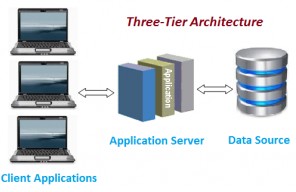
## FILE SYSTEM VS DBMS

* + The difference between file system and dbms is that file system helps to store a collection of raw data files into the hard disk while dbms helps to easily store, retrieve and manipulate data in a database. In brief, dbms provides more flexibility in accessing and managing data than the file system.
  + We are following client server architecture, so data is share with both client and server also so we use dbms. Also ,searching faster using queries, metadata independent, concurrency(multiple people are accessing the data) have proper protocol

### 2 TIER an 3 TIER STRUCTURE

* + The **two-tier** is based on Client Server architecture. The two-tier architecture is like client server application. The direct communication takes place between client and server. There is no intermediate between client and server. Because of tight coupling a 2 tiered application will run faster.



**Three-tier Architecture**

1. **SCHEMA(structure)**:- the logical representation of the database.
   * Three layer of schema:-(for data abstraction)
     + View level // how data are presented to user
     + Logical level // how data are store or what way it is store(ER model,blueprint)
     + Physical level //where it is actually store(size,..)

## KEYS:-

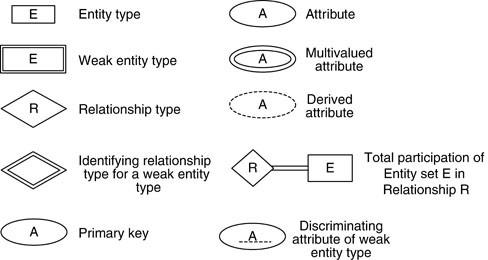
### Candidate Key:-

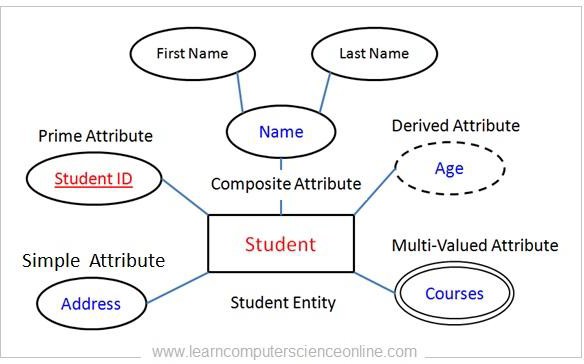
* + A candidate key is a part of a key known as Super Key , where the super key is the super set of all those attributes that can uniquely identify a table but also it may contain null.

1. **Primary Key:-**
   * A primary key is the column or columns that contain values that uniquely identify each row in a table. A database table must have a primary key for optim to insert, update, restore, or delete data from a database table.
2. **Foriegn Key:-**
   * It is an attribute or set of attribute that references to the primary key of same table or another table
   * It maintains r**eferential integrity**, the rule deﬁnes that a foreign key has a matching primary key. Reference from a table to another table should be valid.
   * Referenced table.
     + On insert on violation.
     + On update it may cause violation
       - To solve issue use ON DELETE CASCADE or ON DELETE SET NULL
     + On delete may cause violation
       - To solve issue use ON DELETE CASCADE or ON DELETE SET NULL
   * Referencing table.
     + On insert may cause violation
     + On Delete will not cause violation
     + On update may cause violation
   * **Example:-**
   * CREATE TABLE COURSE( COURSE\_ID VARCHAR(10),

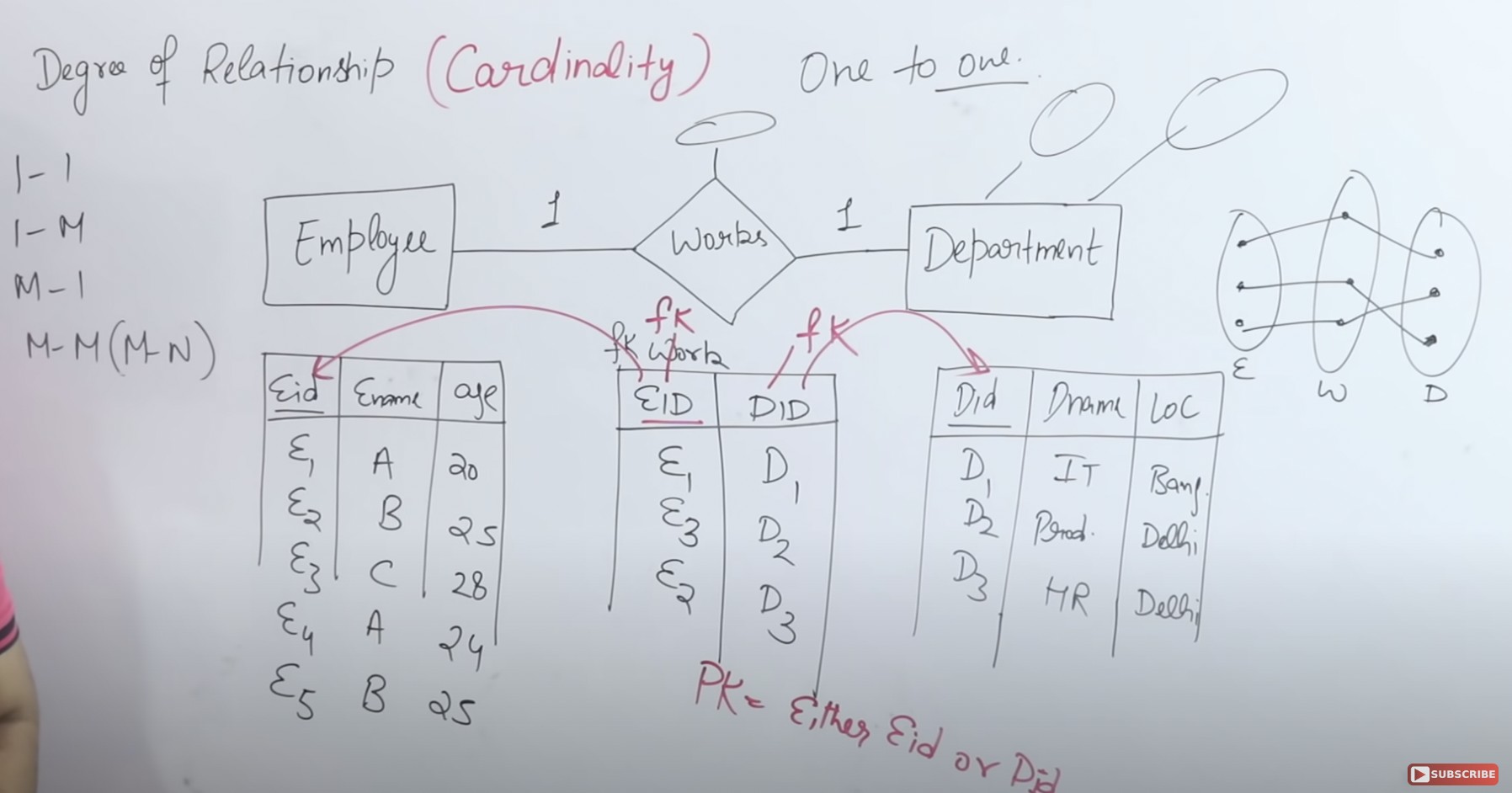
COURSE\_NAME VARCHAR(20), ROLLNO INT REFERENCES STUDENT(ROLL NO)

