Advanced Python Programming for Data Science

CH – 02

Data Sources and APIs

Baikuntha Acharya (baikunth2a@gmail.com)

Senior Lecturer, Sagarmatha Engineering College, Sanepa, Lalitpur

Structured Data

- ✓ Structured data refers to data that is **organized according to an explicit data model**, where each attribute has a defined meaning, type, and position.
- ✓ Such data is typically stored in **tables** and is well suited for **relational databases**.
  - Conforms to a fixed schema
  - Stored in rows and columns
  - Supports direct querying and relational operations
  - Enables efficient storage, retrieval, and analysis
- ✓ Examples: CSV, Excel, JSON

Unstructured data

The university has 5600 students.
John's ID is number 1, he is 18 years old and already holds a B.Sc. degree.
David's ID is number 2, he is 31 years old and holds a Ph.D. degree. Robert's ID is number 3, he is 51 years old and also holds the same degree as David, a Ph.D. degree.

Semi-structured data

```
<University>
  <Student ID="1">
    <Name>John</Name>
    <Age>18</Age>
    <Degree>B.Sc.</Degree>
  </Student>
  <Student ID="2">
    <Name>David</Name>
    <Age>31</Age>
    <Degree>Ph.D. </Degree>
  </Student>
  ....
</University>
```

Structured data

ID	Name	Age	Degree
1	John	18	B.Sc.
2	David	31	Ph.D.
3	Robert	51	Ph.D.
4	Rick	26	M.Sc.
5	Michael	19	B.Sc.

Unstructured Data

- ✓ Unstructured data refers to data that **lacks a predefined data model,** often stored in raw, free-form formats like text, images, or video and cannot be directly organized into relational tables without transformation.
- ✓ Key Characteristics:
  - **No fixed schema or structure**, can include varied formats like text, images, audio, and video.
  - **Requires advanced processing techniques** such as NLP (for text), computer vision (for images), and speech recognition (for audio).
  - **Typically stored in NoSQL** databases (e.g., MongoDB, Cassandra) or file systems (e.g., HDFS).
  - **Difficult to query** using standard querying languages; requires tools like MapReduce or Spark for distributed processing.

Reading and Writing Structured Data

✓ CSV Files

- CSV is one of the most common formats for tabular data.
- Values separated by commas/delimiters
- No data types or formatting
- Pandas a popular data manipulation library in Python offers a straightforward way to read CSV files.

```python
import pandas as pd

# Read a CSV file into a DataFrame
df_csv = pd.read_csv('your_file.csv')

# Write a DataFrame to a CSV file
df_csv.to_csv('output_file.csv', index=False)
```

```python
# CSV with different delimiters
df1 = pd.read_csv("data.csv", sep=';')

# CSV without header
df2 = pd.read_csv("data.csv", header=None)

# Selecting specific columns
df3 = pd.read_csv("data.csv", usecols=["name","age"])

# Large CSV (memory efficient)
df4 = pd.read_csv("data.csv", chunksize=10000)

# CSV from URL or online source
df5 = pd.read_csv("data.csv")

# CSV with encoding issues
df6 = pd.read_csv("data.csv", encoding="latin1")
```

Reading and Writing Structured Data (Contd..)

✓ JSON Files

- Stores data as key–value pairs
- Supports nested structures using objects and arrays
- Can represent complex and hierarchical data
- Human-readable and machine-friendly
- Widely used in web APIs and configuration files


```
{
  "DocumentType": 1,
  "No.": "S-ORD101001",
  "SellToCustNo": "10000",
  "PostingDate": "2023-04-02",
  "Lines": [
    {
      "LineNo": 10000,
      "Type": 2,
      "No": "1996-S",
      "Quantity": 12,
      "UnitPrice": 1397.3
    },
    {
      "LineNo": 20000,
      "Type": 2,
      "No": "1900-S",
      "Quantity": 4,
      "UnitPrice": 192.8
    }
  ]
}
```


```python
import pandas as pd

# Read a JSON file into a DataFrame
df_json = pd.read_json('your_file.json')

# Write a DataFrame to a JSON file
df_json.to_json('output_file.json', orient='records', lines=True)

# For nested JSON, pd.json_normalize() is a useful function
```

