DATABASE
DOCUMENT
SELECT * FROM users
WHERE is_active = 1;

Table of Contents

Introduction 1
Purpose 1
Scope 1
Procedure 1
Brief Problem 1
Overview 2
1. Meaning of Database 2
2. Meaning of DBMS (Database Management System) 3
3. Meaning of Database System 4
4. Advantages of Databases 5
5. Disadvantages of Databases 5
6. Three-Schema Architecture 6
7. Database Environments Comparison 8
8. Database Users 10
1. System Analyst 10
2. Database Designer 11
3. Database Administrator (DBA) 12
4. Application Programmer / Developer 12
SQL 13
Categories of SQL Commands 13
SQL Data Types 14
SQL Constraints 14
Primary Key & Foreign Key relationship 15
Creation order (best practice): 15
Deletion rules: 15
Sample Database: “SchoolDB” 16
DDL (Data Definition Language) 17
1. CREATE 17
2. ALTER 19
3. DROP 20
4. TRUNCATE 20
5. RENAME 21
DML (Data Manipulation Language) 22
1. INSERT 22

2. UPDATE 23
3. DELETE 23
4. SELECT (DQL) 24
5. Advanced Select (DQL) 25
DCL (Data Control Language) 38
1. GRANT 38
2. REVOKE 39
Modeling *40*
Example ERD: University System 40
Entities and Attributes 41
Attributes 41
Weak Entity 41
Relationships 42
a) Degree of Relationship 42
b) Cardinality (Multiplicity) 42
c) Participation 43
Mapping ERD to Relational Database 44
Rule 1: Mapping Strong Entities 44
Rule 2: Mapping Weak Entities 44
Rule 3: Mapping Relationships (Binary-Unary) 45
Rule 4: Mapping Ternary Relationships 47
Rule 5: Mapping Attributes 47
ERD After Mapping 48
Normalization 49
First Normal Form (1NF) 49
Second Normal Form (2NF) 50
Third Normal Form (3NF) 51
Specialization and Generalization 52
Conclusion *59*
Feedback & Contribution *59*
Copyright & Usage *59*
Acknowledgments *60*

Introduction

Purpose

The purpose of this document is to provide a comprehensive overview of **database systems**, their design principles, and the implementation of relational databases using **SQL (Structured Query Language)**. It aims to bridge the conceptual understanding of how data is modeled, stored, and managed, with the practical skills required to design and manipulate databases efficiently.

Scope

This document covers the entire lifecycle of database development — beginning from fundamental definitions and architecture, exploring the roles of database users, and then moving into the **SQL language**, including Data Definition, Manipulation, and Control commands.
It also delves into **Entity–Relationship Diagrams (ERD)**, explaining entities, attributes, relationships, and the process of mapping conceptual designs into relational schemas.
Additionally, the document explains **normalization** techniques (1NF, 2NF, 3NF) for optimizing data structure and minimizing redundancy, and concludes with **advanced concepts** like specialization, generalization, and entity hierarchies.

Procedure

The development process followed a structured methodology:

1. **Understanding the problem domain** — A university database was selected to represent real-world entities such as students, courses, and enrollments.
2. **Conceptual design** — Using an ERD to visualize entities, their attributes, and relationships (including weak entities and participation types).
3. **Logical design and mapping** — Translating the ERD into relational tables following standard mapping rules.
4. **Implementation** — Creating and manipulating the database using SQL commands, testing with example data, and displaying the resulting outputs.
5. **Normalization and optimization** — Applying normalization rules to ensure data integrity, eliminate redundancy, and improve performance.
6. **Advanced modeling** — Exploring hierarchy and inheritance concepts (subclasses and superclasses) using various mapping options.

Brief Problem

Modern organizations deal with large volumes of interrelated data that must be stored, accessed, and updated efficiently. Without a structured database system, data inconsistency, redundancy, and loss of integrity can occur.

Overview

1. Meaning of Database

A **database** is an organized collection of data that is stored and managed so that it can be easily accessed, updated, and analyzed.
Instead of keeping information in random files or spreadsheets, a database provides a structured way to store data in **tables**, which consist of **rows (records)** and **columns (fields)**.

For example, in a university database:

- A table called **Students** may store student IDs, names, and courses.
- A table called **Courses** may store course codes, names, and instructors.

Databases are designed to handle large volumes of data efficiently and to support multiple users at the same time, ensuring data integrity, consistency, and security.

SALES

purchase_number	date_of_purchase	customer_id	item_code
1	03/09/2016	1	A_1
2	02/12/2016	2	C_1
3	15/04/2017	3	D_1
4	24/05/2017	1	B_2
5	25/05/2017	4	B_2
6	06/06/2017	2	B_1
7	10/06/2017	4	A_2
8	13/06/2017	3	C_1
9	20/07/2017	1	A_1
10	11/08/2017	2	B_1

2. Meaning of DBMS (Database Management System)

A **Database Management System (DBMS)** is the **software** that allows users to define, create, maintain, and control access to the database.
It acts as an **interface** between the user (or application) and the database itself.

Functions of a DBMS:

- **Data Definition:** Create and modify database structures (tables, indexes, views, etc.).
- **Data Manipulation:** Insert, update, delete, and retrieve data.
- **Data Security:** Manage user permissions and access control.
- **Backup and Recovery:** Protect data from loss or corruption.
- **Concurrency Control:** Ensure correct results when multiple users access data simultaneously.
- **Integrity Enforcement:** Maintain correctness and consistency of stored data.

Examples:

- MySQL
- Oracle Database
- Microsoft SQL Server
- PostgreSQL
- MongoDB (for NoSQL data)

4 types of database management systems

1. **Relational database management system**
   Stores data in separate tables consisting of rows and columns. All tables are linked using data relationships.
2. **Object-oriented database management system**
   Stores data in the form of objects and offers high data control when connecting the DBMS with other business applications.
3. **Hierarchical database management system**
   Organizes data into a hierarchical structure, with each level representing a different category of information.
4. **Network database management system**
   Stores, retrieves, and manages data within a networked environment. It ensures data is consistent across network-connected devices.

3. Meaning of Database System

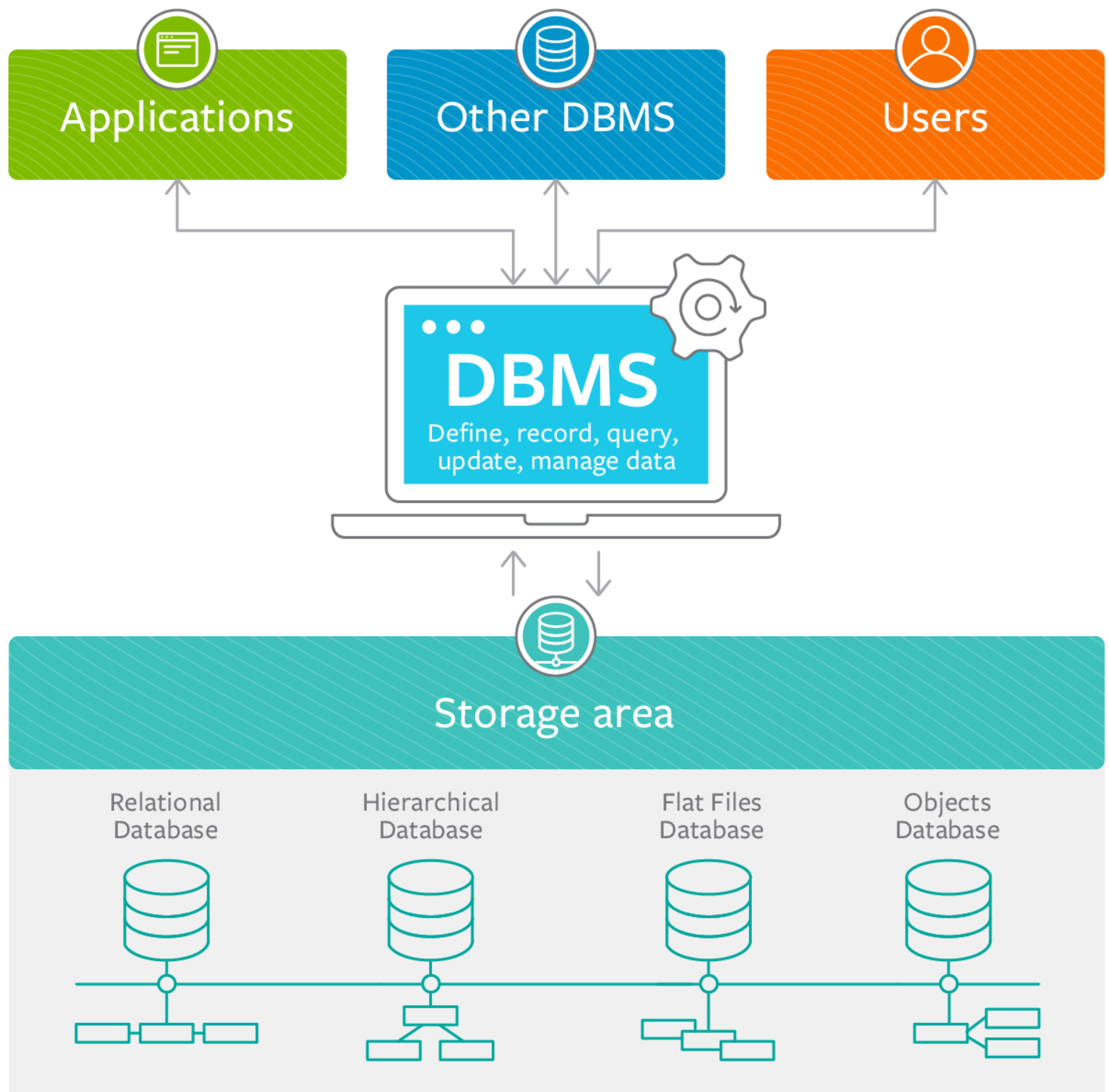
A **database system** is a broader concept that includes:

1. The **database** itself (the data),
2. The **DBMS software**, and
3. The **applications and users** that interact with it.

So:

Database System = Database + DBMS + Application Programs + Users

It represents the complete environment used to store, process, and manage data efficiently.



Made by Youssef Amgad Elkhatib

4. Advantages of Databases

Advantage	Explanation
Data Redundancy Control	Data is stored in one central location, minimizing duplication.
Data Consistency	Since data is not duplicated, updates are consistent across all users and applications.
Data Sharing	Authorized users can share data concurrently in a controlled manner.
Data Integrity	Constraints (e.g., primary keys, foreign keys) ensure accuracy and correctness.
Data Security	Access control, encryption, and authentication protect sensitive data.
Backup and Recovery	Automatic backup and recovery systems prevent data loss.
Data Independence	Changes in data structure do not affect application programs (logical and physical independence).
Efficient Query Processing	DBMS optimizes data retrieval and manipulation.

---

5. Disadvantages of Databases

Disadvantage	Explanation
High Cost	Setting up and maintaining a DBMS requires expensive software and hardware.
Complexity	Designing and managing databases requires skilled professionals.
Performance Overhead	DBMS adds additional layers between user and data, which may slow performance for simple tasks.
Single Point of Failure	Centralized databases may be affected if the main server fails.
Security Risks	Centralized storage increases the impact of data breaches if not properly secured.

6. Three-Schema Architecture

The **three-schema architecture** was introduced by ANSI/SPARC to separate the database structure into three levels — providing **data abstraction** and **independence**.

a. Internal Schema (Physical Level):

- Describes **how data is physically stored** on hardware.
- Includes file organization, indexes, data compression, and access paths.
- Concerned with efficiency, performance, and storage optimization.

b. Conceptual Schema (Logical Level):

- Describes the **logical structure** of the entire database for the community of users.
- Defines entities, attributes, relationships, and constraints.
- Independent of physical storage — focuses on meaning and relationships.

c. External Schema (View Level):

- Describes **how individual users** or applications view the data.
- Provides customized views (subsets) of the database for security and simplicity.
- Hides irrelevant details from each user.

Benefits:

- Data independence (logical and physical)
- Security via user views
- Simplified management and scalability

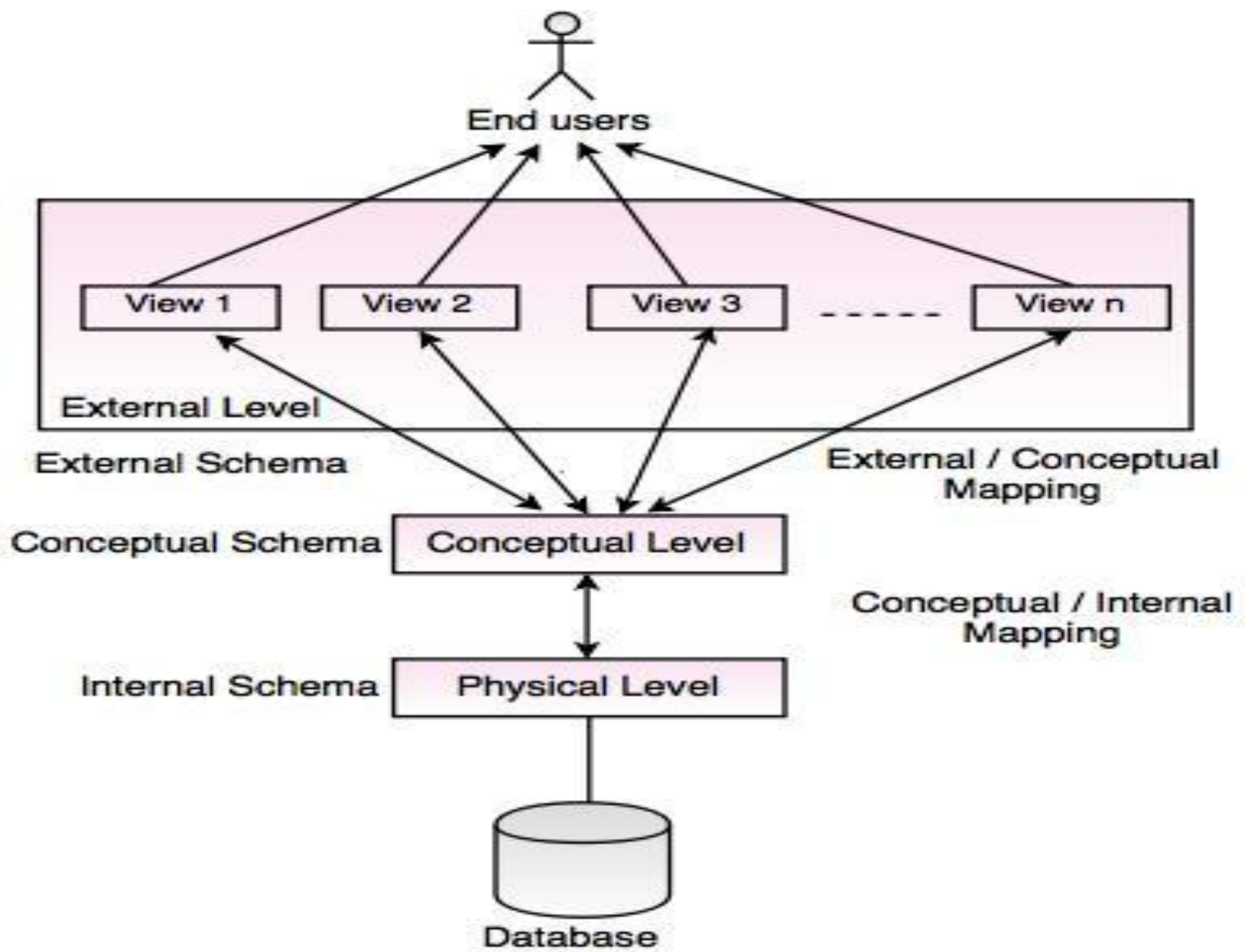
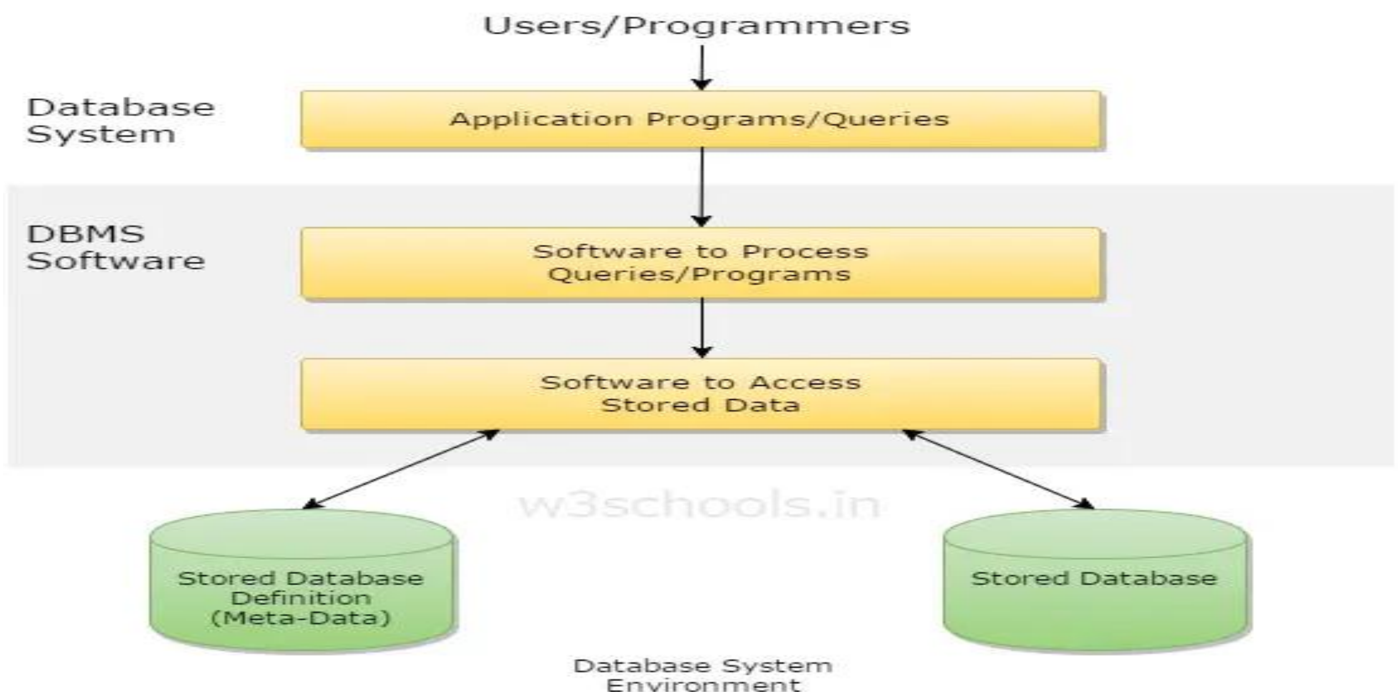
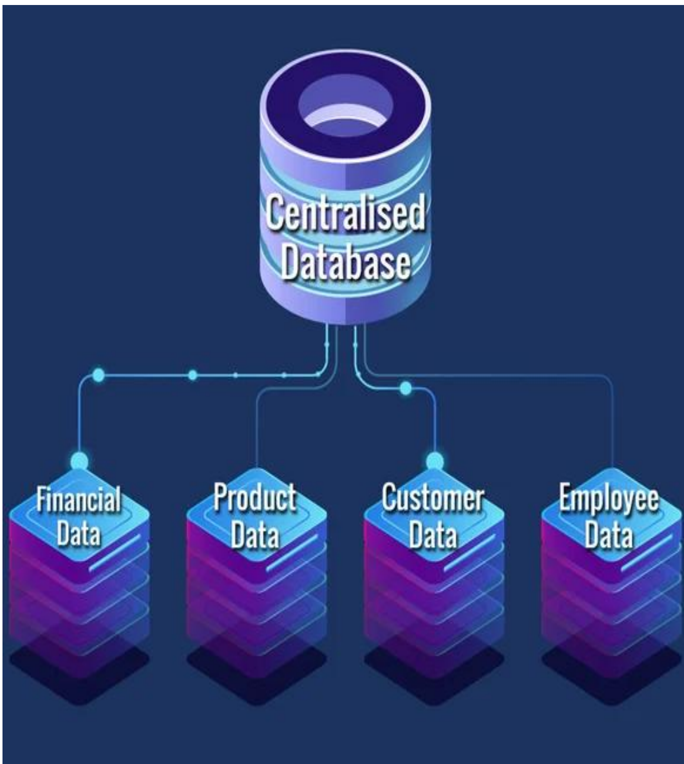