## ER MODEL

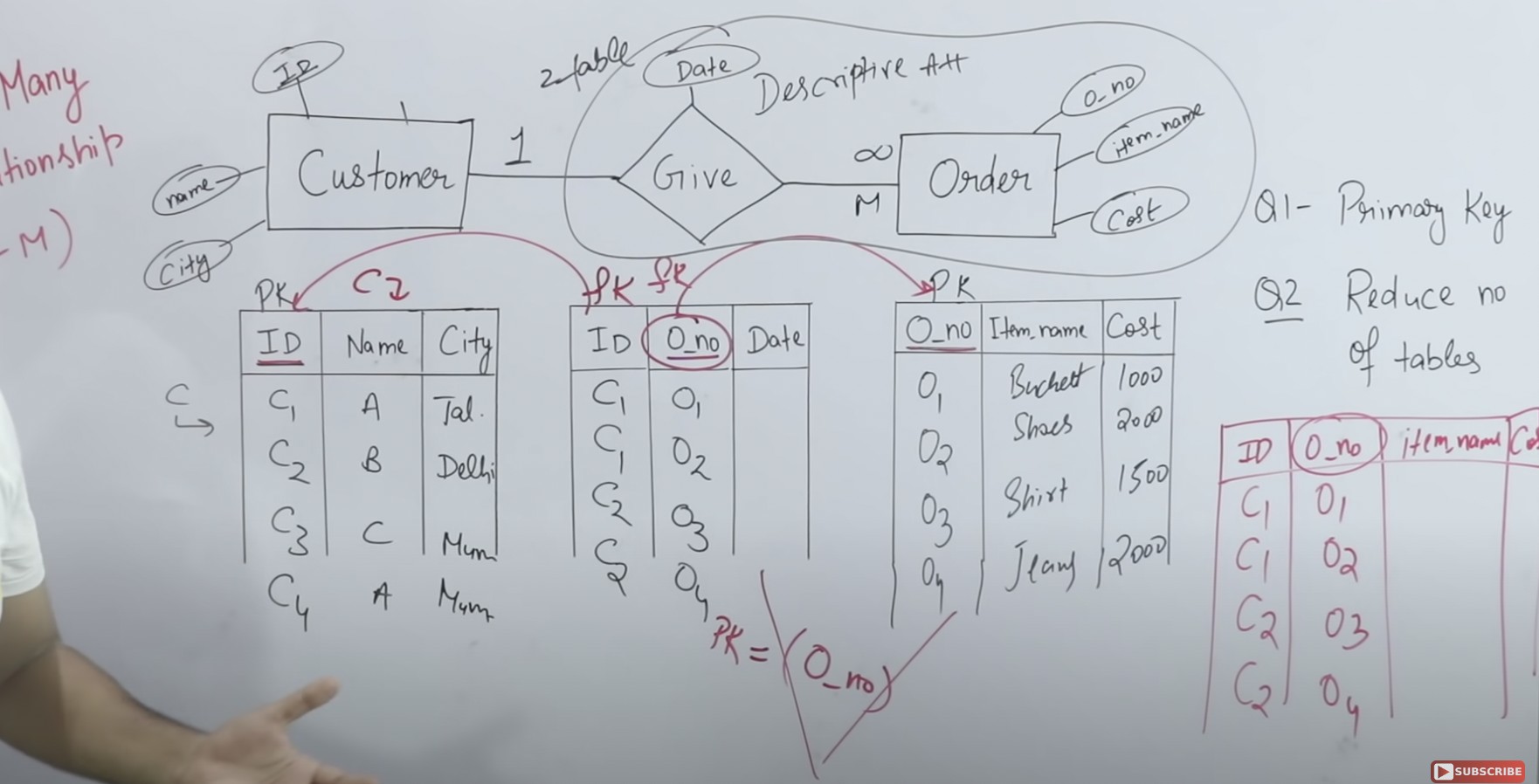




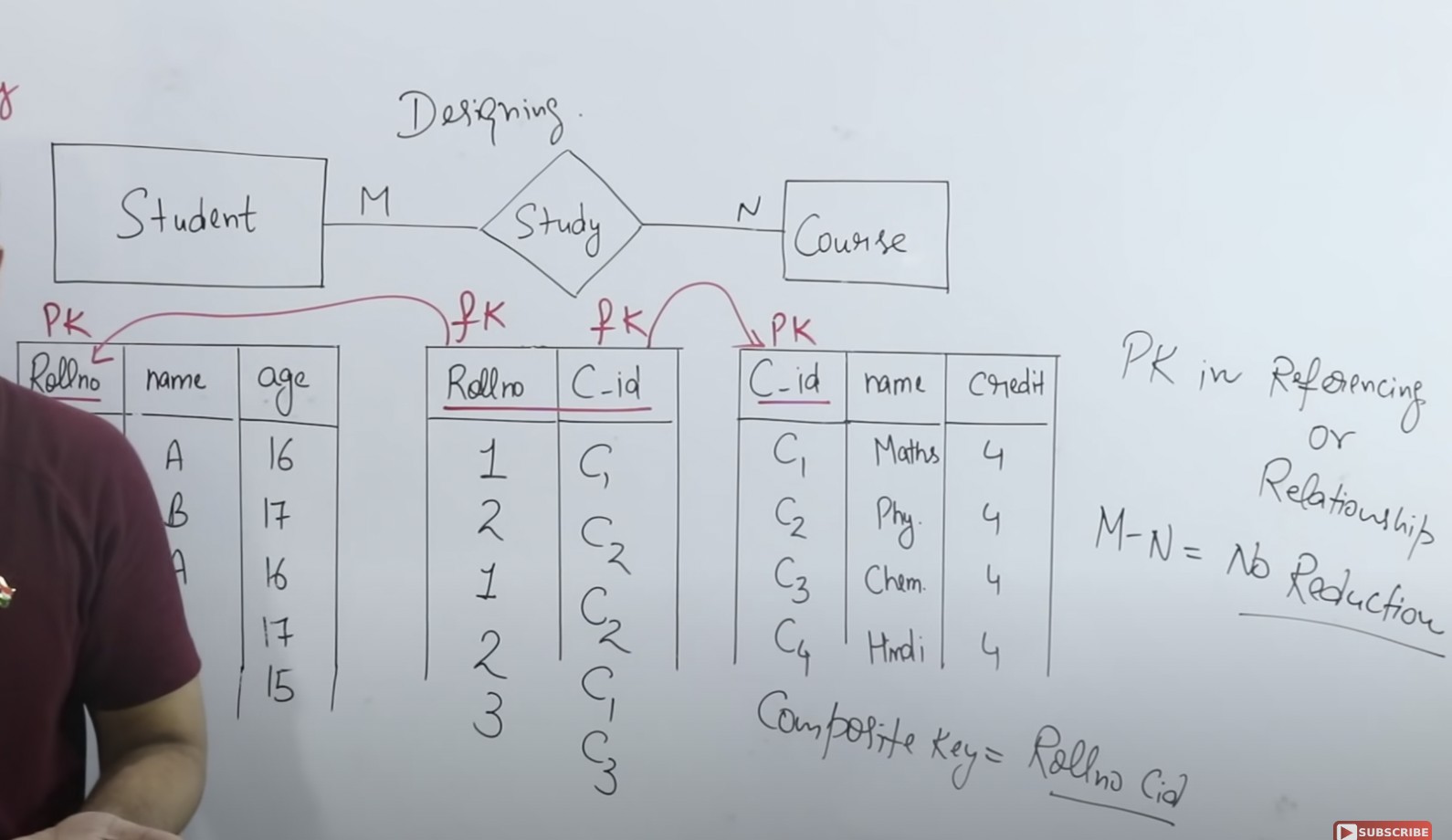
* + **One to one relationship**



* + **One to many relationship**



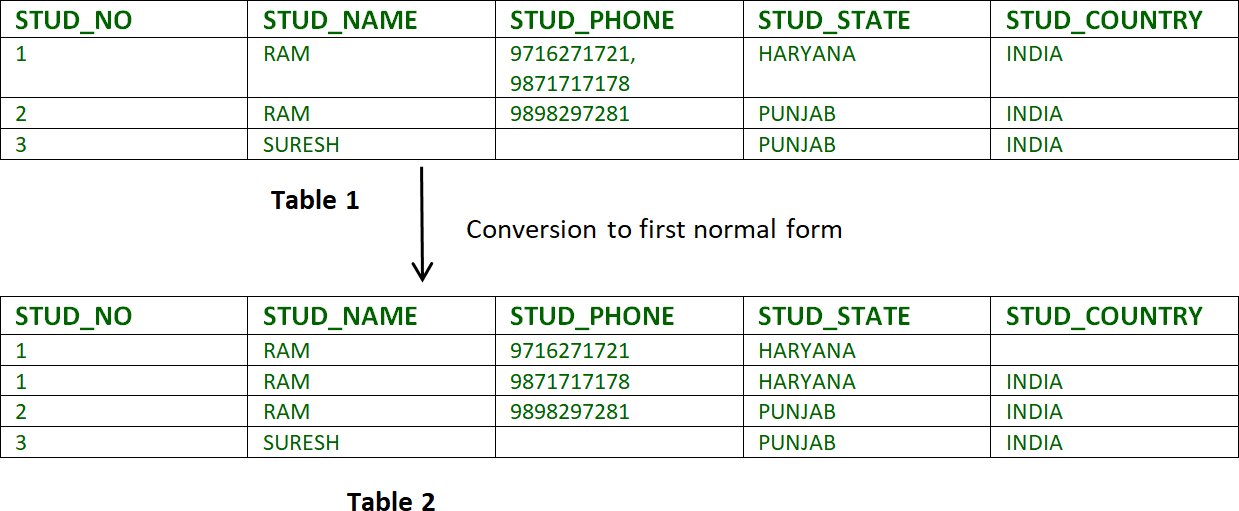
* + **Many to many relationship**



## NORMALIZATION

* + Technique to remove or reduce the redundancy from table.
  + Problems
    - Row duplicates // use primary key
    - Column duplicates // cause problem like
    - Causes anomaly like
      * Insertion anomaly //Let’s say we have a table that has 4 columns. Student ID, Student Name, Student Address and Student Grades. Now when a new student enrolls in school, even though the first three attributes can be filled but 4th attribute will have NULL value because he doesn't have any marks yet.
      * Deletion anomaly //If any student leaves the school then the entry related to that student will be deleted. However, that deletion will also delete the course information even though the course depends upon the school and not the student.
      * Update anomaly //Let say we have 10 columns in a table out of which 2 are called employee Name and employee address. Now if one employee changes its location then we would have to update the table. But the problem is, if the table is not normalized one employee can have multiple entries and while updating all of those entries one of them might get missed.

