Introduction of DBMS using Python

Introduction

- A **Database Management System (DBMS)** is a software system that allows users to define, create, maintain, and control access to a database.
- It acts as an intermediary between the user or application program and the database, providing a structured way to manage vast amounts of data efficiently and securely.

Introduction

- **Data:**
  - Raw facts and figures which, when processed, become meaningful information.
  - For example, 'John', 23, 'Bangalore' are data points.
- **Database:**
  - An organized collection of related data, typically stored and accessed electronically, that can be easily accessed, managed, and updated.
- **Database Management System (DBMS):**
  - A software tool (like MySQL or Oracle) that allows users to define, create, maintain, and control access to the database.
  - It acts as an interface between the user/application and the database.

Introduction

- **Database**
  - A **structured collection of data** stored and accessed electronically from a computer system. It's designed to efficiently store, manage, and retrieve information.
- **Schema**
  - The **blueprint** or logical design of the entire database. It defines the structure of the data, including table names, column (attribute) names, data types, and the relationships and constraints between different data elements.
- **Attributes (Columns/Fields)**
  - A **characteristic** or property that describes an entity (like a person or an item). In a relational database table, attributes are represented as **columns**.
- **Records (Rows/Tuples)**
  - A single, complete **entry** or unit of information in a table. In a relational database table, records are represented as **rows**.

Introduction

- Database technologies are primarily categorized into two major types, largely differentiated by their underlying data model and the language used to interact with them:
  - SQL (Relational Databases)
  - NoSQL (Non-Relational Databases)

Introduction

SQL (Relational Databases)

- SQL (Structured Query Language) is the standard language for managing data in **Relational Database Management Systems (RDBMS)**.
- **Data Model:** Data is organized into fixed, pre-defined **tables** with rows and columns. Relationships between tables are established using keys (like **Primary Keys** and **Foreign Keys**).
- **Schema:** They have a **rigid, predefined schema**. All data inserted must conform to this structure. Changes to the schema are generally complex.
- **Properties:** They adhere to **ACID** properties (**A**tomicity, **C**onsistency, **I**solation, **D**urability), ensuring high reliability and data integrity, especially critical for complex transactions (like financial systems).
- **Examples:** MySQL, Oracle, SQL Server.

Introduction

NoSQL (Non-Relational Databases)

- NoSQL (Not only SQL) databases are a class of DBMs that use diverse, non-tabular models for data storage and retrieval.
- **Data Model:** Data is stored in various non-tabular formats, such as:
  - **Key-Value Stores:** Data is stored as a collection of key-value pairs (like a dictionary).
  - **Document Databases:** Data is stored in **documents** (often JSON or BSON format), which can have different structures.
  - **Graph Databases:** Data is stored in nodes and edges to represent relationships.
- **Schema:** They have a **flexible or dynamic schema** (often called "schema-less"). New fields can be added without affecting existing data, offering greater agility.
- **Properties:** Often prioritize **Availability** and **Partition Tolerance** over strict consistency (following the **BASE** paradigm) for high-volume, distributed applications.
- **Examples:** MongoDB (Document), Redis (Key-Value), Cassandra (Wide-Column), Neo4j (Graph).

Introduction

- **Structural Terms (Relational Model)**
  - Relational databases, the most common type, organize data into tables.
  - **Table (Relation):** A collection of data organized into rows and columns, representing an **entity** (like "Student" or "Course").
  - **Row (Tuple):** A single record in a table.
  - **Column (Attribute/Field):** A specific piece of data within a record, such as "Student ID" or "Student Name."
  - **Schema:** The overall structure that defines the organization of data in the database, including the tables, fields, and relationships.

Introduction

- **Keys and Integrity**
  - Keys are attributes used to uniquely identify records and establish relationships.
  - **Primary Key:** An attribute (or set of attributes) that **uniquely identifies** each record (row) in a table. It cannot contain duplicate or null values.
  - **Foreign Key:** An attribute in one table that refers to the **Primary Key** of another table, used to **establish relationships** between them.
  - **Data Integrity:** Ensuring the **accuracy** and **consistency** of data over its entire lifecycle. Integrity constraints (like keys) are rules that enforce this.
  - **Referential Integrity:** A specific type of integrity ensured by Foreign Keys, which guarantees that relationships between tables are valid.

Key-Value Stores

- Key-Value stores represent the simplest data model: data is stored as a collection of **keys** and their associated **values**. This model is often referred to by the general term **NoSQL**.