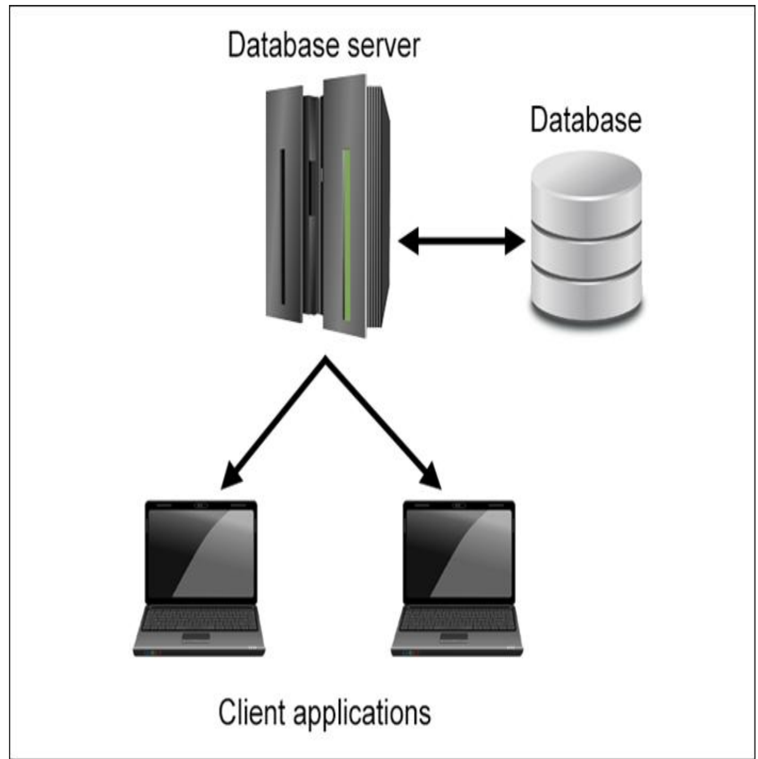
Fig. Three Level Architechture of DBMS

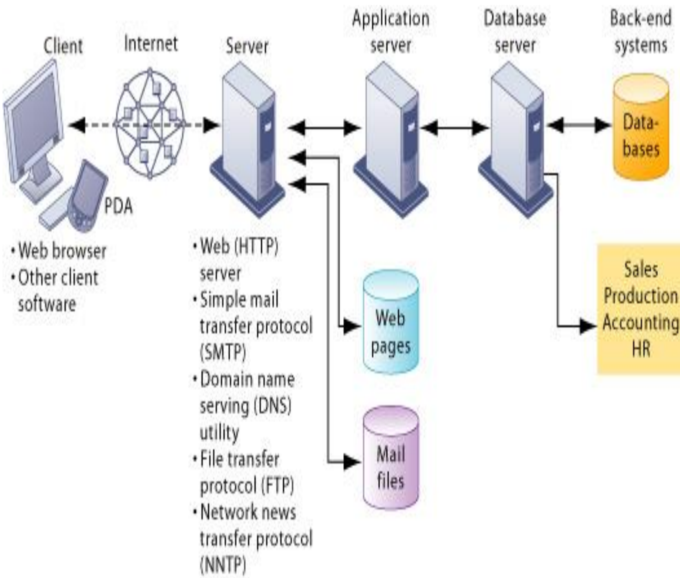
7. Database Environments Comparison

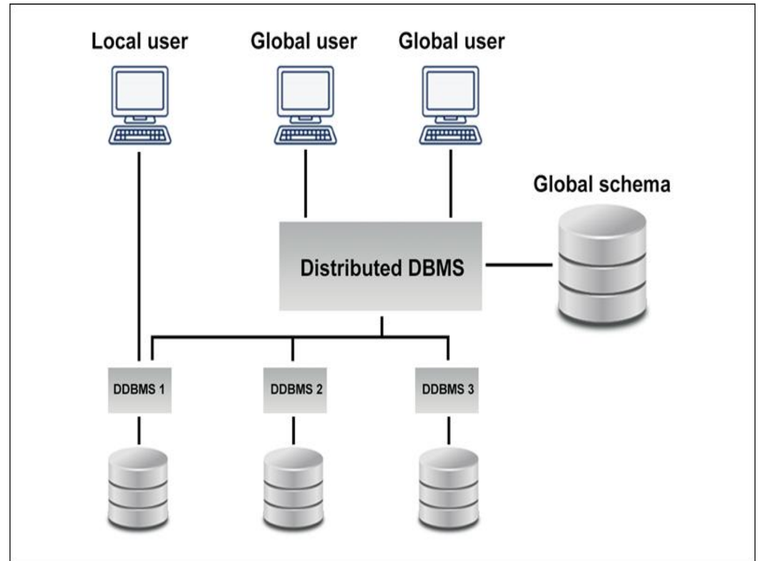
Aspect	Centralized Database	Client-Server Database	Distributed Database	Internet (Web-Based) Database
Location of Data	All data stored on one central server	Data stored on a central server, accessed by client applications(Thick Client)	Data distributed across multiple sites	Data stored on web servers, accessed via browsers
Accessibility	Local or limited remote access	Local network (LAN) or limited WAN	Multiple geographical locations	Global access via the Internet
Performance	Fast for small systems but can bottleneck under heavy load	Balanced workload between client and server	High scalability; depends on network reliability	Depends on Internet speed and server performance
Reliability	Risk of total failure if central server crashes Database, Application layer are single points of failure	Partial failure may still affect clients Database is a single point of failure	Higher fault tolerance (replication, fragmentation possible)	Depends on cloud/web infrastructure Database, Application layer is single point of failure (N-tier archetichture has multiple application servers to avoid single point of failure