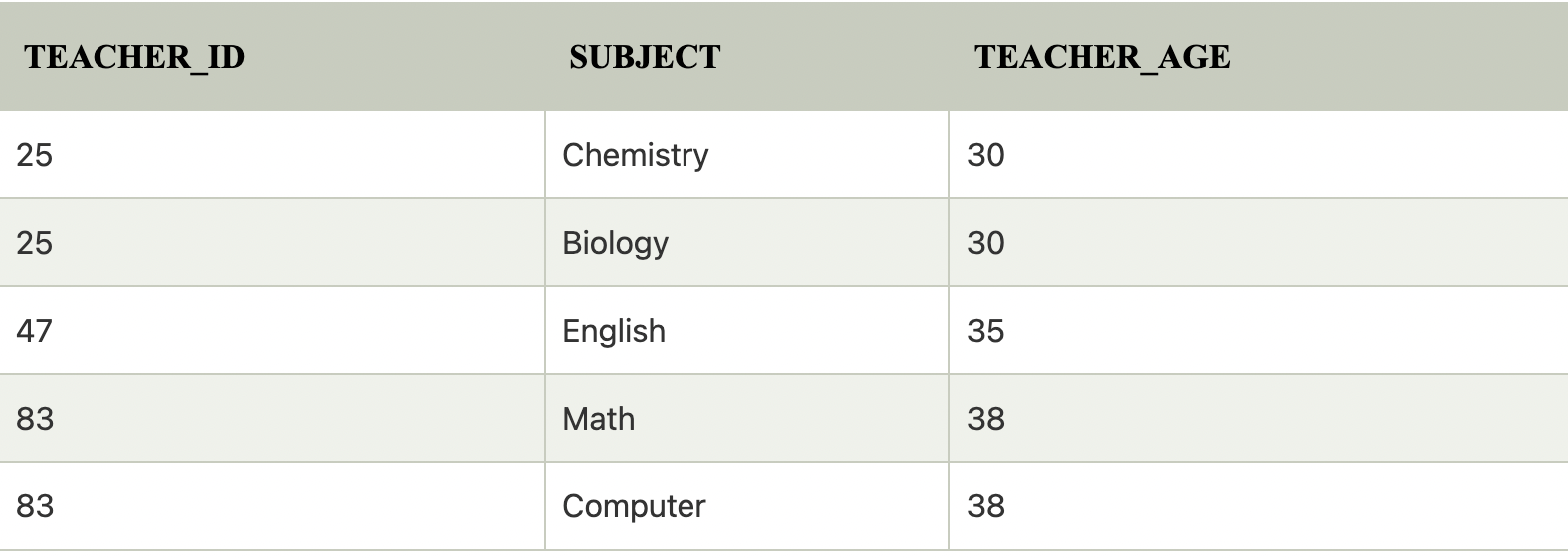
1. **First Normal Form(1NF)**
   * If any table contains the multi attribute value, then the table is considered it is not in first normal form.



* + Three ways to divide the table
    - Create multiple tuples by spreading the multiattribute value in each tuple which creates many duplicate tuples.
    - Create different columns of multiattribute value but other tuples may have more null value across the table.
    - Dividing the table base table and referencing table.

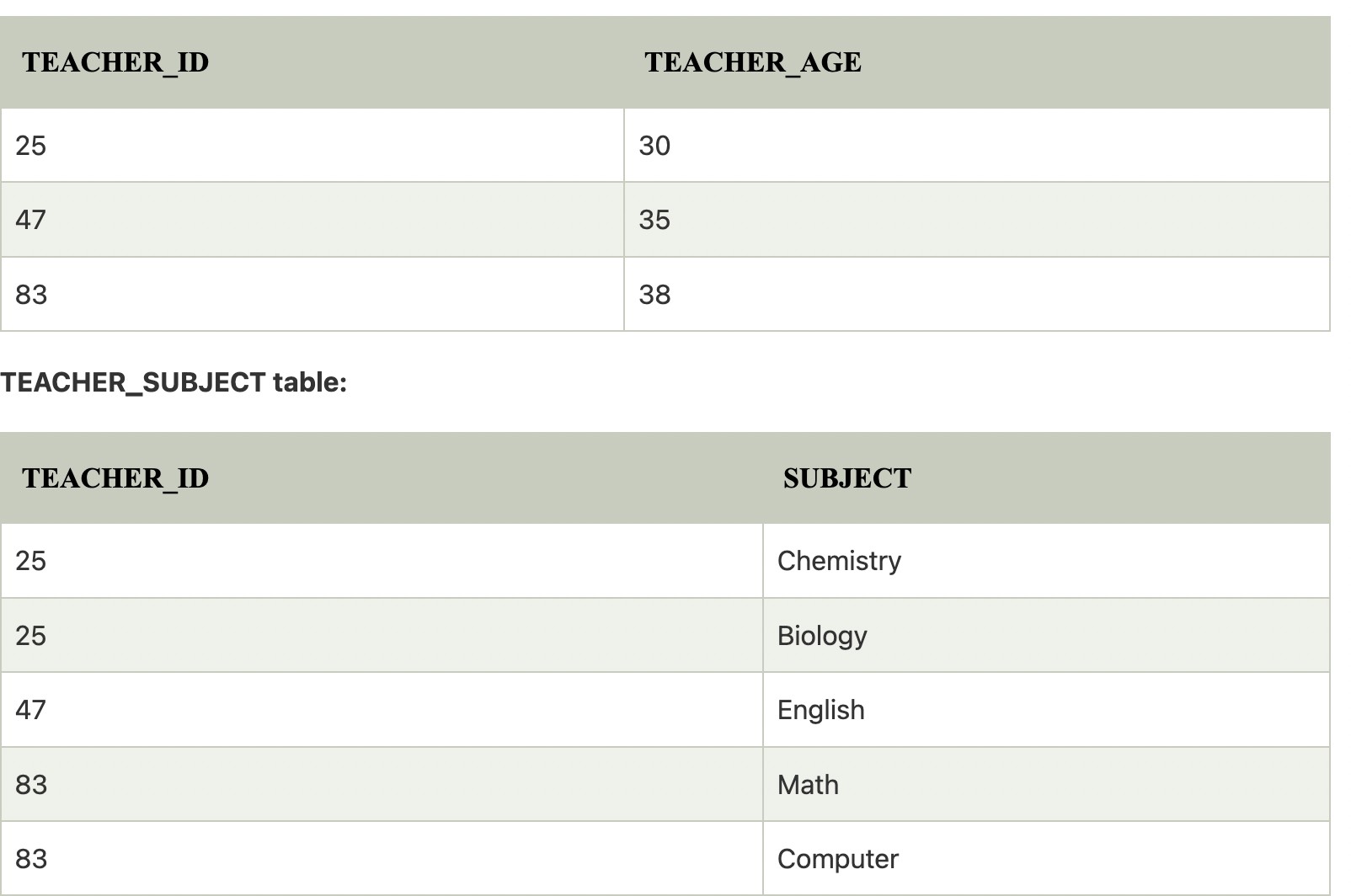
### Second Normal Form(2NF)

* + **Rules**
    - Table or relationship should be in 1nf
    - All non-prime attributes should be fully functionally/not partially dependent on the primary key.(candidate key)
      * Non-prime attribute means the attribute which is not participating in the formation of the candidate key.



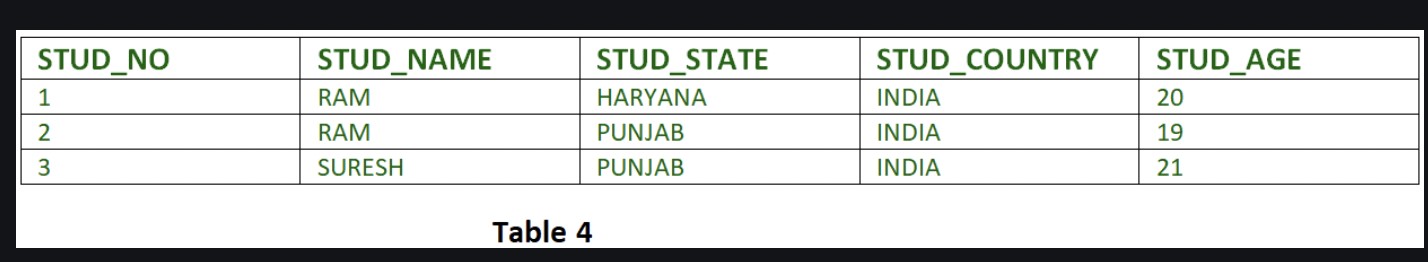
* + In the given table, non-prime attribute TEACHER\_AGE is dependent on TEACHER\_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:



### Third Normal Form (3NF)

* + **Rules**
    - Table should be in 2nf.
    - There should be no transitive dependency in the table.(non prime attribute should not be determined by non prime attribute)
    - Example
      * x(primary key) -> y(non prime attribute) & y->z(non prime attribute)
      * transitive property, x -> z through non prime attribute y
      * So such transitive dependency should not exist.



* Fd set:

{stud\_no -> stud\_name, stud\_no -> stud\_state, stud\_state -> stud\_country, stud\_no -> stud\_age}

* candidate key:

{stud\_no}

* for this relation in table 4, stud\_no -> stud\_state and stud\_state -> stud\_country are true. So stud\_country is transitively dependent on stud\_no. It violates the third normal form. To convert it in third normal form, we will decompose the relation student (stud\_no, stud\_name, stud\_phone, stud\_state, stud\_country\_stud\_age) as:
* Student (stud\_no, stud\_name, stud\_phone, stud\_state, stud\_age) state\_country (state, country)

### Boyce Code Normal Form (BCNF)

* + **Rule**
    - BCNF is the advanced version of 3NF. It is stricter than 3NF.
    - A table is in BCNF if every functional dependency X →

Y, X is the super key of the table.

* + - For BCNF, the table should be in 3NF, and for every FD, LHS is a super key.

