

### 3. Natural Join

\* grp



- **Natural join** can only be performed **if there is at least one common attribute** (column) that exist between two relations. In addition, the **attributes** must have the **same name and domain**.
- Natural join does not use any comparison operator.
- ***It is same as equi join which occurs implicitly*** by comparing all the **common attributes (columns)** in both relation, **but** difference is that **in Natural Join the common attributes appears only once**. The resulting schema will change.
- **Notation:** **A  $\bowtie$  B**
- **The result of the natural join is the set of all combinations of tuples in two relations A and B that are equal on their common attribute names.**

# Example 1:

$$r = (\underline{A}, \underline{B}, \underline{C}, \underline{D}) \quad s = (\underline{B}, \underline{D}, \underline{E})$$

- Resulting schema of  $\underline{r} \bowtie \underline{s} = (\underline{\underline{A}}, \underline{\underline{B}}, \underline{\underline{C}}, \underline{\underline{D}}, \underline{\underline{E}})$
- $r \bowtie s$  is defined as:

$$\prod_{\underline{r.A}, \underline{r.B}, \underline{r.C}, \underline{r.D}, \underline{s.E}} (\sigma_{r.B = s.B \wedge r.D = s.D} (\underline{\underline{r}} \bowtie \underline{\underline{s}}))$$

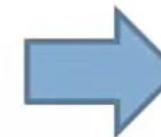
**r**

A	B	C	D
$\alpha$	1	$\alpha$	a
$\beta$	2	$\gamma$	a
$\gamma$	4	$\beta$	b
$\alpha$	1	$\gamma$	a
$\delta$	2	$\beta$	B

**s**

B	D	E
1	a	$\alpha$
3	a	$\beta$
1	a	$\gamma$
2	b	$\delta$
3	b	$\epsilon$



**$r \bowtie s$**

A	B	C	D	E
$\alpha$	1	$\alpha$	a	$\alpha$
$\alpha$	1	$\alpha$	a	$\gamma$
$\alpha$	1	$\gamma$	a	$\alpha$
$\alpha$	1	$\gamma$	a	$\gamma$
$\delta$	2	$\beta$	b	$\delta$



SUBSCRIBE

# Example 2: Natural Join

Courses

CID	Course	Dept
CS01	Database	CS
ME01	Mechanics	ME
EE01	Electronics	EE

HOD

Dept	Head
CS	Rohan
ME	Sara
EE	Jiya

Courses  $\bowtie$  HOD

CID	Course	Dept	Head
CS01	Database	CS	Rohan
ME01	Mechanics	ME	Sara
EE01	Electronics	EE	Jiya

Equivalent to:  $\Pi_{CID, Course, Courses.Dept, Head} (\sigma_{Courses.Dept = HOD.Dept} (Courses \times HOD))$

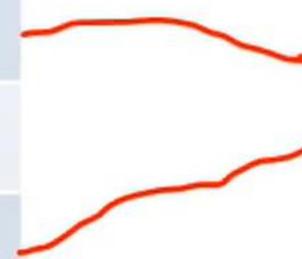
# Example 3: Natural Join

**S1**

sid	name	rating	age
22	dustin	7	45.0
31	lubber	8	55.0
58	rusty	10	35.0

**R1**

sid	bid	day
22	101	10/10/96
58	103	11/12/96



**S1  $\bowtie$  R1**

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
58	rusty	10	35.0	103	11/12/96

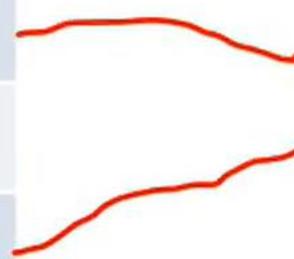
# Example 3: Natural Join

**S1**

sid	name	rating	age
22	dustin	7	45.0
31	lubber	8	55.0
58	rusty	10	35.0

**R1**

sid	bid	day
22	101	10/10/96
58	103	11/12/96



**S1  $\bowtie$  R1**

sid	sname	rating	age	bid	day
✓ 22	dustin	7	45.0	101	10/10/96
✓ 58	rusty	10	35.0	103	11/12/96

Equivalent to:  $\prod_{S1.sid, sname, rating, age, bid, day} (\sigma_{S1.sid = R1.sid} (S1 \times R1))$



# Natural Join



- **Natural join** can only be performed **if there is at least one common attribute** (column) that exist between two relations. In addition, the **attributes** must have the **same name and domain**.
- In **Natural Join** resulting relation the **common attributes appears only once**.
- **Notation:**  $A \bowtie B$       (Natural join does not use any comparison operator)
- The **result of the natural join** is the **set of all combinations of tuples** in two relations **A** and **B** that are **equal on their common attribute names**.
- The **Natural Join** of two relations can be **obtained** by applying a **projection operation to equi join** of two relations. And in terms of basic operators:

**Natural Join = Cartesian product + Selection + Projection**

# Query 1:

- Find the names of all customers who have a loan at the bank, along with the loan number and the loan amount.

- Using Cartesian Product:

$$\Pi_{\text{customer-name}, \text{loan.loan-number}, \text{amount}} (\sigma_{\text{borrower.loan-number} = \text{loan.loan-number}} (\text{borrower} \times \text{loan}))$$

- Using Natural Join:

$$\Pi_{\text{customer-name}, \text{loan.loan-number}, \text{amount}} (\text{borrower} \bowtie \text{loan})$$

branch (branch-name, branch-city, assets)  
 customer (customer-name, customer-street, customer-city)  
 account (account-number, branch-name, balance)  
 ✓ loan (loan-number, branch-name, amount)  
 depositor (customer-name, account-number)  
 ✓ borrower (customer-name, loan-number)

# Query 2:

- Find the names of all branches with customers who have an account in the bank and who live in Harrison.

$\Pi_{\text{branch-name}} (\sigma_{\text{customer-city}=\text{"Harrison"}} (\text{customer} \bowtie \text{account} \bowtie \text{depositor}))$

branch (branch-name, branch-city, assets)
✓ customer(customer-name, customer-street, customer-city)
✓ account(account-number, branch-name, balance)
✓ loan(loan-number, branch-name, amount)
✓ depositor(customer-name, account-number)
borrower(customer-name, loan-number)

- Note: Natural join is associative

$$((\text{customer} \bowtie \text{account}) \bowtie \text{depositor}) = (\text{customer} \bowtie (\text{account} \bowtie \text{depositor}))$$

So we can write it

(**customer**  $\bowtie$  **account**  $\bowtie$  **depositor**) ✓

# Outer Join

- An **Inner join** includes only those tuples with matching attributes and the rest are discarded in the resulting relation. Therefore, we need to use **outer joins** to include all the rest of the tuples from the participating relations in the resulting relation.
- ✓ The **outer join** operation is an **extension of the join operation** that **avoids loss of information**.
- **Outer Join** contains matching tuples that satisfy the matching condition, along with some or all tuples that do not satisfy the matching condition.
  - It is based on both matched or unmatched tuple.
  - It contains all rows from either one or both relations are present
- It uses **NULL** values.
  - NULL signifies that the value is unknown or does not exist

