UNIT-V (Database programming)

Introduction:

→ **persistent storage**: In any application, there is a need for persistent storage. There are three basic storage mechanisms: files, a relational database systems (RDBMS) (or) some sort of hybrid, i.e an API (application programmers Interface) that sits on the top of one of those existing systems, an object relational mapper (ORM), file manager, spreadsheet, configuration file, etc.

→ **Basic Database operations and SQL**

ⓐ **underlying storage**: Databases usually have a fundamental persistent storage using the file system, i.e normal operating files, special O.S files and even raw disk partitions.

ⓑ **User Interface**: Most database systems provide a Command-line tool with which to issue SQL commands & queries. There are also some GUI tools that use the Command-line clients (or) the database client library, giving users a much nicer interface.

ⓒ **Databases**:

An RDBMS can usually manage multiple databases eg. Sales, marketing, customer support, etc, all on the same server. MySQL is a server-based RDBMS.

SQLite, Gadfly, PostgreSQL, MongoDB, MariaDB, ...

④ components:

The table is the storage abstraction for databases. Each row of data will have fields that correspond to database columns. The set of tabe definitions of columns and datatypes per table all put together to define the database schema.

Databases are created and dropped. The same is true for tables. Adding new rows to a database is called inserting, chaning existing rows in a table is called updating, and removing existing rows in a table is called deleting. These actions are referred to as database commands or operations. Requesting rows from a database with optional criteria is called querying. Some databases use the concept of a cursor for issuing SQL commands, queries, and grabbing results, either all at once (or) one row at a time.

⑤ SQL: Database commands and queries are given to a database by SQL. Most databases are configured to be case-insensitive, especially commands. Use CAPS for database keywords.

→ CREATE DATABASE test; → Creating a database

→ USE test; → using a database

→ DROP DATABASE test; removes all tables & data

Creating a table:

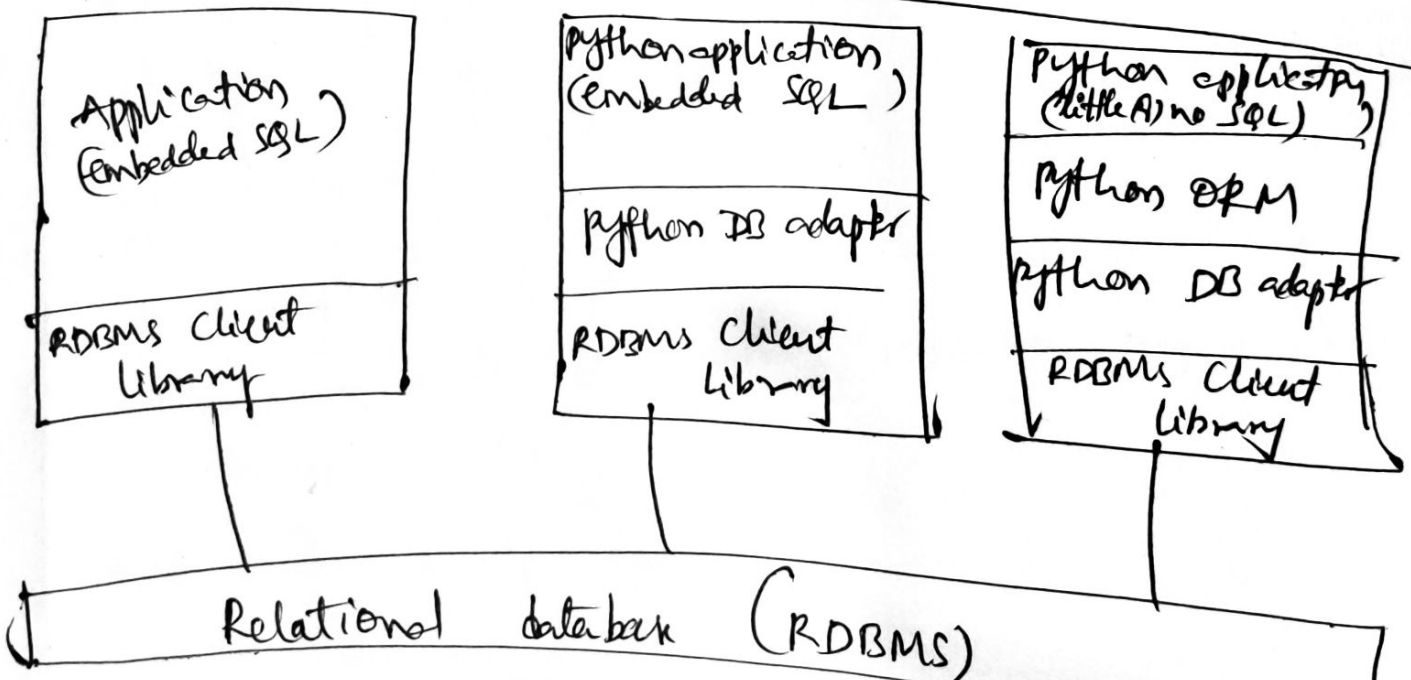
```
CREATE TABLE users ( login VARCHAR(8), uid INT,
                                       prid INT );
```