1. **3nf always ensures ‘dependency preservation decomposition’ but not in BCNF. However, both 3nf and bcnf ensures lossless decomposition.**

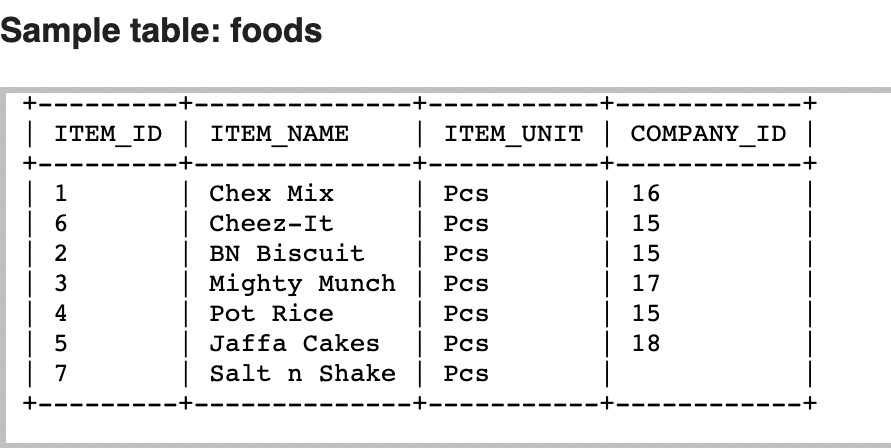
## SQL JOINS

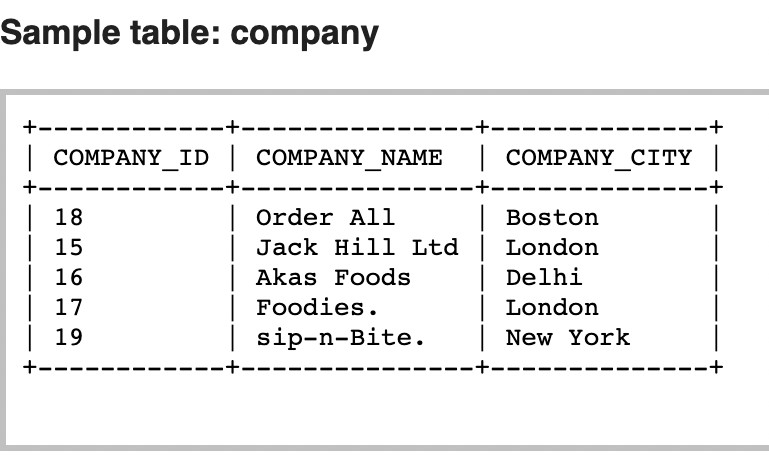
### Rules

* + - Should have common attribute in both tables.
    - Cross product + condition

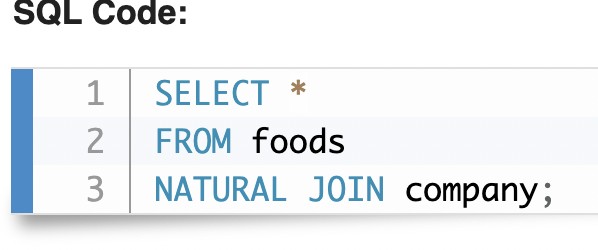
### Natural join:-

* + Natural join is an sql join operation that creates join on the base of the common columns in the tables. To perform natural join there must be one common attribute(column) between two tables. Natural join will retrieve from multiple relations.





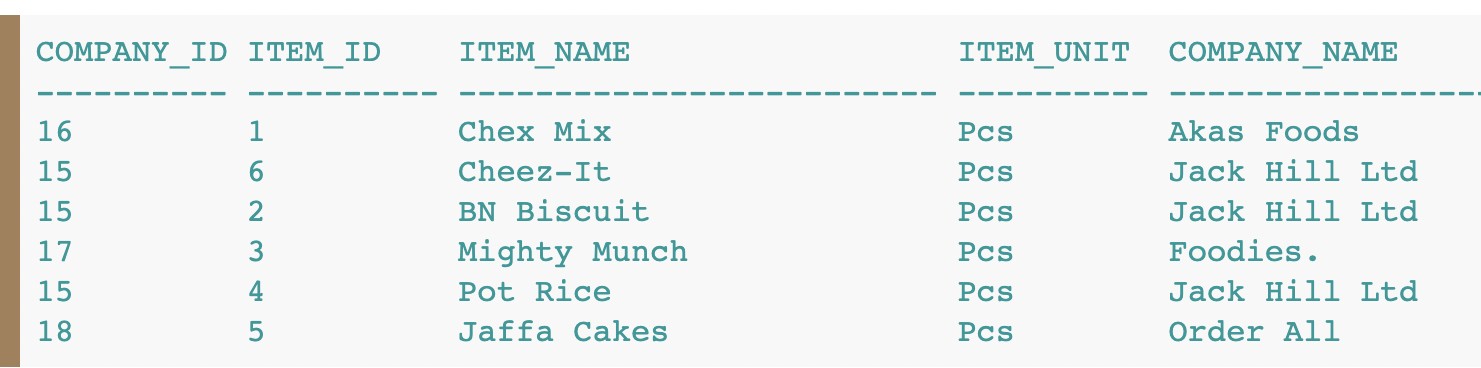
### Code:-



**Meaning can be understand as :-**

SELECT \* FROM foods,company WHERE foods.company\_id = company.company\_id

### Output:-

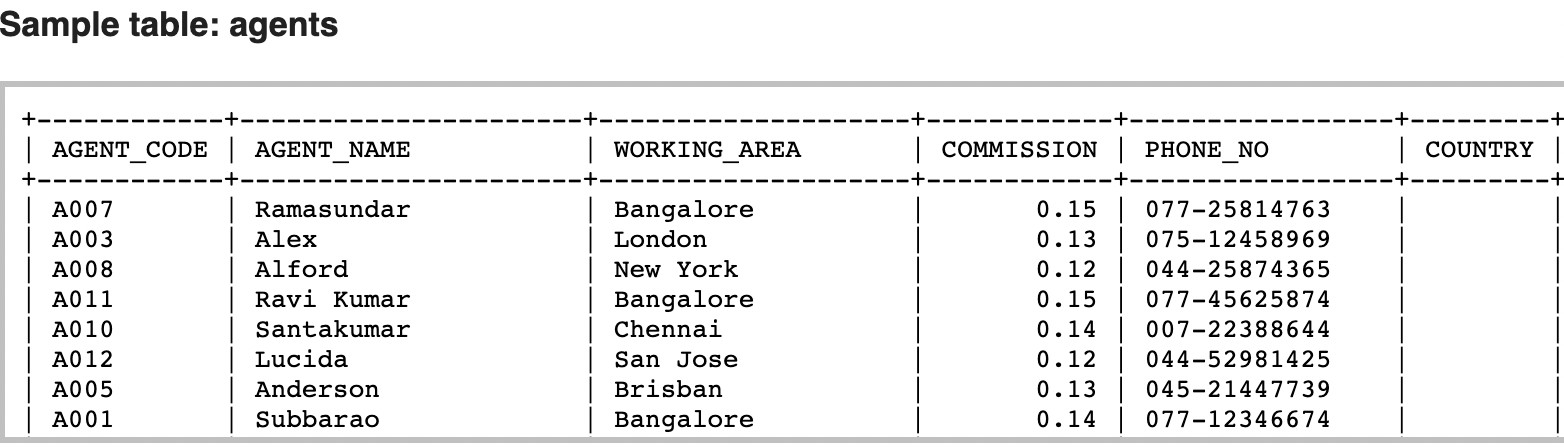


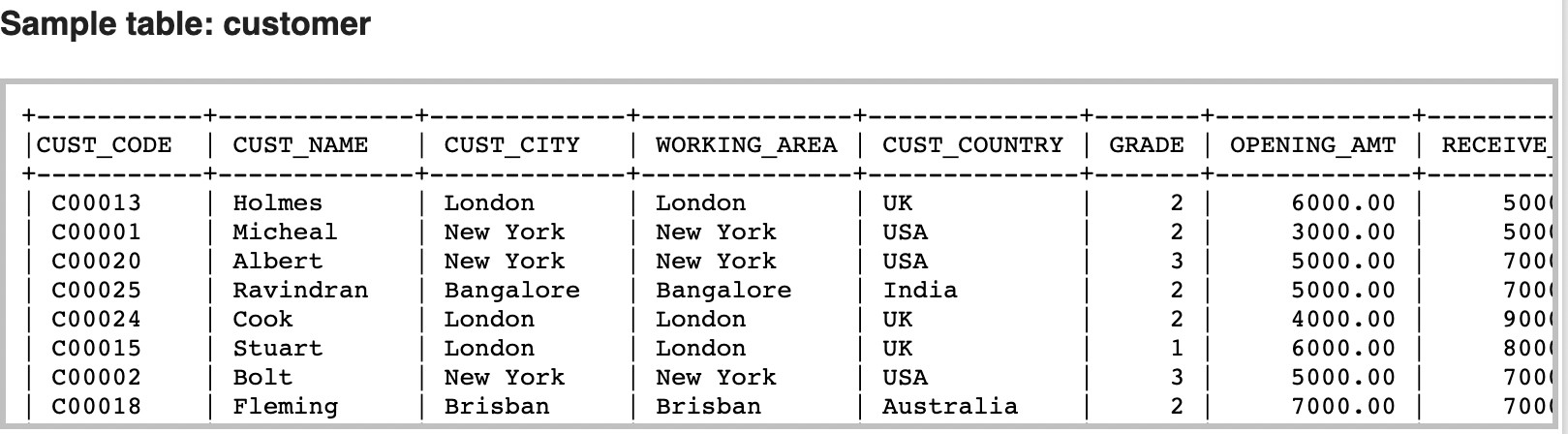
1. **SELF JOIN**
   * A self join is a regular join, but the table is joined with itself.
   * Select column\_name(s)

