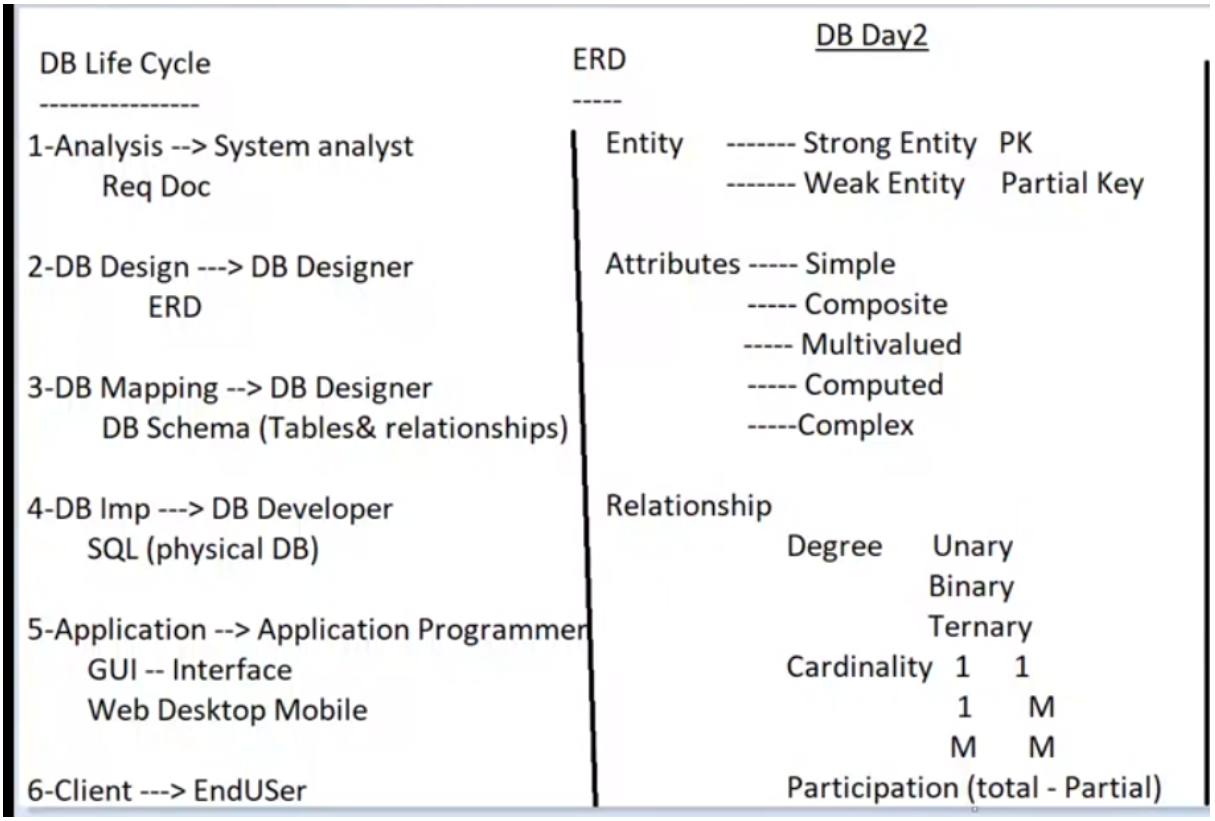
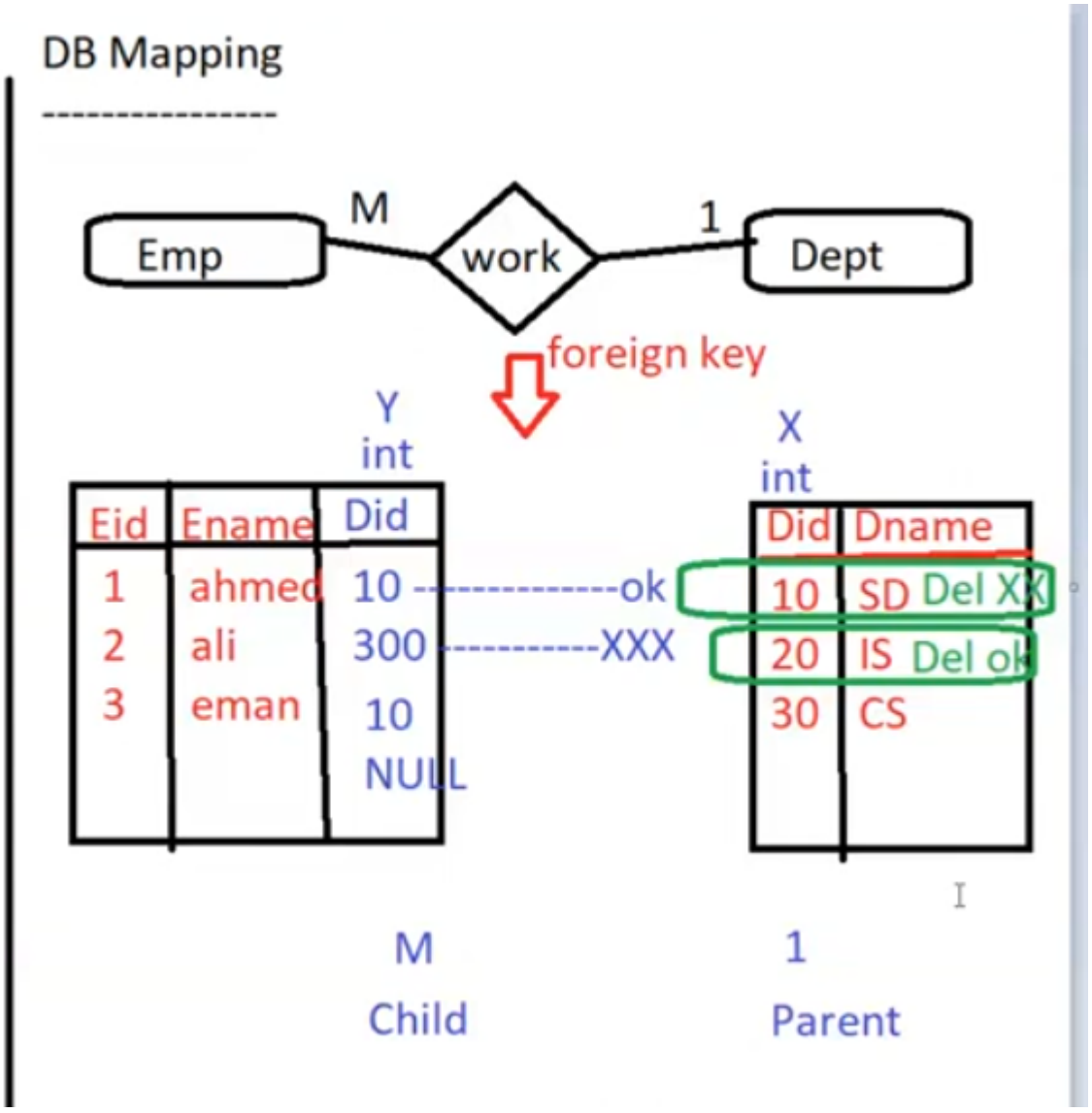
SQL Mapping, DB Schema, SQL statements, Creating DB (Day 2)

Revision day 1



Database Mapping

- we can not delete or edit parent has child like edit one in the relationship 1 to M.
- Server⇒DB⇒Schema⇒Objects(table)⇒columns, rows.
- every column has domain⇒data type.
- data type ensures quality, correct value(size),constrains & values.
- السهم بدايته تخرج من ال foreign key
- وراسه يذهب الي ال primary key.



ER-to-Relational Mapping

Step 1: Mapping of Regular Entity Types

- Create table for each entity type → if there is no 1-1 relationship mandatory from 2 sides
- Choose one of key attributes to be the primary key

Step 2: Mapping of Weak Entity Types

- Create table for each weak entity.
- Add foreign key that correspond to the owner entity type.
- Primary key composed of:
    - Partial identifier of weak entity

- Primary key of identifying relation (strong entity)

Step 3: Mapping of Binary 1:1 Relation Types

Step 4: Mapping of Binary 1:N Relationship Types.