Feature	Description
Data Model	Simple **Key-Value pairs**.
Schema	**Schema-less** or **No Schema**. You can store different types of values for different keys without pre-defining the structure.
Querying	**Simple Key Lookup**. Data is only retrieved by its unique key. There are no powerful query languages, joins, or complex filtering.
Performance	**Extremely fast** read/write operations for single key lookups due to the direct mapping of the key to the physical storage location. **Low overhead.**

Key-Value Stores

- When to Use DBM / Key-Value Stores:
  - This type of store is best suited for scenarios where data access is simple and direct:
    - **Config Files (Configuration):** Storing application settings (e.g., user_limit: 100, server_ip: "192.168.1.1").
    - **Small Caches:** Storing the results of expensive operations temporarily for fast retrieval, often with a time-to-live (TTL).
    - **Storing Small Amounts of Structured Data Quickly:** When you need to rapidly store and retrieve records, and the primary access method is a simple key.

What is a Key-Value Store?

- A **Key-Value store** is the simplest type of non-relational (NoSQL) database. It is essentially a large, persistent, and organized dictionary or hash map where:
  - **Key:** A unique identifier, like a primary key in a SQL database (e.g., a User ID, a product name, or a session token).
  - **Value:** The data associated with that key (e.g., a user profile, a product description, or an entire user session object).
- The core operation is a direct lookup: you provide the **Key**, and the system instantly returns the associated **Value**.

What is a Key-Value Store?

Characteristic	Description
Simplicity	They do not enforce relationships between data, complex querying, or predefined schemas.
Speed	Extremely fast at reading and writing data because they use the key to directly calculate or locate the value's storage address (known as hashing).
No Schema	The structure of the **Value** is not defined by the database. You can store a user's name as a string for one key and a complex JSON object for another key in the same store.

Key-Value Store

- Python provides two built-in modules, **dbm** and the more commonly used **shelve**, that implement file-based key-value storage.
- The **shelve module** is preferred because it can store almost any Python object.
- The **shelve module** creates a persistent dictionary-like object on a disk file.
- It uses the pickle module to serialize (convert) almost any arbitrary Python object into a string of bytes before storing it as the **Value**.

Key-Value Store

Function/Concept	**Purpose**
shelve.open(filename, flag='c')	Opens a shelf file. The **filename** is the base name of the file(s) created. The **flag='c'** means create the file if it doesn't exist.
Dictionary-like Access	You interact with the shelf object exactly like a dictionary: **s[key] = value** to write, and **s[key]** to read.
Keys/Values/Items	Supports standard dictionary methods like **s.keys(), s.values()**, and **s.items().**
del s[key]	Deletes the key-value pair from the store.
s.close()	**Crucial step:** Closes the shelf file and ensures all pending writes are flushed to disk, guaranteeing data persistence.

```python
import shelve
import os

# Define a filename for the shelf database
DB_FILE = 'my_key_value_store'

# -----------------
# 1. WRITE (CREATE/UPDATE) DATA
# -----------------

print(f"--- 1. Writing Data to the Shelf: {DB_FILE} ---")

# Open the shelf file with 'c' (create) flag
# The 's' variable acts like a standard Python dictionary
with shelve.open(DB_FILE, flag='c') as s:
    # Store simple string data (key: 'username', value: 'Alice')
    s['username'] = 'Alice'

    # Store a complex Python object (list)
    s['products_list'] = ['Laptop', 'Monitor', 'Keyboard']

    # The data is automatically saved to the file when 'with' block exits
    print(f"Stored 'username' and 'products_list'.")
# s.close() is automatically called here
```

```python
# -----------------
# 2. READ (RETRIEVE) DATA
# -----------------

print("\n--- 2. Reading Data from the Shelf ---")

# Re-open the shelf file
with shelve.open(DB_FILE) as s:
    # Retrieve simple data
    user = s['username']
    print(f"Retrieved username: {user}")

    # Retrieve complex data (it comes back as a Python list)
    products = s['products_list']
    print(f"Retrieved products: {products}")

    # Accessing all keys
    all_keys = list(s.keys())
    print(f"All keys in the store: {all_keys}")
```

```python
# -----------------
# 3. DELETE DATA
# -----------------

print("\n--- 3. Deleting Data from the Shelf ---")

# Open and delete a key
with shelve.open(DB_FILE) as s:
    if 'products_list' in s:
        del s['products_list']
        print(f"Deleted 'products_list'.")

    # Verify deletion
    remaining_keys = list(s.keys())
    print(f"Remaining keys: {remaining_keys}")
```

Key-Value Store

- Because shelve must re-read and re-write data to the disk, direct modification of mutable values (like lists or dictionaries) in the shelf often fails unless explicitly handled.

```python
import shelve

DB_FILE = 'mutability_test'

with shelve.open(DB_FILE) as s:
    # Initial setup
    s['shopping_list'] = ['milk', 'eggs']
    print(f"Initial list: {s['shopping_list']}")

    # INCORRECT WAY: Modifies the in-memory copy, not the shelf
    s['shopping_list'].append('bread')

# Re-open the shelf to check the persistent data
with shelve.open(DB_FILE) as s:
    # Output will still be ['milk', 'eggs']
    print(f"After INCORRECT update: {s['shopping_list']}")
```