# Example: Inner join (Natural Join)



Courses

CID ✓	Course
100 -	Database
101	Mechanics
102 -	Electronics

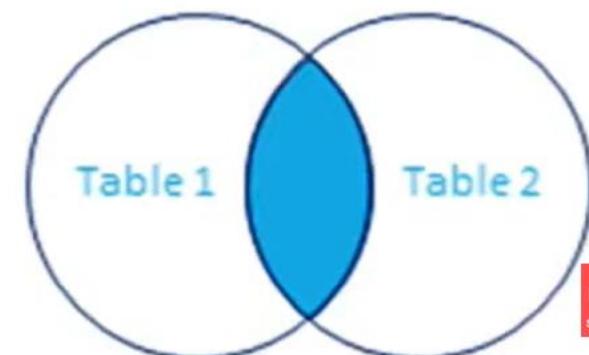
HOD

CID ✓	Name
100	Rohan
102	Sara
104	Jiya

Courses ⚫ HOD

CID	Course	Name
100	Database	Rohan
102	Electronics	Sara

Inner Join ✓



SUBSCRIBE



# Outer Join Types

- Outer Join = Natural Join + Extra information



(from left table, right table or both table)

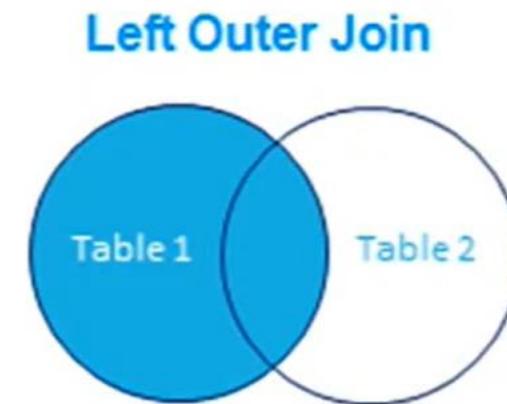
- There are **three kinds of outer joins:**

1. **Left outer join** ✓
2. **Right outer join** ✓
3. **Full outer join** ✓

# 1. Left outer join ( $R1 \bowtie R2$ )

When applying **join** on two relations  $R1$  and  $R2$ , some tuples of  $R1$  or  $R2$  does not appear in result set which does not satisfy the join conditions. **But..**

- In **Left outer join**, **all the tuples from the Left relation  $R1$  are included in the resulting relation.** The tuples of  $R1$  which do not satisfy join condition will have values as **NULL** for attributes of  $R2$ .
- In short:
  - All record from **left** table
  - Only matching records from right table
- **Symbol:**  $\bowtie$
- **Notation:**  $R1 \bowtie R2$



# Example: Left outer join



Courses

CID	Course
100	Database
101	Mechanics
102	Electronics



HOD

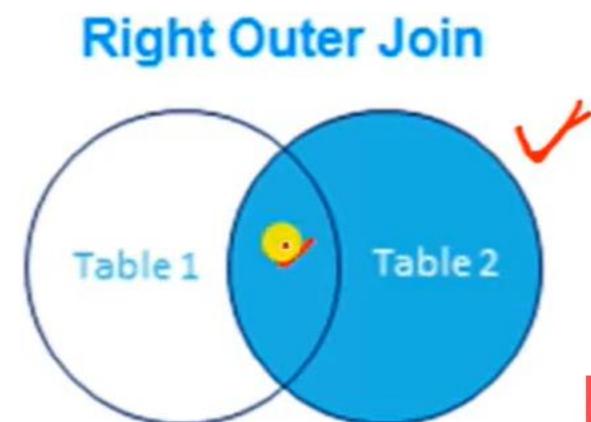
CID	Name
100	Rohan
102	Sara
104	Jiya

Courses  $\bowtie$  HOD

CID	Course	Name
100	Database	Rohan
101	Mechanics	NULL
102	Electronics	Sara

## 2. Right outer join ( $R1 \bowtie R2$ )

- In **Right outer join**, all the tuples from the right relation  $R2$  are included in the resulting relation. The tuples of  $R2$  which do not satisfy join condition will have values as **NULL** for attributes of  $R1$ .
- In short:
  - All record from **right** table ✓
  - Only matching records from left table
- **Symbol:**  $\bowtie$
- **Notation:**  $R1 \bowtie R2$



# Example: Right outer join

**Courses**

CID	Course
100	Database
101	Mechanics
102	Electronics

**HOD**

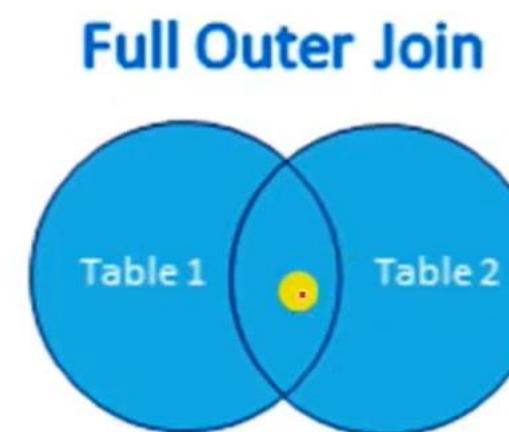
CID	Name
100	Rohan
102	Sara
104	Jiya

**Courses  $\bowtie$  HOD**

CID	Course	Name
100	Database	Rohan
102	Electronics	Sara
104	NULL	Jiya

### 3. Full outer join ( $R1 \bowtie R2$ )

- In **Full outer join**, *all the tuples from both Left relation R1 and right relation R2 are included in the resulting relation.* The tuples of both relations R1 and R2 which do not satisfy join condition, their respective unmatched attributes are made **NULL**.
- In short:
  - All record from **all** table
- **Symbol:**  $\bowtie$
- **Notation:**  $R1 \bowtie R2$



# Example: Full outer join

Courses

CID	Course
100	Database
101	Mechanics
102	Electronics

HOD

CID	Name
100	Rohan
102	Sara
104	Jiya

Courses  $\bowtie$  HOD

CID	Course	Name
100	Database	Rohan
101	Mechanics	NULL
102	Electronics	Sara
104	NULL	Jiya

# Outer Join – Example



## □ Relation **Loan**

loan-number	branch-name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

## □ Relation **borrower**

customer-name	loan-number
Jones	L-170
Smith	L-230
Hayes	L-155

# Inner Join (Natural Join) – Example



**loan**

<b>loan-number</b>	<b>branch-name</b>	<b>amount</b>
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

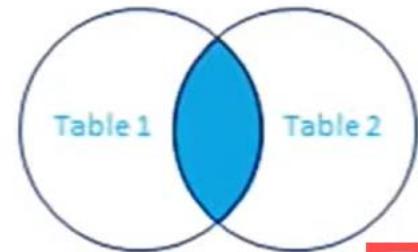
**borrower**

<b>customer-name</b>	<b>loan-number</b>
Jones	L-170
Smith	L-230
Hayes	L-155

**Loan  $\bowtie$  borrower**

<b>loan-number</b>	<b>branch-name</b>	<b>amount</b>	<b>customer-name</b>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

**Inner Join**



# Left Outer Join – Example



**loan**

<b>loan-number</b>	<b>branch-name</b>	<b>amount</b>
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

