DEPARTMENT OF BIOMEDICAL ENGINEERING

24AI671- FUNDAMENTALS OF DATA SCIENCE

UNIT II

DATA COLLECTION, CLEANING AND PREPROCESSING

1. DATA ACQUISITION METHODS

Data acquisition (also called **data collection**) is the process of **gathering, measuring and recording data from one or more sources** in a systematic way so that it can be stored, cleaned and analysed. It is the **first step of every data science and machine learning project**. The quality of the final model or report depends directly on the quality of the data that was collected – the well-known rule is **"Garbage In, Garbage Out (GIGO)"**.

In a real project, raw data is almost never ready to use. It may come from many places (sensors, websites, forms, databases), in many formats (numbers, text, images) and with many errors (missing values, duplicates, wrong entries). The syllabus topics of this unit follow the order in which a data scientist actually works: **acquire → store → clean → engineer features → transform and scale**.

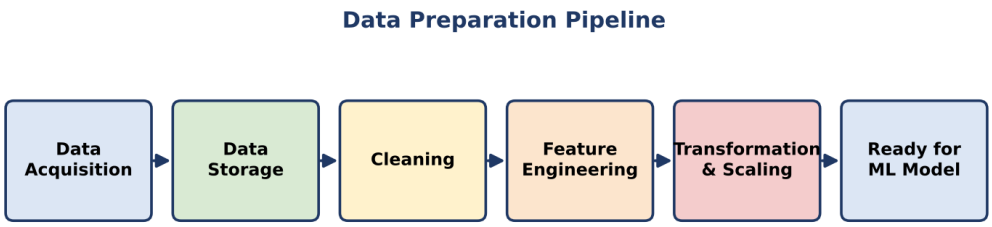


Fig 1.1 Data preparation pipeline covered in this unit

1.1 Why Data Acquisition Matters

- **Foundation of analysis** – no model can be better than the data used to train it.

- **Decides the type of problem** – the kind of data collected (labelled / unlabelled, numeric / text) decides whether supervised or unsupervised learning is possible.
- **Saves cost later** – collecting the right data the first time avoids expensive re-collection and re-cleaning.
- **Supports decision making** – business, engineering and scientific decisions are based on evidence gathered through data.

Understanding Data Acquisition in Simple Words

Think of a data scientist as a **cook** and the data as the **ingredients**. Before cooking, the cook decides what dish is to be made, buys the right ingredients from a reliable shop and checks that they are fresh. In the same way, before analysis we must decide **what question has to be answered**, find **where the needed data lives**, collect it in a suitable way and check that it is **good enough**. If the ingredients are bad, even the best recipe (algorithm) cannot give a good dish (result).

A well-planned data collection follows these steps:

1. **Define the objective** – what problem is to be solved? (e.g. 'predict which students may fail').
2. **Identify the required data** – which attributes (attendance, internal marks, study hours) are needed.
3. **Select the sources and method** – survey, existing database, API, sensors, etc.
4. **Design the collection instrument** – questionnaire, scraping script, sensor setup.
5. **Collect the data** – with proper consent, in the agreed format and sampling plan.
6. **Verify and record metadata** – check quality and write down the source, date, units and meaning of each column (a **data dictionary**).

The **data life cycle** therefore moves through *generation* → *collection* → *storage* → *processing* → *analysis* → *sharing / archiving*. This unit concentrates on the first half of the life cycle, which prepares the data so that the analysis in the later units becomes reliable.

1.2 Characteristics of Good Data (Data Quality Dimensions)

While collecting data, we must check whether it satisfies the following quality dimensions. Poor performance on any one of them leads to cleaning work in the later steps of this unit.

Dimension	Meaning	Example of a problem
Accuracy	Data correctly represents the real-world value	Age entered as 210 instead of 21
Completeness	All required values are present	Phone number missing for 30% of customers
Consistency	Same fact is stored in the same way everywhere	'TN', 'Tamil Nadu' and 'T.N.' used for one state
Timeliness	Data is up to date when it is needed	Using 2015 prices to predict 2026 sales
Validity	Values follow the allowed format and range	Date of birth written as 31-02-2000
Uniqueness	Each real-world entity is recorded only once	Same student appears twice in the list

1.3 Types of Data

Before choosing a collection method, we need to know what kind of data is expected. Data can be classified in two common ways – by **structure** and by **nature of the values**.