Step 5: Mapping of Binary M:N Relationship Types.

Step 6: Mapping of N-ary Relationship Types.

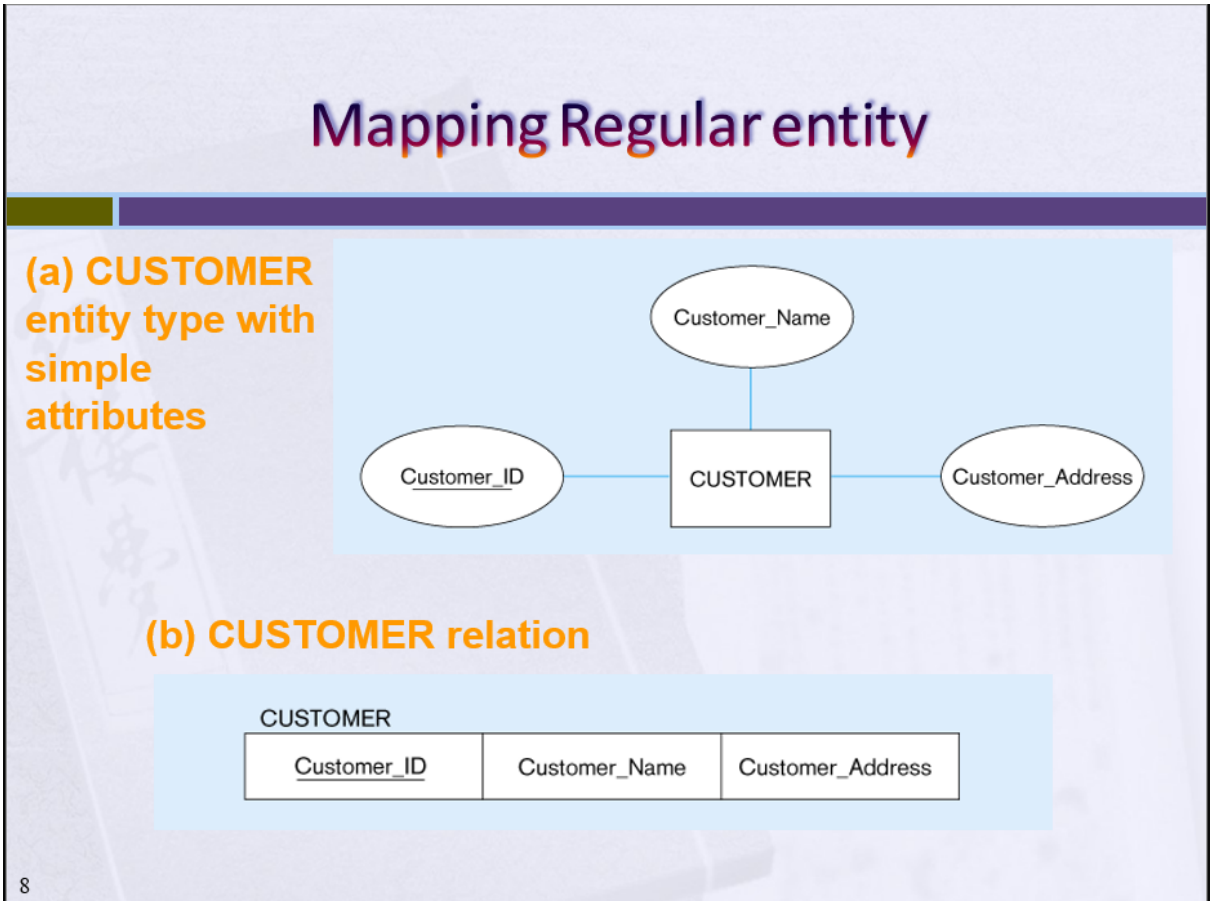
Step 7: Mapping of Unary Relationship.

⇒ mapping rules in standing as following:

Solve one to one and total participation (from two sides.) relationships first.

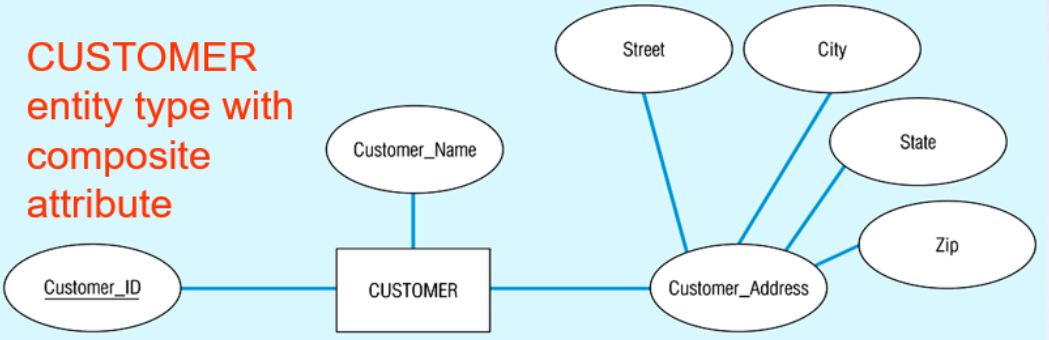
A- any strong entity become a table at first:

1-there is no composite attribute in DB so we take attribute and but it in DB as columns.
2-multi-valued attribute became a table with foreign key from the parent table.
3-deriven attribute we don't put it in table.



Mapping Composite attribute

CUSTOMER entity type with composite attribute



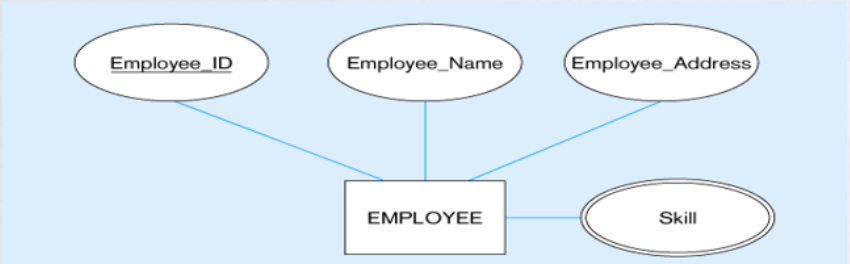
CUSTOMER relation with address detail

CUSTOMER

Customer_ID	Customer_Name	Street	City	State	Zip

9

Mapping Multivalued Attribute



Multivalued attribute becomes a separate relation with foreign key

EMPLOYEE

Employee_ID	Employee_Name	Employee_Address

EMPLOYEE_SKILL

Employee_ID	Skill

1 – to – many relationship between original entity and new relation

10

B- weak entity:

1- become table.
2- primary key of it is a composite primary key.
3- composite key consists of(primary key of the strong entity as foreign key + partial key that inside the weak entity(name is not primary key but used as partial key.)).