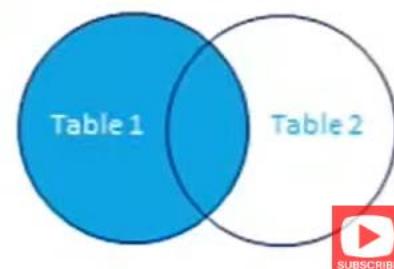
**borrower**

<b>customer-name</b>	<b>loan-number</b>
Jones	L-170
Smith	L-230
Hayes	L-155

**Loan  $\bowtie$  borrower**

<b>loan-number</b>	<b>branch-name</b>	<b>amount</b>	<b>customer-name</b>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	NULL

Left Outer Join



# Right Outer Join – Example

**loan**

<b>loan-number</b>	<b>branch-name</b>	<b>amount</b>
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

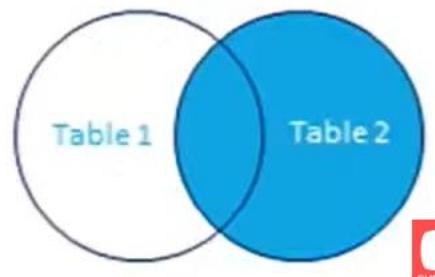
**borrower**

<b>customer-name</b>	<b>loan-number</b>
Jones	L-170
Smith	L-230
Hayes	L-155

**Loan  $\bowtie$  borrower**

<b>loan-number</b>	<b>branch-name</b>	<b>amount</b>	<b>customer-name</b>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	NULL	NULL	Hayes

Right Outer Join



# Full Outer Join – Example



**loan**

<b>loan-number</b>	<b>branch-name</b>	<b>amount</b>
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

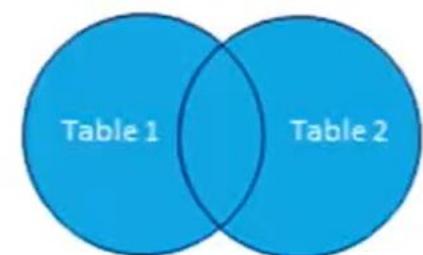
**borrower**

<b>customer-name</b>	<b>loan-number</b>
Jones	L-170
Smith	L-230
Hayes	L-155

**Loan  $\bowtie$  borrower**

<b>loan-number</b>	<b>branch-name</b>	<b>amount</b>	<b>customer-name</b>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	NULL
L-155	NULL	NULL	Hayes

**Full Outer Join**



SUBSCRIBE

# Division Operator ( $\div$ , /)

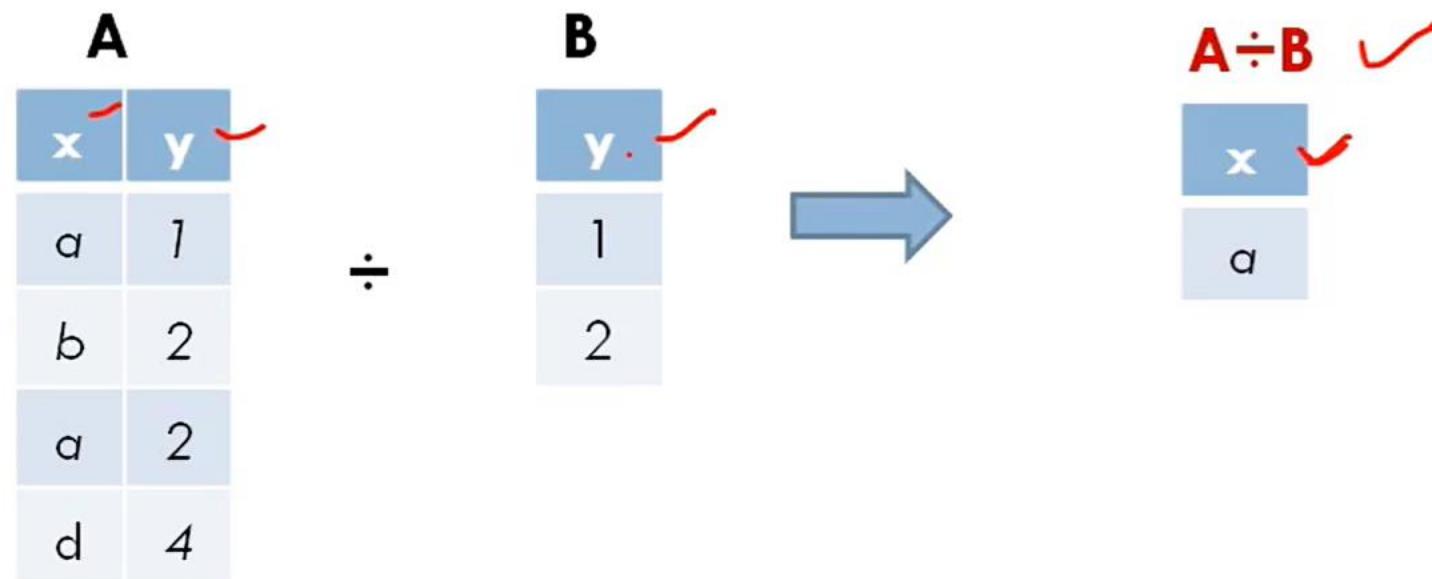
- Division operator is a Derived Operator, not supported as a primitive operator
- Suited to queries that include the keyword “all”, or “every” like “at all”, “for all” or “in all”, “at every”, “for every” or “in every”. Eg:
  - Find the person that has account in all the banks of a particular city
  - Find sailors who have reserved all boats.
  - Find employees who works on all projects of company.
  - Find students who have registered for every course.
- In all these queries, the description after the keyword “**all**” or “**every**” defines a set which contains some elements and the **final result** contains those records who satisfy these requirements.
- **Notation:**  $A \div B$  or  $A/B$  where A and B are two relations

# Division Operator ( $\div$ )...

**Division operator** can be applied if and only if:

- ✓ Attributes of B is **proper subset** of Attributes of A.
- The relation returned by **division operator** will have **attributes = (All attributes of A – All attributes of B)**.
- The relation returned by **division operator** will **return those tuples from relation A which are associated to every B's tuple**.

Example 1:



## Example 2: shows how it works



A ✓

sno	pno
S1	P1
S1	P2 ✓
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

B1 ✓

pno
P2

B2 ✓

pno
P2
P4

B3 ✓

pno
P1
P2
P4

A/B1 ✓

sno
S1
S2
S3
S4

A/B2 ✓

sno
S1 ✓
S4 ✓

A/B3

sno
S1



# Expressing A/B Using Basic Operators



- Division is a **derived operator** (or additional operator).
- Division can be expressed in terms of **Cross Product, Set-Difference and Projection**.

## Idea:

- For  $A/B$ , compute all  $x$  values that are not 'disqualified' by some  $y$  value in  $B$ .
  - $x$  value is disqualified if by attaching  $y$  value from  $B$ , we obtain an **xy tuple** that is not in  $A$ .

**Disqualified x values:**

$$\Pi_X((\Pi_X(A) \times B) - A)$$

So  $\underline{A/B} = \underline{\Pi_X(A)} - \underline{\text{all disqualified tuples}}$

$$\underline{A/B} = \underline{\Pi_X(A)} - \underline{\Pi_X((\Pi_X(A) \times B) - A)}$$

$a$        $a, b, c, d$

$b, d$

A		B		A÷B	
x	y	y	x		
a	1	1		a	
b	2	2		-	
a	2				
d	4				

Handwritten annotations: A red circle highlights the yellow dot in the  $\Pi_X(A)$  term. Red arrows point from the  $\underline{\text{all disqualified tuples}}$  text to the  $(a,2)$  and  $(b,2)$  entries in set A. Red circles highlight the  $(a,1)$  and  $(b,2)$  entries in set A, and the  $(2)$  entry in set B. Red arrows point from the  $\underline{\Pi_X((\Pi_X(A) \times B) - A)}$  term to the  $(a,2)$  and  $(b,2)$  entries in set A.