(a) Based on structure

Type	Description	Examples
Structured	Organised in rows and columns with a fixed schema; easy to store in tables	SQL tables, CSV files, Excel sheets
Semi-structured	Has tags or keys but no rigid table structure	JSON, XML, HTML, log files
Unstructured	No predefined format; needs special processing	Images, audio, video, e-mails, free text

(b) Based on nature of values

Category	Sub-type	Meaning / Example
Qualitative (categorical)	Nominal	Labels with no order – blood group, city, colour
Qualitative (categorical)	Ordinal	Labels with a meaningful order – Low < Medium < High, grade A/B/C
Quantitative (numerical)	Discrete	Countable whole numbers – number of defects, students in a class
Quantitative (numerical)	Continuous	Any value in a range – temperature 36.7 °C, height 172.4 cm

1.4 Sources and Methods of Data Acquisition

Data can be obtained in many ways. The main methods are shown in Fig 1.2 and are explained one by one below.

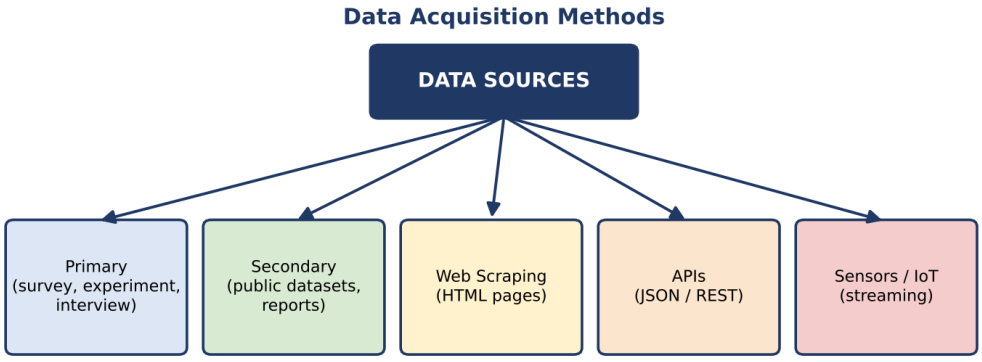


Fig 1.2 Common data acquisition methods

1.4.1 Primary Data Collection

Primary data is collected **first-hand by the researcher** for a specific purpose. It is original, up to date and specific to the problem, but it needs more time, money and effort.

- **Surveys and questionnaires** – a fixed set of questions (online forms, paper forms) given to a sample of people. Good for opinions, habits and feedback. Tools: Google Forms, SurveyMonkey.
- **Interviews** – direct one-to-one questioning (structured, semi-structured or unstructured). Gives deep information but is time consuming.

- **Observation** – watching and recording behaviour or events directly, e.g. counting vehicles at a junction.
- **Experiments** – controlled tests in which one variable is changed and the effect is measured, e.g. A/B testing of two web-page designs.
- **Focus groups** – a small group is asked to discuss a product or topic; useful for qualitative insights.

Designing a good questionnaire. The quality of survey data depends on the quality of the questions. **Closed questions** (Yes/No, multiple choice, rating scale 1–5) give data that is easy to analyse, while **open questions** ('Suggest improvements') give rich but unstructured text. Questions must be short, clear and **neutral** – a leading question such as 'Don't you agree that our service is excellent?' pushes the respondent towards one answer and introduces **bias**. It is good practice to run a small **pilot survey** with a few people before the full survey to find confusing questions.

1.4.2 Secondary Data Collection

Secondary data was **already collected by someone else** for another purpose and is re-used. It is cheap and quick, but it may not exactly match the new problem and its quality must be verified.

- **Public datasets and repositories** – UCI Machine Learning Repository, Kaggle, Open Government Data Platform India (data.gov.in), World Bank Open Data.
- **Published sources** – research papers, books, government reports, census, annual reports of companies.
- **Internal records** – sales records, transaction logs, customer databases and ERP data of an organisation.