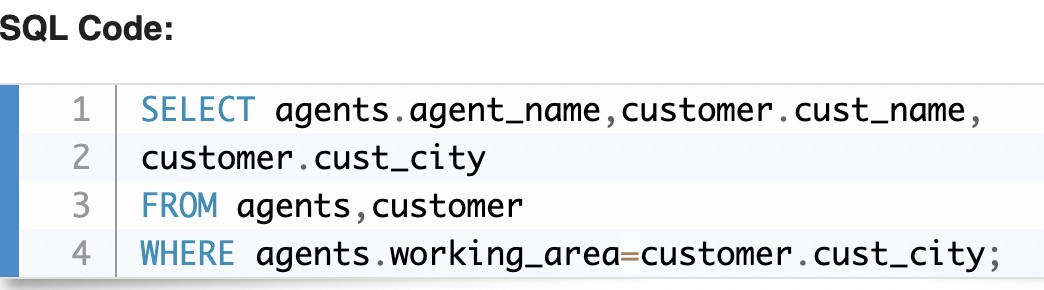
from table1 t1, table1 t2 // cross product with itself where condition;

* + Example:- **find student id who is enrolled in at least two courses**
    - SELECT t1.id from study AS t1, study AS t2 WHERE t1.sid = t2.sid AND t1.cid <> t2.cid

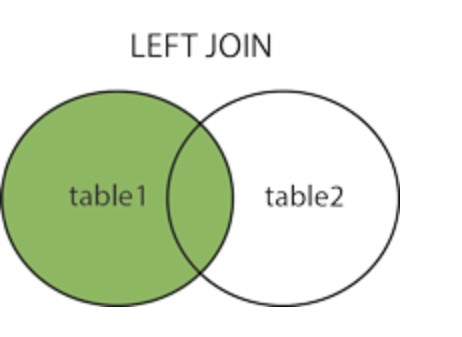
1. **Equi Join:-**
   * In natural join, (‘=’) used for common/same attribute in table like emp.id = dept.id;
   * But in Equi join, (‘=’) used with any attribute like my.address = dept.location
   * The equi join creates a JOIN for equality or matching of the single or multiple column values of the relative tables. Apart from that, the equi join also creates the JOIN by using JOIN along with the ON clause and then providing the names of the columns with their relative tables to check equality using the equal operator.







1. **Outer Joins**
2. **Left outer join**
   * The LEFT JOIN keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

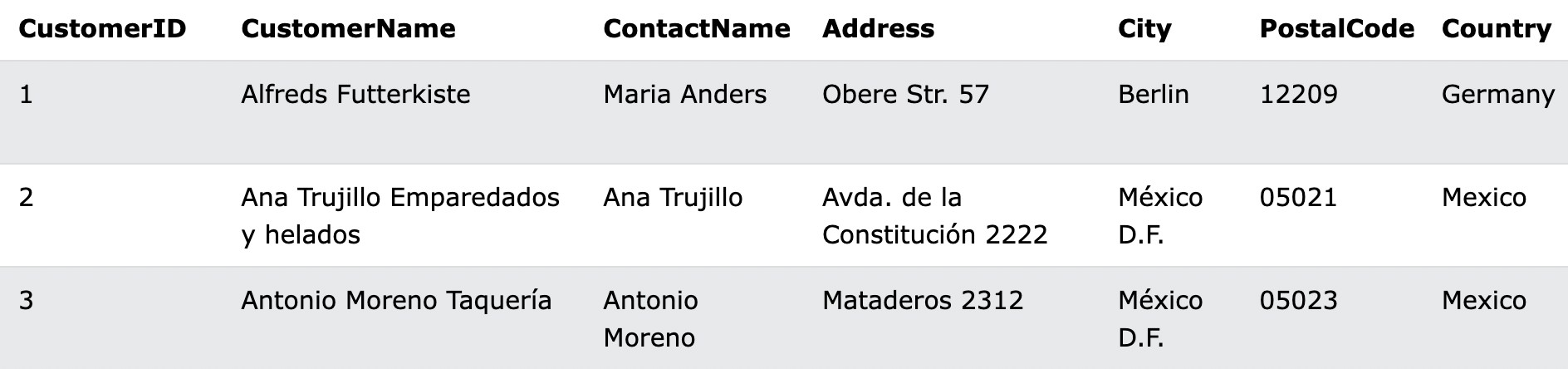


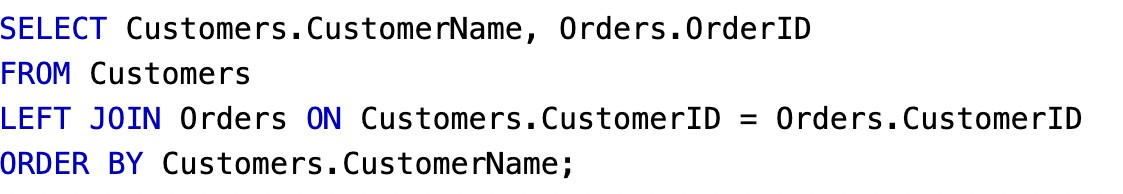
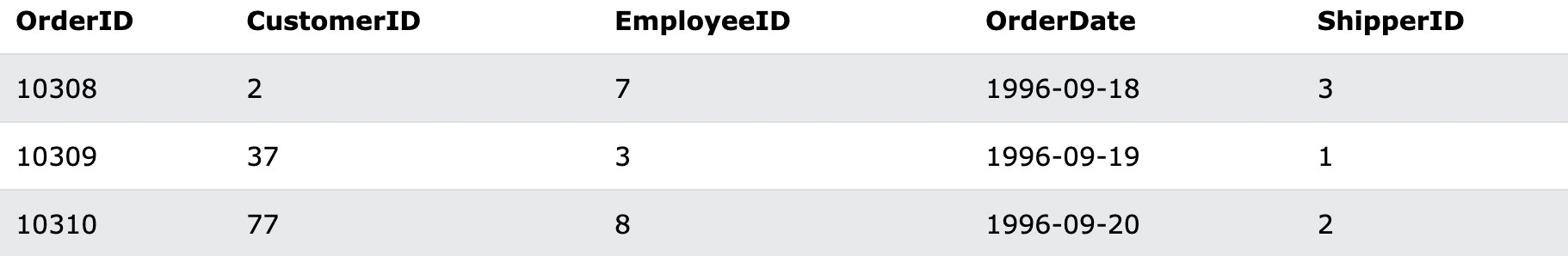
* + **Syntax**
  + SELECT *column\_name(s)*

FROM *table1*

LEFT JOIN *table2*

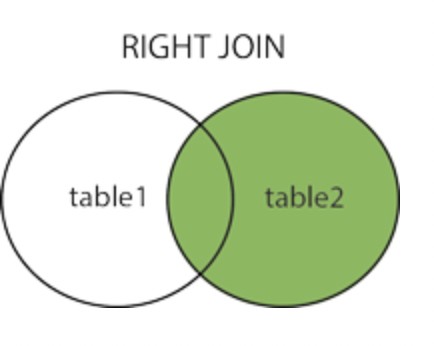
ON *table1.column\_name* = *table2.column\_name*;





### Right Outer Join

* + The right join keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.

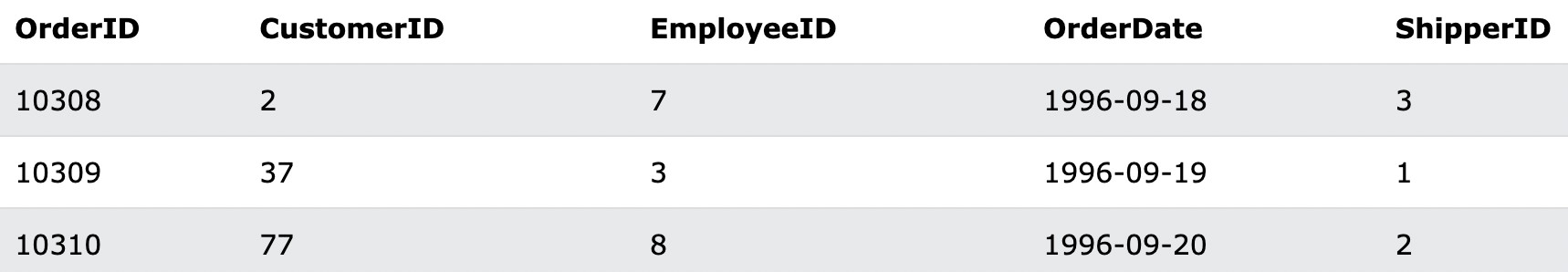


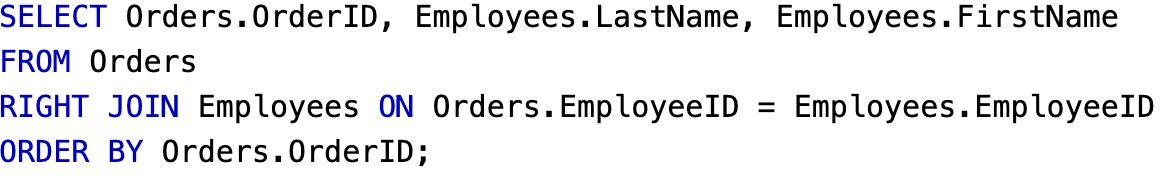
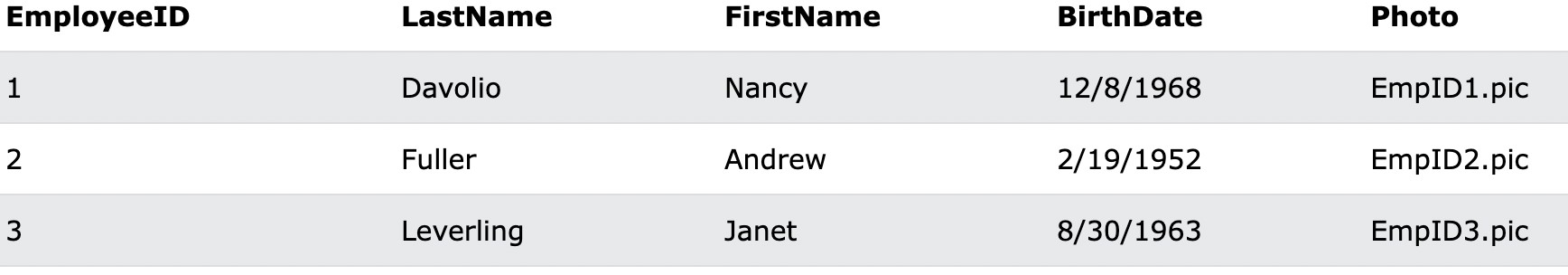
* + **syntax:-**
  + SELECT *column\_name(s)*