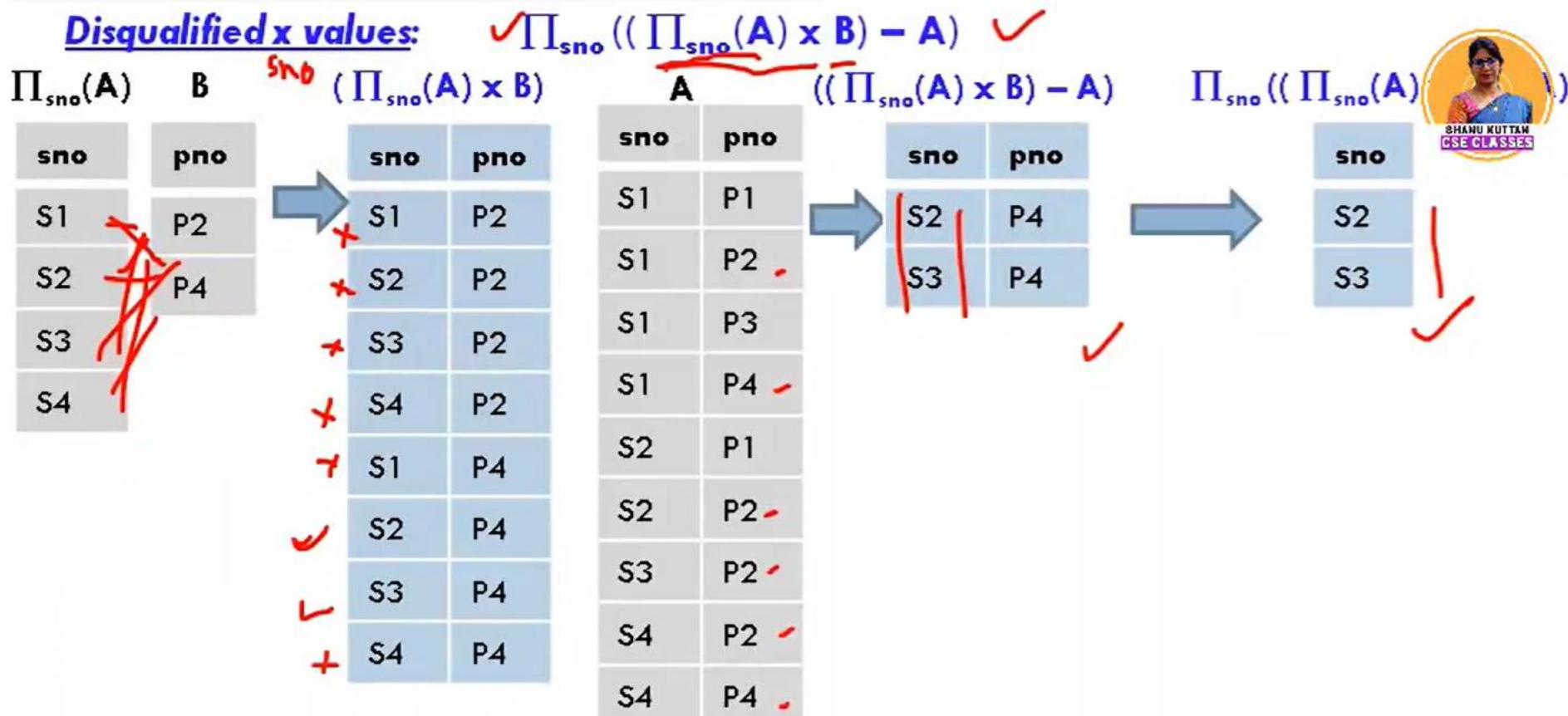
SUBSCRIBE



✓ A

B

sno	pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4



✓  $A/B = \Pi_{sno}(A) - \text{all disqualified tuples}$

✓  $A/B = \Pi_{sno}(A) - \Pi_{sno}((\Pi_{sno}(A) \times B) - A)$  ✓



SUBSCRIBE

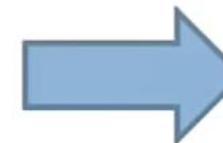
## Example 3:



A				
A	B	C	D	E
$\alpha$	a	$\alpha$	a	1
.	.	.	.	.
$\alpha$	a	$\gamma$	a	1
$\alpha$	a	$\gamma$	b	1
$\beta$	a	$\gamma$	a	1
$\beta$	a	$\gamma$	b	3
$\gamma$	a	$\gamma$	a	1
$\gamma$	a	$\gamma$	b	1
$\gamma$	a	$\beta$	b	1

÷

B	
D	E
a	1
b	1



$A \div B$

$A \div B$		
A	B	C
$\alpha$	a	$\gamma$
$\gamma$	a	$\gamma$



# Assignment Operator

- The **assignment operation** ( $\leftarrow$ ) **provides** a **convenient way** to express complex queries.
  - It **writes query** as a **sequential program** consisting of:
    - a **series of assignments**
    - followed by an expression whose value is displayed as a **result** of the query.
  - Assignment must always be made to a **temporary relation variable**.

$t_1 \rightarrow$   
 $x_2 \leftarrow \sim$

$R \leftarrow t_1 - t_2$

# Example:

- **Division operation**  $A \div B = \prod_x(A) - \prod_x((\prod_x(A) \times B) - A)$
- Write  $A \div B$  as

**temp1**  $\leftarrow \prod_x(A)$

**temp2**  $\leftarrow \prod_x((\text{temp1} \times B) - A)$

**result**  $\leftarrow \text{temp1} - \text{temp2}$

- The result to the right of the  $\leftarrow$  is assigned to the relation variable on the left of the  $\leftarrow$
- May use variable in subsequent expressions.

# SQL (Structured Query Language)

- ✓ SQL is a Domain-Specific Language. ✓ Structured
- ✓ SQL is Declarative language i.e. a non-procedural language.
  - ❑ SQL allows to declare what we want to do, but not how to do.
    - ❑ In SQL, we can specify through queries that what data we want but does not specify how to get those data.
    - ❑ Whereas, C is procedural language as it uses functions, procedures, loops, etc. to specify what to do and how to do it.

Question: How the query is actually performed in SQL?

Answer: The database/SQL engine is very powerful. When we write an SQL query, it first parses it; then it figures out its internal algorithms and it tries to select an algorithm which will be the best for that particular query. So, then it applies that algorithm, finds the answer and returns it.

- SQL is based on Relational Algebra and Tuple Relational Calculus.



# What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

# Types of SQL Commands



- **Data Definition Language (DDL)** commands are used to define the structure and schema of the database.

- **CREATE** - to create new table or database
- **ALTER** - to alter the structure of table
- **DROP** - to delete a table from database
- **TRUNCATE** - to delete all records from table
- **RENAME** - to rename a table

- **Data Manipulation Language (DML)** commands are used for accessing and manipulating the data stored in the database.

