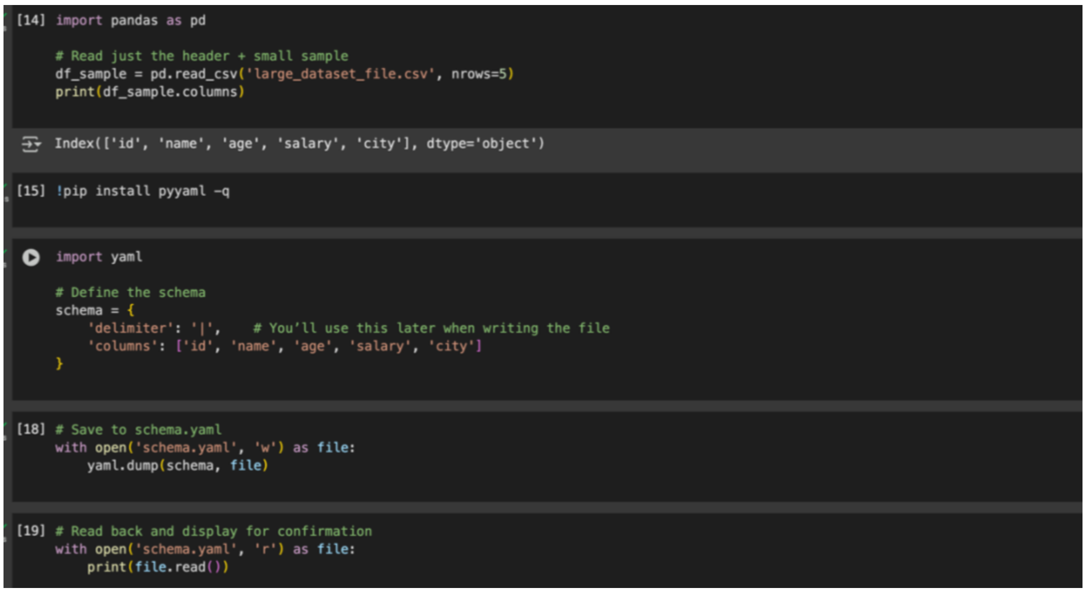
Before proceeding further, I checked and cleaned the column names in the dataset. I made sure there were no special characters, extra spaces, or inconsistencies. I confirmed that the columns were already clean, but this step is important to ensure smooth validation and export later.

Step 4: Creating the YAML Schema

Next, I created a YAML file (schema.yaml) to store the expected structure of the dataset. The YAML file included:

- The delimiter I planned to use (|)
- The column names in the correct order

This schema acted as a reference point for validating the dataset later in the pipeline.



```
[14] import pandas as pd

     # Read just the header + small sample
     df_sample = pd.read_csv('large_dataset_file.csv', nrows=5)
     print(df_sample.columns)
```

```
Index(['id', 'name', 'age', 'salary', 'city'], dtype='object')
```

```
[15] !pip install pyyaml -q
```

```
     import yaml

     # Define the schema
     schema = {
         'delimiter': '|',     # You'll use this later when writing the file
         'columns': ['id', 'name', 'age', 'salary', 'city']
     }
```

```
[18] # Save to schema.yaml
     with open('schema.yaml', 'w') as file:
         yaml.dump(schema, file)
```

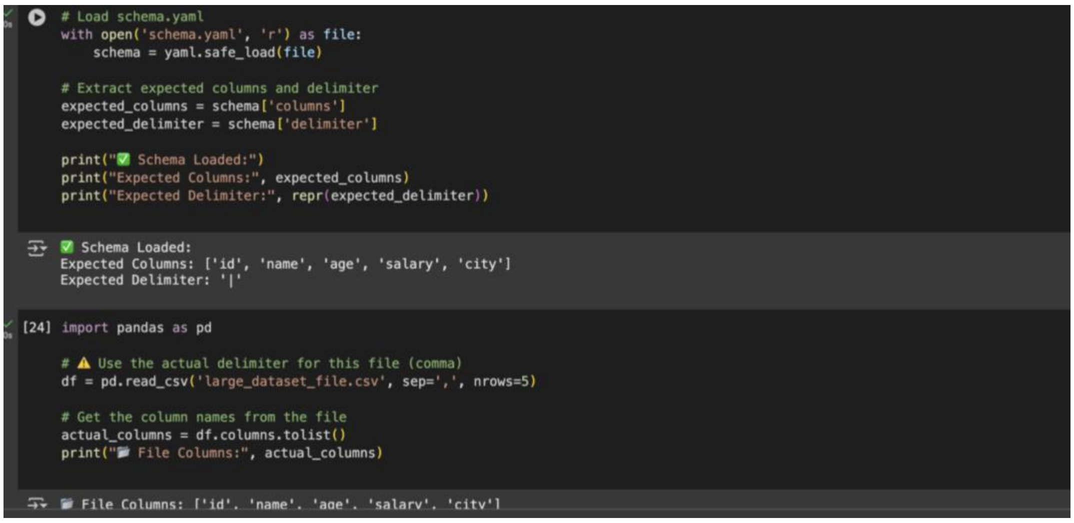
```
[19] # Read back and display for confirmation
     with open('schema.yaml', 'r') as file:
         print(file.read())
```