Point	Primary Data	Secondary Data
Source	Collected first-hand by the researcher	Collected earlier by someone else
Cost and time	High cost, more time	Low cost, quick to obtain
Relevance	Exactly matches the problem	May be only partly relevant
Control over quality	Researcher controls the process	Quality must be verified by the user
Examples	Own survey, lab experiment	Census data, Kaggle datasets

1.4.3 Web Scraping

Web scraping is the automatic extraction of data from the HTML pages of websites using a program. It is used when the data is visible on a web page but no download or API is offered – for example, collecting product prices, news headlines or job postings.

Basic steps: (1) send an HTTP request to the page URL, (2) receive the HTML response, (3) parse the HTML and locate the required tags, (4) extract the text or attributes, and (5) save them in a structured form such as a CSV file.

Code

```
import requests
from bs4 import BeautifulSoup
import pandas as pd

url = "https://example.com/products"
resp = requests.get(url, headers={"User-Agent": "student-project"},
timeout=10)

if resp.status_code == 200:                    # 200 = request successful
    soup = BeautifulSoup(resp.text, "html.parser")
    rows = []
    for item in soup.find_all("div", class_="product"):
        name  = item.find("h2").get_text(strip=True)
        price = item.find("span", class_="price").get_text(strip=True)
        rows.append({"name": name, "price": price})
    df = pd.DataFrame(rows)
    df.to_csv("products.csv", index=False)       # store the scraped data
```

How the scraping program works. A web page is written in **HTML**, which is made of nested tags such as , <h2> and , and each tag may have attributes like class. The program first downloads the HTML text with requests.get(). The status code **200** means success. BeautifulSoup then converts the text into a tree so that find_all("div", class_="product") can pick every product block, and get_text() extracts the visible text inside the tags. Finally the list of dictionaries is converted into a DataFrame and saved as a CSV file. If the website uses JavaScript to load its content, plain Requests cannot see it and tools such as **Selenium** or **Playwright** (which control a real browser) are used.

1.4.4 APIs (Application Programming Interfaces)

An **API** lets one program request data from another program in a clean, structured way, usually in **JSON** format. Most modern services (weather, maps, social media, payment, stock prices) expose **REST APIs** that are accessed through URLs and HTTP methods (GET, POST, PUT, DELETE). APIs are more reliable than scraping because the data format is fixed and the provider allows the access, often through an **API key**.

Code

```
import requests, pandas as pd

url = "https://api.open-meteo.com/v1/forecast"
params = {"latitude": 12.92, "longitude": 80.12,          # Tambaram
          "hourly": "temperature_2m", "forecast_days": 1}

data = requests.get(url, params=params, timeout=10).json()   # JSON -> dict
df = pd.DataFrame({"time": data["hourly"]["time"],
                   "temp_C": data["hourly"]["temperature_2m"]})
print(df.head())
```

Understanding API responses. When a program calls an API, the server replies with a **status code** and usually a **JSON** body. JSON stores data as key–value pairs and lists, which map directly to Python dictionaries and lists:

```
{
  "city": "Chennai",
  "temperature": 31.5,
  "humidity": 74,
  "forecast": ["sunny", "cloudy", "rain"]
}
```

Status code	Meaning	What to do
200	OK – request successful	Read the data
400	Bad request – wrong parameters	Check the query parameters

Status code	Meaning	What to do
401 / 403	Unauthorised / forbidden – key missing or not allowed	Check the API key and permissions
404	Resource not found	Check the URL
429	Too many requests – rate limit crossed	Wait and slow down the requests
500	Server error	Retry later

1.4.5 Sensors and IoT Devices

In engineering and industrial applications, data is generated continuously by **sensors** – temperature, pressure, vibration, current, GPS, cameras. The data is sent through microcontrollers (Arduino, Raspberry Pi, ESP32) over protocols such as **MQTT** or HTTP to a server or cloud. Important design points are the **sampling rate** (how many readings per second), **calibration** of the sensor, and handling of **noise and dropped readings**, which lead directly to the missing-data and noise-reduction techniques discussed later in this unit.