Example JSON (records format)

```json
[
  {"name": "Ram", "age": 20},
  {"name": "Sita", "age": 22}
]
```

How pandas interprets it

- Each **dictionary → one row**
- Each **key → a column**
- Each **value → cell data**

`orient="records"` means ***"each JSON object is a row in the DataFrame."***

Reading and Writing Structured Data (Contd..)

✓ Different Ways to Read JSON Files

```
# 1. Standard JSON file
with open("data.json") as f:
    data1 = json.load(f)   # data as Python dict or list

# 2. JSON from a string
json_str = '{"name": "Ram", "age": 20}'
data2 = json.loads(json_str)

# 3. JSON from URL or online source
with urllib.request.urlopen("https://example.com/data.json") as f:
    data3 = json.load(f)

# 4. Line-delimited JSON (JSON objects per line)
data4 = []
with open("data.json") as f:
    for line in f:
        data4.append(json.loads(line))

# 5. Nested JSON flattening (optional helper)
from pandas import json_normalize
data5 = json_normalize(data1)   # converts nested JSON to flat table
```

Reading and Writing Structured Data (Contd..)

✓ Excel Files

- Spreadsheet-based file format
- Data stored in cells organized into rows and columns
- Supports formulas, charts, pivot tables, and formatting
- Allows multiple worksheets in a single file
- Useful for data analysis and reporting
- Less suitable for automated processing of large datasets
- Pandas provides functions to read from and write to Excel files (requires libraries like openpyxl for .xlsx files).

```
import pandas as pd

# Read an Excel file (supports .xls and .xlsx)
df_excel = pd.read_excel('your_file.xlsx', sheet_name='Sheet1')

# Write a DataFrame to an Excel file
df_excel.to_excel('output_file.xlsx', index=False, sheet_name='Sheet1')
```

Reading and Writing Structured Data (Contd..)

✓ Different Ways to Read EXCEL Files

```python
# Read default sheet (first sheet)
df1 = pd.read_excel("data.xlsx")

# Read specific sheet by name
df2 = pd.read_excel("data.xlsx", sheet_name="Sheet2")

# Read specific sheet by index (0-based)
df3 = pd.read_excel("data.xlsx", sheet_name=1)

# Read multiple sheets at once (returns dict of DataFrames)
dfs = pd.read_excel("data.xlsx", sheet_name=["Sheet1", "Sheet3"])
```

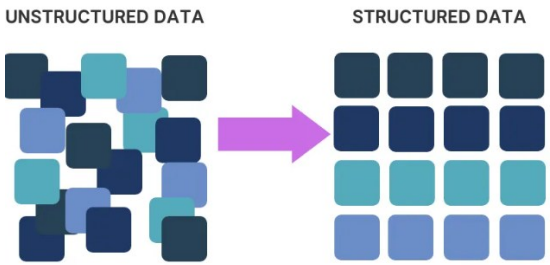
```python
from openpyxl import load_workbook

# Load workbook
wb = load_workbook("data.xlsx")

# List all sheet names
print(wb.sheetnames)

# Access a specific sheet
sheet = wb["Sheet2"]

# Read cell values
for row in sheet.iter_rows(values_only=True):
    print(row)
```

```python
import xlrd

book = xlrd.open_workbook("data.xls")
sheet = book.sheet_by_name("Sheet1")

# Read a cell
value = sheet.cell_value(0, 0)  # row 0, column 0
```

Reading and Writing Structured Data (Contd..)