- **SELECT** - to retrieve data from the database
- **INSERT** - to insert a new row into a table
- **UPDATE** - to update existing row in a table
- **DELETE** - to delete a row from a table

Programmers call these DML operations "**CRUD**".  
**CRUD** stands for:  
Performed by:

**Create, Read, Update, Delete**  
**INSERT, SELECT, UPDATE, DELETE**



# Types of SQL Commands...

- **Data Control Language (DCL)** are the commands to control the access of the data stored in the database
  - **GRANT** - grant permission to user for database access
  - **REVOKE** - take back granted permission from user
- **Transaction Control Language (TCL)** commands are used to manage the changes made by the DML statement.
  - **COMMIT** - to permanently save the transaction
  - **ROLLBACK** - to undo transaction
  - **SAVEPOINT** - to temporarily save a transaction so that you can rollback to that point whenever necessary
- **Note:** TCL Commands are used for only DML commands while DDL and DCL commands are auto-commited



# SQL Rules

- **SQL** is NOT case sensitive. ✓
  - For example: select is the same as SELECT
- But names of databases, tables and columns are case sensitive.
  - For example: In given SQL query, ACCOUNT is different from account
    - SELECT \* FROM ACCOUNT;
    - SELECT \* FROM account;
- **SQL statements** can use **multiple lines**
  - End each SQL statement with a semi-colon ;

# SQL: Data Definition Language (DDL)



□ **Data Definition Language (DDL)** is used for creating and modifying the database objects such as tables, indices, views and users.

□ **DDL Commands** are used to define the structure and schema of the database

□ All the command of DDL are *auto-committed* that means it permanently save all the changes in the database.

□ **Data Definition Language (DDL) Commands:**

- **CREATE** - to create new table or database

- **ALTER** - to alter (or modify) the structure of table

- **DROP** - to delete a table from database

- **TRUNCATE** - to delete all records from table

- **RENAME** - to rename a table





# 1. CREATE

- **CREATE** statement is used to create database schema and to define the type and structure of the data to be stored in the database.
- **CREATE** statement can be used for
  - **Creating a Database,**
  - **Creating a Table**, etc.



# I. Creating a DATABASE:

✓ **CREATE DATABASE** statement is used to create a database in RDBMS.

## Syntax:

**CREATE DATABASE database\_name;**

## Example:

**CREATE DATABASE my\_db;**

### Note:

❖ **SHOW statement:** To see existing **databases** and **tables**  
**SHOW databases;**  
**SHOW tables;**

❖ **USE statement:** To use or select any existing database

**Syntax**      **USE database\_name;**

**Example:**    **USE my\_db;**



# II. Creating a TABLE:



➤ **CREATE TABLE** statement is used **to create a new table in a database.**

- It specifies **column names** of the table, its **data types** (e.g. varchar, integer, date, etc.) and can also specify **integrity constraints** (e.g. Primary key, foreign key, not null, unique).

**Syntax:**

**CREATE TABLE** **table\_name**

```
(  
    column_name1  datatype,  
    column_name2  datatype,  
    ...           ...  
    column_name n  datatype,  
    (Integrity Constraints)  
)
```





# Example: Create table

## Example 1:

```
CREATE TABLE Emp(  
    Emp_ID int,  
    Name varchar(20),  
    Age int,  
    Address varchar(100),  
    Salary numeric(10,2)  
);
```



Emp_ID	Name	Age	Address	Salary

## Note:

💡 **DESC (or DESCRIBE) statement:** To describe the details of the table structure

Syntax:

**DESC table\_name;**

Example:

**DESC Emp;**

## Example 2:

```
CREATE TABLE branch(  
    ✓branch-name varchar(15),  
    branch-city varchar(30),  
    assets integer,  
    primary key (branch-name),  
    check(assets>0)  
);
```



branch-name	branch-city	assets



# SQL DATA TYPES



- **Each column** in a database **table** is required to have a name and a data type.
  - The data type is a guideline for SQL to understand what type of data is expected inside of each column, and it also identifies how SQL will interact with the stored data.
- ✓ **Note:** Data types might have different names in different database. And even if the name is the same, the size and other details may be different! Always check the documentation!





Data type	Description
CHAR(size)	A FIXED length string (can contain letters, numbers, and special characters), with user-specified length size.
VARCHAR(size)	A VARIABLE length string (can contain letters, numbers, and special characters), with user-specified length size. Full From Character Varying
INT / INTEGER	A medium Integer (Signed range is from -2,147,483,648 to 2,147,483,647. Unsigned range is from 0 to 4294967295). 4 bytes
SMALLINT	A small integer (Signed range is from -32,768 to 32,767. Unsigned range is from 0 to 65,535.) 2 bytes
BIGINT	A large integer (Signed range is from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,808. Unsigned range is from 0 to 18446744073709551615). 8 bytes
NUMERIC(size,d)/ DECIMAL(size,d)/ FLOAT(size,d)	A small floating point number. Its size parameter specifies the total number of digits. The number of digits after the decimal point is specified by d parameter. For NUMRIC(3,1), 46.2 can be stored but not 466.0 or 0.32
REAL, DOUBLE PRECISION	Floating point and double-precision floating point numbers, with machine-dependent precision.
FLOAT(n)	A floating point number, with user-specified precision of at least n digits.
DOUBLE(size, d)	A large floating point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter





Data type	Description
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
TIME	A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'
DATETIME / TIMESTAMP	A date and time combination. Format: YYYY-MM-DD hh:mm:ss.
YEAR	A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000.

Note: There are many other data types.....

# SQL Constraints



- **SQL constraints** are used to specify rules for the data in a table.
- **Constraints** are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.
- **Constraints** can be **column level** or **table level**.
  - **Column level** constraints apply to a column
  - **Table level** constraints apply to the whole table.
- **Constraints** can be specified:
  - when a table is created with the **CREATE TABLE** statement, or
  - we can use the **ALTER TABLE** statement to create constraints even after the table is created.

# SQL Constraints:



- **NOT NULL**: Ensures that a column cannot have NULL value.
- **DEFAULT**: Provides a default value for a column when none is specified.
- **UNIQUE**: Ensures that all values in a column are different.
- **PRIMARY KEY**: A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- **FOREIGN KEY**: Uniquely identifies a row/record in another table.
- **CHECK**: Ensures that all the values in a column satisfies certain conditions
- **INDEX**: Used to create and retrieve data from the database very quickly



## Example:

CREATE TABLE Emp

(

=

Emp\_ID int NOT NULL,

Name varchar(20) **NOT NULL**,

Age int **NOT NULL**,

Address varchar(100) **DEFAULT** 'India',

Salary numeric(10,2) **NOT NULL**,

DeptID int,

PRIMARY KEY(Emp\_ID),

FOREIGN KEY (Dept\_ID) REFERENCES Dept(Dept\_ID),

CHECK(Age>=18)

);



# 2. ALTER



- ❑ **ALTER TABLE** statement is **used to modify the structure of the existing table**
- ❑ It is used to add, delete, modify or rename columns in an existing table.
- ❑ It is also used to add and drop various constraints on an existing table.