Sampling rate explained. If a machine vibrates 50 times per second (50 Hz), the sensor must read it **at least twice as fast** – 100 readings per second – to capture the vibration correctly (the **Nyquist rule**). Reading too slowly misses the real pattern, and reading too fast fills the storage with repeated values. Sensors also suffer from **drift** (slow change in the reading with time) and need regular **calibration** against a known standard.

1.4.6 Other Sources

- **Databases and data warehouses** – extracting records using SQL queries.
- **Log files and transaction records** – web-server logs, click-stream data, bank transactions.
- **Social media and text data** – tweets, reviews and comments collected through platform APIs for sentiment analysis.
- **Crowdsourcing** – many people contribute or label data online, e.g. Amazon Mechanical Turk, OpenStreetMap.
- **Files and spreadsheets** – CSV, Excel, JSON and XML files shared by other teams.

1.5 Comparison of Acquisition Methods

Method	Data type	Speed	Cost	Typical use
Survey / questionnaire	Structured	Medium	Low–Medium	Opinions, feedback, market research
Experiment	Structured	Slow	High	Testing cause and effect, A/B tests
Public dataset	Mostly structured	Very fast	Free	Learning, benchmarking, research
Web scraping	Semi / unstructured	Fast	Low	Prices, news, reviews
API	Semi-structured (JSON)	Fast	Free / paid	Weather, finance, maps, social media
Sensors / IoT	Time-series	Real time	Medium–High	Machine monitoring, smart grid

1.6 Sampling Methods

It is usually impossible to collect data from the **entire population**, so a smaller group called a **sample** is selected. A good sample must be **representative** of the population, otherwise the results will be biased.

Category	Method	How the sample is chosen
Probability sampling	Simple random	Every member has an equal chance (lottery method, random numbers)
Probability sampling	Systematic	Every k-th member after a random start (every 10th record)
Probability sampling	Stratified	Population divided into groups (strata) and random samples taken from each group
Probability sampling	Cluster	Whole groups (clusters) such as colleges or villages are selected at random

Category	Method	How the sample is chosen
Non-probability sampling	Convenience	Whoever is easily available; quick but biased
Non-probability sampling	Purposive / judgement	Researcher selects members using expert judgement
Non-probability sampling	Quota	Fixed number from each category is filled in a non-random way
Non-probability sampling	Snowball	Existing participants recommend further participants

Example of stratified sampling. A college has 1,000 students – 600 boys and 400 girls – and a sample of 100 is needed. With simple random sampling the sample might accidentally contain 80 boys and only 20 girls. In stratified sampling the sample copies the population proportions: boys = 100 × 600/1000 = **60** and girls = 100 × 400/1000 = **40**. Each group is then picked randomly, so every group is fairly represented.

1.7 Challenges and Ethical Issues in Data Collection

- **Data quality problems** – errors, duplicates, missing and inconsistent entries.
- **Bias** – an unrepresentative sample or leading questions give a model that is unfair to some groups.
- **Privacy and consent** – personal data (name, phone, health, location) must be collected only with consent and protected. In India the **Digital Personal Data Protection Act, 2023** governs this; in Europe it is the **GDPR**.
- **Volume, velocity and variety** – big and fast-changing data needs proper storage and tools.
- **Legal and licence issues** – a dataset may be restricted to research use or may need attribution.
- **Anonymisation** – remove or mask names, IDs and other identifiers before sharing data.

1.8 Loading Collected Data into Python

After acquisition, the data is loaded into a Pandas **DataFrame** for cleaning. Pandas can read many formats with a single function call.

Code

```
import pandas as pd, sqlite3

df1 = pd.read_csv("students.csv")                  # CSV file
df2 = pd.read_excel("marks.xlsx", sheet_name="Sem1") # Excel sheet
df3 = pd.read_json("data.json")                    # JSON file
df4 = pd.read_html("https://example.com/table")[0]  # first HTML table on a
page

con = sqlite3.connect("college.db")
df5 = pd.read_sql("SELECT * FROM students", con)     # SQL query result
```

2. DATA STORAGE: CSV, SQL, NoSQL

After data is acquired, it must be **stored safely and in a form that can be retrieved easily** for analysis. The choice of storage depends on the **structure of the data, its size, how often it changes, how many users access it, and the type of queries required**. The three storage options in this unit are **flat files (CSV), relational databases (SQL)** and **non-relational databases (NoSQL)**.

