

# **MoP for Processing Big Datasets Efficiently (Chunk-Based Reading)**

## **GOAL:**

Students will learn how to:

- Process a massive dataset (e.g., 5 million customer usage rows) without crashing memory.
- Use chunk-based reading in pandas to handle big files in smaller batches.
- Incrementally calculate results such as total usage and count of heavy data users.

1) Pre-Lab Setup - Software

**Python 3 (Anaconda preferred):** Programming environment

**Jupyter Notebook:** To run the experiment interactively

**Pandas library** : For chunk-based data processing

1.1 Install Required Packages:

Run in Anaconda Prompt or Jupyter notebook:

pip install pandas

2) Create a Large Sample Dataset (Simulated 5M rows)

Create new folder: "training\_bigdata"

Create new notebook in jupyter browse in above folder: "bigdata"

We will simulate a large telecom dataset so we don't need a ~500 MB file. This step generates a CSV with random values.

```
import pandas as pd
import numpy as np

# Create a large dataset of 5 million telecom records
rows = 5_000_000
np.random.seed(42)

data = {
    'customer_id': np.arange(1, rows + 1),
    'data_used_gb': np.random.uniform(0.1, 100, size=rows).round(2),
    'call_duration': np.random.randint(1, 500, size=rows),
    'region': np.random.choice(['Delhi', 'Mumbai', 'Chennai', 'Kolkata', 'Bangalore'], size=rows)
}

df = pd.DataFrame(data)
df.to_csv('big_telecom_data.csv', index=False)
```

```
print("✅ big_telecom_data.csv created successfully with 5 million rows.")
```

**File created:** big\_telecom\_data.csv (~300–400 MB, depending on hardware).

### 3) Process Large Dataset in Chunks (Main Experiment)

```
import pandas as pd
```

```
# Step 1: Read file in chunks of 100,000 rows
chunks = pd.read_csv('big_telecom_data.csv', chunksize=100000)

# Step 2: Initialize counters
total_usage = 0
heavy_users = 0

# Step 3: Process each chunk separately
for chunk in chunks:
    total_usage += chunk['data_used_gb'].sum()      # incremental sum
    heavy_users += (chunk['data_used_gb'] > 50).sum() # count high-usage users

# Step 4: Print results
print(f"✅ Total data usage: {total_usage:.2f} GB")
print(f"✅ Heavy users (>50GB): {heavy_users};")
```

## What's Happening

- chunksize=100000: Loads only 100k rows at a time (instead of all 5M).
- for chunk in chunks: Iterates through each subset sequentially.
- sum(): Adds each chunk's total usage to the running total.
- (data\_used\_gb > 50).sum() Counts users who used more than 50GB.

This incremental approach avoids memory overflow.

If you check your system task manager, memory usage stays stable (~200–300 MB).

### 4) Add a Progress Tracker - to visualize progress:

```
import pandas as pd
from tqdm import tqdm # pip install tqdm

chunks = pd.read_csv('big_telecom_data.csv', chunksize=100000)
total_usage = 0
heavy_users = 0

for chunk in tqdm(chunks, desc="Processing..."):
    total_usage += chunk['data_used_gb'].sum()
    heavy_users += (chunk['data_used_gb'] > 50).sum()
```

```
print(f"\n✓ Total Data Usage: {total_usage:.2f} GB")
print(f"✓ Heavy Users (>50GB): {heavy_users};")
```

5) Validate Results – let's run sanity checks:

```
print("Average data used per user (approx.):", round(total_usage / 5_000_000, 2), "GB")
```

Expected output: around 50 GB average (since we generated random 0-100 values).

6) Discussion & Real-World Connection

- **Chunk Processing:** How PySpark or Hadoop MapReduce divides massive datasets across workers
- **Memory Efficiency:** Important for servers with limited RAM
- **Incremental Aggregation:** Used in ETL pipelines, data lakes, and warehouse ingestion jobs
- **Performance Benefit:** 100× faster and 90% less memory than trying to load everything into RAM

Benefits: Chunk-based processing

- Reads partial data in memory -> perfect for massive CSVs.
- Reduces crash risk on limited-memory systems.
- Foundation for distributed big data tools like PySpark, Dask, or Hadoop.
- Teaches real-world telecom data handling efficiency.