FROM *table1*

RIGHT JOIN *table2*

ON *table1.column\_name* = *table2.column\_name*;





### Full Outer Join

* + SELECT *column\_name(s)*

FROM *table1*

FULL OUTER JOIN *table2*

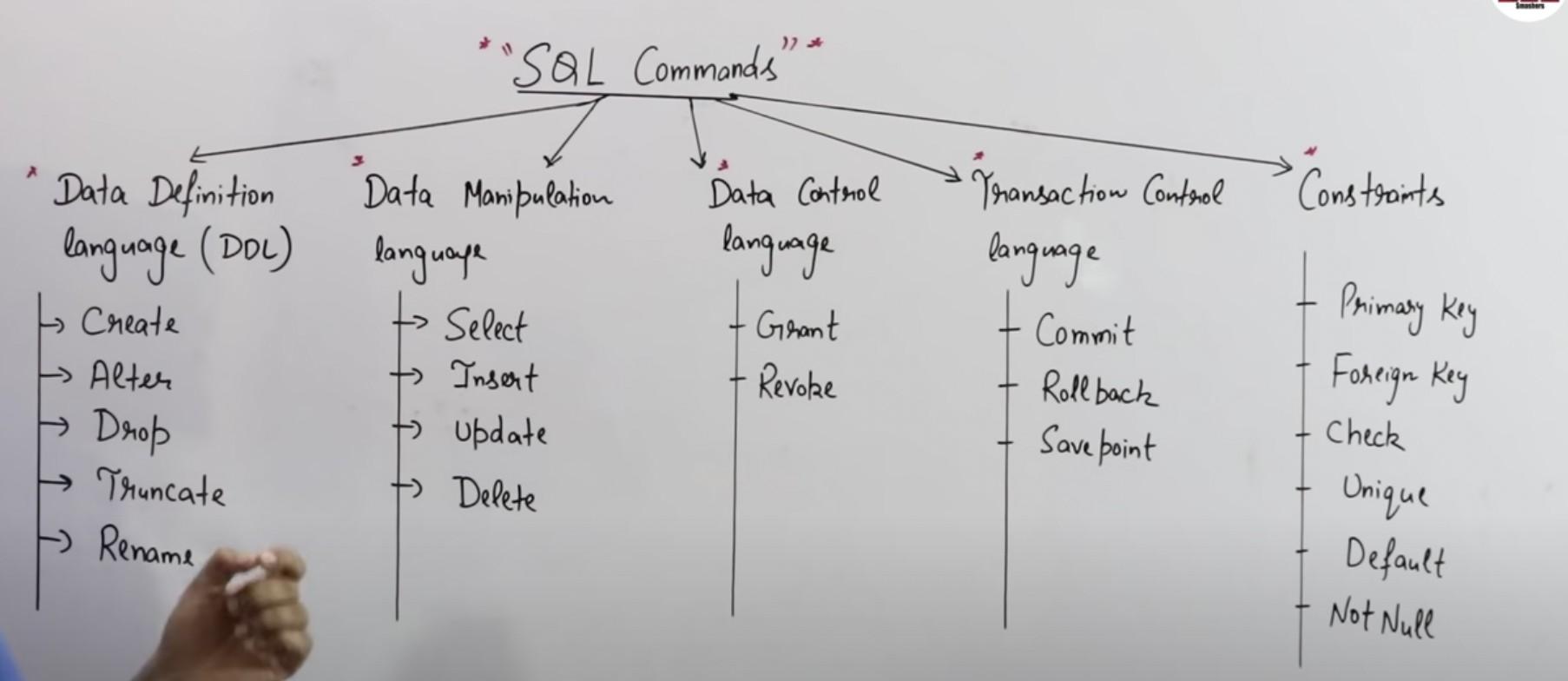
ON *table1.column\_name* = *table2.column\_name*

WHERE *condition*;

1. **InnerJoin**
   * Inner join joins two tables on the basis of the column which is explicitly specified in the on clause. The resulting table will contain all the attributes from both the tables including the common column also.

### Structured Query Language Commands(SQL commands)

* + Fixed structure, domain specific language(only on relational database)
  + Declarative language(what to do , but not how to do but in case of pl sql both included )
  + 1970, Ef code with the help of relation model, relation algebra then ibm implement it.



### Data Definition Language

* + **Alter is DDL** and **update is DML**, alter used for altering the structure of schema on the other hand update used for manipulation of data not schema
  + **Delete is DML** which deletes the data one per queries, **Drop is DDL** removes the whole schema whereas **Truncate is DDL** but it removes all tuples at one go.

## CREATE:-

* + CREATE TABLE EMP( ID INT,

NAME VARCHAR2(20); SALARY NUMBER(10);

)

DESC EMP // describe the creation of the table

### Alter:-

* + Add and remove columns, modify data type and data type length, add and remove constraint, rename columns and tables in schema.
  + ALTER TABLE *table\_name*

ADD *column\_name datatype*;

* + ALTER TABLE *table\_name*

DROP COLUMN *column\_name*;

### Drop:-

* + DROP DATABASE databasename;
  + DROP TABLE *table\_name*;

### Constraint Commands

* + [NOT NULL](https://www.w3schools.com/sql/sql_notnull.asp) - Ensures that a column cannot have a NULL value
  + [UNIQUE](https://www.w3schools.com/sql/sql_unique.asp) - Ensures that all values in a column are different
  + [PRIMARY KEY](https://www.w3schools.com/sql/sql_primarykey.asp) - A combination of NOT NULL and UNIQUE. Uniquely identifies each row in a table
  + [FOREIGN KEY](https://www.w3schools.com/sql/sql_foreignkey.asp) - Prevents actions that would destroy links between tables
  + [CHECK](https://www.w3schools.com/sql/sql_check.asp) - Ensures that the values in a column satisfies a specific condition
  + [DEFAULT](https://www.w3schools.com/sql/sql_default.asp) - Sets a default value for a column if no value is specified
  + [CREATE INDEX](https://www.w3schools.com/sql/sql_create_index.asp) - Used to create and retrieve data from the database very quickly
  + **Example:- (not null, primary, foriegn key)**
    - **CREATE TABLE Orders (**

**OrderID int NOT NULL PRIMARY KEY,**

**OrderNumber int NOT NULL,**

**PersonID int FOREIGN KEY REFERENCES**

**Persons(PersonID)**

**);**

* + **Example:- Unique**
    - **CREATE TABLE Persons (**

**ID int NOT NULL UNIQUE,**

**LastName varchar(255) NOT NULL, FirstName varchar(255),**

**Age int**

**);**

* + **Example:- check**
    - **CREATE TABLE Persons (**

**ID int NOT NULL,**

**LastName varchar(255) NOT NULL, FirstName varchar(255),**

**Age int CHECK (Age>=18)**

**);**

* + **Example:- default**
    - **CREATE TABLE Persons (**

**ID int NOT NULL,**

**LastName varchar(255) NOT NULL, FirstName varchar(255),**

**Age int,**

**City varchar(255) DEFAULT 'Sandnes'**

**);**

### Data Manipulation Language:-

1. **Insert:-**
   * INSERT INTO *table\_name* (*column1*, *column2*, *column3*, ...) VALUES (*value1*, *value2*, *value3*,

...); OR

* + INSERT INTO *table\_name* VALUES (*value1*, *value2*, *value3*, ...);

1. **Select:-**
   * SELECT *column1*, *column2, ...*

FROM *table\_name*;

1. **Update:-**
   * UPDATE *table\_name* SET *column1* = *value1*, *column2*

= *value2*, ...