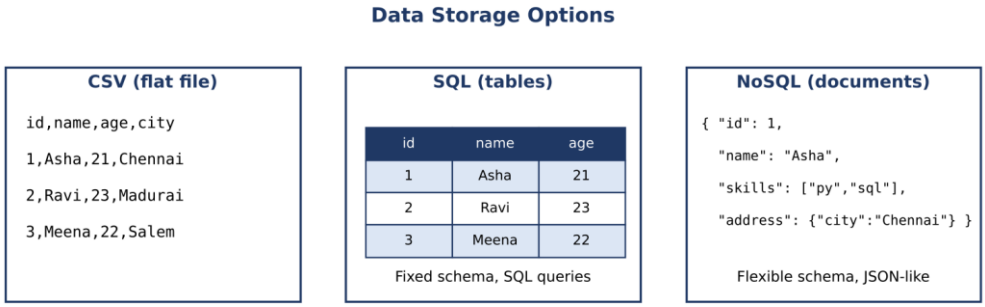


Fig 2.1 The same student record stored in CSV, SQL and NoSQL forms

Why Do We Need Data Storage?

Variables in a Python program live in **RAM**, which is fast but **temporary** – everything is lost when the program ends or the computer is switched off. Data that must be kept for later use has to be written to **permanent (persistent) storage** such as a hard disk, SSD or cloud server. The simplest way is to save a **file**; but as the data becomes larger and more people need to read and change it at the same time, files become difficult to manage – duplicate copies appear, two people overwrite each other's changes, and searching a huge file takes a

long time. A **database** solves these problems by providing a **structured, shared, secure and searchable** store with rules that protect the data.

In short: **files** are enough for a single person working with a small dataset, **relational databases** are chosen for structured data with strict rules, and **NoSQL databases** are chosen when data is huge, fast changing or does not fit neatly into tables.

2.1 CSV (Comma-Separated Values) Files

A **CSV file** is a simple **plain-text file in which each line is one record and the values in a line are separated by a comma** (or another delimiter such as tab, semicolon or pipe). The first line normally contains the column names (header). CSV is the most common format for exchanging small and medium datasets because every tool – Excel, Python, R, SQL databases – can read and write it.

```
roll,name,dept,marks
101,Asha,AI&DS,88
102,Ravi,CSE,76
103,Meena,AI&DS,
104,"Kumar, S",ECE,91
```

Observe that record 103 has an **empty value** (missing data) and record 104 has a comma inside the name, so the value is enclosed in **double quotes**.

How a CSV file is read. The program reads the file line by line. The first line is taken as the **header**, and every following line is split at the delimiter (comma) into values that become the fields of one record. Because the file stores only text, Pandas **guesses the data type** of each column: a column containing only digits becomes a number, and a column with any text stays as text (object). If a numeric column contains a stray word like 'NA', the whole column may be read as text – this is why the `na_values` option is important. Dates are also read as plain text unless `parse_dates=["date"]` is given.

Advantages of CSV

- Simple, human readable and can be opened in any text editor or spreadsheet.
- Small file size compared to Excel or XML; very fast to read and write.
- Supported by practically all software; ideal for sharing datasets (Kaggle, UCI).

Limitations of CSV

- No data types – everything is text; the program must guess whether a column is number, date or string.
- No support for relationships between tables, indexing, security or multiple users.
- Not suitable for very large data (the whole file usually has to be scanned) or for nested data.
- Problems with delimiters, line breaks and character encoding (UTF-8 vs ANSI) inside values.

Reading and writing CSV in Python

Code

```
import csv, pandas as pd

# 1. Using the built-in csv module
with open("students.csv", newline="", encoding="utf-8") as f:
    for row in csv.DictReader(f):
        print(row["name"], row["marks"])

# 2. Using Pandas (preferred for data analysis)
df = pd.read_csv("students.csv",
                 sep=",",                # delimiter
                 header=0,                 # first row has column names
                 usecols=["roll", "name", "marks"],
                 na_values=["", "NA", "?"],# treat these strings as NaN
                 encoding="utf-8")

df.to_csv("clean_students.csv", index=False)  # write back, without row index
```

2.2 SQL – Relational Databases

A **relational database management system (RDBMS)** stores data in **tables (relations)** made of **rows (records / tuples)** and **columns (attributes / fields)**. Tables are linked through keys. The language used to define, insert, retrieve and control data is **SQL – Structured Query Language**. Popular RDBMS are **MySQL, PostgreSQL, Oracle, Microsoft SQL Server and SQLite**.