→ DROP TABLE users → for dropping

Syntax → INSERT INTO users VALUES ("ECE", 311, 1);

syntax { UPDATE users SET prid=4 ~~where~~ WHERE prid=2;
       { UPDATE users SET prid=1 ~~otherwise~~ WHERE uid=311

deleting { DELETE FROM users WHERE prid = 4
         { DELETE FROM users ; → for all rows.



Multilayered communication b/w application and database.

→ ORMs can simplify an application by handling all of the database-specific details.

→ A database adapter is an implementation of a database connector.

UNIT 5 : Python (Databases)

The standard for database interfaces is the python DB-API. It supports a wide range of database servers such as:

Oracle, Sybase, Interbase, MySQL, mSQL, Gadfly, postgreSQL

→ we must download a separate DB API module for each database we need to access.

The DB API provides a minimal standard for working with databases using python structures and syntax wherever possible. This API includes:

① Importing the API module
② Acquiring a connection with the database
③ Issuing SQL statements and stored procedures.
④ closing the connection.

MySQL Database

→ Install MYSQL.

python needs a MySQL driver to access the MySQL database. use 'MySQL Connector'.

→ python -m pip install mysql-connector-python

→ `import mysql.connector`

① import mysql.connector

② Establish connection using connect()

```
mydb = mysql.connector.connect( host = "host_name",
                                user = "user_name",
                                passwd = "password",
                                database = "db_name")
```

③ Create cursor object

```
obj = mydb.cursor()
```

④ Execute queries.

①
```
import mysql.connector
mydb = mysql.connector.connect( host = "localhost",
                                user = "root",
                                passwd = "")

print("Database connected")

c = mydb.cursor()
c.execute("CREATE DATABASE 4ECE")
print("Database created")
```

op: Database connected
Database created

in the mysql database 4ECE will be created.

Create table

(2)

```
# create table
import mysql.connector
mydb = mysql.connector.connect(host = "localhost",
                               user = "root",
                               passwd = "",
                               database = "4ECE")

print("Database connected")
c = mydb.cursor()
c.execute("CREATE TABLE TB1 (NO INT(20), NAME VARCHAR(20))")

print("TABLE created")
```

(3)

```
# insert row
import mysql.connector
mydb = mysql.connector.connect(host = "localhost",
                               user = "root",
                               passwd = "", database = "4ECE")

print("database connected")
c = mydb.cursor()
c.execute("INSERT INTO tb1 (NO, NAME) VALUES
                               (303, 'mgit')")

Print("row created")

mydb.commit()
```

create → C → (create database / table)
read → R → (select * FROM ...)
update → U → (Update / INSERT..)
Delete → D → (DELETE / DROP . . .)

} CRUD operation on a Database

(4)

```
# select
import mysql.connector
mydb = mysql.connector.connect ( host = "local host",
                                 user = "root",
                                 passwd = "", database = "4EECE")

print ("database connected")
c = mydb.cursor()
c.execute ("SELECT * FROM TB1")
r = c.fetchall()
for i in r:
    print(i)
```

o/p database connected
(303, 'mgit')

fetchall() → fetches all the rows of a query result. It returns all the rows as a list of tuples. An empty list is returned if there is no record to fetch.