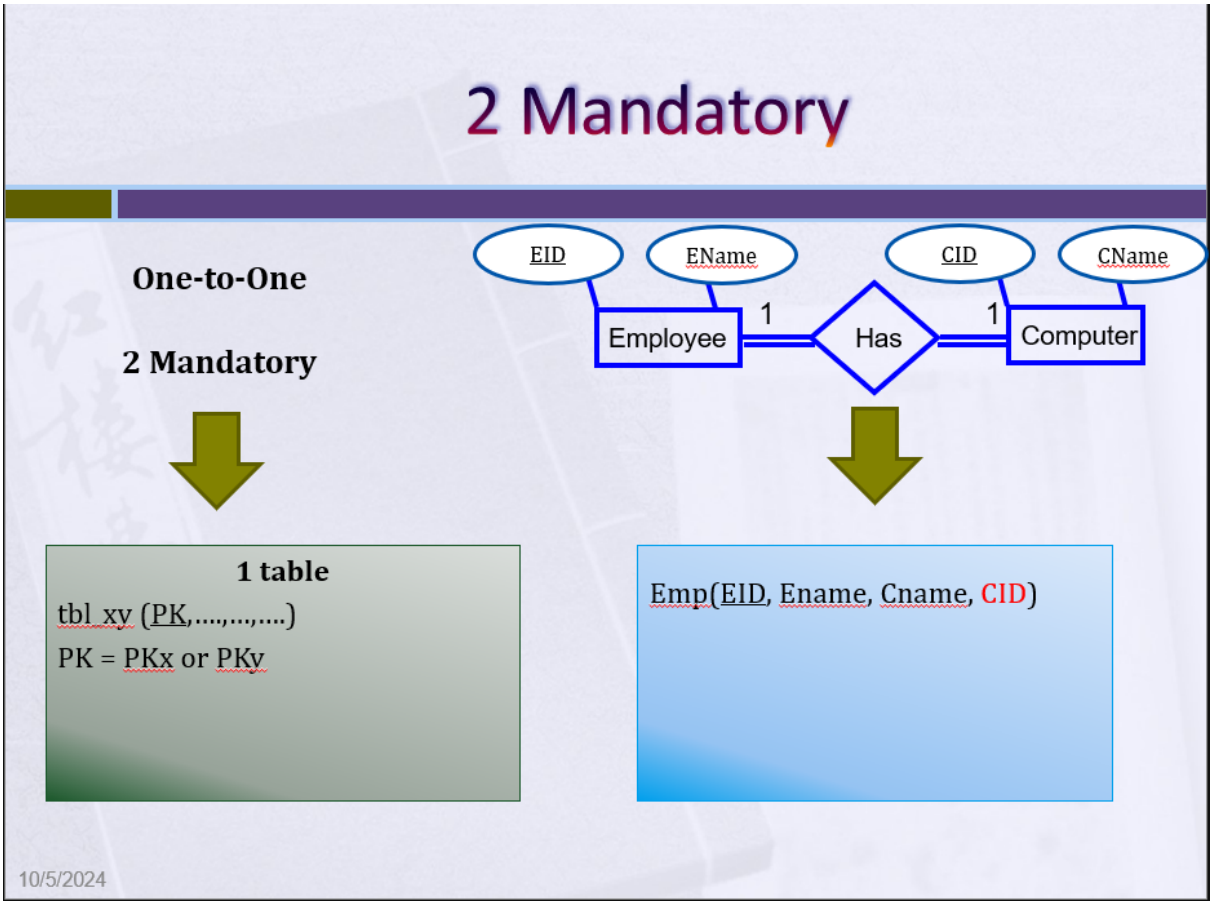
Binary relationship has 6 Cases:

1. Binary 1:1 Total participation from two table
2. Binary 1:1 Partial participation from two table
3. Binary 1:1 Partial participation from the first table and total participation from the second
4. Binary 1:N from many is a Total participation (N ⇒ Total participation)
5. Binary 1:N from many is a Partial participation (N ⇒ Partial participation)
6. Binary N:M

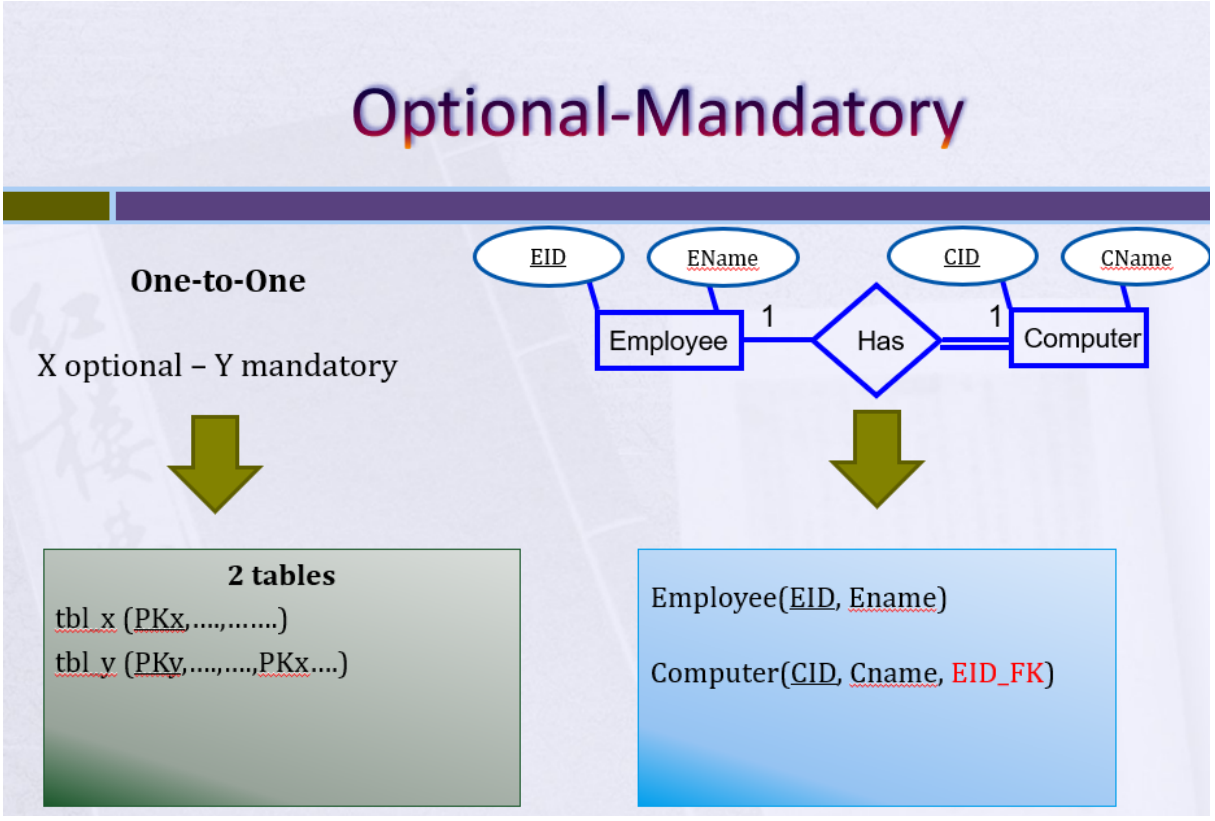
C- mapping 1:1 relationship:

⇒ consists of six cases:

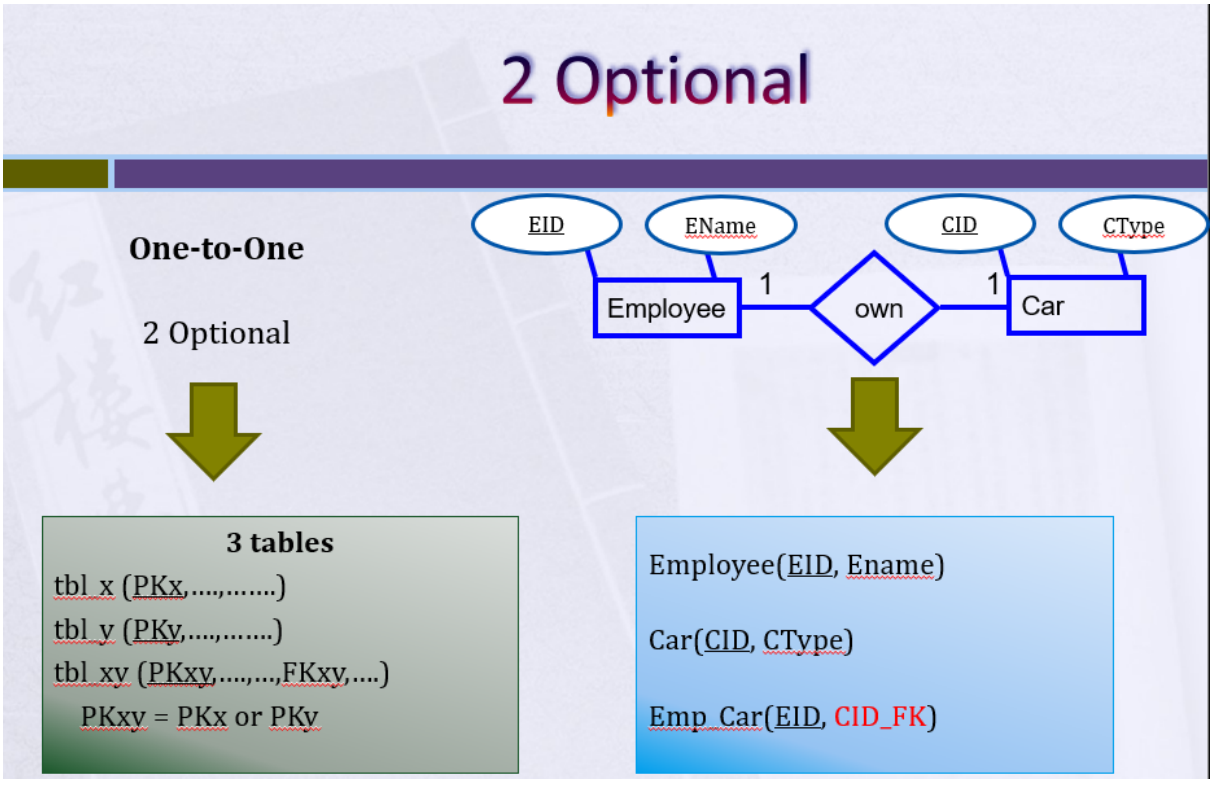
1- total participation from the two sides: they become one table and the primary key is any one of the two primary keys.