Important terms

Term	Meaning
Schema	The fixed design of the database – table names, column names, data types and constraints
Primary key	Column (or set of columns) that uniquely identifies each row; cannot be NULL or repeated
Foreign key	Column that refers to the primary key of another table and creates the relationship
Index	Special structure that makes searching on a column faster
Normalisation	Dividing data into related tables to remove redundancy and update problems (1NF, 2NF, 3NF)
Transaction	A group of SQL operations treated as a single unit – either all succeed or none

Categories of SQL commands

Category	Full form	Commands	Purpose
DDL	Data Definition Language	CREATE, ALTER, DROP, TRUNCATE	Define or change the structure of tables
DML	Data Manipulation Language	INSERT, UPDATE, DELETE	Change the data inside tables
DQL	Data Query Language	SELECT	Retrieve data
DCL	Data Control Language	GRANT, REVOKE	Give or remove access rights
TCL	Transaction Control Language	COMMIT, ROLLBACK, SAVEPOINT	Manage transactions

Common SQL operations with examples

Code

```
-- DDL : create two related tables
CREATE TABLE department (
    dept_id   INTEGER PRIMARY KEY,
    dept_name VARCHAR(30) NOT NULL
);
CREATE TABLE student (
    roll     INTEGER PRIMARY KEY,
    name     VARCHAR(40) NOT NULL,
    marks    INTEGER CHECK (marks BETWEEN 0 AND 100),
    dept_id  INTEGER REFERENCES department(dept_id)   -- foreign key
);

-- DML : insert, update, delete
INSERT INTO department VALUES (1, 'AI&DS'), (2, 'CSE');
INSERT INTO student VALUES (101, 'Asha', 88, 1), (102, 'Ravi', 76, 2);
UPDATE student SET marks = 80 WHERE roll = 102;
DELETE FROM student WHERE marks < 35;

-- DQL : filtering, sorting, aggregation
SELECT name, marks FROM student WHERE marks >= 80 ORDER BY
marks DESC;
SELECT dept_id, AVG(marks) AS avg_marks, COUNT(*) AS total
FROM student GROUP BY dept_id HAVING AVG(marks) > 60;

-- JOIN : combine data from two tables
SELECT s.name, d.dept_name
FROM student s INNER JOIN department d ON s.dept_id = d.dept_id;
```

Normalisation – why data is split into several tables

Storing everything in one big table causes **redundancy** (the same information repeated) and **anomalies** when data is changed. Look at this single table:

roll	name	dept	HOD
101	Asha	AI&DS	Dr. Kumar

roll	name	dept	HOD
102	Ravi	CSE	Dr. Priya
103	Meena	AI&DS	Dr. Kumar

The Head of Department 'Dr. Kumar' is stored again for every AI&DS student. If the HOD changes, every row must be updated (**update anomaly**); if all AI&DS students are deleted, the information about the department is lost (**deletion anomaly**). **Normalisation** solves this by splitting the data into **student(roll, name, dept_id)** and **department(dept_id, dept_name, HOD)** linked by the key dept_id, so each fact is stored only once.

Types of JOIN

A **JOIN** combines rows from two tables using a related column. The same ideas are used in the Pandas merge() function later in this unit.