Centralised Database
Financial Data
Product Data
Customer Data
Employee Data


Database server
Database
Client applications


Client
Internet
Server
Application server
Database server
Back-end systems
Data-bases
PDA
• Web browser
• Other client software
• Web (HTTP) server
• Simple mail transfer protocol (SMTP)
• Domain name serving (DNS) utility
• File transfer protocol (FTP)
• Network news transfer protocol (NNTP)
Web pages
Mail files
Sales
Production
Accounting
HR


Local user
Global user
Global user
Global schema
Distributed DBMS
DDBMS 1
DDBMS 2
DDBMS 3

8. Database Users

1. System Analyst

Role:

The **system analyst** acts as a bridge between the organization's business requirements and the technical database system.
They analyze what the organization needs and translate those needs into **functional specifications** for the database and applications.

Responsibilities:

- Study existing systems and identify problems or areas for improvement.
- Gather and document user requirements.
- Design the **overall system flow**, including how data will move between processes.
- Communicate with database designers and developers to ensure the database supports business operations.
- Ensure that the system meets performance, usability, and reliability goals.

2. Database Designer

Role:

The **database designer** is responsible for **creating the logical and physical structure** of the database — determining how data will be organized, stored, and related.

There are usually two stages of design:

- **Logical Design:** Define entities, attributes, relationships, and constraints using models like **ER (Entity-Relationship)** or **Relational Schema**.
- **Physical Design:** Decide how to implement the logical design in the DBMS — including **storage structures, indexes, partitions**, etc.

Responsibilities:

- Identify the entities and relationships needed to represent real-world data.
- Create a **conceptual schema** and normalize it to reduce redundancy.
- Define **primary and foreign keys**, and integrity constraints.
- Optimize database structure for performance and scalability.
- Collaborate with the DBA to implement the database on a specific DBMS.

3. Database Administrator (DBA)

Role:
The **DBA** is the **manager and custodian** of the database system.
They are responsible for maintaining, securing, and ensuring the database operates efficiently and reliably once it's implemented.

Responsibilities:

- **Database Installation & Configuration:** Set up the DBMS software and tune it for performance.
- **User Management:** Create user accounts, assign roles, and control access permissions.
- **Security:** Implement authentication, authorization, and encryption mechanisms.
- **Backup & Recovery:** Plan and execute data backup and recovery strategies to prevent data loss.
- **Performance Tuning:** Monitor system performance, optimize queries, and allocate resources.
- **Maintenance:** Apply patches, upgrades, and monitor logs for potential issues.
- **Data Integrity & Consistency:** Enforce constraints and ensure correctness of stored data.

---

4. Application Programmer / Developer

Role:
The **application programmer** (or **developer**) writes **application programs** that interact with the database through queries and APIs.
They focus on how users will access and manipulate the data via software interfaces.

Responsibilities:

- Write programs using SQL (Structured Query Language) or APIs to read/write data.
- Design and develop front-end interfaces (e.g., forms, web pages) for user interaction.
- Implement business logic that uses data stored in the database.
- Optimize application performance with efficient database queries.
- Test and debug database-related functionality.

SQL

SQL (Structured Query Language) is the **standard language** used to manage and manipulate relational databases.
It is used to:

- **Define** database structures (tables, schemas, etc.),
- **Insert, update, delete, and query** data,
- **Control access and permissions** to the database.

SQL is **declarative**, meaning you specify *what* you want to do (e.g., "get all students whose grade > 80") and the DBMS figures out *how* to do it.

Categories of SQL Commands

SQL commands are classified into **five major categories:**

Category	Full Form	Purpose
DDL	Data Definition Language	Define and modify database structures (tables, schemas, indexes, etc.)
DML	Data Manipulation Language	Manage data stored in database tables (insert, update, delete, query)
DCL	Data Control Language	Control access and permissions for users
TCL	Transaction Control Language	Manage transactions (commit, rollback, savepoint)
DQL	Data Query Language	Retrieve data from tables (SELECT command)

SQL Data Types

Different DBMSs (like MySQL, PostgreSQL, SQL Server) have slightly different data types, but here are the common ones:

Category	Data Type	Description
Numeric	INT, DECIMAL(p,s), FLOAT, DOUBLE	Store numbers (integer or decimal)
Character/String	CHAR(n), VARCHAR(n), TEXT	Store text of fixed or variable length
Date/Time	DATE, TIME, DATETIME, TIMESTAMP	Store date and/or time values
Boolean	BOOLEAN	True or False values
Binary	BLOB	Store binary large objects (e.g., images, files)

SQL Constraints

Constraints are **rules** applied to table columns to maintain data integrity and accuracy.

Constraint	Description
PRIMARY KEY	Uniquely identifies each record in a table
FOREIGN KEY	Ensures referential integrity between related tables
NOT NULL	Ensures a column cannot have NULL values
UNIQUE	Ensures all values in a column are different
CHECK	Ensures data meets a specific condition
DEFAULT	Assigns a default value if no value is provided

Primary Key & Foreign Key relationship

Creation order (best practice):

1. **Create parent table** (the table holding the PRIMARY KEY) — e.g., Students.
2. **Create child table** and add FOREIGN KEY referencing parent — e.g., Enrollments → Students(StudentID).

Deletion rules:

- **Default (restrict):** if Enrollments has a FK referencing Students without cascade, trying to delete the parent row fails:

```
DELETE FROM Students WHERE StudentID = 1;
-- ERROR: delete violates foreign key constraint FK_Enrollments_Students
```

- **ON DELETE CASCADE:** make FK delete related child rows automatically:

```
ALTER TABLE Enrollments
DROP CONSTRAINT FK_Enrollments_Students; -- if exists, then
ALTER TABLE Enrollments
ADD CONSTRAINT FK_Enrollments_Students
FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
ON DELETE CASCADE;
```

Now:

```
DELETE FROM Students WHERE StudentID = 1;
```

Effect / expected result

- The Students row with StudentID=1 is removed.
- Any Enrollments rows with StudentID=1 are also removed automatically.
- After deletion, SELECT * FROM Enrollments; would show only enrollments for StudentID 2 & 3.

ON UPDATE CASCADE: similarly, if you change a parent PK value (rare but possible), ON UPDATE CASCADE updates child FK values.

We can also use Set Null, Set Default when updating or deleting.

Sample Database: “SchoolDB”

We’ll use this **School Database (SchoolDB)** throughout the document.

Entities (Tables):

1. **Students**
2. **Courses**
3. **Enrollments**

Table 1: Students

StudentID	FirstName	LastName	Gender	Age
1	Ali	Hassan	M	21
2	Sara	Nabil	F	20
3	Omar	Adel	M	22

Table 2: Courses

CourseID	CourseName	CreditHours
C101	Database Systems	3
C102	Programming Fundamentals	4
C103	Data Structures	3

Table 3: Enrollments

EnrollmentID	StudentID	CourseID	Grade
1	1	C101	A
2	2	C102	B
3	3	C103	A

DDL (Data Definition Language)

DDL commands **define and modify** the structure of the database.

Main Commands:

- CREATE
- ALTER
- DROP
- TRUNCATE
- RENAME

---

1. CREATE

Purpose:
Used to create databases, tables, views, or indexes.