(8)

```
# insert multiple rows
import mysql.connector
mydb = mysql.connector.connect ( host = "local host",
                                 user = "root",
                                 passwd = "", database = "4EECE")

print ("database connected")
c = mydb.cursor()
qry = ("INSERT INTO tb1 (NO, NAME) VALUES (%s,%s)")
rows = [(102, "abc")
```

→ Commit() → It is used to make sure the changes made to the database are consistent. It basically provides the database confirmation regarding the changes made by a user (or) an application in the database.

rollback() → It is used to revert the last changes made to database. If a condition arises where one is not satisfied with the changes made to the database (or) transaction fails, it can be used retrieve the original data.

(6) # insert multiple rows.

```
import mysql.connector
mydb= mysql.connector.connect ( host= "localhost",
                                user= "root",
                                passwd= "",
                                database = "4CCE")

print ("Database connected")
c= mydb.cursor()
qry= ("INSERT INTO tb1 (NO,NAME) VALUES (%s,%s)")
rows= [ (102, "abc"),
        (11, "xyz"),
        (10, "pqr"),
        (15, "mno")]

try:
    c.executemany ( qry, rows)
    mydb.commit()
except:
    mydb.rollback()

print (" rows inserted ")
```

executemany() : used to insert multiple rows into a table

executemany (query, list)

- query → SQL query to insert
- list → list of tuples containing the data to be inserted

for updating

→ c.execute("update tb1 set no = 7001 where name = 'mgjt'")

for deleting

→ c.execute("DELETE FROM tb1 WHERE no = 102")

→ c.execute("SELECT * FROM tb1 WHERE no = 1").

→ c.execute("DROP TABLE tb1")

→ c.execute('DROP TABLE IF EXISTS TB1')

Python DB-API [Database Application programmer's Interface]

→ The API is a specification that states a set of required objects and database access mechanisms to provide consistent access across the various adapters and underlying database systems.

SIG (Special Interest Group) ⇒ DB-API version 1.0

→ The API provides for a consistent interface to a variety of relational databases, and porting code between different databases is much simpler.

DB-API module Attributes:

Attribute	Description
apilevel	→ version of DB-API module is complaint with
threadsafety	→ Level of thread safety of this module
paramstyle	→ SQL statement parameter style of this module
Connect()	→ Connect() function

⇒ Data Attributes:

① apilevel → This string indicates the highest version of the DB-API the module is compliant with i.e '1.0', '2.0' etc. If absent, "1.0" should be assumed as the default value.

② threadsafety:

0 → Not threadsafe; so threads should not share the module at all.

1 → Minimally threadsafe: threads can share the module but not connections

2 → Moderately threadsafe: threads can share the module and connections but not cursors.

3 → Fully thread safe: threads can share the module, connections and cursors.

⑥ paramstyle

The API supports a variety of ways to indicate how parameters should be integrated into an SQL statement that is eventually sent to the server for execution. It is a string that specifies the form of string substitution used when building rows for a query (or) command.

parameter style	Description	Example
numeric	Numeric positional style	WHERE name = :1
named	Named style	WHERE name = :name
pyformat	python dictionary print() format conversion	WHERE name = %(name)s
qmark	Question mark style	WHERE name = ?
format	ANSI C printf() format conversion	WHERE name = %s

⇒ Function Attributes

⑥ connect() function attributes

parameters	description
user	Username
password	password
host	hostname
database	database name
dsn	Data source name

Ex: mysql.connector.connect (host = "localhost",
user = "root",
passwd = "",
database = "db_name")

⇒ DB-API Exception classes

Exception	Description
Warning	: Root warning exception class
Error	: Root error exception class
→ InterfaceError	: Database Interface error
→ DatabaseError	: Database error
→ DataError	: problems with the processed data
→ OperationalError	: Error during database operation execution
→ IntegrityError	: Database relational integrity error
→ InternalError	: Error that occurs within the database
→ ProgrammingError	: SQL command failed
→ NotSupportedError	: Unsupported operation occured.