## 1. ALTER TABLE – ADD COLUMN

- For adding new columns in a table

## 2. ALTER TABLE - DROP COLUMN

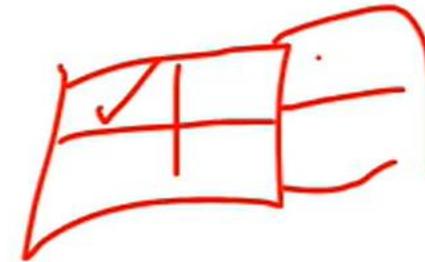
- For removing existing columns in table

## 3. ALTER TABLE - MODIFY COLUMN

- To modify existing columns in a table

## 4. ALTER TABLE – RENAME COLUMN

- To rename an existing column in a table



# 1. ALTER TABLE – ADD COLUMN



- For adding new columns in a table ✓
- Syntax:

**ALTER TABLE** *table\_name*  
**ADD** *column\_name(s)* *datatype(s);*

Example:

```
ALTER TABLE Emp  
ADD Email varchar(100);
```

## 2. ALTER TABLE - DROP COLUMN



- For removing existing columns in a table
- Syntax:

```
ALTER TABLE table_name
DROP column_name(s);
```

Example:

```
ALTER TABLE Emp
DROP Email;
```

### 3. ALTER TABLE - MODIFY COLUMN



- To modify existing columns in a table
  - To change data type of any column or to modify its size.
- Syntax:

```
ALTER TABLE table_name .  
MODIFY column_name datatype;
```

Example:

```
ALTER TABLE Emp  
MODIFY Name varchar(100);
```

## 4. ALTER TABLE – RENAME COLUMN



- To rename an existing column in a table
- Syntax:

**ALTER TABLE** *table\_name*

**RENAME** *old\_column\_name* **To** *new\_column\_name*;

Example:

```
ALTER TABLE Emp  
MODIFY Address To Location;
```

# 3. DROP



➤ **DROP TABLE** statement **completely removes a table from the database.**

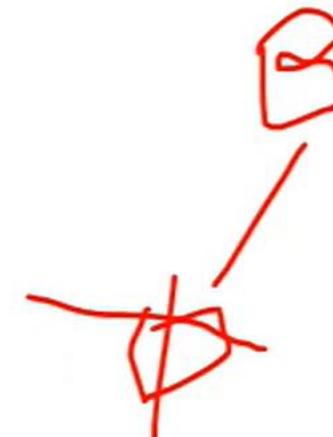
- This command will destroy the table structure and the data stored in it.

**Syntax:**

**DROP TABLE table\_name;**

**Example:**

**DROP TABLE Emp;**



**DROP** command is also be used on Databases.

**DROP DATABASE** statement is used to delete the complete database.

**Syntax:**

**DROP DATABASE database\_name;**

**Example:**

**DROP DATABASE my\_db;**



SUBSCRIBE

# 4. TRUNCATE



- **TRUNCATE TABLE** statement is used to remove all rows (complete data) from a table. It is similar to the DELETE statement with no WHERE clause.
- **TRUNCATE TABLE Vs DROP TABLE**
  - DROP TABLE command can also be used to delete complete table but it deletes table structure too. TRUNCATE TABLE doesn't delete the structure of the table.
- **Syntax:**

✓ **TRUNCATE TABLE table\_name;**

✓

Emp				
Emp_ID	Name	Age	Location	Salary
1	Rahul	32	Delhi	2000
2	Kamal	25	Pune	1500

**Example:**

✓ **TRUNCATE TABLE Emp;**



Emp_ID	Name	Age	Location	Salary



SUBSCRIBE

# SQL: Data Manipulation Language (DML)



- **Data Manipulation Language (DML)** commands are used **for accessing and manipulating the data stored in the database**.
- The DML commands are *not auto-committed* that means it can't permanently save all the changes in the database. They can be rollback.

## □ **Data Manipulation Language (DML)**

- R      □ **SELECT** - to retrieve data from the database
- C      □ **INSERT** - to insert a new row into a table
- U      □ **UPDATE** - to update existing row in a table
- D      □ **DELETE** - to delete a row from a table
- ←      **Data Query language (DQL)**
- }      **Modification of Database**

Programmers call these DML operations "**CRUD**".

**CRUD** stands for:

**Performed** by:

*Create, Read, Update, Delete*

**INSERT, SELECT, UPDATE, DELETE**





# 1. SELECT (mostly used SQL command/query)

- **SELECT** statement is used to select a set of data from a database table. Or Simply **SELECT** statement is used to retrieve data from a database table.
  - It returns data in the form of a result table. These result tables are called result-sets.
- **SELECT** is also called DQL because it is used to query information from a database table
- **SELECT** statement specifies column names, and **FROM** specifies table name
- **SELECT** command is used with different Conditions and CLAUSES.

## Basic Syntax:

**To retrieve selected fields from the table:**

```
SELECT column1, column2, ... columnN  
FROM table_name(s);
```

**To retrieve all the fields from the table:**

```
SELECT *  
FROM table_name(s);
```





# Example Table: Emp

Columns/Fields/Attributes

✓

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Records/  
Rows/  
Tuples



SUBSCRIBE

# Example: SELECT

## Example 1:

To retrieve columns *Emp\_ID*, *Name* from *Emp* table

```
SELECT Emp_ID, Name
FROM Emp;
```

Emp_ID	Name
1	Rahul
2	Kamal
3	Karan
4	Chirag
5	Harsh
6	Kajal
7	Mahi

## Example 2:

To retrieve all the fields from the *Emp* table

```
SELECT *
FROM Emp;
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00



SUBSCRIBE

# WHERE CLAUSE



- ❑ ✓ WHERE Clause is used to select a particular record based on a condition. It is used to filter records.
- ❑ WHERE Clause is used to specify a condition while fetching the data from a single table or by joining with multiple conditions
- ❑ The WHERE clause is not only used in the SELECT statement, but it is also used in the UPDATE, DELETE statement.
- ❑ ✓ Syntax: “SELECT with WHERE Clause”

```
SELECT column1, column2, ... columnN  
      - - - - -  
      |  
      FROM table_name(s)
```

Example: | WHERE condition; ✓ .

To retrieve Emp\_ID, Name from Emp table whose Salary is greater than 2000

```
SELECT Emp_ID, Name  
      - - -  
      |  
      FROM Emp  
      - - -  
      |  
      WHERE Salary > 2000;
```



Emp_ID	Name
4	Chirag
5	Harsh
6	Kajal
7	Mahi

# Basic Query Structure

Suggested: Relational Model & Relational Algebra



- SQL is based on set and relational operations with certain modifications and enhancements.

- A typical SQL query has the form:

```
select A1, A2, ..., An
from r1, r2, ..., rm
where P
```

- A<sub>i</sub>s represent attributes
  - r<sub>j</sub>s represent relations
  - P is a predicate (or condition).
- This SQL query is equivalent to the relational algebra expression.

$$\Pi_{A_1, A_2, \dots, A_n} (\sigma_P (r_1 \times r_2 \times \dots \times r_m))$$

- The result of an SQL query is a relation (or table).



# Optional Clauses in SELECT statement

Suggested: Relational Model & Relational Algebra



- ✓ **[WHERE Clause]** : It specifies which rows to retrieve by specifying **conditions**.
- **[GROUP BY Clause]** : **Groups rows** that share a **property** so that the **aggregate function** can be applied to each group.
- **[HAVING Clause]** : It **selects** among the **groups** defined by the **GROUP BY** clause by specifying **conditions**..
- **[ORDER BY]** : It specifies an order in which to return the rows.
- **[DISTINCT Clause]**: It is used to remove duplicates from results set of a **SELECT** statement. (**SELECT DISTINCT**)



# 1. SQL Arithmetic Operators

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulo

## Example:

To retrieve Emp\_ID, Name, Salary plus 3000 from Emp table