Key-Value Store

- To update a mutable object, students must explicitly fetch the object, modify it, and then **reassign** it back to the key.

```python
# CORRECT WAY: Fetch, Modify, Reassign
with shelve.open(DB_FILE) as s:
    # 1. Fetch the data (gets a copy)
    temp_list = s['shopping_list']

    # 2. Modify the in-memory copy
    temp_list.append('coffee')

    # 3. Reassign the modified copy back to the shelf key
    s['shopping_list'] = temp_list

# Re-open the shelf to check the persistent data
with shelve.open(DB_FILE) as s:
    # Output will be ['milk', 'eggs', 'coffee']
    print(f"After CORRECT update: {s['shopping_list']}")
```

Key-Value Store

- **The Writeback Feature**
  - To simplify the Mutability Trap, shelve offers an optional **writeback=True** flag.
  - When opened with writeback=True, the shelf keeps all fetched items in memory (in a cache).
  - Any modification to those in-memory copies *is* automatically written back to the disk when the shelf is closed.
  - While it solves the mutability problem, it can consume a lot of memory for large shelves and slows down the closing operation as it flushes all changes to the disk at once.

Key-Value Store

```python
with shelve.open('writeback_demo', writeback=True) as s:
    s['data'] = {'count': 0}

    # CORRECT with writeback=True
    s['data']['count'] += 1 # Direct modification works!

# The change is automatically saved when the 'with' block exits
```

SQL (Relational Databases)

- SQL databases, also known as Relational Database Management Systems (**RDBMS**), organize data into tables with pre-defined structures.

Feature	**Description**
Data Model	Data is organized into **Tables** (Relations) with a fixed number of **Columns** (Attributes) and an unlimited number of **Rows** (Tuples).
Schema	**Strict Schema**. The structure of the data (table names, column names, data types) must be defined before data is inserted.
Querying	**Complex Queries** using the **Structured Query Language (SQL)**. Supports joins across multiple tables, aggregation, and powerful filtering.
Integrity & Reliability	**High Integrity** enforced through constraints like **Primary Keys** and **Foreign Keys (FKs)**, which maintain relationships. Supports **ACID** properties.

SQL (Relational Databases)

- SQL is the established standard for applications requiring robust data consistency and complex data relationships:
- **Multiple Related Tables:** When your data naturally breaks down into multiple entities that need to be linked (e.g., customers, orders, and products). **Joins** are used to query across these relationships.
- **Complex Queries:** When you need to answer sophisticated questions involving aggregates, grouping, ordering, and combining data from many sources.
- **Integrity (FKs):** When **data consistency** is critical (e.g., in a financial system or e-commerce platform), you rely on Foreign Keys to prevent orphan records.
- **Multi-user:** When multiple users or application instances need to access and modify the data concurrently. The database handles locking and concurrency control.
- **ACID Semantics:** When transactional reliability is paramount. The database guarantees that operations are **A**tomic, **C**onsistent, **I**solated, and **D**urable.

SQL (Relational Databases)

- The Python **sqlite3** module is the standard library for working with the **SQLite** database. It allows you to use the full power of SQL within your Python programs without needing a separate database server.
- SQLite is an open-source relational database management system (RDBMS) that is distinct from traditional databases like MySQL or PostgreSQL.
- **Serverless:** Unlike other RDBMSs, SQLite is "serverless." It doesn't run as a separate service or daemon.
- **Zero-Configuration:** It requires no setup or administration.
- **File-Based:** The entire database—including definitions, tables, indices, and data—is stored in a single, standard disk file

SQL (Relational Databases)

- The workflow for using the sqlite3 module involves four main steps: **Connect, Cursor, Execute, Commit/Close.**
- **Connecting to the Database**
  - You start by connecting to the database file. If the file doesn't exist, SQLite will create it.

Concept	**Python Code**
Connection Object	conn = sqlite3.connect('example.db')
In-Memory DB	conn = sqlite3.connect(':memory:') (The database exists only for the duration of the script's execution.)

SQL (Relational Databases)

- **Creating a Cursor**
  - The **Cursor** object is what allows you to execute SQL commands and fetch results.
  - **cursor = conn.cursor()**
- **Executing SQL Commands**
  - The cursor's execute() method is used to run any SQL command, whether it's for defining the structure (DDL) or manipulating the data (DML).