✓ Text Files (Unstructured)

- Stores data as plain text characters
- No predefined structure or schema
- Can be used to store logs, instructions, or simple data
- Universally supported across platforms
- Python's built-in open() function is the primary way to read and write basic text files.




```python
# Read from a text file
with open('your_text_file.txt', 'r') as file:
    content = file.read()
    # Process content (e.g., use string manipulation, regex)

# Write to a text file
with open('output_text_file.txt', 'w') as file:
    file.write("Structured output here.")
```

Reading and Writing Structured Data (Contd..)

✓ Different Ways to Read Text Files

```python
# Different ways to read text files in Python
from pathlib import Path
import urllib.request

# 1. Read entire file
txt1 = open("data.txt").read()

# 2. Read all lines into a list
with open("data.txt") as f:
    txt2 = f.readlines()

# 3. Iterate line by line (memory efficient)
with open("data.txt") as f:
    txt3 = [line.strip() for line in f]

# 4. Read single line
with open("data.txt") as f:
    txt4 = f.readline()
```

```python
# 5. Read fixed number of characters
with open("data.txt") as f:
    txt5 = f.read(100)

# 6. Using pathlib
txt6 = Path("data.txt").read_text()

# 7. Read large file in chunks
txt7 = []
with open("data.txt") as f:
    while chunk := f.read(1024):
        txt7.append(chunk)

# 8. Read from URL
with urllib.request.urlopen("https://example.com/data.txt") as f:
    txt8 = f.read().decode('utf-8')
```

Database Access with Relational and Non-Relational Databases

- ✓ **Databases** are used to store, manage, and retrieve data efficiently.
- ✓ Database access refers to the process of **connecting to, querying, inserting, updating, and deleting data** stored in a database using software applications or programming languages.
  - Applications interact with databases through query languages or APIs
- ✓ Two major database categories:
  - Relational databases
    - Data is stored in **tables** (rows and columns)
    - Enforce a **fixed schema** and data integrity constraints
  - Non-relational (NoSQL) databases
    - Data is stored in non-tabular formats.
    - Schema is flexible or schema-less

Relational

student			
id	**name**	**surname**	**age**
1	John	Brown	19
2	Emma	Carly	23

Non-Relational

student.json file body:

```
[
 {
  "id": 1,
  "name": "John",
  "surname": "Brown",
  "age": 19
 },
 {
  "id": 2,
  "name": "Emma",
  "surname": "Carly",
  "age": 23
 }
]
```

Relational Database

- ✓ A **relational database** stores data in **tables (relations)** consisting of rows and columns, where each table represents an entity and relationships are defined between tables.
  - Data follows a predefined schema
  - Relationships are maintained using keys
  - Data is accessed using SQL (Structured Query Language)
- ✓ **Accessing Relational Databases**
  - Applications access relational databases using:
    - SQL queries (SELECT, INSERT, UPDATE, DELETE)
    - Database drivers and connectors
    - Object Relational Mapping (ORM) tools
- ✓ **Common Systems :**MySQL, PostgreSQL, Oracle, SQL Server.

Non-Relational Database

- ✓ A **NoSQL database** is a non-relational database designed to handle **large volumes of structured, semi-structured, or unstructured data** without requiring a fixed schema.
  - Schema-less or flexible schema
  - Optimized for distributed systems
  - Designed for scalability and availability
- ✓ **Accessing NoSQL Databases**
  - Applications access NoSQL databases using:
    - Database-specific APIs
    - Query languages or SDKs
    - REST-based interfaces
- ✓ **Common Systems:** MongoDB, Redis, Cassandra, Neo4j

Non-Relational Database

✓ Popular Types:

- **Document databases**: MongoDB, CouchDB → store JSON-like documents
- **Key-Value stores**: Redis, DynamoDB → simple key → value storage
- **Column-family stores**: Cassandra → optimized for large datasets
- **Graph databases**: Neo4j → store nodes and relationships

```
# 1. Document database (MongoDB): JSON-like documents
# Example: storing user profile with nested data
doc_db = {
    "user_id": 101,
    "name": "Ram",
    "age": 25,
    "skills": ["Python", "AI"],
    "address": {"city": "Kathmandu", "country": "Nepal"}
}
```