Join	Rows returned	Example result
INNER JOIN	Only rows with a match in both tables	Students who belong to a listed department
LEFT JOIN	All rows of the left table + matches from the right (NULL if none)	All students, even those without a department
RIGHT JOIN	All rows of the right table + matches from the left	All departments, even those without students
FULL OUTER JOIN	All rows of both tables	Everything, with NULL where no match exists

ACID properties

Relational databases guarantee reliable transactions through the **ACID** properties:

Property	Meaning	Example (bank transfer)
Atomicity	All operations of a transaction happen or none happen	Debit and credit both succeed, or both are cancelled

Property	Meaning	Example (bank transfer)
Consistency	Database moves from one valid state to another valid state	Total money in both accounts stays the same
Isolation	Concurrent transactions do not interfere with each other	Two transfers at the same time do not mix up balances
Durability	Once committed, data is permanent even after a crash	Transfer remains after power failure

Using SQL from Python

Code

```
import sqlite3, pandas as pd

con = sqlite3.connect("college.db")          # creates the file if absent
cur = con.cursor()
cur.execute("CREATE TABLE IF NOT EXISTS student(roll INT, name
TEXT, marks INT)")
cur.executemany("INSERT INTO student VALUES (?,?,?)",
                [(101, "Asha", 88), (102, "Ravi", 76), (103, "Meena", 91)])
con.commit()                                 # make changes permanent

df = pd.read_sql("SELECT * FROM student WHERE marks > 80", con)
print(df)
# DataFrame -> table
df.to_sql("toppers", con, if_exists="replace", index=False)
con.close()
```

Output

```
   roll   name  marks
0   101  Asha     88
1   103  Meena    91
```

Advantages and limitations of SQL databases

Advantages	Limitations

Advantages	Limitations
Well-defined structure and data integrity through constraints	Fixed schema – changing it later is difficult
Powerful, standard query language (SQL)	Hard to scale horizontally across many servers
ACID transactions – reliable for banking and ERP	Not suited to unstructured data such as images or free text
Multiple users, security and backup supported	Complex joins become slow for very large data

2.3 NoSQL Databases

NoSQL ("Not Only SQL") databases store data in **flexible, non-tabular formats**. They were developed to handle **big data, very high speed, huge numbers of users and changing or unstructured data** which relational databases find difficult. They usually **scale horizontally** – more machines are added to share the load – and have a **dynamic schema**, meaning each record may have different fields.

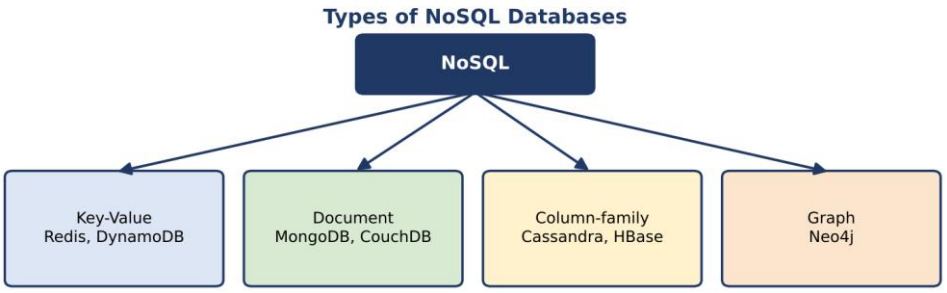


Fig 2.2 Four main types of NoSQL databases

Type	How data is stored	Examples	Typical use
Key-Value	Simple pairs: key → value (like a Python dictionary)	Redis, DynamoDB	Caching, sessions, shopping cart
Document	JSON / BSON documents, each can have its own fields	MongoDB, CouchDB	Catalogues, user profiles, content management

Type	How data is stored	Examples	Typical use
Column-family	Data grouped by columns rather than rows	Cassandra, HBase	Huge time-series, log and IoT data
Graph	Nodes (entities) and edges (relationships)	Neo4j, Amazon Neptune	Social networks, recommendations, fraud rings

Example – MongoDB (document database)

In MongoDB a **database** contains **collections**, and a collection contains **documents** (similar to a table containing rows). Note that the two documents below have different fields, which is allowed.

