Database Management System

Two types:
Structured and unstructured
Basic operation: CRUD Operations
Create - row based
Update - row based
Read, - can be done for row and table
 Delete - can be done for row and table
PRIMARY KEY - non editable and mandatory, table specific
FOREIGN KEY -

A **foreign key** in one table points to the **primary key** in another table.

Example:

Table 1: `Students`

StudentID (PK)	Name
1	Alice
2	Bob

Table 2: `Enrollments`

EnrollmentID	StudentID (FK)	Course
101	1	Math
102	2	Science

- Here, `StudentID` in the **Enrollments** table is a **foreign key** that references `StudentID` in the **Students** table.
- This ensures that you can't insert a record into `Enrollments` with a `StudentID` that doesn't exist in `Students`.

Unlike other languages python which are general purpose, SQL is domain specific

Groupby
Having
Orderby
Limit
Offset

✅ When to use `AND` vs `OR`:

- Use `**AND**` when **all conditions must be true** for the same row.
- Use `**OR**` when **any one of multiple combinations** is acceptable.

Absolutely! Let's break down this SQL query line by line so you understand **exactly what it's doing**:

✅ Query:

SELECT winner, subject
FROM nobel
WHERE yr = 1984
ORDER BY subject IN ('Chemistry', 'Physics'), subject, winner;

🔍 Line-by-line Explanation:

1. `SELECT winner, subject`

- You're selecting two columns from the `nobel` table: `winner` and `subject`.

2. `FROM nobel`

- You're querying the data from the `nobel` table.

3. `WHERE yr = 1984`

- You're filtering the data to include **only the winners from the year 1984**.

4. `ORDER BY subject IN ('Chemistry', 'Physics'), subject, winner`

- This is the most interesting part. Let's break it down:

🧠 How `ORDER BY subject IN ('Chemistry', 'Physics')` Works:

- The expression `subject IN ('Chemistry', 'Physics')` returns:
    - `TRUE` (which is treated as `1`) if the subject is **Chemistry or Physics**
    - `FALSE` (which is treated as `0`) for other subjects

So this ordering **puts Chemistry and Physics at the end**, because:

- SQL sorts in ascending order by default → `0` (others) comes **before** `1` (Chemistry, Physics)

🧾 Full Sort Order Explained:

- First, rows where subject is **not** Chemistry or Physics (i.e., `IN (...)` is `0`)
- Then, within those, sort by **subject alphabetically**
- Then, if subjects match, sort by **winner alphabetically**

✅ Example:

subject	**winner**	`IN ('Chemistry','Physics')`
Literature	Alice Munro	0
Peace	John Smith	0
Chemistry	Jane Doe	1
Physics	Albert Einstein	1

Sorted Result:

1. Literature – Alice Munro
2. Peace – John Smith
3. Chemistry – Jane Doe
4. Physics – Albert Einstein

🔗 JOIN

Purpose: Combine rows from two or more tables based on a related column.

Example:

SELECT orders.order_id, customers.customer_name

FROM orders

JOIN customers ON orders.customer_id = customers.customer_id;

Use case: Get customer names for each order by matching `customer_id`.

📊 GROUP BY

Purpose: Group rows with the same values in specified columns and perform aggregate functions like `COUNT`, `SUM`, `AVG`.

Example:

SELECT customer_id, COUNT(*) AS total_orders

FROM orders

GROUP BY customer_id;

Use case: Count how many orders each customer placed.

⏳ WINDOW FUNCTIONS

Purpose: Perform calculations across a "window" of rows related to the current row without collapsing them into one (like `GROUP BY` does).