Interacting with Relational Database

- ✓ A standard way to represent rows of data in Python is as a sequence/list of tuples. For example:

```
stocks = [
    ('GOOG', 100, 490.1),
    ('AAPL', 50, 545.75),
    ('FB', 150, 7.45),
    ('HPQ', 75, 33.2),
]
```

  - Operations are performed using SQL statements
  - Input and output rows are handled as tuples
  - Same interface works for: SQLite, MySQL ,PostgreSQL, ODBC-based databases

- ✓ **Establish Connection**

```
>>> import sqlite3
>>> db = sqlite3.connect('database.db')
>>>
```

  - **connect()** opens (or creates) a database file
  - Returns a **connection object**
  - Connection manages transactions and persistence

Interacting with Relational Database (Contd..)

✓ Create Cursor and Table

- Cursor is used to **execute SQL commands**
- SQL defines the **schema**
- **commit()** makes changes permanent

```
>>> c = db.cursor()
>>> c.execute('create table portfolio (symbol text, shares integer, price real)')
<sqlite3.Cursor object at 0x10067a730>
>>> db.commit()
>>>
```

✓ Inserting Data into the Database

- **executemany()** inserts multiple rows efficiently
- **?** acts as a **placeholder** for values
- Prevents unsafe SQL string construction

```
>>> c.executemany('insert into portfolio values (?,?,?)', stocks)
<sqlite3.Cursor object at 0x10067a730>
>>> db.commit()
>>>
```

Interacting with Relational Database (Contd..)

✓ Retrieving Data from the Database

- **Querying the Table**
  - Each result row is returned as a tuple
  - Query results can be processed using standard Python loops

```
>>> for row in db.execute('select * from portfolio'):
...     print(row)
...
('GOOG', 100, 490.1)
('AAPL', 50, 545.75)
('FB', 150, 7.45)
('HPQ', 75, 33.2)
>>>
```

✓ Parameterized Queries

- **Query with User Input**
  - Always use parameterized queries when handling external or user-provided data.
  - Never construct SQL statements using string formatting or concatenation.
  - The **?** placeholder safely inserts user input via a tuple, preventing SQL injection.

```
>>> min_price = 100
>>> for row in db.execute('select * from portfolio where price >= ?',
                          (min_price,)):
...     print(row)
...
('GOOG', 100, 490.1)
('AAPL', 50, 545.75)
>>>
```

Interacting with Non-Relational Database

✓ Example: MongoDB (Document Database)

- **users.insert_one()** → adds a document (like a row in SQL).
- **users.find()** → queries using dictionary-style filters, not SQL.
- Flexible schema allows storing varying data per document.

```
doc = col.find_one({"name": "Ram"})

for doc in col.find().limit(2):
    print(doc)
```

```python
from pymongo import MongoClient

# Connect to local MongoDB server
client = MongoClient("mongodb://localhost:27017/")

# Create / select database
db = client["example_db"]

# Create / select collection (like table in relational DB)
col = db["users"]

# Insert documents
col.insert_one({"name": "Ram", "age": 25})
col.insert_many([{"name": "Sita", "age": 22}, {"name": "Hari", "age": 30}])
```

```python
# Query documents
docs = col.find({"age": {"$gt": 20}})
for doc in docs:
    print(doc)

# Update a document
col.update_one({"name": "Ram"}, {"$set": {"age": 26}})

# Delete a document
col.delete_one({"name": "Hari"})

# Close connection
client.close()
```

```
docs = col.find()   # no filter → all records
Same as: SELECT * FROM users;
find() returns a cursor, not a list (iterator)
```

Interacting with Non-Relational Database (Contd..)