Syntax:

```
CREATE TABLE table_name (
    column_name datatype constraint,
    ...
);
```

Example (create Students table):

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    [First Name] VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) Default "Ahmed",
    Gender CHAR(1) CHECK (Gender IN ('M', 'F')),
    Age INT Unique
);
```

Result:
A table named Students is created with defined columns and constraints.

If you want a column name that is a reserved word (e.g., Order), quote it:

```
CREATE TABLE "OrderTable" (
  "OrderID" INT PRIMARY KEY,
  "Order" VARCHAR(50)
);
```

(Use double-quotes for standard SQL / Postgres, square brackets [Order] for SQL Server, backticks `Order` for MySQL.)

If you want an auto Primary Key value, use IDENTITY

```
CREATE TABLE StudentsAuto (
  StudentID INT IDENTITY(1,1) PRIMARY KEY,
  FirstName VARCHAR(50),
  LastName VARCHAR(50),
  Gender CHAR(1),
  Age INT
);
```

(For MySQL use AUTO_INCREMENT, for Postgres use SERIAL or GENERATED ... AS IDENTITY.)

2. ALTER

Purpose:

Modify an existing table (add, drop, or modify columns).

Syntax:

```
ALTER TABLE table_name
ADD column_name datatype;
```

Example:

```
ALTER TABLE Students
ADD Email VARCHAR(100);
```

Result:

A new column Email is added to Students.

```
ALTER TABLE Enrollments
ADD CONSTRAINT FK_Enrollments_Students
FOREIGN KEY (StudentID) REFERENCES Students(StudentID);
```

Result: DB now enforces referential integrity: every StudentID in Enrollments must exist in Students.

3. DROP

Purpose:
Delete a table or database permanently.

Syntax:

```
DROP TABLE table_name;
```

Example:

```
ALTER TABLE Enrollments
Drop column Grade

DROP TABLE Enrollments;
```

Result:

The Column Grade is removed from the Enrollments table.

The Enrollments table is permanently removed from the database.

---

4. TRUNCATE

Purpose:
Remove all rows from a table **without deleting the table structure**.

Syntax:

```
TRUNCATE TABLE table_name;
```

Example:

```
TRUNCATE TABLE Students;
```

Result:
All student records are deleted, but the Students table still exists.

5. RENAME

Purpose:

Change the name of a table.

Syntax:

```
RENAME TABLE old_name TO new_name;
```

Example:

```
RENAME TABLE Students TO StudentInfo;
```

Result:

Table name changes from Students to StudentInfo.

DML (Data Manipulation Language)

DML commands manipulate data **inside tables**.

Main Commands:

- INSERT
- UPDATE
- DELETE
- SELECT (also called DQL)

---

1. INSERT

Purpose:
Add new records into a table.

We can insert values directly without specifying order but we take into account order and all columns must be filled with a value.

Syntax:

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

Example:

```
INSERT INTO Students (StudentID, FirstName, LastName, Gender, Age)
VALUES (4, 'Laila', 'Fouad', 'F', 23);
```

Result:
A new student record is added to Students.

2. UPDATE

Purpose:

Modify existing records.

Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2
WHERE condition;
```

Example:

```
UPDATE Students
SET Age = 22
WHERE StudentID = 2;
```

Result:

Sara Nabil's age is updated to 22.

---

3. DELETE

Purpose:

Remove specific records.

If no condition is applied, all rows are deleted.

Syntax:

```
DELETE FROM table_name
WHERE condition;
```

Example:

```
DELETE FROM Students
WHERE StudentID = 3;
```

Result:

Omar Adel's record is deleted from Students.

4. SELECT (DQL)

Purpose:
Retrieve data from tables.

Without a WHERE condition, the entire column is selected.

SELECT * -> Selects all columns.

Syntax:

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

Example:

```
SELECT FirstName, LastName, Age
FROM Students
WHERE Gender = 'F';
```

Result:

FirstName	LastName	Age
Sara	Nabil	22
Laila	Fouad	23

5. Advanced Select (DQL)

1) Logical operators (AND, OR, NOT, IN, BETWEEN)

```
-- students older than 20 and male
SELECT FirstName, LastName, Age, Gender
FROM Students
WHERE Age > 20 AND Gender = 'M';
```

Result

FirstName	LastName	Age	Gender
Ali	Hassan	21	M
Omar	Adel	22	M

```
-- Age between 20 and 22 OR female
SELECT FirstName, Age, Gender
FROM Students
WHERE Age BETWEEN 20 AND 22 OR Gender = 'F';
```

Result

FirstName	Age	Gender
Ali	21	M
Sara	20	F
Omar	22	M

<> -> not equal

2) LIKE (pattern match)

% -> any string of zero or more characters

- -> any single character

```
SELECT FirstName, LastName
FROM Students
WHERE FirstName LIKE 'A%';
```

Result

FirstName	LastName
Ali	Hassan

---

3) Column & Table Aliases

```
SELECT
  StudentID AS id,
  CONCAT(FirstName, ' ', LastName) AS FullName,
  Age AS Years
FROM Students;
```

Result

id	FullName	Years
1	Ali Hassan	21
2	Sara Nabil	20
3	Omar Adel	22

(If your DBMS uses + for concat, use FirstName + ' ' + LastName.)

4) ORDER BY

```
SELECT FirstName, LastName, Age
FROM Students
ORDER BY Age DESC, LastName ASC;
```

Result

FirstName	LastName	Age
Omar	Adel	22
Ali	Hassan	21
Sara	Nabil	20

---

5) DISTINCT

Selects Unique Values

```
SELECT DISTINCT Grade
FROM Enrollments;
```

Result

Grade
A
B

6) JOINs

INNER JOIN (students with their course and grade)

```
SELECT s.FirstName, s.LastName, c.CourseName, e.Grade
FROM Students s
INNER JOIN Enrollments e ON s.StudentID = e.StudentID
INNER JOIN Courses c ON e.CourseID = c.CourseID;
```

Result

FirstName	LastName	CourseName	Grade
Ali	Hassan	Database Systems	A
Sara	Nabil	Programming Fundamentals	B
Omar	Adel	Data Structures	A

LEFT JOIN

All students, with any enrollment.

```
SELECT s.FirstName, s.LastName, e.CourseID, e.Grade
FROM Students s
LEFT OUTER JOIN Enrollments e ON s.StudentID = e.StudentID;
```

Result (same as inner here because every student has 1 enrollment)

FirstName	LastName	CourseID	Grade
Ali	Hassan	C101	A
Sara	Nabil	C102	B
Omar	Adel	C103	A

RIGHT JOIN (all enrollments, show course info)

```sql
SELECT c.CourseName, e.Grade
FROM Courses c
RIGHT OUTER JOIN Enrollments e ON c.CourseID = e.CourseID;
```

Result

CourseName	Grade
Database Systems	A
Programming Fundamentals	B
Data Structures	A

FULL OUTER JOIN (the union of left+right)

```sql
SELECT s.FirstName, e.CourseID, e.Grade
FROM Students s
FULL OUTER JOIN Enrollments e ON s.StudentID = e.StudentID;
```

Result

FirstName	CourseID	Grade
Ali	C101	A
Sara	C102	B
Omar	C103	A

CROSS JOIN (cartesian product)

```
SELECT s.FirstName, c.CourseName
FROM Students s
CROSS JOIN Courses c;
```

Result (3 students × 3 courses = 9 rows)

FirstName	CourseName
Ali	Database Systems
Ali	Programming Fundamentals
Ali	Data Structures
Sara	Database Systems
Sara	Programming Fundamentals
Sara	Data Structures
Omar	Database Systems
Omar	Programming Fundamentals
Omar	Data Structures

SELF JOIN

Find students who share the same grade via enrollments

```
SELECT s1.FirstName AS Student1, s2.FirstName AS Student2, e1.Grade
FROM Enrollments e1
JOIN Enrollments e2
  ON e1.Grade = e2.Grade AND e1.StudentID < e2.StudentID
JOIN Students s1 ON e1.StudentID = s1.StudentID
JOIN Students s2 ON e2.StudentID = s2.StudentID;
```

Result (pairs of students with same grade; e1.studentid < e2.studentid avoids duplicate pairs)

Student1	Student2	Grade
Ali	Omar	A

7) Subqueries

a. Simple IN subquery

students enrolled in "Database Systems"

```
SELECT FirstName, LastName
FROM Students
WHERE StudentID IN (
  SELECT StudentID FROM Enrollments WHERE CourseID =
    (SELECT CourseID FROM Courses WHERE CourseName = 'Database Systems')
);
```