Example:

SELECT

  customer_id,

  order_id,

  COUNT(*) OVER (PARTITION BY customer_id) AS total_orders_per_customer

FROM orders;

📥 Reading Data from Various Sources

In data science, you often work with data from different formats:

Source	Tool / Function	Example
CSV File	`pandas.read_csv()`	`pd.read_csv("data.csv")`
Excel File	`pandas.read_excel()`	`pd.read_excel("data.xlsx")`
JSON File	`pandas.read_json()`	`pd.read_json("data.json")`
SQL DB	`pandas.read_sql()`	`pd.read_sql(query, connection)`
Web API	`requests + json` or `pandas.read_json(url)`	`pd.read_json("https://api.example.com/data")`
Python List / Array	`np.array()` or `pd.DataFrame()`	`np.array([1,2,3])`, `pd.DataFrame([[1,2],[3,4]])`

🧮 Data Structures Explained

Type	Library	Description	Example
Array	NumPy	Basic data structure for numerical data (fixed size, homogeneous)	`np.array([1, 2, 3])`

Series	Pandas	1D labeled array (like a column in Excel)	`pd.Series([10, 20, 30])`
DataFrame	Pandas	2D table with rows and columns (like a spreadsheet)	`pd.DataFrame({"A":[1,2], "B":[3,4]})`
Vector	Often NumPy or SciPy	A 1D array used in linear algebra or ML	`np.array([5, 10, 15])`

🔍 Key Differences

- **Array**: Efficient numerical operations (good for math-heavy tasks)
- **Series**: Like a single column in Excel with row labels
- **DataFrame**: Like a full Excel table with rows and columns
- **Vector**: Mathematically treated as direction & magnitude (but stored as arrays)

Data Ingestion and Its Workflow

Data Ingestion is the process of collecting raw data from various sources and transferring it to a storage or processing system like a data lake, warehouse, or analytics engine.

🔁 Data Ingestion Workflow

1. **Source**: Databases, APIs, IoT devices, logs, etc.
2. **Ingestion Layer**: Batch or streaming tool (e.g., Kafka, NiFi, Airflow)
3. **Processing Layer**: Transform data (cleaning, enriching)
4. **Storage Layer**: Data lake or warehouse (e.g., S3, BigQuery, Snowflake)

5. **Analytics Layer**: Reporting, dashboards, ML models

Types of Ingestion Systems

---

1. 🧊 Batch Processing

Definition: Data is collected over a time interval (e.g., hourly, daily) and processed all at once.

✅ Characteristics:

- High throughput
- Cost-efficient
- Easier to manage

❌ Limitations:

- High latency (not real-time)
- Not suitable for instant decision-making

🧰 Tools:

- Apache Hadoop
- Apache NiFi
- AWS Glue
- Apache Airflow (for orchestration)

🔧 Architecture:

Data Source → Ingestion Tool (Airflow) → Processing Engine (Spark) → Data Warehouse

📌 Example:

Generating daily sales reports by reading a day's worth of data at midnight.

---

2. ⚡ Real-Time Streaming

Definition: Data is ingested and processed instantly as it is generated.

✅ **Characteristics:**

- Low latency (real-time updates)
- Supports event-driven processing
- Ideal for monitoring, fraud detection, etc.

❌ **Limitations:**

- Higher complexity
- Requires fault-tolerant infrastructure
- Can be costly

🧰 **Tools:**

- **Apache Kafka** (most popular)
- Apache Flink
- Spark Structured Streaming
- Amazon Kinesis

🔧 **Kafka Architecture:**

Producer → Kafka Topic → Consumer (Stream Processor) → Data Sink

📌 **Example:**

Live processing of transaction data to detect fraud instantly.

---

3. 🕒 Micro-Batching

Definition: A hybrid model where small batches of data are collected and processed frequently (e.g., every few seconds or minutes).

✅ **Characteristics:**

- Near real-time
- Easier to implement than full streaming
- Better resource utilization

❌ **Limitations:**

- Slight delay compared to true real-time
- Requires buffering logic

🧰 **Tools:**

- **Spark Structured Streaming**
- Kafka Streams
- Azure Stream Analytics
- Google Dataflow (with windowing)

🔧 **Architecture:**

Data Source → Kafka → Spark Micro-Batches → Data Warehouse

📌 **Example:**

Monitoring website activity with updates every 10 seconds for dashboards.

🔄 Comparison Table

Feature	Batch	Micro-Batching	Real-Time Streaming

Latency	High (minutes to hours)	Low (seconds to minutes)	Very Low (ms to seconds)
Data Handling	Large chunks	Small frequent chunks	Per event/message
Complexity	Low	Medium	High
Cost	Low	Medium	High
Use Case	Reports, backups	Dashboards, alerts	Fraud detection, IoT data
Tools	Airflow, Glue, Hadoop	Spark Structured Streaming	Kafka, Flink, Kinesis

📥 What is Data Ingestion?