```
SELECT Emp_ID, Name, Salary + 3000  
FROM Emp;
```



Emp_ID	Name	Salary
1	Rahul	5000.00 ✓
2	Kamal	4500.000
3	Karan	5000.00
4	Chirag	9500.00
5	Harsh	11500.00
6	Kajal	7500.00
7	Mahi	13000.00



# 2. SQL Comparison Operators

Operator	Description
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<> or !=	Not equal to

## Example:

To retrieve *Emp\_ID, Name* from *Emp* table whose *Salary* is greater than 2000

```
SELECT Emp_ID, Name, Salary  
FROM Emp  
WHERE Salary > 2000;
```



Emp_ID	Name	Salary
4	Chirag	6500.00
5	Harsh	8500.00
6	Kajal	4500.00
7	Mahi	10000.00



# 3. SQL Logical Operators

Operator	Description
AND	It displays a record if <u>all</u> the conditions separated by AND are TRUE. <i>(1 AND 2)</i>
OR	It displays a record if <u>any</u> of the conditions separated by OR is TRUE <i>(1 OR 2)</i>
NOT	<ul style="list-style-type: none"> <li>✓ It displays a record if the condition(s) is <u>NOT</u> TRUE.</li> <li>✓ It reverses the meaning of any operator with which it is used. This is a <u>negate operator</u>.</li> </ul> <p>Eg: NOT EXISTS, NOT BETWEEN, NOT IN, IS NOT NULL, etc.</p>

## Example:

To retrieve all records from Emp table whose salary is greater than 2000 and age is less than 25

```
SELECT *
FROM Emp
WHERE Salary > 2000 AND Age < 25;
```



Emp_ID	Name	Age	Address	Salary
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

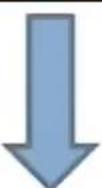
# 4. SQL Special Operators

Operator	Description
ALL	It compares a value to all values in another value set (in subqueries)
ANY	It compares the values in the list according to the condition (in subqueries).
BETWEEN	It is used to search for values that are within a set of values (given minimum and maximum values).
EXISTS	It is used to search for the presence of a row in a specified table (in subqueries).
IN	It compares a value to that specified list value (and in subqueries).
LIKE	It compares a value to similar values using wildcard operator (% , _ ).
IS NULL	It checks for missing data or NULL values

## Example:

To retrieve all records from *Emp* table with salary between 2000 and 5000

```
SELECT *
FROM Emp
WHERE Salary BETWEEN 2000 AND 5000;
```



Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
3	Karan	23	Pune	2000.00
6	Kajal	22	Bilaspur	4500.00

## 2. INSERT

- The **INSERT INTO** statement is used to insert new records in a table.
- It is possible to write the **INSERT INTO** statement in two ways.

**Syntax 1:** Specify both the column names and the values to be inserted

```
INSERT INTO table_name(column1, column2, column3,...columnN)
VALUES (value1, value2, value3,...valueN);
```

**Syntax 2:** Specify only values to be inserted. But needed to remember the column order

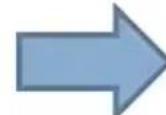
```
INSERT INTO table_name
VALUES (value1, value2, value3,...valueN);
```

# Example: INSERT INTO

- ✓ **INSERT INTO Emp (Emp\_ID, Name, Age, Address, Salary) VALUES (1, 'Rahul', 32, 'Delhi', 2000);**
  - **INSERT INTO Emp (Emp\_ID, Name, Age, Address, Salary) VALUES (2, 'Kamal', 25, 'Pune', 1500);**
  - **INSERT INTO Emp (Emp\_ID, Name, Age, Address, Salary) VALUES (3, 'Karan', 23, 'Pune', 2000);**
  - **INSERT INTO Emp VALUES (4, 'Chirag', 25, 'Mumbai', 6500);**
  - **INSERT INTO Emp VALUES (5, 'Harsh', 32, 'Mumbai', 8500);**
  - **INSERT INTO Emp VALUES (6, 'Kajal', 22, 'Bilaspur', 4500);**
  - **INSERT INTO Emp VALUES (7, 'Mahi', 24, 'Patna', 10000);**
- 1<sup>st</sup> way
- 2<sup>nd</sup> way

✓ To Check:

```
SELECT *
FROM Emp;
```



Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00



SUBSCRIBE

# 3. UPDATE



- The **UPDATE** statement is used to modify the existing records in a table.
- **Note:** Always use the WHERE clause with the UPDATE statement to update the selected rows, otherwise all the rows would be affected.

## Syntax:

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



# Example: UPDATE

Example:

To update the address of Emp\_ID: 3 to Chennai

✓ UPDATE Emp

SET Address='Chennai'

WHERE Emp\_ID=3;

✓ To Check:

SELECT \*  
FROM Emp;



Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Chennai	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00



# 4. DELETE



- The **DELETE** statement is used to delete existing records in a table. ✓
- Note: **Always use the WHERE clause with a DELETE statement to delete the selected rows, otherwise all the records would be deleted.**

1-MINUTE  
TO

## Syntax:

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

# Example: DELETE

**Example:**

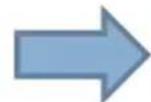
To delete the employee records of **Pune** location

**DELETE FROM Emp**

**WHERE Address='Pune'; ✓**

✓ **To Check:**

```
SELECT *  
FROM Emp;
```



Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

# SELECT DISTINCT: “DISTINCT Clause”

- The **SELECT DISTINCT** statement is used to return only distinct (different) values.  
Or simply it is used to return only unique values.
- ☒ **DISTINCT Clause** is used to remove duplicates from results set of a **SELECT** statement.
  - Inside a table, a **column** often **contains many duplicate values**; and sometimes you only want to list the different (distinct) values.

## Syntax:

```
SELECT DISTINCT column1, column2, ... columnN  
FROM table_name;
```



# SELECT DISTINCT....

**Note:** ✓

- When only one column (expression) is **provided** in the **DISTINCT clause**, the query will return the unique values for that **column**.  

- When **more than one column** (expression) is **provided** in the **DISTINCT clause**, the query will retrieve **unique combinations** for the **columns** listed.
- In SQL, the **DISTINCT clause** doesn't **ignore NULL values**. So when using the **DISTINCT clause** in your SQL statement, your result set will include **NULL** as a distinct value.



# SQL: AND, OR, NOT OPERATOR

- The **WHERE** clause can be combined with **AND**, **OR**, and **NOT** operators.
  - To make more precise conditions for fetching data from database by combining more than one condition together.
- The **AND** and **OR operators** are used to filter records based on more than one condition
  - The **AND** operator displays a record if all the conditions separated by **AND** are **TRUE**.
  - The **OR** operator displays a record if any of the conditions separated by **OR** is **TRUE**.
- The **NOT operator** displays a record if the condition(s) is **NOT TRUE**
  - **NOT** reverses the meaning of any operator with which it is used. This is a **negate operator**.
  - Eg: **NOT IN**, **NOT BETWEEN**, **NOT EXISTS**, **IS NOT NULL**, etc.

# Example Table: Emp



Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

# SQL: AND OPERATOR



- AND operator (or AND condition) is used to combine multiple conditions with WHERE clause, in SELECT, UPDATE or DELETE statements.
- All conditions must be satisfied for a record to be selected.
- Syntax:

**SELECT column1, column2, ... columnN**

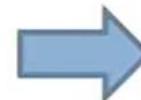
**FROM table\_name**

**WHERE condition1 AND condition2.....AND conditionN;**

## Example:

Selects all records from Emp table whose salary is greater than 2000 and age is less than 25

