PRACTICAL NO. :- 1

Design a Comprehensive Data Model for a Business Scenario

Objective: To design and implement a normalized data model for a business case using Python and SQL.

Software/Tools Required:

- Python 3.x
- SQLite / MySQL Workbench
- VS Code or PyCharm
- DB Browser for SQLite
- dbdiagram.io (optional for ERD)

Theory / Background: (Covers data modeling concepts, normalization (up to 3NF), identifying entities, relationships, primary and foreign keys.)

Data modeling is foundational in database design. It helps in understanding how data elements relate to one another and ensures efficient storage and retrieval. There are three levels:

- **Conceptual model**: High-level design identifying entities and their relationships.
- **Logical model**: Applies normalization, identifies keys, and refines relationships.
- **Physical model**: Implements the structure in a database system.

Normalization (1NF, 2NF, 3NF) is used to eliminate redundancy:

- **1NF**: Eliminate repeating groups.
- **2NF**: Eliminate partial dependency.
- **3NF**: Eliminate transitive dependency.

ERDs (Entity Relationship Diagrams) are visual tools to represent entities, attributes, and relationships.

Problem Statements:

- High: Online retail business
- Moderate: Library management system
- Poor: Student-course tracking system

Procedure / Steps:

1. Identify key entities and relationships

2. Normalize up to 3NF
3. Sketch ERD
4. Write raw SQL in Python
5. Execute and validate

Expected Output:

- Normalized tables with appropriate constraints
- Working primary and foreign keys
- Tables created and verified successfully

Result: Database schema implemented successfully with normalization.

Conclusion: Learned how to design logical and physical models using SQL and Python.

Viva Questions:

- What is 3NF?
- Why normalize data?
- What is an ERD?

PRACTICAL NO. :- 2

Aim: Automate the Creation of a Data Model using Python

Objective: To create and execute Python scripts for building database schema using raw SQL.

Software/Tools Required:

- Python 3.x
- SQLite or MySQL
- Text Editor / IDE

Theory / Background: (Focus on executing SQL through Python, automation, connection handling, raw SQL execution.)

Automation with Python improves consistency and reduces manual errors in database creation. Using libraries such as sqlite3 or mysql.connector, students can write scripts to:

- Connect to the database
- Execute SQL commands (CREATE TABLE, INSERT, etc.)
- Automate schema creation

This method supports **DevOps practices** like CI/CD for database applications. Students also learn about **DBMS connectivity**, error handling in scripts, and how SQL statements integrate into Python workflows.

This practical provides an introduction to **programmatically managing databases**, which is essential in back-end development.

Problem Statements:

- Level 1: E-commerce platform schema
- Level 2: Student registration system
- Level 3: Students and courses table

Procedure / Steps:

1. Define schema (entities, attributes)
2. Write raw SQL create statements
3. Use Python's sqlite3 or MySQL connector
4. Execute SQL through script

Expected Output:

- Tables created via Python script
- Foreign key relationships in place
- Data model automates DB schema creation

Result: Automated data model creation via Python achieved.

Conclusion: Automating DB schema increases reusability and reduces error.

Viva Questions:

- How to execute SQL in Python?
- Difference between SQLite and MySQL?
- What is a cursor object?

PRACTICAL NO. :- 3

Aim:Implement Database Schema and Tables Based on Naming Standards and Normalization

Objective: To apply database naming conventions and normalization to design schemas and create tables.

Software/Tools Required:

- MySQL Workbench / PostgreSQL + pgAdmin
- Python 3.x (for optional automation)

Theory / Background: Emphasizes on naming standards for tables/columns and structured normalization (1NF to 3NF).

Using **standard naming conventions** (e.g., tbl_Student, col_Name) ensures databases are understandable, scalable, and maintainable across teams.

Best practices include:

- Using snake_case or PascalCase consistently
- Prefixing table names for clarity
- Avoiding reserved words

Coupling naming standards with normalization leads to a **clean, error-free schema**. It's especially critical in organizations where multiple teams access the same database.

This practical enforces professional database development habits and improves collaboration among developers.

Problem Statements:

- High: Banking application database
- Moderate: Inventory tracking system
- Poor: Employee-department schema

Procedure / Steps:

1. Define business rules and naming standards
2. Identify and normalize entities
3. Apply naming conventions for all objects
4. Create SQL tables and test structure

Expected Output:

- Well-named tables following standards
- Correctly normalized structures
- Foreign key relationships applied

Result: Standardized, normalized tables ready for use

Conclusion: Proper naming and normalization improve maintainability and performance.

Viva Questions:

- What are common naming conventions?
- What is a surrogate key?
- What happens when you skip normalization?

PRACTICAL NO. : 4

Aim: Create and Manage Indexes and Views in a MySQL Database

Objective: To improve performance and usability by creating indexes and views in a relational database.

Software/Tools Required:

- MySQL Workbench
- DB Browser (optional)

Theory / Background: Explains how indexes improve query speed and views simplify access to data.

Indexes improve query performance by enabling faster searches and sorting. Common types:

- **Primary index** (automatically on primary key)
- **Secondary index** (on other columns)

Views are virtual tables created from SELECT queries. Benefits:

- Encapsulate complex joins and logic
- Restrict column access (for security)
- Reuse logic in applications

This practical introduces **performance optimization and abstraction**, essential for large databases.

Problem Statements:

- High: Create indexed views for large e-commerce orders
- Moderate: Create views for student performance
- Poor: Index customer name in small retail DB

Procedure / Steps:

1. Create tables and populate data
2. Design views with joins and filters
3. Create indexes on performance-critical columns
4. Measure query performance

Expected Output:

- Properly created indexes
- Views reflecting business requirements
- Query results improved

Result: Indexes and views successfully implemented

Conclusion: Indexes enhance speed; views enhance abstraction.