Data ingestion is the process of collecting and importing data for use or storage in a database, data warehouse, or analytics platform.

🧊 Batch Data Ingestion

Definition:
In batch processing, data is collected over a period and then processed as a single unit.

🔹 Features:

- High latency (not real-time)
- Suitable for large volumes of historical data

- Runs on schedule (e.g., hourly, daily)

🔹 **Example Tools:**

- Apache Nifi
- Talend
- AWS Glue
- Hadoop

🔹 **Use Case:**

Generating daily sales reports from logs collected throughout the day.

```
# Example: Reading a CSV batch file using pandas

import pandas as pd

df = pd.read_csv('daily_sales.csv')
```

⚡ Streaming Data Ingestion

Definition:
Streaming ingestion means processing data in real-time as it arrives.

🔹 **Features:**

- Low latency (near real-time)
- Continuous and ongoing
- Ideal for time-sensitive data

🔹 **Tools:**

- **Apache Kafka** (popular for streaming)
- Apache Flink

- Apache Spark Streaming
- Amazon Kinesis

🔹 Use Case:

Live dashboard for monitoring server health, transaction alerts, or IoT sensor data.

```
# Kafka streaming conceptual code
from kafka import KafkaConsumer

consumer = KafkaConsumer('sensor_data', bootstrap_servers='localhost:9092')
for message in consumer:
    print(message.value)
```

🔄 Batch vs Streaming: Comparison Table

Feature	Batch Processing	Streaming Processing
Latency	High (minutes/hours)	Low (real-time/seconds)
Data Size	Large files or tables	Small events/messages
Use Case	Reports, backups	Alerts, dashboards
Complexity	Easier to manage	More complex infrastructure
Example Tool	Hadoop, AWS Glue	Kafka, Spark Streaming

ETL vs. ELT: Building Robust Data Pipelines using Apache Airflow

🔄 ETL (Extract, Transform, Load)

- **Extract** data from source systems (e.g., databases, APIs).
- **Transform** the data (cleaning, filtering, aggregating) before loading.
- **Load** the transformed data into the target system (usually a data warehouse).

✅ *Best when transformation needs to happen before loading.*
🔧 Common in traditional systems like Hadoop or on-prem databases.

⚙️ ELT (Extract, Load, Transform)

- **Extract** and **Load** raw data directly into the data warehouse.
- **Transform** inside the warehouse using SQL or processing engines (e.g., BigQuery, Snowflake).

✅ *Best for cloud-based data warehouses with strong processing power.*
💡 Efficient for large-scale analytics using modern platforms.

📌 Apache Airflow for ETL/ELT

Apache Airflow is an open-source **workflow orchestration tool** used to schedule, monitor, and manage data pipelines.

Why use Airflow?

- **DAG-based Pipelines**: Define workflows as Directed Acyclic Graphs (DAGs).
- **Scheduling**: Run tasks hourly/daily.
- **Monitoring**: Web UI with logs & visual DAGs.
- **Integration**: Works well with ETL tools, Python, Spark, AWS, GCP, etc.

Example ETL DAG in Airflow:

```python
from airflow import DAG

from airflow.operators.python import PythonOperator

from datetime import datetime


def extract(): pass

def transform(): pass

def load(): pass


with DAG('etl_pipeline', start_date=datetime(2023, 1, 1), schedule_interval='@daily') as dag:

    t1 = PythonOperator(task_id='extract', python_callable=extract)

    t2 = PythonOperator(task_id='transform', python_callable=transform)

    t3 = PythonOperator(task_id='load', python_callable=load)


    t1 >> t2 >> t3
```

Data Warehousing Concepts: OLAP, OLTP & Dimensional Modeling

⚡ OLTP (Online Transaction Processing)

- Optimized for **fast inserts, updates, and deletes**
- Used in **day-to-day operations** (e.g., ATM transactions, order booking)
- Normalized schema (many small tables)

💡 *Think: Real-time operational databases*

OLAP (Online Analytical Processing)

- Optimized for **complex queries and reporting**
- Used for **historical data analysis**, BI dashboards
- Denormalized structure (star/snowflake schema)

💡 *Think: Business insights & decision-making*

Dimensional Modeling

A technique to design OLAP-friendly schemas.