✓ Example: Redis (Key-Value Store)

- Ideal for caching, session storage, or fast lookups.
- Stores simple key-value pairs efficiently.

```
import redis


r = redis.Redis(host='localhost', port=6379, db=0)
r.set('username', 'bob')      # Set a key-value pair
print(r.get('username'))       # Retrieve value
```

Accessing and processing data from APIs

- ✓ **Definition:**
  - An **API** is a set of **rules, protocols, and tools** that allows one software application to **communicate with another**. It defines how requests are made, how data is sent and received, and how responses are formatted.
  - Think of an API as a **messenger** between applications.
- ✓ Two popular approached:
  - **REST (Representational State Transfer)**
  - **SOAP (Simple Object Access Protocol)**
- ✓ Key Concepts:
  - **Endpoints:**
    - URL paths where the API listens for requests.
    - Example: https://api.example.com/users

Accessing and processing data from APIs

✓ Key Concepts (Cont..):

- **Request Methods (HTTP verbs):**
  - GET → Retrieve data
  - POST → Send or create data
  - PUT / PATCH → Update data
  - DELETE → Remove data

```python
import requests

# API endpoint
url = "https://api.example.com/users"

# GET request to fetch data
response = requests.get(url, params={"id":101})

# Parse JSON response
data = response.json()
print(data)
```

- **Request Parameters:**
  - Data sent to the API to filter, modify, or specify the response.
  - Can be in **URL query** (?id=101) or **request body** (JSON for POST/PUT).
- **Response:**
  - Data returned by the API, usually in **JSON** or **XML** format.
  - Includes **status codes** (e.g., 200 = success, 404 = not found).

Accessing and processing data from APIs (Contd..)

- ✓ **REST APIs (Representational State Transfer)**
  - REST is an **architectural style** that uses standard **HTTP methods** like **GET, POST, PUT, DELETE.**
  - It typically exchanges **lightweight JSON data** but can support XML, text, or other formats.
  - Widely used for modern web and mobile applications.
  - **Stateless:** Each request contains all information needed; server does not store session state.
- ✓ Using Python's **requests** module to access and process JSON data from a REST API.

```python
import requests

API_URL = "https://newsapi.org/v2/top-headlines"
API_KEY = "your_api_key_here"

params = {
    "country": "us",
    "apiKey": API_KEY,
    "pageSize": 5
}

response = requests.get(API_URL, params=params)
data = response.json()
print(data)
```

Accessing and processing data from APIs (Contd..)

- ✓ **SOAP APIs (Simple Object Access Protocol)**
  - SOAP is a **protocol** that strictly uses **XML envelopes** for requests and responses.
  - SOAP APIs often define their interface via **WSDL (Web Services Description Language)**.
  - Suitable for enterprise, legacy systems with *formal standards and strict contracts*.
- ✓ Using the **Zeep library** to interact with a SOAP API — SOAP communicates via **structured XML messages**.

```python
from zeep import Client


# Create a SOAP client with the WSDL URL
client = Client("http://www.example.com/service?wsdl")


# Call a SOAP method
result = client.service.MethodName(arg1="value1", arg2="value2")
print(result)
```

Web scraping using Requests and BeautifulSoup

✓ What Is Web Scraping?

- **Web scraping** is the automated process of extracting **publicly available data** from websites
- The data collected does **not require authentication** (no login or account)
- It replaces manual copy-paste with **programmatic data collection**
- Scraped data can be exported into formats such as **CSV, JSON, or databases**
- Commonly implemented using **programming languages like Python**

✓ Why Web Scraping Is Important

- Enables **rapid data collection** from multiple web sources
- Essential skill for **data scientists and engineers**
- Helps in building **large, real-world datasets**
- Supports **data-driven decision making**
- Saves time and reduces human error compared to manual methods.