Step 5: Validating Dataset with YAML Schema

I read the YAML file and compared it against the actual dataset to make sure:

- The number of columns matched
- The column names matched exactly, in the correct order

I used a small sample from the CSV file to do this validation. Once both checks passed, I confirmed the dataset structure was valid and matched the schema.

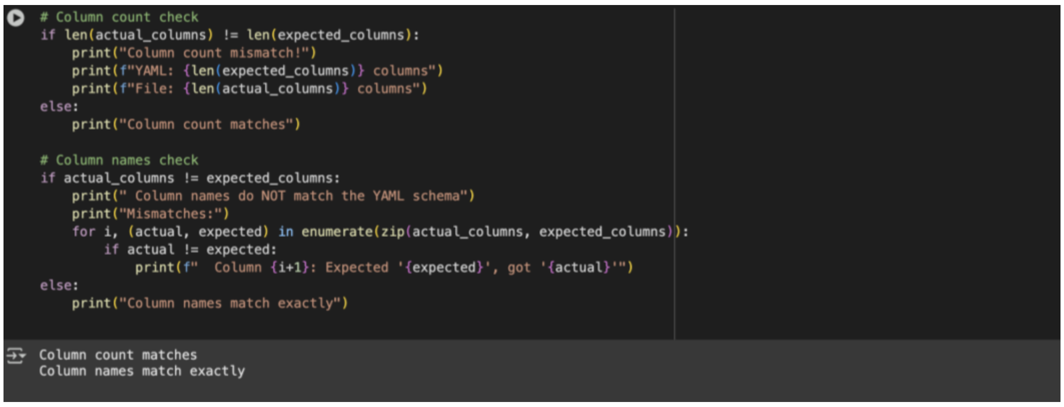


```
     # Load schema.yaml
     with open('schema.yaml', 'r') as file:
         schema = yaml.safe_load(file)

     # Extract expected columns and delimiter
     expected_columns = schema['columns']
     expected_delimiter = schema['delimiter']

     print("✅ Schema Loaded:")
     print("Expected Columns:", expected_columns)
     print("Expected Delimiter:", repr(expected_delimiter))
```

```
✅ Schema Loaded:
Expected Columns: ['id', 'name', 'age', 'salary', 'city']
Expected Delimiter: '|'
```

```
[24] import pandas as pd

     # ⚠ Use the actual delimiter for this file (comma)
     df = pd.read_csv('large_dataset_file.csv', sep=',', nrows=5)

     # Get the column names from the file
     actual_columns = df.columns.tolist()
     print("📁 File Columns:", actual_columns)
```

```
📁 File Columns: ['id', 'name', 'age', 'salary', 'city']
```



```python
# Column count check
if len(actual_columns) != len(expected_columns):
    print("Column count mismatch!")
    print(f"YAML: {len(expected_columns)} columns")
    print(f"File: {len(actual_columns)} columns")
else:
    print("Column count matches")

# Column names check
if actual_columns != expected_columns:
    print(" Column names do NOT match the YAML schema")
    print("Mismatches:")
    for i, (actual, expected) in enumerate(zip(actual_columns, expected_columns)):
        if actual != expected:
            print(f"  Column {i+1}: Expected '{expected}', got '{actual}'")
else:
    print("Column names match exactly")
```