SQL (Relational Databases)

- Use DDL commands to create the structure (schema).

```
import sqlite3

# 1. Connect
conn = sqlite3.connect('company_data.db')
cursor = conn.cursor()

# 2. Execute DDL: CREATE TABLE
cursor.execute('''
    CREATE TABLE employees (
        id INTEGER PRIMARY KEY,
        name TEXT NOT NULL,
        department TEXT,
        salary REAL
    )
''')

# 3. Commit the changes to the database file
conn.commit()
```

SQL (Relational Databases)

- Use commands (INSERT, UPDATE, DELETE) to manage the data.

```
# DML: INSERT
# The data values are passed as a tuple in the second argument
cursor.execute(
    "INSERT INTO employees (name, department, salary) VALUES (?, ?, ?)",
    ('Alice', 'Sales', 60000.0)
)

# DML: UPDATE
cursor.execute(
    "UPDATE employees SET salary = ? WHERE name = ?",
    (65000.0, 'Alice')
)

conn.commit()
```

SQL (Relational Databases)

- **Querying Data (SELECT and Fetching)**
  - The SELECT statement retrieves data, but you need a separate method to pull the results into Python memory.

```python
# Query 1: SELECT a single row
cursor.execute("SELECT * FROM employees WHERE name = ?", ('Alice',))
one_record = cursor.fetchone() # Fetches the next single row (as a tuple)
print(f"Single Record: {one_record}")

# Query 2: SELECT multiple rows
cursor.execute("SELECT name, department FROM employees WHERE salary > 50000")
all_records = cursor.fetchall() # Fetches all remaining rows (as a list of tuples)
print(f"All Records: {all_records}")

# Loop through the results row by row
cursor.execute("SELECT name FROM employees")
print("Employee Names:")
for row in cursor:
    print(f"- {row[0]}")
```

SQL (Relational Databases)

- **Closing the Connection**
  - Always close the connection to ensure that all committed changes are written to the disk and to free up resources. The **recommended practice** is to use the with statement, which handles automatic commit and closing, even if errors occur.

```python
import sqlite3

# Recommended practice: The 'with' statement handles commit and closing
try:
    with sqlite3.connect('company_data.db') as conn:
        cursor = conn.cursor()

        # Insert a new record
        cursor.execute(
            "INSERT INTO employees (name, department, salary) VALUES (?, ?, ?)",
            ('Bob', 'IT', 72000.0)
        )
        # conn.commit() is called automatically here

except sqlite3.Error as e:
    print(f"An error occurred: {e}")
    # In case of an error, the transaction is rolled back (changes are discarded)
```

SQL (Relational Databases)

- The typical, most common SQL query you will write follows this structure:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- **The SELECT Clause: What Data to Display**
  - The SELECT clause is always the **first** part of a query and defines **what columns (fields) you want to see** in the final result.

Term	**Purpose**	**Example**
column1, column2	You specify the exact columns you want. This is a best practice to avoid unnecessary data transfer.	SELECT Name, Email, Salary
*** (Asterisk)**	The wildcard character that means "all columns." Use this for quick checks, but avoid it in production code.	SELECT *

Example Goal: I want to see only the names and prices of products.

SQL: SELECT ProductName, Price

SQL (Relational Databases)

- **The FROM Clause: Where to Find the Data**
  - The FROM clause is the **second** part of a query and specifies **which table** in the database contains the data you are querying.

Term	Purpose	Example
table_name	This is the specific table (like a spreadsheet sheet) that the database should search within.	FROM Products
Example Goal: The product data is stored in a table called Products.	**SQL:** FROM Products	

SQL (Relational Databases)

- **The WHERE Clause: How to Filter the Rows**
  - The WHERE clause is an **optional filter** that specifies **which rows (records)** should be included in the final result set. Without it, the query will return data from *every single row* in the table.

Term	Purpose	Example
condition	This is a logical test that evaluates to **True** or **False** for every row. Only rows where the condition is **True** are kept.	WHERE Price > 50
Operators	You use comparison operators like: **=** (equal), **>** (greater than), **<** (less than), **!=** or **<>** (not equal), **LIKE** (pattern matching).	WHERE Category = 'Electronics'
Example Goal: I only want to see products where the price is less than $100.	**SQL:** WHERE Price < 100	

SQL (Relational Databases)

- Let's assume we have a table named Books with the following structure:

ID	Title	Author	Price	InStock
1	The Cat	A. Smith	15.00	Yes
2	SQL Basics	J. Doe	55.50	Yes
3	Data Science	P. Jones	99.99	No
4	The Novel	A. Smith	12.00	Yes

SQL (Relational Databases)

- Retrieve the **Title** and **Price** for all books written by **'A. Smith'** that are **In Stock**.

```
SELECT Title, Price
FROM Books
WHERE Author = 'A. Smith' AND InStock = 'Yes';
```

Title	Price
The Cat	15.00
The Novel	12.00