2- total from one side and partial from the other:
-rule says that while partial relations increase number of tables increase.
-so we take primary key of partial put it inside total table.

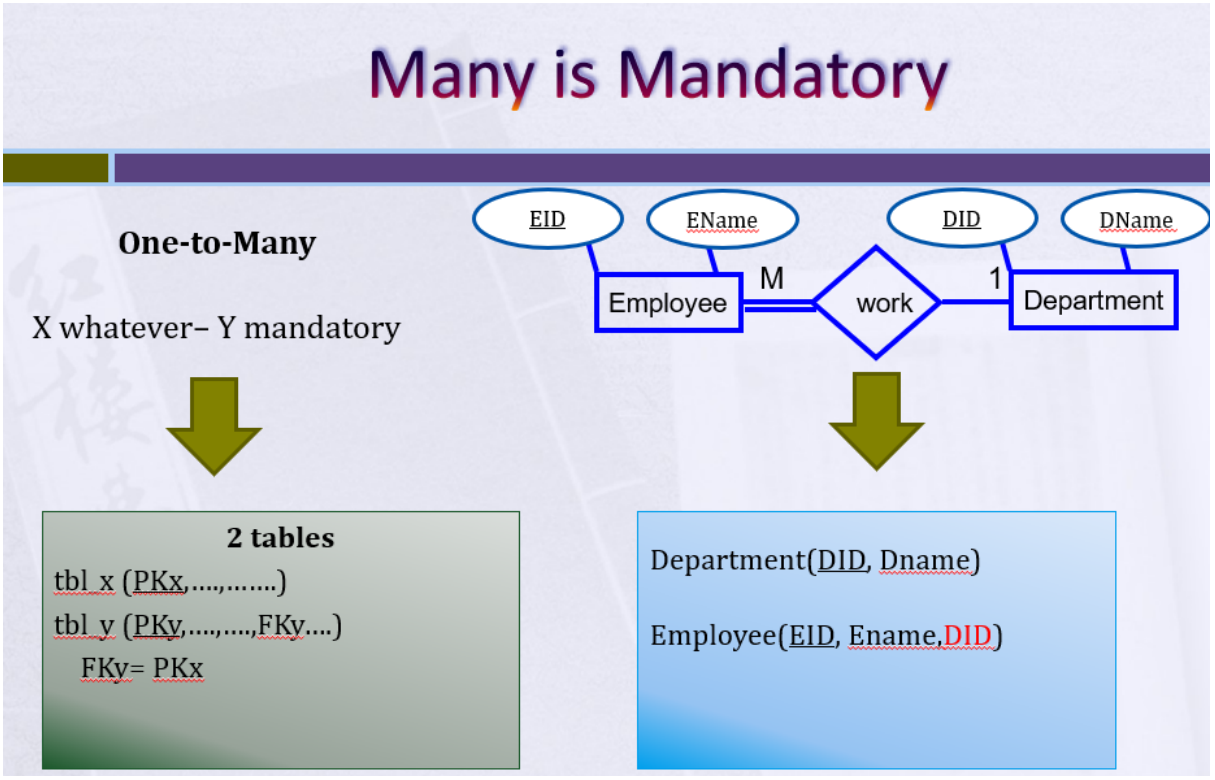


3- partial from the two sides: so we create 3 table ⇒ two tables from the partial and one table collect the primary key of the one side and foreign key from the other.

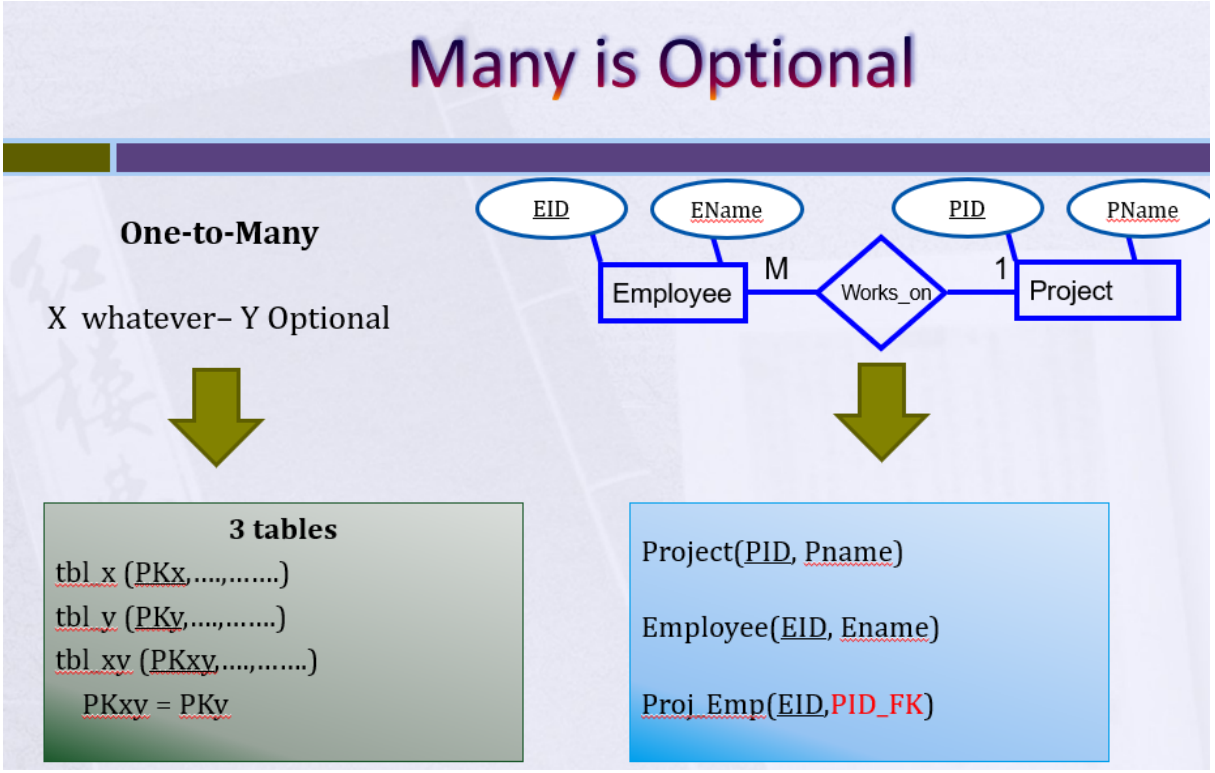


D- 1:N relationship: we focus on side of the many of the relation.