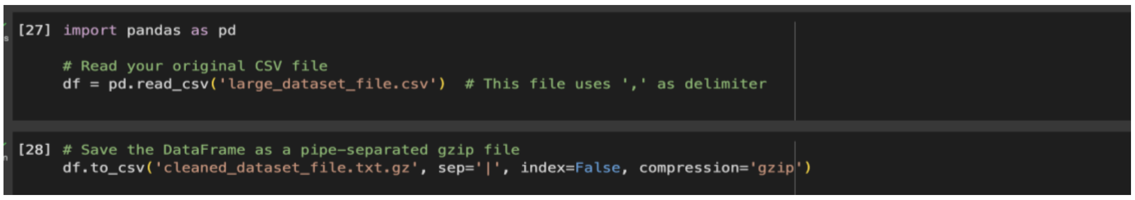
```
Column count matches
Column names match exactly
```

Step 6: Saving the Cleaned File

After validation, I saved the dataset in a new format using:

- Pipe (|) as the separator (instead of comma)
- Gzip compression to reduce file size

The final file was saved as cleaned_dataset_file.txt.gz. This format is efficient for storage and matches the structure defined in the YAML schema.



```python
[27] import pandas as pd

     # Read your original CSV file
     df = pd.read_csv('large_dataset_file.csv')  # This file uses ',' as delimiter
```

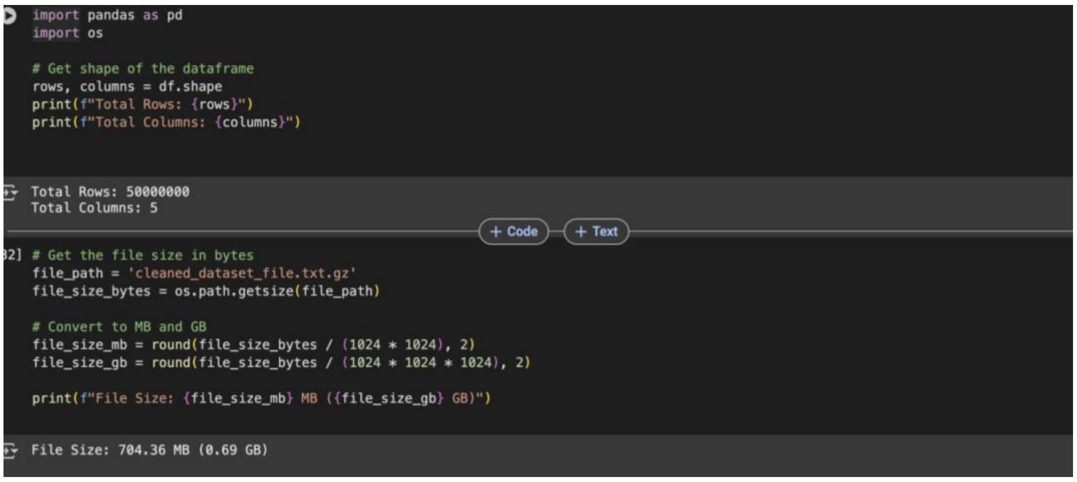
```python
[28] # Save the DataFrame as a pipe-separated gzip file
     df.to_csv('cleaned_dataset_file.txt.gz', sep='|', index=False, compression='gzip')
```

Step 7: Summary of the Final File

In the final step, I checked:

- The number of rows and columns in the cleaned file
- The file size in MB and GB

The output confirmed that the file had 50 million rows, 5 columns, and a compressed size of around 704 MB. This step helped verify that everything was exported correctly.



```
import pandas as pd
import os

# Get shape of the dataframe
rows, columns = df.shape
print(f"Total Rows: {rows}")
print(f"Total Columns: {columns}")
```

```
Total Rows: 50000000
Total Columns: 5
```

```
# Get the file size in bytes
file_path = 'cleaned_dataset_file.txt.gz'
file_size_bytes = os.path.getsize(file_path)

# Convert to MB and GB
file_size_mb = round(file_size_bytes / (1024 * 1024), 2)
file_size_gb = round(file_size_bytes / (1024 * 1024 * 1024), 2)

print(f"File Size: {file_size_mb} MB ({file_size_gb} GB)")
```

```
File Size: 704.36 MB (0.69 GB)
```

Conclusion

This project helped me understand how to work with large datasets step by step, from data generation to validation and final export. I explored different tools like Pandas, Dask, Modin, and Ray to see how they handle big data in terms of speed and performance. I also learned how to create and use a YAML schema to keep the data structure consistent, and how to save the cleaned data in a compressed format that is easier to store and share.

Overall, this task gave me hands-on experience with real-world data processing challenges and taught me how to manage memory, validate data, and use scalable tools in a cloud environment like Google Colab.