* + WHERE *condition*;

1. **Delete:-**
   * DELETE FROM *table\_name* WHERE *condition*;

### Sql Statements & Operators:-

1. **Distinct:-**
   * return only distinct (different) values.
   * **Example:-**
   * SELECT DISTINCT *column1*, *column2, …* FROM *table\_name*;

### And, Or, Not:-

* + Where clauses can be combined with And, Or, and Not.
  + AND and OR operators are used to filter records based on more than one condition
  + **And:-**
    - SELECT *column1*, *column2, ...*
    - FROM *table\_name*
    - WHERE *condition1* AND *condition2* AND *condition3*

*...*;

* + **OR:-**
    - SELECT *column1*, *column2, ...*
    - FROM *table\_name*
    - WHERE *condition1* OR *condition2* OR *condition3*

*...*;

* + **Not:-**
    - SELECT *column1*, *column2, ...*
    - FROM *table\_name*
    - WHERE NOT *condition*;

### Order By:-

* + The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.
  + Example:-
    - SELECT *column1*, *column2, ...*
    - FROM *table\_name*
    - ORDER BY *column1, column2, ...* ASC|DESC;

### Aggregate:-

* **max()**
  + SELECT MAX(*column\_name*)FROM *table\_name*

WHERE *condition*;

## MIN()

* + SELECT MIN(*column\_name*)FROM *table\_name*

WHERE *condition*;

### Count()

* + SELECT COUNT(*column\_name*)FROM *table\_name*

WHERE *condition*;

### Avg()

* + SELECT AVG(*column\_name*)FROM *table\_name*

WHERE *condition*;

### sum()

* + SELECT SUM(*column\_name*)FROM *table\_name*

WHERE *condition*;

### Like operator(with where):-

* + The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.
  + The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.
    - There are two wildcards often used in conjunction with the LIKE operator:
    - The percent sign (%) represents zero, one, or multiple characters
    - The underscore sign (\_) represents one, single character
  + Syntax:-
    - SELECT *column1, column2, ...*
    - FROM *table\_name*
    - WHERE *columnN* LIKE *pattern*;
  + Different like operator with %, \_
    - Where customername like 'a%' // find any value start with ‘a’
    - Where customername like '%a' // find any value end with ‘a’
    - Where customername like '%or%' // find any value that have ‘or’ in any position
    - Where customername like '\_r%' // find any value ‘r’ in second position
    - Where customername like 'a\_%' // find any value start with a and at least two character length
    - Where customername like 'a %' // find any value start with a and at least three character length
    - Where customername not like 'a%o' // find any value start with a end with o

### WildCards(with like):-

* + A wildcard character is used to substitute one or more characters in a string.
  + Wildcard characters are used with the [LIKE](https://www.w3schools.com/sql/sql_like.asp) operator. The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.
  + Some wild character:-
    - ‘%’ -> zero or more characters like bl%-> black,blue…
    - ‘\_’ -> single characters like h\_t -> hat, hot…
    - ‘[] ‘-> any character within the bracket like h[oa]t -> hot,hat,...
    - ‘^’ -> any character not in the bracket like h[^oa]t ->hit not hot, hat
    - ‘-’ -> any character within specified range like c[a-d]t -> cat,cdt

### In operator:-

* + The IN operator allows you to specify multiple values in a

WHERE clause.

* + The IN operator is a shorthand for multiple OR conditions.
  + Syntax:-
    - SELECT *column\_name(s)*
    - FROM *table\_name*
    - WHERE *column\_name* IN (*value1*, *value2*, ...);

# or

* + - SELECT *column\_name(s)*
    - FROM *table\_name*
    - WHERE *column\_name* IN (*SELECT STATEMENT*);
  + **Example:-**
    - SELECT \* FROM Customers
    - WHERE Country IN (SELECT Country FROM Suppliers);

1. **Between:-**
   * SELECT *column\_name(s)*
     + FROM *table\_name*
     + WHERE *column\_name* BETWEEN *value1* AND *value2;*
2. **Alias:-**
   * SELECT *column\_name* AS *alias\_name*
     + FROM *table\_name;*
3. **Union:-**
   * The UNION operator is used to combine the result-set of two or more SELECT statements.
     + Every SELECT statement within UNION must have the same number of columns
     + The columns in every SELECT statement must also be in the same order
     + The columns must also have similar data types
   * **Syntax:-**
     + SELECT *column\_name(s)* FROM *table1*
     + UNION
     + SELECT *column\_name(s)* FROM *table2*;
   * **Example:-**
     + SELECT City, Country FROM Customers
     + WHERE Country='Germany'
     + UNION
     + SELECT City, Country FROM Suppliers
     + WHERE Country='Germany'
     + ORDER BY City;
4. **Group By:-**
   * The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".
   * The GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.
   * **Syntax:-**
     + SELECT *column\_name(s)*
     + FROM *table\_name*
     + WHERE *condition*
     + GROUP BY *column\_name(s)*
     + ORDER BY *column\_name(s);*
   * ***Example:-***
     + SELECT COUNT(CustomerID), Country
     + FROM Customers
     + GROUP BY Country
     + ORDER BY COUNT(CustomerID) DESC;

### Having:-

* + The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.
  + **Syntax:-**
    - SELECT *column\_name(s)*
    - FROM *table\_name*
    - WHERE *condition*
    - GROUP BY *column\_name(s)*
    - HAVING *condition// aggregate function*
    - ORDER BY *column\_name(s);*
  + ***Example*:-**
    - *SELECT COUNT(CustomerID), Country*
    - *FROM Customers*
    - *GROUP BY Country*
    - *HAVING COUNT(CustomerID) > 5*
    - *ORDER BY COUNT(CustomerID) DESC;*

### Exists:-

* + The EXISTS operator is used to test for the existence of any record in a subquery.
  + The EXISTS operator returns TRUE if the subquery returns one or more records.
  + **Syntax:-**
    - SELECT *column\_name(s)*
    - FROM *table\_name*
    - WHERE EXISTS
    - (SELECT *column\_name* FROM *table\_name* WHERE

*condition*);

* + **Example:-**
    - SELECT SupplierName
    - FROM Suppliers
    - WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID = Suppliers.supplierID AND Price < 20);

### Any/All:-

* + The ANY and ALL operators allow you to perform a comparison between a single column value and a range of other values.
  + The ANY operator:
    - returns a boolean value as a result
    - returns TRUE if ANY of the subquery values meet the condition
  + **Syntax:-**
    - SELECT *column\_name(s)*
    - FROM *table\_name*
    - WHERE *column\_name operator* ANY
    - (SELECT *column\_name*
    - FROM *table\_name*
    - WHERE *condition*);
  + ALL means that the condition will be true only if the operation is true for all values in the range.
  + The ALL operator:
    - returns a boolean value as a result
    - returns TRUE if ALL of the subquery values meet the condition
    - is used with SELECT, WHERE and HAVING statements
  + **Syntax:-**
    - SELECT ALL *column\_name(s)*
    - FROM *table\_name*
    - WHERE *condition*;

# Or

* + - SELECT *column\_name(s)*
    - FROM *table\_name*
    - WHERE *column\_name operator* ALL
    - (SELECT *column\_name*
    - FROM *table\_name*
    - WHERE *condition*);

**O**.**Summary :-**

* **Distinct** used with **Select**.
* **Where** clauses can be combined with **And, Or, and Not.**
* **Like** operator used with **where**.
* **Wildcard** used with **like**.
* **Aggregate** can be used as **normal** and also can be used with

**Having** Clause.

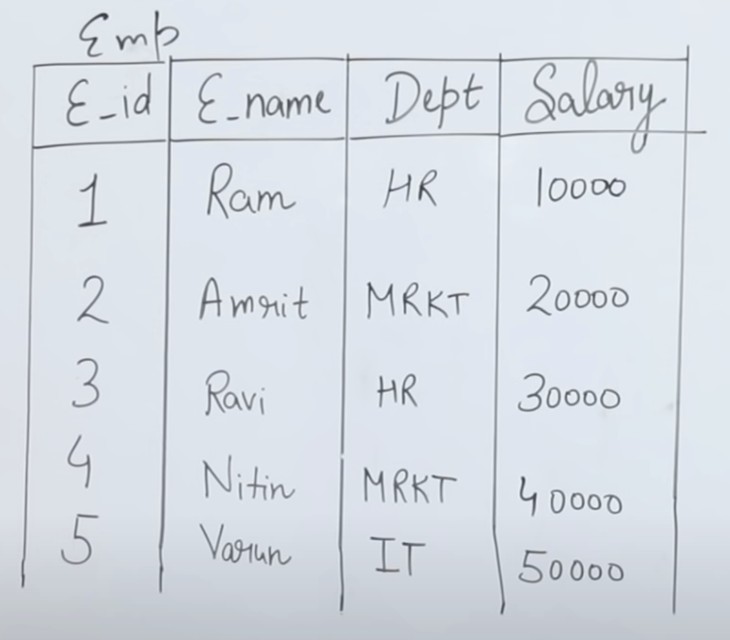
* **GroupBy** often used with **Aggregate** function
* **OrderBy** can be used at the end of a statement,compatible with anything.
* **IN** operator is a shorthand for multiple **OR** conditions also used in nested queries .
* **Exist** used for subqueries.
* **Any/All** used for subqueries.

### Extra:-

* + Replace()
  + Intersection()
  + Minus()
  + Concat()
  + Mid() / SubString()
  + Locate() / CharIndex()
  + Upper()/ Lower()
  + Using
  + Ltrim()/ Rtrim()
  + Null
  + Now() / getDate()

**10. Practice Question**

1. **Subqueries:-**



* + **Write a sql query to display maximum salary from emp table.**
    - SELECT MAX(salary) FROM Emp;

### Write a sql query to display the employee name who has a maximum salary.

* + - SELECT e\_name FROM Emp WHERE Salary = (SELECT MAX(Salary) FROM Emp);

### Write a sql query to display the second highest salary from emp table.

* + - SELECT MAX(Salary) FROM Emp WHERE Salary != (SELECT MAX(Salary) FROM Emp);

### Write a sql query to display the employee name who has a second highest salary from emp table.

* + - SELECT e\_name FROM Emp WHERE Salary = ((SELECT MAX(Salary) FROM Emp WHERE Salary != (SELECT MAX(Salary) FROM Emp));

### Write a query to display all the department names along with the number of emps working in that?

* + - SELECT dept, COUNT(\*) from Emp Group By dept;
    - If **group by** mention the **dept column** then in the **select** section , **dept** should be mentioned **no other column** attribute can be mentioned, except aggregate function.