✓ Ethical and Legal Considerations

- Web scraping can raise **ethical and legal concerns**
- Public availability does **not always imply permission**
- Scrapers must respect:
  - Website policies
  - Copyright laws
  - Server resources

✓ Best Practices in Web Scraping

- **Check robots.txt**
  - Indicates whether scraping is allowed
  - Example: www.example.com/robots.txt
- **Respect Terms of Use**
  - Especially for commercial usage
- **Set Rate Limits**
  - Avoid sending too many requests
  - Prevent server overload
- **Avoid Login-Protected Content**
  - Contact website owners if access is required
- **Do Not Scrape Copyrighted Creative Content**

Web scraping using Requests and BeautifulSoup (Contd..)

✓ Python Libraries for Web Scraping

- **Requests**
  - Sends HTTP requests (GET, POST, etc.)
  - Fetches webpage content
- **BeautifulSoup**
  - Parses HTML/XML
  - Extracts and navigates webpage elements

✓ Why BeautifulSoup?

- Beginner-friendly and Pythonic
- Simplifies:
  - Searching HTML tags
  - Navigating DOM structure
  - Extracting text and attributes
- Returns clean textual data. For large projects → use **Scrapy or Selenium**

✓ Importing Required Libraries and sending HTTP Request

- The **requests** library is used to send an HTTP GET request to the target website and retrieve its HTML content. The response status code confirms whether the request was successful.

```
from bs4 import BeautifulSoup
import requests
import csv

url = "https://quotes.toscrape.com/"
response = requests.get(url)

if response.status_code == 200:
    print("Response successful")
else:
    print("Request failed")
```

✓ Parsing HTML with BeautifulSoup

- BeautifulSoup parses the raw HTML content and converts it into a structured format, making it easy to navigate, search, and extract data from web pages.

```
soup = BeautifulSoup(response.text, "html.parser")
print(soup)
```

Web scraping using Requests and BeautifulSoup (Contd..)

✓ Extracting Required Data

- The **find_all()** method is used to locate specific HTML elements based on tags and attributes, allowing targeted data extraction.

```
quotes = soup.find_all('span', class_='text')
authors = soup.find_all('small', class_='author')
```

✓ Processing Extracted Data

- Extracted elements are processed using a loop and stored in a structured list for easy handling and further analysis.

```
data = []

for quote, author in zip(quotes, authors):
    data.append([quote.text.strip(), author.text.strip()])

print(data)
```

✓ Saving Data to CSV

- The scraped data is saved into a CSV file, making it portable and easy to use for analysis or reporting.

```
with open('Quotes_scraped.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['Quote', 'Author'])
    writer.writerows(data)
```

✓ Error Handling in Web Scraping - Example

- Defines a function **getTitle()** to safely extract the <h1> title from a webpage
- Uses **try–except** to handle HTTP errors when the webpage cannot be accessed
- Parses HTML using BeautifulSoup only if the page is successfully retrieved
- Catches **AttributeError** when expected HTML elements (like **<h1>**) are missing
- Returns **None** if any error occurs, ensuring the program does not crash
- Demonstrates robust and fault-tolerant web scraping practices

```
from urllib.request import urlopen
from urllib.error import HTTPError
from bs4 import BeautifulSoup

def getTitle(url):
    try:
        html = urlopen(url)
    except HTTPError as e:
        return None
    try:
        bs = BeautifulSoup(html.read(), 'html.parser')
        title = bs.body.h1
    except AttributeError as e:
        return None
    return title

title = getTitle('http://www.pythonscraping.com/pages/page1.html')
if title == None:
    print('Title could not be found')
else:
    print(title)
```

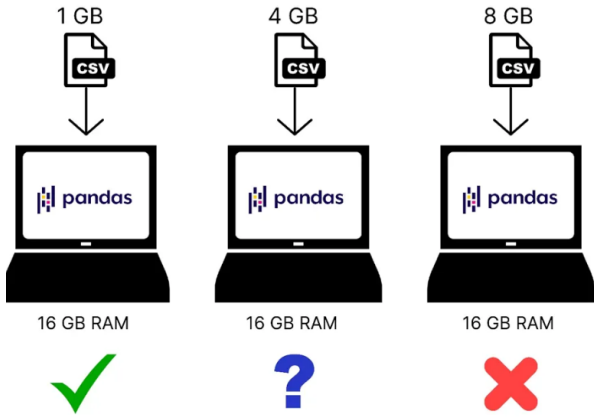
Handling large datasets with chunking and lazy evaluation

✓ Problem with Large Datasets

- Modern datasets often exceed available system memory (RAM).
- Loading entire datasets at once leads to slow performance or crashes.
- Traditional tools like Pandas operate *in-memory* and are limited by RAM.