⭐ **Star Schema:**

- **Fact Table**: Contains measurable data (e.g., sales amount, quantity)
- **Dimension Tables**: Contain descriptive attributes (e.g., customer, product, region)

```
Product   Customer   Time
    \       |       /
     \      |     /
     ---- FACT ----
```

❄️ **Snowflake Schema:**

- A normalized version of the star schema with more hierarchy.

Summary Table

Concept	OLTP	OLAP

Purpose	Day-to-day transactions	Analytical processing
Speed	Fast writes	Fast reads
Schema	Normalized	Denormalized (Star/Snowflake)
Use Case	Banking, Booking	Sales trend, Market insights
Tools	MySQL, PostgreSQL	Redshift, BigQuery, Snowflake

```python
# -*- coding: utf-8 -*-
"""Data Ingestion - batch Vs Stream.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1oq-oP9FXh3CN4oXeGRkPaOSdGm436YjA

Batch Ingestion (All-at-Once)
"""

import pandas as pd

# Load the dataset
url = "https://raw.githubusercontent.com/velicki/Weather_Data_Analysis_Project/main/Weather_Data.csv"
df = pd.read_csv(url)
df.head()

#PREPROCESSING

# Rename columns for easier access
df.columns = df.columns.str.strip().str.replace(' ', '_').str.replace('/', '_')

# Convert Date/Time to datetime object
df['Date_Time'] = pd.to_datetime(df['Date_Time'], errors='coerce')

# Drop rows with missing Date_Time or Temp_C
df = df.dropna(subset=['Date_Time', 'Temp_C'])

# Optional: Convert temperature to numeric
df['Temp_C'] = pd.to_numeric(df['Temp_C'], errors='coerce')

df.shape
df.head()

"""Stream Ingestion (Row-by-Row Simulation)"""

import pandas as pd
import time

# Step 1: Load the original data
url = "https://raw.githubusercontent.com/velicki/Weather_Data_Analysis_Project/refs/heads/main/Weather_Data.csv"
df = pd.read_csv(url)

# PREPROCESSING (added for streaming section)
# Rename columns for easier access
df.columns = df.columns.str.strip().str.replace(' ', '_').str.replace('/', '_')

# Convert Date/Time to datetime object
df['Date_Time'] = pd.to_datetime(df['Date_Time'], errors='coerce')

# Drop rows with missing Date_Time or Temp_C
df = df.dropna(subset=['Date_Time', 'Temp_C'])

# Optional: Convert temperature to numeric
df['Temp_C'] = pd.to_numeric(df['Temp_C'], errors='coerce')

def alert_high_temp(row):
    if row['Temp_C'] > 5:
        print(f"вљ пёЏ ALERT: High temperature detected at {row['Date_Time']} - {row['Temp_C']}В°C")

# Apply during streaming
def stream_with_alert(data, delay=0.5):
    for idx, row in data.iterrows():
        # print(f"{row['Date_Time']} - Temp: {row['Temp_C']}В°C")
        print(f"Streaming row {idx} в†’ {row.to_dict()}")
        alert_high_temp(row)
        time.sleep(delay)

stream_with_alert(df)
```

```python
# -*- coding: utf-8 -*-
"""ETL TASKS.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/123BxwJSDUHfzl8AxDJzjZq6vEGPZ17JX
"""

import numpy as np
import pandas as pd

url = "https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_month.csv"

# prompt: Load the dataset into a DataFrame

df = pd.read_csv(url)

df.head()

# Get the shape (number of rows, number of columns)
print(df.shape)

# prompt: в—Џ Identify number of records and unique locations

# Number of records is the number of rows

print(f"Number of records:", len(df))

# Identify unique locations
unique_locations = df['place'].nunique()
print(f"Number of unique locations:", unique_locations)

# prompt: в—Џ Print top 5 rows and column names

print(df.columns.tolist())
print(df.head())

# Extract the part after the last comma in the 'place' column
#This splits each string in the 'place' column at the comma, turning it into a list.