Code

```
from pymongo import MongoClient

client = MongoClient("mongodb://localhost:27017")
col = client["college"]["students"]              # database -> collection

col.insert_one({"roll": 101, "name": "Asha", "skills": ["Python", "SQL"],
                "address": {"city": "Chennai", "pin": 600045}})
col.insert_one({"roll": 102, "name": "Ravi", "marks": 76})   # different fields

for doc in col.find({"marks": {"$gt": 70}}, {"_id": 0}):      # query
    print(doc)

col.update_one({"roll": 102}, {"$set": {"marks": 80}})
col.delete_one({"roll": 101})
```

Scaling up vs scaling out

When a database becomes slow, there are two choices. **Vertical scaling (scale up)** means buying a bigger and more powerful server – it is simple but expensive and has an upper limit. **Horizontal scaling (scale out)** means adding more ordinary servers and sharing the data between them (**sharding**). Relational databases find it difficult to spread tables with many joins and strict transactions over many machines, whereas NoSQL databases are designed for horizontal

scaling from the beginning – this is the main reason for their popularity with large web companies.

Modelling data in a document database

In a relational database an order and its items would be stored in two tables and joined at query time. In a document database the items are usually **embedded inside the order document**, so a single read returns everything:

```
{ "order_id": 5001,
  "customer": "Asha",
  "items": [ {"product": "Pen",  "qty": 10, "price": 5},
             {"product": "Book", "qty": 2,  "price": 120} ],
  "total": 290 }
```

This makes reading fast and simple, but the same product details may be repeated in many orders, which is the trade-off of NoSQL (speed and flexibility against some duplication).

CAP theorem and BASE

The **CAP theorem** states that a distributed database can guarantee **only two of these three** properties at the same time: **Consistency** (all nodes show the same data), **Availability** (every request gets a response) and **Partition tolerance** (system works even if the network between nodes fails). Since network partitions cannot be avoided, NoSQL systems choose between consistency and availability. Many follow the **BASE** model – **B**asically **A**vailable, **S**oft state, **E**ventually consistent – in contrast to ACID.

Advantages and limitations of NoSQL

Advantages	Limitations
Flexible schema – easy to store changing or nested data	Weaker or no joins and standard query language
Scales horizontally; handles very large data volumes	Often only eventual consistency instead of full ACID
High read / write speed for simple queries	Less mature tools; skills vary between products
Good for JSON, logs, IoT, social and real-time data	Data may be duplicated, increasing storage and update effort

2.4 Comparison: SQL vs NoSQL

Feature	SQL (Relational)	NoSQL (Non-relational)
Data model	Tables with rows and columns	Key-value, document, column, graph
Schema	Fixed, defined in advance	Dynamic, flexible
Query language	SQL (standard)	Product specific (e.g. MongoDB query API)
Scaling	Vertical (bigger server)	Horizontal (more servers)
Transactions	ACID	Usually BASE (eventual consistency)
Relationships	Strong – joins and foreign keys	Weak – usually embedded or application-managed
Best for	Structured data, banking, ERP	Big data, real-time, unstructured data
Examples	MySQL, PostgreSQL, Oracle	MongoDB, Redis, Cassandra, Neo4j

2.5 Choosing the Right Storage

Situation	Best choice	Reason
Small dataset to share or train a model quickly	CSV	Simple, portable, no server needed
Structured data, many users, strict accuracy (student marks, bank)	SQL	Integrity, ACID, powerful queries
Fast-changing JSON data such as product catalogue or app logs	NoSQL – document	Flexible schema
Temporary fast look-up (session, cache, leaderboard)	NoSQL – key-value	Very fast read / write

Situation	Best choice	Reason
Relationship-heavy data (friends, recommendations)	NoSQL – graph	Relationships are first-class
Very large analytical data kept for years	Data warehouse / data lake (Parquet files)	Cheap, scalable analysis

2.6 Data Warehouse and Data Lake (for large-scale analytics)

Organisations that collect data from many systems keep it in a central place for analysis. A **data warehouse** stores **cleaned, structured data** with a fixed schema, optimised for reports and business intelligence. A **data lake** stores **raw data of every type** (tables, logs, images, audio) in its original form at low cost; the structure is applied only when the data is read (**schema-on-read**).

Point	Data Warehouse	Data Lake
Data kept	Processed, structured data	Raw data – structured, semi-structured and unstructured
Schema	Defined before loading (schema-on-write)	Applied while reading (schema-on-read)
Users	Business analysts and managers	Data scientists and engineers
Cost	Higher	Lower
Example	Amazon Redshift, Google BigQuery, Snowflake	Amazon S3 / Hadoop (HDFS) based lakes