### Write a query to display all the department names where number of emps are less than 2?

* + - SELECT dept FROM Emp GROUP BY dept
    - HAVING COUNT(\*) < 2;

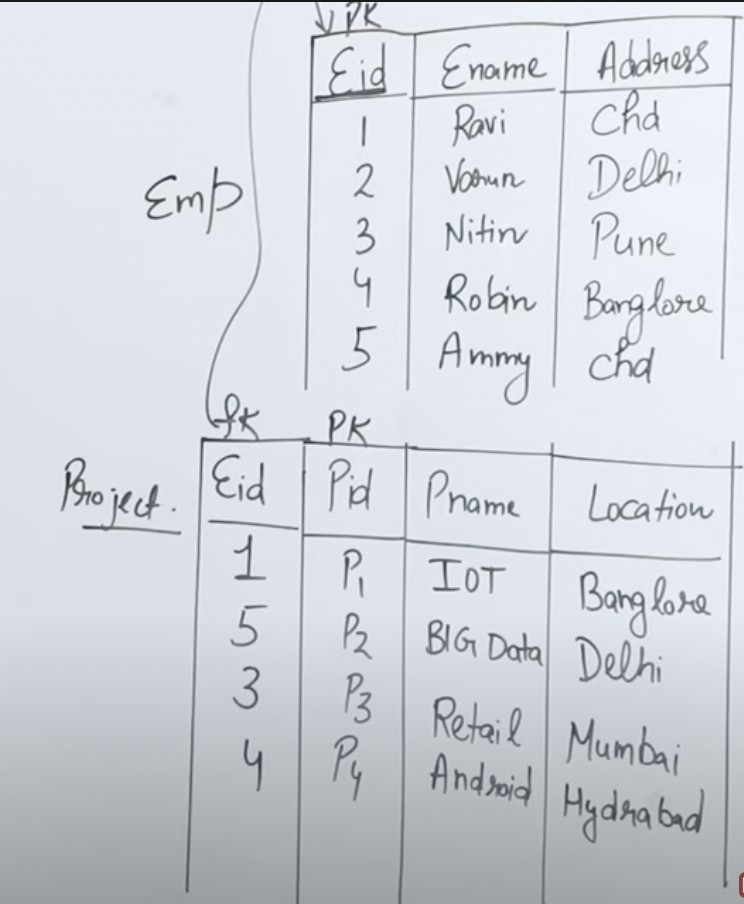
### Write a query to display the employee name where the number of employee in department is less than 2

* + - SELECT e\_name FROM Emp WHERE dept IN (SELECT dept FROM Emp GROUP BY dept HAVING COUNT(\*) < 2);
    - Inner query return dept having count less 2
    - Outer query just checks with multiple dept received from inner query.

### Write a query to display the highest salary department wise and name of the employee who is having that salary.

* + - SELECT e\_name from Emp WHERE Salary IN (SELECT MAX(Salary) FROM Emp GROUP BY dept)

**In/ Not In**



### Detail of employee whose address is either delhi or chd or pune.

* + - SELECT \* FROM Emp WHERE Address IN (‘Delhi’,’Chd’,Pune’)

### Find the name of the employee who is working on the project?

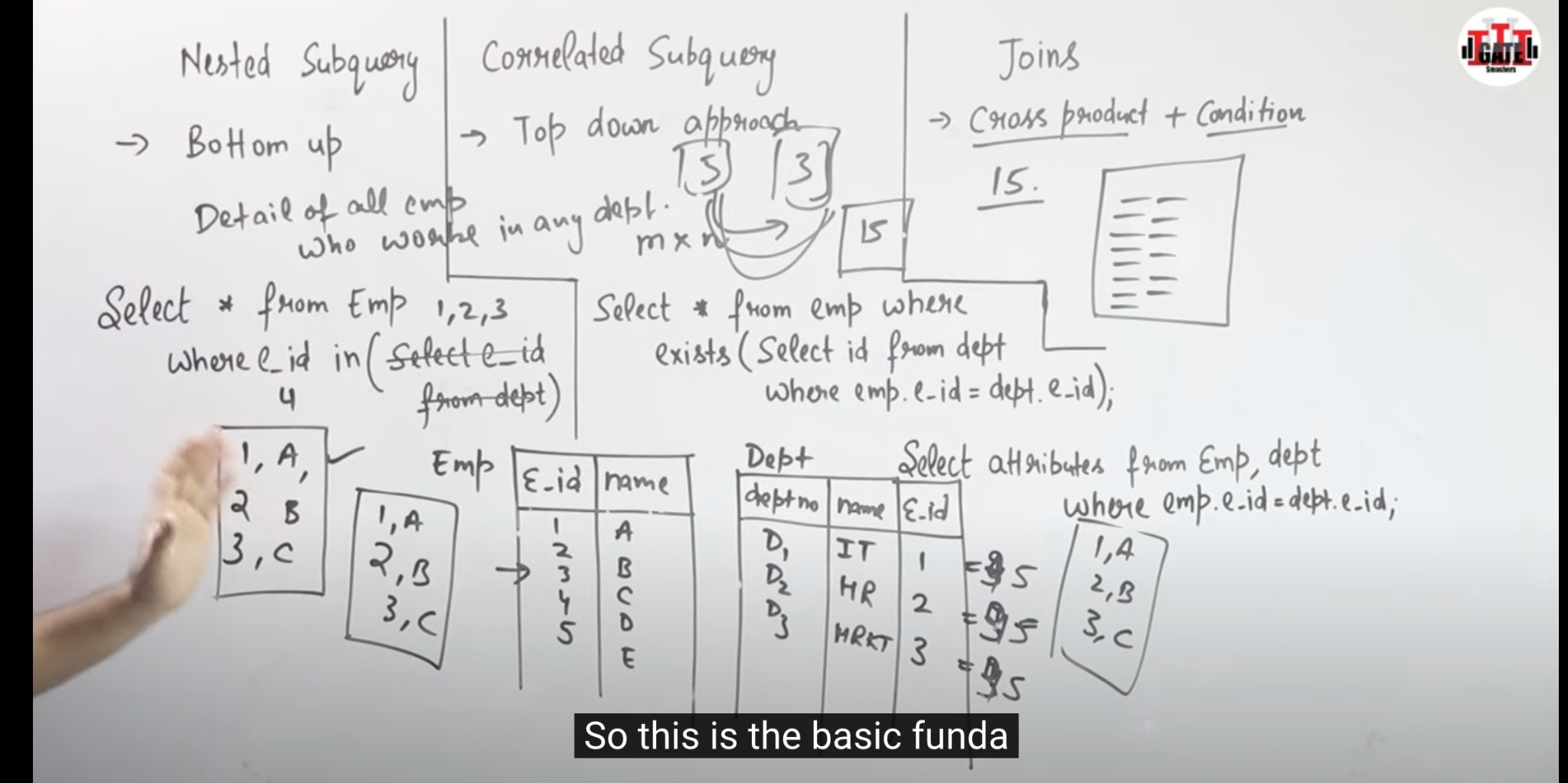
* + - SELECT Ename FROM Emp WHERE Eid In (SELECT Eid FROM Project)

### Correlated Queries:-

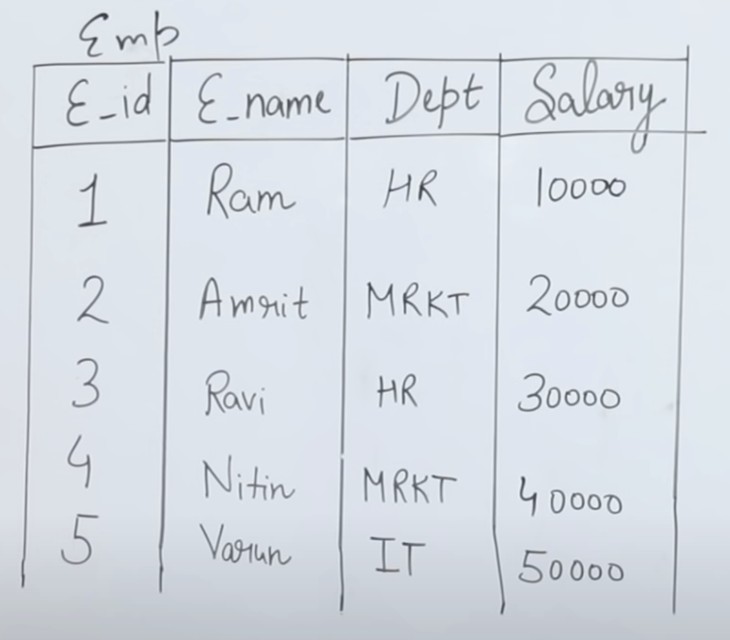
* + There is a relation between inner query and outer query.
  + Above table example.
  + Especially use Exist / Not Exist , relational operator, if/any, and/or
  + What is the difference between nested queries and correlated queries ? :- **well nested queries execute only once and return value while exist return only boolean value in correlated queries**
  + ***Find the details of Emp who is working on at least one project?***
    - SELECT \* FROM Emp WHERE Eid EXISTS (SELECT Eid

FROM Project WHERE Emp.Eid = Project.Eid)

1. **Nested Queries vs Correlated Queries vs Join:-**



1. **Nth highest salary**



* + **Find Nth highest using sql**
    - SELECT E\_id, Salary FROM Emp e1 WHERE N-1 = (SELECT COUNT(Distinct Salary) FROM Emp e2 WHERE e2.salary > e1.salary);

1. **Remaining Topics:**

* **ER design. views**
* **Transaction concurrency in RDBMS.**
* **Relation Algebra in RDBMS.**
* **No SQL and Big Data.**