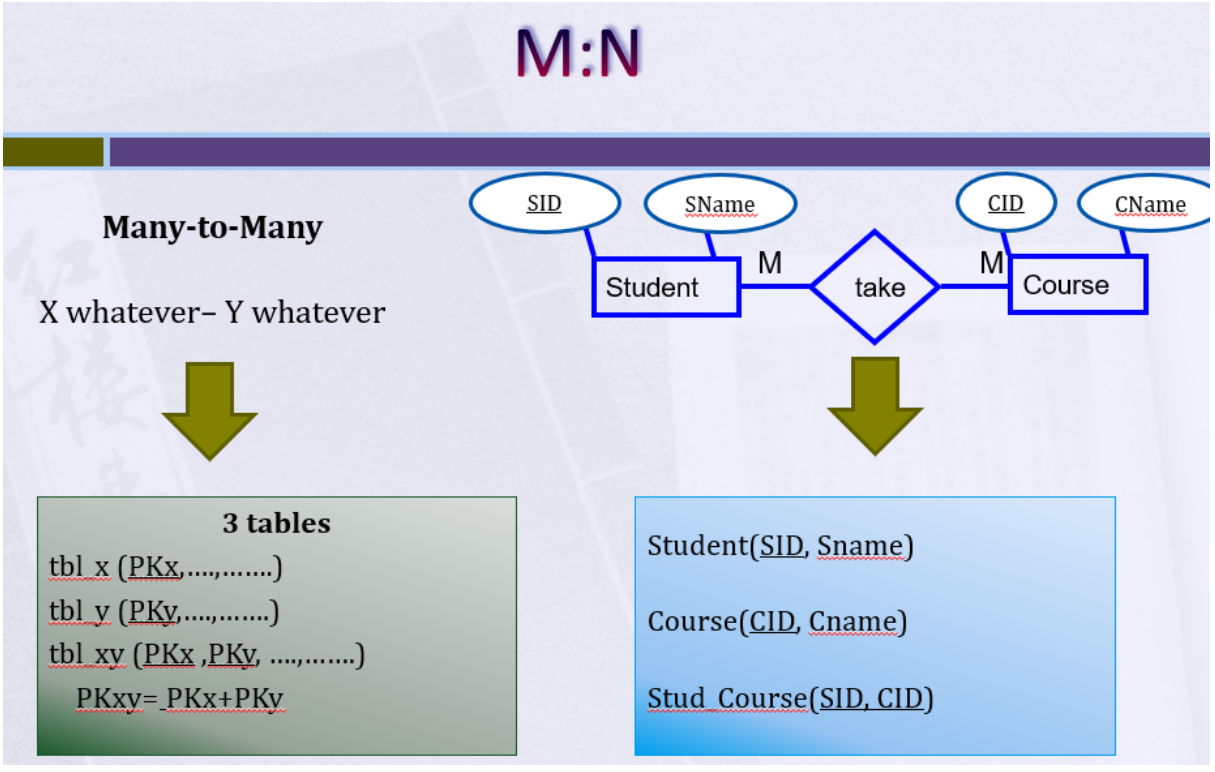
1-if the many is total: take the primary key of the one and put it inside the many as foreign key.



2-if the many side is partial: so we create 3 table ⇒ two tables from the partial and one table collect the primary key of the many side and foreign key from the one.



E- M:N: we create new table directly and take two primary key as composite primary key from the two tables.



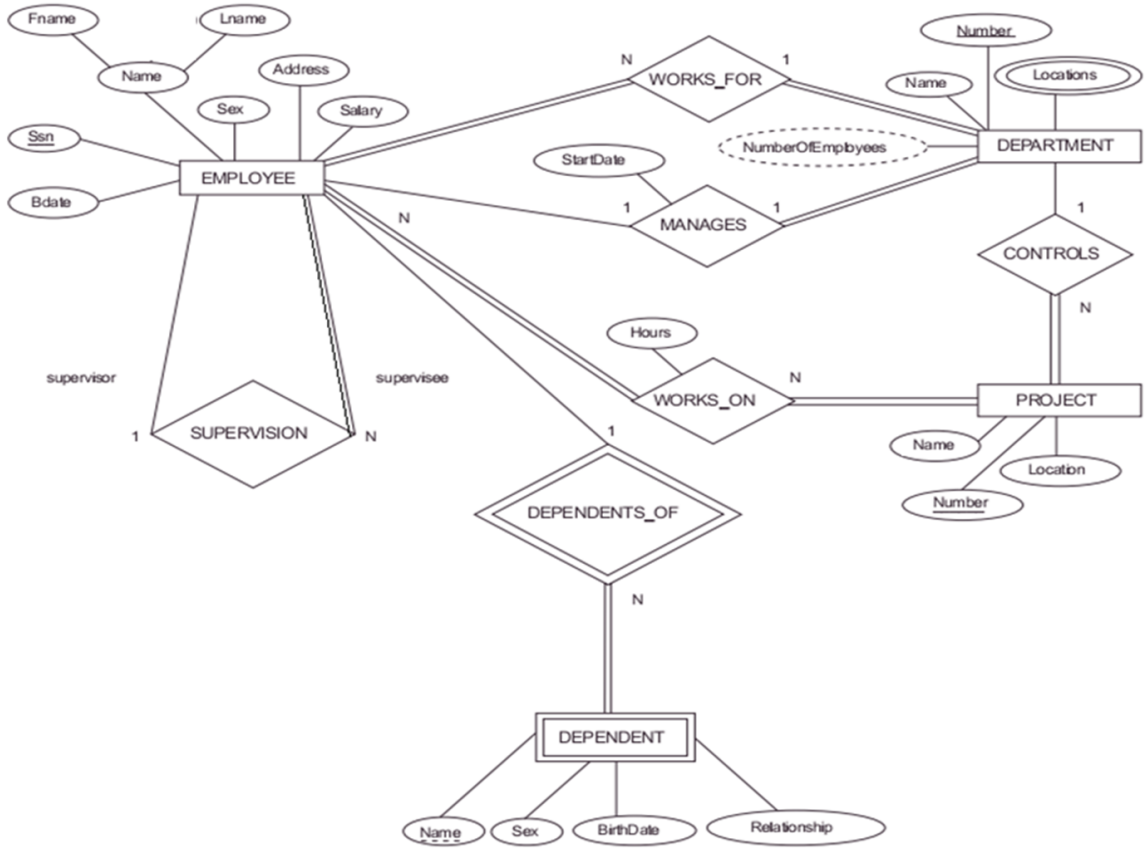
F- Mapping ternary (N-ary) relationship: create new table directly.