✓ Why Pandas Struggles with Large Files

- Pandas requires **more memory than raw file size** due to internal data structures.
- Intermediate operations (filtering, grouping) consume additional RAM.
- Pandas primarily uses **a single CPU core**, limiting performance.



Concept of Chunking

- **Chunking** means processing data in smaller, fixed-size portions.
- Only a part of the dataset is loaded into memory at a time.
- Ideal for aggregation, filtering, and sequential processing.

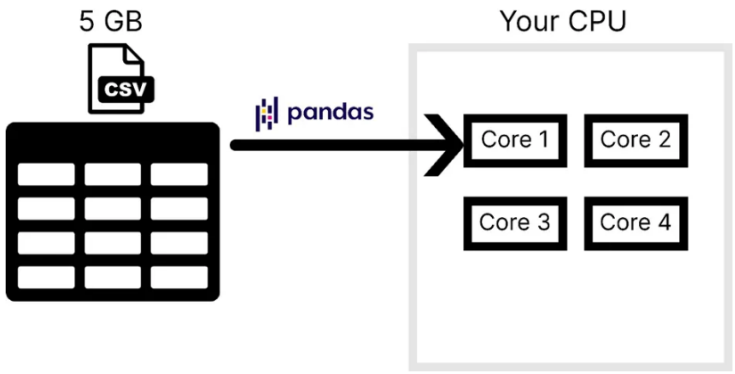
```
import pandas as pd

total_revenue = 0
for chunk in pd.read_csv("large_sales.csv", chunksize=100000):
    total_revenue += chunk["revenue"].sum()

print(total_revenue)
```

- Data is read and processed incrementally.
- Memory usage remains constant regardless of file size.

Handling large datasets with chunking and lazy evaluation (Contd..)

✓ CPU Limitation in Pandas

- Pandas executes operations using one CPU core.
- Other cores remain idle during computation.
- Results in slow execution for large datasets.



✓ This is where parallel and lazy systems become useful.

Handling large datasets with chunking and lazy evaluation (Contd..)

✓ What is Lazy Evaluation?

- **Lazy Evaluation** is a core principle of generators where data is produced **only when requested**, rather than computed upfront.

  Computation is deferred until needed, which **reduces memory usage** and **improves execution efficiency** by avoiding unnecessary work.

✓ Lazy Evaluation with Generators

```
def read_large_file(file_path):
    with open(file_path) as f:
        for line in f:
            yield line

for line in read_large_file("large_log.txt"):
    print(line)
```

- Data is generated one line at a time.
- Entire file is never loaded into memory.

Handling large datasets with chunking and lazy evaluation (Contd..)

✓ Lazy Evaluation in Data Pipelines

- Each stage processes **one element at a time**.
- Memory-efficient and composable.

```
def numbers():
    for i in range(10):
        yield i

def squares(nums):
    for n in nums:
        yield n * n

pipeline = squares(numbers())
```

✓ Dask – Chunking + Lazy Evaluation

- Dask automatically chunks data.
- Uses lazy execution and multiple CPU cores

```
import dask.dataframe as dd

df = dd.read_csv("huge_dataset.csv")
result = df["sales"].mean()
final = result.compute()
```

When to Use What?

- **Chunking (Pandas):** Simple aggregations, limited resources
- **Generators:** File streaming, pipelines
- **Dask:** Very large datasets, parallel execution