Viva Questions:

- What is a clustered vs non-clustered index?
- Can you update a view?
- When to use indexes?

PRACTICAL NO.:5

Aim: Perform Forward and Reverse Engineering of a Data Model

Objective: To perform database generation from model (forward) and create model from database (reverse).

Software/Tools Required:

- MySQL Workbench or pgAdmin
- dbdiagram.io or Lucidchart

Theory / Background: Forward engineering converts ERD to schema; reverse engineering derives ERD from schema.

Forward Engineering: Translating a data model (ERD) into SQL code for database creation.

Reverse Engineering: Extracting a database's structure to generate an ER diagram.

Tools like **MySQL Workbench** and **dbdiagram.io** enable visual and code-based approaches to database design. This practical teaches students how to switch between **design (visual)** and **development (code)** perspectives, a vital skill for DBAs and software engineers.

Problem Statements:

- High: Large academic system (reverse ERD + forward script)
- Moderate: Medium HR database (reverse into diagram)
- Poor: Two-table database reverse into ERD

Procedure / Steps:

1. Reverse engineer DB using tool
2. Modify and validate ERD
3. Forward engineer DB script from ERD
4. Execute script in DB

Expected Output:

- ER diagram created and generated from DB
- Forward-engineered schema script

Result: Successful transformation between model and DB

Conclusion: Bi-directional engineering supports better design and maintenance

Viva Questions:

- What is forward vs reverse engineering?
- How do tools help design?
- What is schema sync?

PRACTICAL NO. :-6

Aim: Implement a Physical Data Model in a Database

Objective: To deploy a physical model that aligns with hardware, storage, and indexing strategies.

Software/Tools Required:

- MySQL / PostgreSQL

Theory / Background: A physical data model is detailed for storage, partitions, indexing, tablespaces, etc.

A **Physical Data Model** (PDM) includes:

- Data types (VARCHAR, INT)
- Constraints (NOT NULL, UNIQUE)
- Indexes
- Tablespaces (in PostgreSQL/Oracle)

While logical models show what data is stored, the physical model shows **how** and **where** it's stored. Students learn how to:

- Optimize for performance
- Ensure data integrity
- Apply indexing strategies

This practical bridges the gap between database design and real-world deployment.

Problem Statements:

- High: Retail system with partitioned fact table
- Moderate: Centralized patient database
- Poor: Contact management system

Procedure / Steps:

1. Define physical needs (e.g., partitions, filegroups)
2. Translate logical model to physical model
3. Apply performance configurations

Expected Output:

- Physical storage mapped schema

- Indexed and partitioned tables

Result: Physical schema created with optimization

Conclusion: Physical design is essential for production-grade systems

Viva Questions:

- What is difference between logical and physical model?
- What is tablespace?
- What are I/O optimization techniques?

PRACTICAL NO. :7

Aim: Design and Implement a Data Warehouse Schema

Objective: To build a star schema-based warehouse with dimensions and fact tables.

Software/Tools Required:

- MySQL / PostgreSQL
- dbdiagram.io (for schema design)

Theory / Background: Covers OLAP design, dimensional modeling, star/snowflake schemas, and SCD types.

Data Warehouse is used for analytical queries and stores historical data.

Common components:

- **Fact Table**: Central table storing measurements (e.g., sales).
- **Dimension Tables**: Contextual info (e.g., time, product, location).

Two main schemas:

- **Star Schema**: Fact table directly connected to dimensions.
- **Snowflake Schema**: Dimensions are normalized.

Also includes concepts like:

- **SCD (Slowly Changing Dimensions)** for tracking changes
- **ETL (Extract, Transform, Load)** for loading data

This practical introduces **OLAP (Online Analytical Processing)** concepts used in business intelligence.

Problem Statements:

- High: Full retail warehouse with SCD Type II
- Moderate: Small sales warehouse
- Poor: Single fact and dimension table

Procedure / Steps:

1. Identify facts and dimensions
2. Design star schema
3. Create tables and populate sample data
4. Apply historical tracking (if high level)

Expected Output:

- Fact and dimension tables created
- Star schema implemented
- SCD Type II if applicable

Result: Data warehouse schema implemented successfully

Conclusion: OLAP modeling supports analytical queries efficiently

Viva Questions:

- What is star vs snowflake schema?
- What is a slowly changing dimension?
- What is a fact table?

PRACTICAL NO. 8:

Aim: Introduction to NoSQL Data Modeling and Document Store

Objective: To design and interact with a NoSQL document-based database.

Software/Tools Required:

- MongoDB
- MongoDB Compass or Robo 3T
- Python (pymongo)

Theory / Background: Explains JSON-based modeling, flexibility in schema, and comparison with RDBMS.

NoSQL databases like **MongoDB** store data in flexible formats, commonly:

- **Document Store (JSON/BSON)**
- **Key-Value Store**
- **Column Store**
- **Graph Store**

In MongoDB:

- A **collection** is like a table.
- A **document** is like a row, but schema-less.

NoSQL excels in:

- Handling unstructured data
- High scalability
- Real-time apps

This practical introduces modern, scalable data solutions used in web/mobile apps.

Problem Statements:

- High: E-commerce app with nested orders
- Moderate: Student-course enrollment in JSON
- Poor: Basic address book in document form

Procedure / Steps:

1. Install MongoDB and connect via Compass
2. Design sample JSON structure
3. Insert documents via Compass or Python
4. Query data using filters

Expected Output:

- JSON documents created and inserted
- Collections formed with embedded documents

Result: Document store populated and queried successfully

Conclusion: NoSQL is ideal for flexible, schema-less applications

Viva Questions:

- What is a document database?
- Compare MongoDB with MySQL
- How does JSON help in NoSQL?