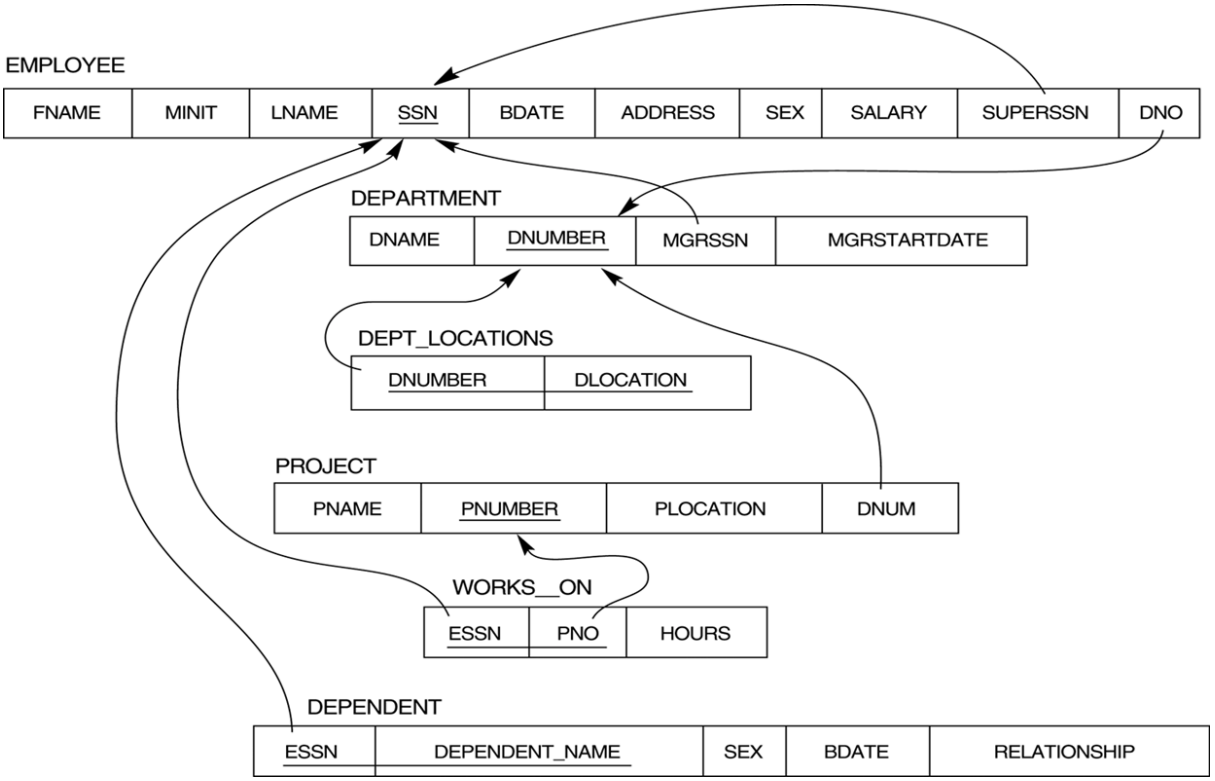
-the cardinality and participation is not important.
-we take foreign key from the tables(n>=3) that have relation.
-number of columns is (number of relations(n>=3) + attribute inside the relation itself).
-primary key of the new table we create it by looking in the attribute or create id from your self.
-we cannot take the foreign keys as primary key because may be copied.

G- mapping unary relationship:

-we add directly the primary key as foreign key inside the same table.

LAB 1





⇒**ANSI:** stands for **American National Standards Institute**.

- ANSI SQL is library has all syntax of sql to the people work with DB.
- ANSI sql ⇒ (Microsoft Transact-SQL, Oracle Pl-SQL, IBM-Pl-SQL, open source(MySQL)).
- Microsoft Transact-SQL ⇒

  ⇒DDL(Data definition lang)

  ⇒DML(Data Manipulation lang)

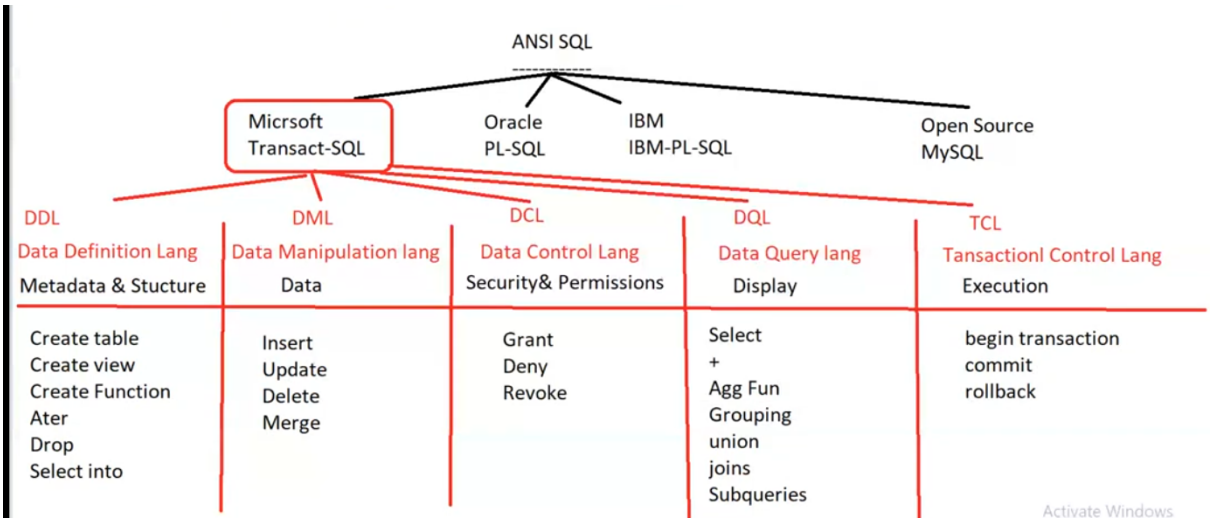
  ⇒DCL(Data control lang)

  ⇒DQL(Data query lang)

  ⇒TCL(transaction control lang).
- DDL(Meta data & structure):Create table, create function, create view, alter, Drop, select
- DML(Data):insert, update, Delete, Merge.
- DCL(Security & permissions):Grant, Deny, Revoke.

- DQL(Display):select+(aggregate function, grouping, union, joins, subqueries).
- TCL(Execution):begin transaction, commit, rollback.

In T-SQL, the various types of SQL commands are categorized based on their functionality. These categories include DDL, DML, DCL, DQL, and TCL.



1. DDL (Data Definition Language)

- **Purpose**: Used to define and manage the structure of database objects such as tables, indexes, and views.

Command	Description	Example
`CREATE`	Used to create database objects such as tables, views, indexes, databases, etc.	`CREATE TABLE Employees (EmployeeID INT, FirstName NVARCHAR(50), LastName NVARCHAR(50));`
`ALTER`	Used to modify existing database objects (e.g., add or delete columns in a table).	`ALTER TABLE Employees ADD Salary DECIMAL(10,2);`
`DROP`	Used to delete database objects (e.g., tables, views, indexes).	`DROP TABLE Employees;`
`TRUNCATE`	Deletes all rows from a table but keeps the table structure.	`TRUNCATE TABLE Employees;`
`COMMENT`	Adds comments to database objects (optional, used for	`COMMENT ON TABLE Employees IS 'Stores employee details';`

	documentation purposes).	
`RENAME`	Renames an existing database object.	`ALTER TABLE Employees RENAME TO Staff;`
`CREATE INDEX`	Creates an index to improve query performance.	`CREATE INDEX idx_lastname ON Employees(LastName);`
`DROP INDEX`	Deletes an index from the database.	`DROP INDEX idx_lastname;`
`CREATE VIEW`	Creates a virtual table (view) based on the result of a `SELECT` query.	`CREATE VIEW EmployeeView AS SELECT FirstName, LastName FROM Employees;`
`DROP VIEW`	Deletes a view from the database.	`DROP VIEW EmployeeView;`
`CREATE DATABASE`	Creates a new database.	`CREATE DATABASE CompanyDB;`
`DROP DATABASE`	Deletes an entire database along with its data.	`DROP DATABASE CompanyDB;`
`CREATE SCHEMA`	Creates a schema to group database objects together.	`CREATE SCHEMA Sales AUTHORIZATION UserName;`
`DROP SCHEMA`	Deletes a schema along with all objects contained in it.	`sql DROP SCHEMA Sales;`
`CREATE SEQUENCE`	Generates a sequence of numbers used for unique identifiers.	`sql CREATE SEQUENCE seq_emp START WITH 1 INCREMENT BY 1;`
`ALTER SEQUENCE`	Modifies an existing sequence.	`sql ALTER SEQUENCE seq_emp INCREMENT BY 5;`
`DROP SEQUENCE`	Deletes a sequence from the database.	`sql DROP SEQUENCE seq_emp;`

2. DML (Data Manipulation Language)

- **Purpose**: Used for data manipulation, such as inserting, updating, deleting, and retrieving data from the database.