#"160 km ESE of Petropavlovsk-Kamchatsky, Russia"в†' ["160 km ESE of Petropavlovsk-Kamchatsky", " Russia"]
#.str[-1] - This picks the last element from the split list вЂ" the part after the last comma.
# ["160 km ESE of Petropavlovsk-Kamchatsky", " Russia"] в†' " Russia"
#.str.strip()This removes any leading or trailing spaces.
#" Russia" в†' "Russia"
df['countries'] = df['place'].str.split(',').str[-1].str.strip()

# View result
print(df[['place', 'countries']].head())

# Get unique values from the 'countries' column
unique_countries = df['countries'].unique()
unique_countries
len(unique_countries)

# Convert 'time' column to datetime
df['time'] = pd.to_datetime(df['time'])

#  Drop rows with missing values in 'latitude', 'longitude', or 'mag'
#df is your DataFrame, likely containing information about earthquakes (based on the column names).

#.dropna() is a Pandas method used to remove rows with missing (NaN) values.

#subset=['latitude', 'longitude', 'mag'] tells Pandas to only check these specific columns.
df = df.dropna(subset=['latitude', 'longitude', 'mag'])

# Filter only earthquakes with magnitude >= 4.0
df = df[df['mag'] >= 4.0]

# Add a new column 'day_of_week' from 'time'
df['day_of_week'] = df['time'].dt.day_name()

#  Create a column severity_level based on magnitude:
# в—‹ <4.0: "Low"
# в—‹ 4.0 - 6.0: "Moderate"
# в—‹ 6.0+: "High"

df['severity_level'] = df['mag'].apply(lambda x: "Low" if x < 4.0 else ("Moderate" if 4.0 <= x < 6.0 else "High"))

# prompt: в—Џ Count number of earthquakes per place

# Count the number of earthquakes per place
earthquake_counts_per_place = df['place'].value_counts()

# Display the counts
print("\nNumber of earthquakes per place:")
earthquake_counts_per_place

# prompt: в—Џ Compute average magnitude and max depth per day
# Step 1: Convert 'time' to datetime
df['time'] = pd.to_datetime(df['time'])

# Step 2: Create a 'date' column
df['date'] = df['time'].dt.date

# Step 3 & 4: Group by date and compute average magnitude and max depth
daily_stats = df.groupby('date').agg({
    'mag': 'mean',
    'depth': 'max'
}).reset_index()

# Optional: Rename columns for clarity
daily_stats.columns = ['date', 'average_magnitude', 'max_depth']

# Display result
print(daily_stats)

# вњ… Save cleaned dataset
df.to_csv('cleaned_earthquakes.csv', index=False)

# Rename columns
daily_stats.columns = ['date', 'average_magnitude', 'max_depth']

# вњ… Save summary dataset
daily_stats.to_csv('earthquake_summary.csv', index=False)

import sqlite3

# Connect to SQLite database (or create it)
conn = sqlite3.connect('earthquakes.db')

# Save cleaned data to table
df.to_sql('cleaned_earthquakes', conn, if_exists='replace', index=False)
```

```
# Save summary data to another table
daily_stats.to_sql('earthquake_summary', conn, if_exists='replace', index=False)

# Close the connection
conn.close()

"""![image.png](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAXUAAAMBCAYAAAAQ0SQ2AAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQUAAAAJcEhZcwAAFiUAABYlAUlSJPAAAEcWSURBVHhe7d1/b

We performed an ETL process on earthquake data to prepare it for analysis:

Extract: We began with a raw CSV dataset containing global earthquake records.

Transform: We converted time columns, removed incomplete records, filtered for significant earthquakes (magnitude в‰Ґ 4.0), and added additional fields like day of the

Load: We saved the cleaned data and summary statistics into both CSV files and a SQLite database for further querying and analysis.

Insight:
Most of the high-magnitude earthquakes (в‰Ґ4.0) with deeper epicenters were observed more frequently on Wednesdays and Fridays, suggesting a pattern worth investigatin