Result

FirstName	LastName
Ali	Hassan

---

b. Correlated subquery (EXISTS, NOT EXISTS)

(example: count existence)

```
-- Example of correlated subquery: list students who have any grade = 'A'
SELECT s.FirstName, s.LastName
FROM Students s
WHERE EXISTS (
  SELECT 1 FROM Enrollments e
  WHERE e.StudentID = s.StudentID AND e.Grade = 'A'
);
```

Result

FirstName	LastName
Ali	Hassan
Omar	Adel

8) Aggregation functions (COUNT, AVG, MIN, MAX, SUM)

```
SELECT
  COUNT(*) AS TotalStudents, # Count(*) -> Counts null values as well
  AVG(Age)   AS AvgAge,
  MIN(Age)   AS Youngest,
  MAX(Age)   AS Oldest
FROM Students;
```

Result

TotalStudents	AvgAge	Youngest	Oldest
3	21.0	20	22

```
SELECT Grade, COUNT(*) AS NumStudents
FROM Enrollments
GROUP BY Grade;
```

Result

Grade	NumStudents
A	2
B	1

```
-- grades with more than 1 student
SELECT Grade, COUNT(*) AS NumStudents
FROM Enrollments
GROUP BY Grade
HAVING COUNT(*) > 1;
```

Result

Grade	NumStudents
A	2

10) VIEW (virtual table)

```
CREATE VIEW StudentCourseView AS
SELECT s.StudentID, CONCAT(s.FirstName, ' ', s.LastName) AS FullName,
    c.CourseName, e.Grade
FROM Students s
JOIN Enrollments e ON s.StudentID = e.StudentID
JOIN Courses c ON e.CourseID = c.CourseID;
```

```
-- read from the view
SELECT * FROM StudentCourseView;
```

Result (view rows)

StudentID	FullName	CourseName	Grade
1	Ali Hassan	Database Systems	A
2	Sara Nabil	Programming Fundamentals	B
3	Omar Adel	Data Structures	A

11) TOP / LIMIT — get first N rows

SQL Server:

```
SELECT TOP 2 FirstName, Age FROM Students ORDER BY Age DESC;
```

MySQL / Postgres:

```
SELECT FirstName, Age FROM Students ORDER BY Age DESC LIMIT 2;
```

Result (top 2 by Age)

FirstName	Age
Omar	22
Ali	21

---

12) INSERT INTO ... SELECT (copy rows from one table to another)

```
-- create a backup table
CREATE TABLE StudentsBackup (
  StudentID INT,
  FirstName VARCHAR(50),
  LastName VARCHAR(50),
  Gender CHAR(1),
  Age INT
);
```

```
INSERT INTO StudentsBackup (StudentID, FirstName, LastName, Gender, Age)
SELECT StudentID, FirstName, LastName, Gender, Age FROM Students;

SELECT * FROM StudentsBackup;
```

Result

StudentID	FirstName	LastName	Gender	Age
1	Ali	Hassan	M	21
2	Sara	Nabil	F	20
3	Omar	Adel	M	22

13) SET Operations

UNION

```
-- two queries with overlap
SELECT FirstName FROM Students WHERE Age >= 21
UNION
SELECT FirstName FROM Students WHERE StudentID IN (1,2);
```

Explanation: UNION removes duplicates.

Result

FirstName
Ali
Omar
Sara

UNION ALL

```
-- UNION ALL keeps duplicates
SELECT FirstName FROM Students WHERE Age >= 21
UNION ALL
SELECT FirstName FROM Students WHERE StudentID IN (1,2);
```

Result (duplicates where they exist)

FirstName
Ali
Omar
Ali
Sara

INTERSECT

```
-- INTERSECT (rows common to both)
SELECT FirstName FROM Students WHERE Age >= 21
INTERSECT
SELECT FirstName FROM Students WHERE StudentID IN (1,2);
```

Result

FirstName
Ali

(Note: INTERSECT is not supported in MySQL older versions; behavior varies by DBMS.)

EXCEPT

```
--  EXCEPT (returns rows from the first query that do not appear in the second query)
SELECT StudentName FROM Students WHERE StudentID <= 4
EXCEPT
SELECT StudentName FROM Students WHERE GPA < 3.0;
```

Result

StudentName
Ali
Sara
Mona

DCL (Data Control Language)

DCL commands manage **user access and permissions**.

Main Commands:

- GRANT
- REVOKE

---

1. GRANT

Purpose:
Give privileges to users.

Syntax:

```
GRANT privilege_name
ON object_name
TO user_name;
```

Example:

```
GRANT SELECT, INSERT
ON Students
TO 'teacher_user';
```

Result:
teacher_user can now view and add records to the Students table.

Example:

```
GRANT SELECT
ON Students
TO Mariam
With GRANT option;
```

Result:

Mariam can now select from the Students table with the ability to grant this privilege to other people

2. REVOKE

Purpose:

Remove privileges from users.

Syntax:

```
REVOKE privilege_name
ON object_name
FROM user_name;
```

Example:

```
REVOKE INSERT
ON Students
FROM 'teacher_user';
```

Result:

teacher_user loses permission to insert records.

Example:

```
REVOKE All
On Students
FROM “Ahmed”;
```

Result:

Removes all privileges from Ahmed on the Students table.

Modeling

An **ERD** (Entity–Relationship Diagram) is a visual model that represents the **logical structure of data** in a database.
It shows:

- **Entities** (objects to store information about),
- **Attributes** (properties of entities),
- **Relationships** (how entities are linked).

Example ERD: University System

We'll use a **university** example throughout (same context as our SQL examples):

Entities:

- **Student**
- **Course**
- **Instructor**

Relationships:

- Students **enroll** in Courses
- Instructors **teach** Courses

Here's the conceptual ERD (visualized in text form):

[STUDENT] —< ENROLLMENT >— [COURSE] —< TAUGHT_BY >— [INSTRUCTOR]

Entities and Attributes

Attributes

Type	Description	Example in University DB
Simple Attribute	Cannot be divided further.	StudentName, CourseName, CourseCode
Composite Attribute	Can be divided into smaller subparts.	StudentAddress → (Street, City, PostalCode)
Derived Attribute	Calculated from other attributes.	StudentAge (derived from DateOfBirth)
Multivalued Attribute	Can have multiple values for one entity.	PhoneNumbers (a student may have more than one)
Key Attribute	Uniquely identifies each entity.	StudentID, CourseID

Weak Entity

A **weak entity** cannot be uniquely identified by its own attributes alone — it depends on another entity (called the **owner** or **strong entity**) through a **foreign key** and a **partial key**.

In our case:

Entity	Description	Key Attributes	Other Attributes
Student	Information about students	StudentID (PK)	Name, Age, Gender
Course	Information about courses	CourseID (PK)	Title, Credits
Instructor	Information about instructors	InstructorID (PK)	Name, Department

Relationships

a) Degree of Relationship

- **Unary (1-entity)** → relationship within the same entity (e.g., Employee supervises Employee).
- **Binary (2-entities)** → most common (e.g., Student enrolls in Course).
- **Ternary (3-entities)** → relationship among three entities.

Our example includes **binary relationships**:

- Student ↔ Enrollment ↔ Course
- Instructor ↔ Course

---

b) Cardinality (Multiplicity)

Specifies **how many instances** of one entity relate to instances of another:

Type	Description	Example
1:1	One entity instance relates to only one of the other	Department has one Head
1:M	One entity relates to many of another	Instructor teaches many Courses
M:N	Many relate to many	Students enroll in many Courses

In our case:

- Student ↔ Course is **M:N**
- Instructor ↔ Course is **1:M**

c) Participation

Defines whether **all instances** of an entity participate in the relationship:

Type	Meaning	Example
Total Participation/ Must (2 lines)	Every entity instance must be related	Every Enrollment must have a Student
Partial Participation/ May (one line)	Some instances may not be related	Some Students might not enroll yet

Example:

- Every **Enrollment** must belong to a **Student** (Total)
- A **Student** may or may not have **Enrollment** records (Partial)

Mapping ERD to Relational Database

We now transform the ERD into **tables** using **ER Mapping Rules**.

Rule 1: Mapping Strong Entities

Each strong entity becomes a **table**.