⇒ Connection object methods

Method Name	Description
close()	close database connection
commit()	commit current transaction
rollback()	Cancel current transaction
cursor()	create a cursor (or) cursor-like object using this connection
errorhandler()	serves as a handler for given connection cursor.

cursor object attributes

Object Attribute	Description
arraysize	: No of rows to fetch at a time fetchmany() defaults to 1
connection	: Connection that created this cursor (optional)
rowcount	: No of rows that the last execute() produced (or) affected.
callproc(func[,args])	: call a stored procedure
* close()	: close cursor
* execute(op)	: execute a database query (or) command
* executemany(op, args)	: prepare and execute a database query (or) command over given arguements.
* fetchone()	: Fetch next row of query result.
* fetchall()	: fetch all rows of a query result
messages	: List of messages received from the database for cursor execution.
next()	: used by iterator to fetch next row of query result.
nextset()	: Move to next results set
rownumber	: Index of cursor on current result set

Type objects and constructors

Type object	Description
Date(yr, mo, dy)	: object for a date value
Time(hr, min, sec)	: object for a time value
STRING	: object describing string-based columns (VARCHAR)
NUMBER	: " numeric columns
ROWID	: " "row ID" columns

Relational Databases:

Commercial RDBMS	open source RDBMS	Database APIs
Informix	MySQL	. JDBC
Sybase	postgreSQL	. ODBC
Oracle	SQLite	
MS SQL server	Gadfly	
DB/2		
SAP		
Rubrbase		
Ingres		

Adapters (python)

MySQL → MySQLdb, mysql

postgreSQL → psycopg, PypgSQL, PyGreSQL

SQLite → pysqlite

Object-Relational Managers (ORMS)

~~A variety of different database systems~~

→ ORM (Object-Relational Mappers)

It is a code library that automates the transfer of data stored in relational database tables into objects that are more commonly used in application code.

Relational database (MYSQL).

id	FIRSTNAME	LASTNAME	PHONE
1	AAAAA	ZZZZZ	111111
2	BBBBB	YYYYY	222222
3	CCCCC	XXXXX	333333

Python objects

class person: "AAAAA" FIRSTNAME = "ZZZZZ" LASTNAME = "111111" PHONE2

class person: "BBBBB" FIRSTNAME = "YYYYY" LASTNAME = "222222" PHONE2

class person: "CCCCC" FIRSTNAME: "XXXXX" LASTNAME: "333333" PHONE:

→ ORMs provide a bridge b/w relational database tables, relationships and fields and python objects.

⇒ ORMs provide a high level abstraction upon a relational database that allows a developer to write python code instead of SQL to create, read, update and delete and schemas in their database. Developers can use the programming language they are comfortable with to work with a database instead of writing SQL statements (or) stored procedures.

Ex: SELECT * FROM USERS WHERE NO = 999; ← without an ORM

users = Users.objects.filter(NO=999) → Django ORM query

→ The ability to write python code instead of SQL can speed up web application development.

→ ORMs also make it theoretically possible to switch an application btw various relational databases.

web framework	None	Flask	Flask	Django
ORM	SQLAlchemy	SQLAlchemy	SQLAlchemy	Django ORM
database connector	(built into python stdlib)	MySQL-Python	psycopg	psycopg
relational database	SQLite	MySQL	postgreSQL	postgreSQL

python ORM implementations

① SQLAlchemy ② The Django ORM ③ SQLObject ④ PDO, Qlim

↓

It is a well-regarded python ORM because it gets the abstractio level "just right" and seems to make complex database queries easier to write than the Django ORM in most cases.

```
class Student(models.Model):
    stuid = models.IntegerField()
    stuname = models.CharField(max_length=70)
    stuemail = models.EmailField(max_length=70)
```

```
CREATE TABLE "enroll_student"
("stuid" integer NOT NULL Primary key,
"stuname" varchar(70) NOT NULL,
"stuemail" varchar(70) NOT NULL);
```

↓

id	stuid	stuname	email