"""
```

```
# -*- coding: utf-8 -*-
"""SQL practice.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1yafoBpjUgAfo-rNcXAQ6sV1PvENiUycy
"""

!pip install Faker

"""Libraries"""

import sqlite3
import pandas as pd
import random
from faker import Faker #to create fake data

"""Initialize"""

conn = sqlite3.connect('ICTAcademy.db')
fake = Faker()
random.seed()
cursor = conn.cursor()

"""# DB Operations

Creating DB
"""

#drop tables if they already exists
#this helps us to build table with the schema of our own
cursor.execute("DROP TABLE IF EXISTS Departments")
cursor.execute("DROP TABLE IF EXISTS Trainer")
cursor.execute("DROP TABLE IF EXISTS Courses")
cursor.execute("DROP TABLE IF EXISTS Students")
cursor.execute("DROP TABLE IF EXISTS Enrollment")

#Creating a table
create_q1 = """CREATE TABLE Departments (
    department_id INTEGER PRIMARY KEY,
    d_name TEXT NOT NULL)""" #(cant keep it null)

create_q2 = """CREATE TABLE Trainer (
    trainer_id INTEGER PRIMARY KEY,
    t_name TEXT NOT NULL,
    department_id INTEGER,
    FOREIGN KEY (department_id) REFERENCES Departments(department_id))"""

create_q3 = """CREATE TABLE Courses (
    course_id INTEGER PRIMARY KEY,
    c_name TEXT NOT NULL,
    department_id INTEGER,
    trainer_id INTEGER,
    credits INTEGER,
    FOREIGN KEY (department_id) REFERENCES Departments(department_id),
    FOREIGN KEY (trainer_id) REFERENCES Trainer(trainer_id))"""

create_q4 = """CREATE TABLE Students (
```

```
    student_id INTEGER PRIMARY KEY,
    s_name TEXT NOT NULL,
    gender TEXT,
    age INTEGER)"""

create_q4 = """CREATE TABLE Enrollment (
    enrollment_id INTEGER PRIMARY KEY,
    student_id INTEGER,
    course_id INTEGER,
    batch INTEGER,
    score INTEGER,
    eligibilty BOOLEAN,
    FOREIGN KEY (student_id) REFERENCES Students(student_id),
    FOREIGN KEY (course_id) REFERENCES Courses(course_id))"""

#executing queries to create tables
cursor.execute(create_q1)
cursor.execute(create_q2)
cursor.execute(create_q3)
cursor.execute(create_q4)

# prompt: give code for printing schema

cursor.execute("SELECT name FROM sqlite_master WHERE type='table';")
tables = cursor.fetchall()
for table in tables:
  print(f"Table: {table[0]}")
  cursor.execute(f"PRAGMA table_info({table[0]});")
  schema = cursor.fetchall()
  for col in schema:
    print(f"  Column: {col[1]} | Type: {col[2]} | NOT NULL: {col[3]} | Primary Key: 
{col[5]}")

"""Populating the database"""

departments = ["DSA","Cybersecurity","full stack", "digital marketting", "AIML"]

#filling departments
for department in departments:
  cursor.execute("INSERT INTO Departments (d_name) VALUES (?)", (department,))
#filling trainers
for i in range(1,11):
  cursor.execute("INSERT INTO Trainer (t_name, department_id) VALUES (?, ?)", 
(fake.name(), random.randint(1,len(departments))))
#filling courses
for i in range(1,11):
  cursor.execute("INSERT INTO Courses (c_name, department_id, trainer_id, credits) 
VALUES (?, ?, ?, ?)", (fake.word(), random))

"""storing the data that is read into dataframw to display it"""

cursor.execute("SELECT * FROM Departments")
df = pd.read_sql_query("SELECT * FROM Departments", conn)
df#should only run it once

cursor.execute("SELECT * FROM Trainer")
df = pd.read_sql_query("SELECT * FROM Trainer", conn)
df#should only run it once
```

```
#edit a value
q1= """UPDATE Trainer SET department_id = 100 WHERE trainer_id = 2"""
cursor.execute(q1)

cursor.execute("SELECT * FROM Trainer")
df = pd.read_sql_query("SELECT * FROM Trainer", conn)
df
```