Example:

```
CREATE TABLE Student (
 StudentID INT PRIMARY KEY,
 Name VARCHAR(50),
 Age INT,
 Gender CHAR(1)
);
CREATE TABLE Course (
 CourseID INT PRIMARY KEY,
 Title VARCHAR(100),
 Credits INT
);
CREATE TABLE Instructor (
 InstructorID INT PRIMARY KEY,
 Name VARCHAR(50),
 Department VARCHAR(50)
);
```

Rule 2: Mapping Weak Entities

For each **weak entity**, create a new table that includes:

- All its own attributes.
- The **primary key of the owner entity** (as a foreign key).
- Together, they form the **composite primary key**.

Rule 3: Mapping Relationships (Binary-Unary)

(a) 1:1 Relationship

Participation Matters Here:

Mapping depends on **total** vs **partial participation**:

Case	Meaning	Mapping
Total participation (both entities must participate)	Every student must have a student card, and every card belongs to one student.	Combine both into one table (merge).
Partial participation (one side optional)	Some students may not yet have a card.	Keep separate tables and use the primary key from the **partially participating** side as a foreign key on the **totally participating** side.
Partial participation (Both sides)	A student may enroll in a course or not; a course might have no students enrolled in it.	Keep separate tables and use the primary key from one **partially participating** side as a foreign key on the **other partially participating** side.

Example – Partial Participation:

```
CREATE TABLE StudentCard (
   CardID INT PRIMARY KEY,
   IssueDate DATE,
   StudentID INT UNIQUE,
   FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
);
```

Here, StudentID is **unique** (1:1 relationship) but **nullable** — meaning not all students have a card yet.

(b) 1:M Relationship

Add the primary key of the "one" side as a foreign key to the "many" side.

Example:

Instructor teaches Courses → add InstructorID to Course:

```
ALTER TABLE Course
ADD InstructorID INT,
FOREIGN KEY (InstructorID) REFERENCES Instructor(InstructorID);
```

(c) M:N Relationship

Create a new **associative (bridge) table** containing both primary keys.

Example:

Students enroll in Courses → create **Enrollment** table:

```
CREATE TABLE Enrollment (
 EnrollmentID INT PRIMARY KEY,
 StudentID INT,
 CourseID INT,
 Grade CHAR(2),
 FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
 FOREIGN KEY (CourseID) REFERENCES Course(CourseID)
);
```

Rule 4: Mapping Ternary Relationships

Create a new table with:

- The **primary keys of all three entities** as **foreign keys**.
- Any **relationship-specific attributes**.

Example Mapping:

Table	Attributes	Primary Key	Foreign Keys
Teaches	InstructorID, CourseID, Semester, HoursPerWeek	(InstructorID, CourseID, Semester)	InstructorID, CourseID

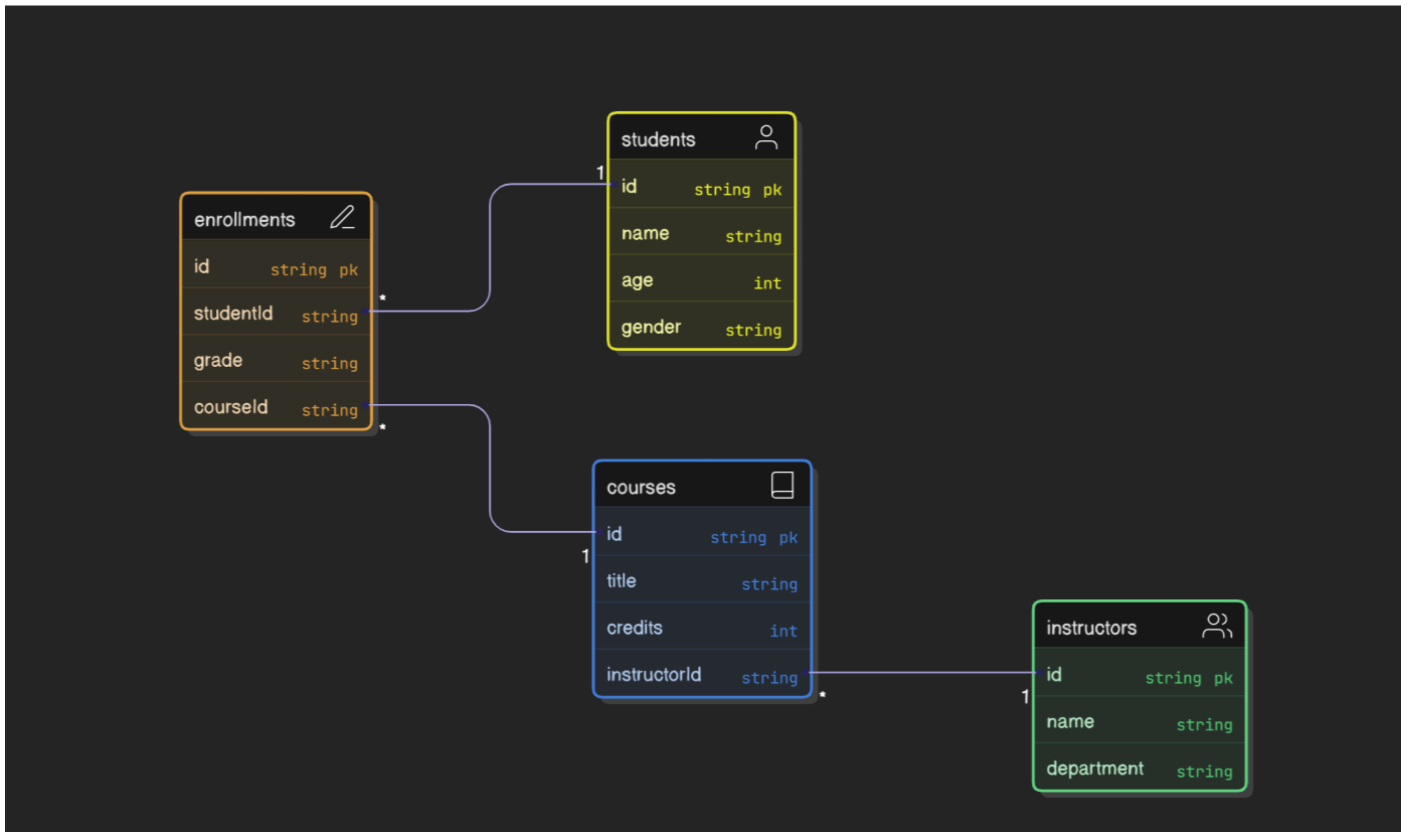
SQL Example:

```
CREATE TABLE Teaches (
  InstructorID INT,
  CourseID INT,
  Semester VARCHAR(10),
  HoursPerWeek INT,
  PRIMARY KEY (InstructorID, CourseID, Semester),
  FOREIGN KEY (InstructorID) REFERENCES Instructors(InstructorID),
  FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);
```

Rule 5: Mapping Attributes

- **Simple Attribute:** becomes a single column (e.g., Age INT).
- **Composite Attribute:** split into multiple columns (e.g., Address → Street, City, Zip).
- **Multivalued Attribute:** create a new table (e.g., StudentPhone(StudentID, PhoneNumber)).
- **Derived Attribute:** not stored, computed as needed (e.g., Age from DOB).

ERD After Mapping



Normalization

Normalization is the process of organizing data to reduce **redundancy** and improve **integrity**.
It's done in **Normal Forms (NF)** — each with stricter rules.

We'll use this unnormalized example:

StudentID	Name	Course1	Course2	Instructor	InstructorDept
1	Ali	DB	Networks	Dr. Samir	CS
2	Sara	DB	NULL	Dr. Samir	CS

First Normal Form (1NF)

Rule:

Each column must contain **atomic (indivisible)** values.
No repeating groups, No Composite or Multivalued Attributes

Split Composite, Multivalued attributes, and repeating groups into multiple columns.

Example

Split repeating Course columns into separate rows:

StudentID	Name	Course	Instructor	InstructorDept
1	Ali	DB	Dr. Samir	CS
1	Ali	Networks	Dr. Samir	CS
2	Sara	DB	Dr. Samir	CS

Second Normal Form (2NF)

Rule:

Be in 1NF, **and** all non-key attributes must depend on the **whole primary key, not part of it (no partial dependency)**.

Non-key attributes and key they partially depend on are placed in a new table with primary key they partially depend on remaining as the primary key in the second table.