Command	Description	Example
`INSERT`	Adds new records (rows) into a table.	`sql INSERT INTO Employees (EmployeeID, FirstName, LastName) VALUES (1, 'John', 'Doe');`
`UPDATE`	Modifies existing records in a table.	`sql UPDATE Employees SET LastName = 'Smith' WHERE EmployeeID = 1;`

`DELETE`	Removes existing records from a table based on a condition.	`sql DELETE FROM Employees WHERE EmployeeID = 1;`
`MERGE`	Inserts, updates, or deletes records based on matching conditions between two tables.	`sql MERGE INTO Employees AS target USING NewData AS source ON target.EmployeeID = source.EmployeeID WHEN MATCHED THEN UPDATE SET target.Salary = source.Salary;`
`BULK INSERT`	Efficiently inserts a large amount of data from an external file into a table.	`sql BULK INSERT Employees FROM 'C:\data\employees.csv' WITH (FIELDTERMINATOR = ',', ROWTERMINATOR = '\n');`
`SELECT INTO`	Copies data from one table into a new table.	`sql SELECT * INTO EmployeesBackup FROM Employees;`
`TRUNCATE TABLE`	Deletes all records from a table, keeping its structure intact (technically a DDL, often used as DML).	`sql TRUNCATE TABLE Employees;`
`INSERT INTO SELECT`	Inserts records into a table based on the result of a `SELECT` query from another table.	`sql INSERT INTO EmployeesBackup (EmployeeID, FirstName, LastName) SELECT EmployeeID, FirstName, LastName FROM Employees;`
`OUTPUT`	Returns the data affected by `INSERT`, `UPDATE`, or `DELETE` operations, useful for auditing or logging.	`sql DELETE FROM Employees OUTPUT DELETED.* WHERE EmployeeID = 1;`

3. DCL (Data Control Language)

- **Purpose**: Deals with permissions and access control for users in the database.

Command	Description	Example
`GRANT`	Provides specific privileges to users or roles on database objects (e.g., tables, views, procedures).	`sql GRANT SELECT, INSERT ON Employees TO 'UserName';`
`REVOKE`	Removes specific privileges previously granted to users or roles.	`sql REVOKE INSERT ON Employees FROM 'UserName';`

`DENY`	Denies a specific permission to a user or role, overriding `GRANT` permissions.	`sql DENY SELECT ON Employees TO 'UserName';`

4. DQL (Data Query Language)

- **Purpose**: Focused on querying the database and retrieving data.

Command	Description	Example
`SELECT`	Retrieves data from one or more tables.	`sql SELECT FirstName, LastName FROM Employees;`
`WHERE`	Filters records based on specific conditions.	`sql SELECT * FROM Employees WHERE LastName = 'Smith';`
`ORDER BY`	Sorts the result set by one or more columns.	`sql SELECT * FROM Employees ORDER BY LastName ASC;`
`GROUP BY`	Groups rows that have the same values in specified columns and allows aggregate functions (e.g., `COUNT`, `SUM`).	`sql SELECT Department, COUNT(*) FROM Employees GROUP BY Department;`
`HAVING`	Filters groups created by `GROUP BY` based on a condition (used with aggregate functions).	`sql SELECT Department, COUNT(*) FROM Employees GROUP BY Department HAVING COUNT(*) > 5;`
`JOIN`	Combines rows from two or more tables based on a related column between them.	`sql SELECT Employees.FirstName, Departments.DepartmentName FROM Employees INNER JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;`
`UNION`	Combines the result sets of two or more `SELECT` queries into a single result set (removes duplicates by default).	`sql SELECT FirstName FROM Employees UNION SELECT FirstName FROM Managers;`
`DISTINCT`	Returns unique values, removing duplicates from the result set.	`sql SELECT DISTINCT Department FROM Employees;`
`LIMIT` / `TOP`	Limits the number of rows returned by the query (MySQL uses	`sql SELECT TOP 10 * FROM Employees;` (SQL Server) `sql SELECT * FROM Employees LIMIT 10;` (MySQL)

	`LIMIT`, SQL Server uses `TOP`).	
`IN`	Filters the result set based on a list of specified values.	`sql SELECT * FROM Employees WHERE DepartmentID IN (1, 2, 3);`
`BETWEEN`	Filters the result set for values within a certain range.	`sql SELECT * FROM Employees WHERE Salary BETWEEN 3000 AND 6000;`
`LIKE`	Filters the result set based on pattern matching.	`sql SELECT * FROM Employees WHERE FirstName LIKE 'J%';`
`EXISTS`	Checks for the existence of rows in a subquery.	`sql SELECT * FROM Employees WHERE EXISTS (SELECT * FROM Managers WHERE Employees.EmployeeID = Managers.EmployeeID);`

5. TCL (Transaction Control Language)

- **Purpose**: Used to manage transactions in the database to maintain the integrity of data.

Command	Description	Example
`COMMIT`	Saves all changes made in the current transaction.	`sql COMMIT;`
`ROLLBACK`	Undoes all changes made in the current transaction since the last `COMMIT`.	`sql ROLLBACK;`
`SAVEPOINT`	Creates a point within a transaction to which you can later roll back.	`sql SAVEPOINT savepoint_name;`
`SET TRANSACTION`	Configures the properties of the current transaction (e.g., isolation level).	`sql SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;`

⇒ Read Notes

- DB that you create consists of two files (file.mdf ⇒ tables,data....etc.,) & (file.ldf ⇒ transaction on this DB).
- After creating DB you cannot copy it or transfer it so you right click on DB ⇒ tasks⇒backup⇒put the path you want.

- If you want to restore any DB⇒right click on main Databases folder in SQL server app ⇒ restore DB ⇒ select device.
- To create table wizard ⇒ open DB you create ⇒ right click on tables ⇒ New ⇒ table ⇒ CRL + S .
- To edit table ⇒ right click on table ⇒ design.
- To allow edit in tables ⇒ from navbar select ⇒ tools ⇒ options ⇒ from left select designers ⇒ remove marked point in prevent saving changes.
- To make column primary key ⇒ right click on column ⇒ set as PK.
- To make composite PK select columns then right click on columns ⇒set as PK.
- To make foreign key ⇒ open DB you create ⇒right click on DB diagram ⇒ New DB diagram.

⇒ Connecting to Microsoft SQL Server via CMD

Step-by-Step Instructions:

1. Open **Command Prompt** (CMD).
2. Use the `sqlcmd` command to connect to SQL Server:

   ```
   bash
   sqlcmd -S [server_name] -U [username] -P [password]
   ```

   - `S`: Specifies the server name (replace `[server_name]` with the actual name of the server).
   - `U`: Specifies the SQL Server username (replace `[username]`).
   - `P`: Specifies the password (replace `[password]`).
3. Once connected, you can execute SQL queries:

   ```
   bash
   USE [database_name];
   GO
   ```

   Example:

```
bash
sqlcmd -S localhost -U sa -P MyPassword123
```

Important Notes:

- Ensure that the database service is running before trying to connect.
- You need appropriate user privileges to access the database.
- For MySQL and PostgreSQL, if the PATH variable is set correctly, you can run the command directly from any directory. If not, navigate to the directory where the DBMS is installed or add it to the system's PATH.

This method allows you to interact with the database through command-line queries and commands, which can be useful for administrative tasks or development purposes.

SQL State ments

DDL queries:

```
-- Use descriptive table and column names
CREATE TABLE Employee (
    EmployeeID INT PRIMARY KEY,                    -- Use me
    EmployeeName VARCHAR(50),                      -- Descri
    Age INT NOT NULL,                              -- Mandat
    HireDate DATE DEFAULT GETDATE(),               -- Defaul
    Address VARCHAR(50) DEFAULT 'Cairo',           -- Descri
    DepartmentID INT                               -- Link t
);

-- Add Salary column to Employee table
ALTER TABLE Employee
ADD Salary INT;

-- Modify Salary column data type to BIGINT for larger values
ALTER TABLE Employee
ALTER COLUMN Salary BIGINT;

-- Insert a new record into Employee table
```

```
INSERT INTO Employee (EmployeeID, EmployeeName, Age, HireDate
VALUES (1, 'Amr', 21, '2002-10-01', 'Assuit', 12);

-- Remove Salary column from Employee table
ALTER TABLE Employee
DROP COLUMN Salary;

-- Drop the Employee table if no longer needed
DROP TABLE Employee;
```

DML queries:

1. Creating the `Employee` Table:

```
CREATE TABLE Employee (
    EmployeeID INT PRIMARY KEY,                    -- Use
meaningful column name
    EmployeeName VARCHAR(50),                      -- Desc
riptive column name for the employee's name
    Age INT NOT NULL,                              -- Mand
atory column for age
    HireDate DATE DEFAULT GETDATE(),               -- Defa
ult to current date
    Address VARCHAR(50) DEFAULT 'Cairo',           -- Defa
ult address set to 'Cairo'
    DepartmentID INT                               -- Fore
ign key to department table (to be added later)
);
```

---

2. Inserting Data:

Single Insert with All Values:

It's important to specify the column names when inserting data, so the order and structure remain clear and prevent issues with table schema changes.

```
-- Insert all values into the Employee table
INSERT INTO Employee (EmployeeID, EmployeeName, Age, HireDa
te, Address, DepartmentID)
VALUES (1, 'Amr', 21, '2002-10-01', 'Assuit', 12);
```

Insert with Specific Columns:

When not inserting all the fields, explicitly mention the columns to be inserted.

```
-- Insert specific columns (EmployeeID, EmployeeName, Age)
INSERT INTO Employee (EmployeeID, EmployeeName, Age)
VALUES (2, 'Amr', 21);
```

Insert Multiple Rows (Constructor):

When inserting multiple rows, use this cleaner syntax to make the code more concise and efficient.

```
-- Insert multiple rows in a single statement
INSERT INTO Employee (EmployeeID, EmployeeName, Age)
VALUES
    (6, 'Amr', 21),
    (3, 'Omar', 21),
    (4, 'Ali', 21),
    (5, 'Abdo', 21);
```

---

3. Updating Data:

Update All Rows:

If updating all rows in the table, this should be made clear with comments.

```
-- Update EmployeeName to 'Omar' for all rows
UPDATE Employee
SET EmployeeName = 'Amr';
```

Update Specific Row (With Condition):

Always include a `WHERE` clause when targeting specific rows, so the update operation only affects those rows.

```
-- Update EmployeeName to 'Omar' where EmployeeID is 4
UPDATE Employee
SET EmployeeName = 'Amr'
WHERE EmployeeID = 4;
```

Increment Age for All Rows:

```
-- Increment the Age column by 1 for all rows
UPDATE Employee
SET Age = Age + 1;
```

Set a Column to NULL:

```
-- Set EmployeeName to NULL for all rows
UPDATE Employee
SET EmployeeName = NULL;
```

---

4. Deleting Data:

Delete All Rows (But Keep Table Structure):

```
-- Delete all rows from the Employee table, but keep the st
ructure intact
DELETE FROM Employee;
```

Delete Specific Row (With Condition):

```
-- Delete the row where EmployeeID is 1
DELETE FROM Employee
WHERE EmployeeID = 1;
```

DQL quaries

1. Selecting All Rows from the Employee Table:

```
-- Select all columns from Employee table
SELECT *
FROM Employee;
```

2. Selecting Specific Row Based on EmployeeID:

```
-- Select all columns where EmployeeID is 1
SELECT *
FROM Employee
WHERE EmployeeID = 1;
```

3. Selecting Specific Columns (EmployeeID, Age):

```
-- Select EmployeeID and Age from Employee table
SELECT EmployeeID, Age
FROM Employee;
```

4. Ordering by Age (Ascending):

```
-- Select EmployeeID and Age, order by Age in ascending ord
er
SELECT EmployeeID, Age
FROM Employee
ORDER BY Age;
```

5. Ordering by Age (Descending):

```
-- Select EmployeeID and Age, order by Age in descending or
der
SELECT EmployeeID, Age
FROM Employee
ORDER BY Age DESC;
```

6. Calculating and Aliasing a Sum of EmployeeID and Age (with Order):

```
-- Select the sum of EmployeeID and Age, alias as 'Sum', or
der by Age in descending order
SELECT EmployeeID + Age AS Sum
FROM Employee
ORDER BY Age DESC;
```

---

7. Handling Aliases with Square Brackets:

Though you can use square brackets, it's cleaner to use `AS` for column aliasing.

```
-- Select the sum of EmployeeID and Age, alias as 'Sum', us
ing proper aliasing syntax
SELECT EmployeeID + Age AS [Sum]
FROM Employee
ORDER BY Age DESC;
```

---

8. Handling NULL Values in `WHERE` Clause:

Use `IS NOT NULL` instead of `!= NULL` for checking nulls.

```
-- Select the sum of EmployeeID and Age where EmployeeName
is not null
SELECT EmployeeID + Age AS [Sum]
FROM Employee
WHERE EmployeeName IS NOT NULL;
```

➔**NULL Comparison**: In SQL, you **cannot use `!=` or `=` to compare with `NULL`**. This is because `NULL` represents an unknown or missing value, and comparing anything to `NULL` with `=` or `!=` does not work as expected.

Correct Way to Handle NULL Comparison:

To check if a column is not `NULL`, you must use the `IS NOT NULL` condition. The proper way to check if `EmplyeeName` is not `NULL` would be:

Corrected SQL Statement:

```
SELECT *
FROM Employee
WHERE EmployeeName IS NOT NULL;
```

Explanation:

1. `SELECT *`: This retrieves all columns from the `Employee` table.
2. `FROM Employee`: Specifies the table from which to retrieve the data (assuming the table is named `Employee`).
3. `WHERE first_name IS NOT NULL`: This condition checks that the `first_name` column does not have a `NULL` value, ensuring that only rows where `first_name` contains a value will be returned.

---

9. Applying Additional Conditions with `AND`:

```
-- Select the sum of EmployeeID and Age where EmployeeName
is not null and Age is greater than 6
SELECT EmployeeID + Age AS [Sum]
FROM Employee
WHERE EmployeeName IS NOT NULL
  AND Age > 6;
```

---

10. Selecting Distinct Values for Age:

```
-- Select distinct Age values from Employee table
SELECT DISTINCT Age
FROM Employee;
```

---

11. Using `IN` for Specific Values:

```
-- Select distinct Age values where Age is 21, 22, or 23
SELECT DISTINCT Age
FROM Employee
WHERE Age IN (21, 22, 23);
```

---

12. Using `BETWEEN` for Range Filtering:

```
-- Select distinct Age values where Age is between 21 and 2
7
SELECT DISTINCT Age
FROM Employee
WHERE Age BETWEEN 21 AND 27;
```

LAB 2

1. **Display all the employees Data.**

   ```
   SELECT *
   FROM Employee;
   ```

2. **Display the employee First name, last name, Salary, and Department number.**

   ```
   SELECT FirstName, LastName, Salary, DepartmentID
   FROM Employee;
   ```

3. **Display all the project names, locations, and the department responsible for it.**

   ```
   SELECT ProjectName, Location, DepartmentID
   FROM Project;
   ```

4. **Display each employee's full name and their annual commission (10% of their annual salary).**

   ```
   SELECT
       CONCAT(FirstName, ' ', LastName) AS FullName,
       Salary * 0.1 AS AnnualCommission
   FROM Employee;
   ```

5. **Display the employee IDs and names who earn more than 1000 LE monthly.**

```
SELECT EmployeeID, CONCAT(FirstName, ' ', LastName) AS F
ullName
FROM Employee
WHERE Salary > 1000;
```

6. **Display the employee IDs and names who earn more than 10000 LE annually.**

```
SELECT EmployeeID, CONCAT(FirstName, ' ', LastName) AS F
ullName
FROM Employee
WHERE Salary * 12 > 10000;
```

7. **Display the names and salaries of the female employees.**

```
SELECT CONCAT(FirstName, ' ', LastName) AS FullName, Sal
ary
FROM Employee
WHERE Gender = 'Female';  -- Adjust the column name for
gender if necessary
```

8. **Display each department ID and name managed by a manager with ID equals 968574.**

```
SELECT DepartmentID, DepartmentName
FROM Department
WHERE ManagerID = 968574;  -- Adjust the column name for
ManagerID if necessary
```

9. **Display the IDs, names, and locations of the projects controlled by department 10.**

```
SELECT ProjectID, ProjectName, Location
FROM Project
WHERE DepartmentID = 10;  -- Adjust the column name for
DepartmentID if necessary
```