Create separate tables for Students, Courses, and Instructors to remove partial dependency:

Students

StudentID	Name
1	Ali
2	Sara

Courses

CourseID	CourseName
1	DB
2	Networks

Instructors

InstructorID	Name	Department
1	Dr. Samir	CS

Enrollments

StudentID	CourseID	InstructorID
1	1	1
1	2	1
2	1	1

Third Normal Form (3NF)

Rule:

Be in 2NF and remove **transitive dependencies** (non-key attributes depending on other non-keys).

Leave the Independent attribute in first table as a foreign key, take it as a primary key with other attributes depending on it in a new table.

Example

Instructor's department depends on the **Instructor**, not the enrollment.
Move it to Instructor table only.

InstructorID	Name	Department
1	Dr. Samir	CS

Final normalized tables:

- **Student(StudentID, Name)**
- **Course(CourseID, CourseName)**
- **Instructor(InstructorID, Name, Department)**
- **Enrollment(StudentID, CourseID, InstructorID)**

Summary Diagram (Conceptual → Logical → Physical)

[ERD]

↓

[Tables via Mapping Rules]

↓

[Normalization to 3NF]

↓

[Efficient Relational Database Schema]

Specialization and Generalization

Specialization

Creating **subclasses** from a **superclass** (more specific categories).
We start with a general entity and divide it into more detailed entities.

Example:

Person

├── Student

└── Instructor

Here, Person is a **superclass** (or parent entity),
Student and Instructor are **subclasses** (or child entities).

Generalization

The reverse process — combining similar entities into a single higher-level entity.

Example:
If we have Student and Instructor entities that both share attributes like Name, Address, DateOfBirth, we can **generalize** them into one Person superclass.

Disjoint vs Overlapping

🔹 **Disjoint (Exclusive)**

An entity instance **can belong to only one subclass**.

Example:
A Person is **either** a Student **or** an Instructor, but **not both**.

Visually, the subclasses are **disjoint (D)**.

🔸 **Overlapping**

An entity instance **can belong to more than one subclass**.

Example:
A Person can be **both a Student and an Instructor** (like a teaching assistant).

Visually, the subclasses are **overlapping (O)**.

Total vs Partial Participation

🟢 **Total Participation**

Every superclass entity **must belong** to at least one subclass.

- Represented by a **double line** from superclass to specialization.

Example:
Every Person is either a Student or an Instructor.

⚪ **Partial Participation**

Some superclass entities **may not belong** to any subclass.

- Represented by a **single line**.

Example:
Some Persons may be just visitors or alumni, not Students nor Instructors.

Mapping Specialization/Generalization to Relations

When converting ERD to tables, there are **four main mapping strategies** (also known as *options or approaches*).
Each has trade-offs depending on **redundancy**, **performance**, and **integrity**.

Let's use this **example ERD**:

Person (PersonID, Name, Address, DOB)

├── Student (StudentID, Major, GPA)

└── Instructor (InstructorID, Department, Salary)

Option 1: One Table for Each Entity (Super + Subclasses)

Each subclass has **its own table**, and the **superclass has a separate table**.

Tables:

- Person(PersonID, Name, Address, DOB)
- Student(PersonID, Major, GPA)
- Instructor(PersonID, Department, Salary)

Mapping:
Each subclass table's PK = FK referencing superclass PK.

```
CREATE TABLE Person (
   PersonID INT PRIMARY KEY,
   Name VARCHAR(50),
   Address VARCHAR(100),
   DOB DATE
);

CREATE TABLE Student (
   PersonID INT PRIMARY KEY,
   Major VARCHAR(50),
   GPA DECIMAL(3,2),
   FOREIGN KEY (PersonID) REFERENCES Person(PersonID)
);

CREATE TABLE Instructor (
   PersonID INT PRIMARY KEY,
   Department VARCHAR(50),
   Salary DECIMAL(10,2),
   FOREIGN KEY (PersonID) REFERENCES Person(PersonID)
);
```

✅ **Pros:**

- Clean separation.
- No redundancy.
- Supports any specialization(partial overlapping, partial disjoint are preferred).

❌ **Cons:**

- Requires joins to get full info about a Student or Instructor.

Option 2: Only Subclass Tables (No Superclass Table)

Each subclass table includes both its own attributes and the **superclass attributes**.

Tables:

- Student(PersonID, Name, Address, DOB, Major, GPA)
- Instructor(PersonID, Name, Address, DOB, Department, Salary)

✅ Example:

```
CREATE TABLE Student (
   PersonID INT PRIMARY KEY,
   Name VARCHAR(50),
   Address VARCHAR(100),
   DOB DATE,
   Major VARCHAR(50),
   GPA DECIMAL(3,2)
);

CREATE TABLE Instructor (
   PersonID INT PRIMARY KEY,
   Name VARCHAR(50),
   Address VARCHAR(100),
   DOB DATE,
   Department VARCHAR(50),
   Salary DECIMAL(10,2)
);
```

✅ **Pros:**

- Simpler — no superclass needed.
- Good for **Total disjoint** specializations.

❌ **Cons:**

- Redundant storage of shared attributes.
- If attributes change (e.g., Address), you must update in multiple tables.

Option 3: Single Table for All (Flattened Structure)

Combine all superclass and subclass attributes into **one table with a primary key(original key + Type)**.

Table:

- Person(PersonID, Name, Address, DOB, Major, GPA, Department, Salary, Type)

✅ Example:

```
CREATE TABLE Person (
   PersonID INT PRIMARY KEY,
   Name VARCHAR(50),
   Address VARCHAR(100),
   DOB DATE,
   Major VARCHAR(50),
   GPA DECIMAL(3,2),
   Department VARCHAR(50),
   Salary DECIMAL(10,2),
   Type VARCHAR(20)  -- 'Student' or 'Instructor'
);
```

✅ **Pros:**

- Simple queries (no joins).
- Works well for disjoint subclasses.

❌ **Cons:**

- Many NULL values (e.g., a Student has no Salary).
- Not good for overlapping subclasses.

Option 4: One Table per Subclass with Common Attributes Repeated (Union Type)

Create only a **superclass** with all attributes of all subclasses and multiple flag columns

Tables:

- Person(PersonID, Name, Address, DOB)
- Student(PersonID, Major, GPA)
- Instructor(PersonID, Department, Salary)

```
CREATE TABLE Person (
   PersonID    INT PRIMARY KEY,
   Name       VARCHAR(100) NOT NULL,
   Address    VARCHAR(200),
   DOB        DATE,

   -- flags indicate which roles apply
   IsStudent   BOOLEAN NOT NULL DEFAULT FALSE,
   IsInstructor BOOLEAN NOT NULL DEFAULT FALSE,

   -- Student-specific attributes (nullable unless IsStudent = TRUE)
   Major      VARCHAR(100),
   GPA        DECIMAL(3,2),

   -- Instructor-specific attributes (nullable unless IsInstructor = TRUE)
   Department  VARCHAR(100),
   Salary     DECIMAL(10,2),
);
```

✅ **Pros:**

- Supports Disjoint overlapping entities
- Flexible.

❌ **Cons:**

- Integrity constraints more complex.

Conclusion

In conclusion, this document has demonstrated the complete journey from understanding database fundamentals to implementing and managing relational databases using SQL.
Through the **university database example**, we explored how conceptual models (ERDs) are transformed into practical relational schemas, how SQL commands can define, control, and manipulate data, and how normalization ensures data integrity.
Furthermore, we addressed complex modeling aspects such as weak entities, ternary relationships, and entity hierarchies, emphasizing their impact on database structure and performance.

Ultimately, a well-designed database is not merely a collection of tables — it is a structured, logical, and efficient representation of real-world data that supports reliable decision-making, consistency, and scalability across all levels of an organization.

Feedback & Contribution

This is a living document. If you have feedback, suggestions, or additional insights that could improve it, I welcome your input. Sharing ideas helps keep this material accurate, relevant, and valuable for everyone.

Copyright & Usage

© **2025 [Youssef Amgad Elkhatib]**.
This document is intended **for learning and reference purposes only**.
You are free to read, use, and share the content for personal or educational use, but:

- **Do not copy or republish this document as your own.**
- Always provide proper credit when referencing.
- Commercial use or redistribution without permission is not allowed.

You may not reproduce this document in whole or in part without attribution. Suggestions and feedback are welcome, and contributors will be acknowledged, but copyright remains with the author.

Acknowledgments

This document is authored and copyrighted by **[Youssef Amgad Elkhatib]**.
Special thanks to the contributors who provided valuable feedback and suggestions for improvements:

Name	Contribution