```
SELECT *  
FROM Emp  
WHERE Salary>2000 AND Age<25;
```



Emp_ID	Name	Age	Address	Salary
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00



# SQL: OR OPERATOR

Suggested: Relational Model & Relational Algebra

- OR operator (or OR condition) is used to combine multiple conditions with WHERE clause, in SELECT, UPDATE or DELETE statements.
- Any one of the conditions must be satisfied for a record to be selected.

## Syntax:

SELECT column1, column2, ... columnN

FROM table\_name

WHERE condition1 OR condition2.....OR conditionN;

## Example:

Selects all records from Emp table whose salary is greater than 2000 or whose age is less than 25

```
SELECT *  
FROM Emp  
WHERE Salary>2000 OR Age<25;
```



Emp_ID	Name	Age	Address	Salary
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00



SUBSCRIBE

# SQL: NOT OPERATOR

- **NOT operator (or NOT condition)** is used to negate the conditions with the **WHERE** clause, in **SELECT, UPDATE or DELETE** statements.
- It displays a record if the condition(s) is **NOT TRUE**
- **NOT** reverses the meaning of any operator with which it is used. This is a **negate operator**.
  - Eg: **NOT IN, NOT BETWEEN, NOT EXISTS, IS NOT NULL**, etc.

## Syntax:

**SELECT column1, column2, ... columnN**

**FROM table\_name**

**WHERE NOT condition;**

## Example:

Selects all records from where address is **not Mumbai**

**SELECT \***

**FROM Emp**

**WHERE NOT Address='Mumbai';**



Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00



SUBSCRIBE

# Example: Combining AND and OR Conditions

- The SQL **AND condition** and **OR condition** can be combined to test for multiple conditions with the **WHERE** clause, in **SELECT, UPDATE or DELETE** statements.
- When combining these conditions, it is important to use parentheses ().
  - The **parentheses determine the order** that the AND and OR conditions are evaluated.
- Precedence order: **NOT, AND, OR**  
**1 2 3**

## Example:

Selects all records from Emp table whose salary is greater than 2000 and age is less than 25 or whose Emp ID is 2.

```
SELECT *  
FROM Emp  
WHERE (Salary>2000 AND Age<25) OR (Emp_ID=2);
```



Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00



# SQL: IN OPERATOR

Suggested: SQL DDL Commands | CREATE,ALTER,DROP,TRUNCA...



- The **IN** operator allows us to specify multiple values in a WHERE clause. ✓
  - It allows us to easily test if an expression matches any value in a list of values
- ✓ The **IN** operator is a shorthand for multiple OR conditions. So it used to replace many OR conditions.
- Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN(value1, value2, ... valueN);
```

OR (In Subqueries)

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN(SELECT STATEMENT);
```

## Subquery

A subquery is a *select-from-where* expression that is nested within another query.



# Example: IN OPERATOR

## Example 1:

It selects all Employees located in Delhi, Bilaspur and Patna

```
SELECT *
FROM Emp
WHERE Address IN ('Delhi', 'Bilaspur','Patna);
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

## Example 2:

It selects all Employees whose salary is either 2000,  
4500 or 10000

```
SELECT *
FROM Emp
WHERE Salary IN (2000, 4500,10000);
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
3	Karan	23	Pune	2000.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

# Example: IN OPERATOR using Subquery



## Example 3:

It selects all employees that are from the same address as the customers

0

```
SELECT *
FROM Emp
WHERE Address IN(SELECT Address FROM Customers);
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
6	Kajal	22	Bilaspur	4500.00

Address
Delhi
Pune
Pune
Mumbai
Mumbai
Bilaspur
Patna

Address
Delhi
Bhopal
Bangaluru
Chennai
Bilaspur

# SQL: NOT IN OPERATOR

- **NOT IN operator** is just opposite of **IN operator**
- Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name NOT IN (value1, value2, ... valueN);
```

## Example:

It selects all Employees not located in Delhi, Bilaspur and Patna

```
SELECT *
FROM Emp
WHERE Address NOT IN ('Delhi','Bilaspur','Patna');
```

Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00



# SQL: BETWEEN OPERATOR

- ❑ ✓**BETWEEN** operator (**BETWEEN...AND**) is used to select values within given range (given minimum and maximum values).
- ❑ The values can be numbers, text, or dates.
- ❑ The **BETWEEN** operator is inclusive: begin and end values are included.
- ❑ Syntax:

$\checkmark 10 - 50 \text{ m.a.}^+$

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

### Example:

It selects all Employees with the Salary between 2000 to 7000

```
SELECT *
FROM Emp
WHERE Salary BETWEEN 2000 AND 7000;
```



Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
6	Kajal	22	Bilaspur	4500.00



# SQL: NOT BETWEEN OPERATOR

- **NOT BETWEEN** operator is opposite of **BETWEEN** operator ✓
- Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name NOT BETWEEN value1 AND value2;
```

## Example:

It selects all Employees with the salary not between 2000 to 7000

```
SELECT *
FROM Emp
WHERE Salary NOT BETWEEN 2000 AND 7000;
```



Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	1000.00

# SQL: LIKE OPERATOR



- The **LIKE operator** is used in a **WHERE clause** to search for a specified **pattern** in a column.
- The **LIKE operator** is used to match string pattern values using two wildcard characters.
- **Wildcard Character:** It is used to substitute one or more characters in a string.
- There are two wildcards used in conjunction with the **LIKE operator**.
  - The percent sign (%): It represents zero, one or multiple characters.
  - The underscore (\_): It represents only a single character.
- The percent sign (%) and the underscore (\_) can also be used in combinations!

## Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE columnN LIKE pattern;
```



SUBSCRIBE

### Example 1:

It selects all Employees with a Name starting with "K"

```
SELECT * FROM Emp
WHERE Name LIKE "K%";
```



Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500
3	Karan	23	Pune	2000.00
6	Kajal	22	Bilaspur	4500.00



### Example 2:

It selects all Employees with a Name that have "a" in second position

```
SELECT * FROM Emp
WHERE Name LIKE "_a%";
```



Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

### Example 3:

It selects all Employees with a Name starting with "M" and are at least 4 characters in length

```
SELECT * FROM Emp
WHERE Name LIKE "M____%";
```



Emp_ID	Name	Age	Address	Salary
7	Mahi	24	Patna	10000.00

### Example 4:

It selects all records from Emp where salary starts with 200

```
SELECT * FROM Emp
WHERE Salary LIKE "200%";
```



Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
3	Karan	23	Pune	2000.00



## Examples : LIKE operator with wildcard operator (% , \_ ):

<u>LIKE Operator</u>	<u>Description</u>
<b>WHERE Name LIKE 'A %'</b>	Finds any values that start with "A", Eg: Aman, Aakriti
<b>WHERE Name LIKE '%a'</b>	Finds any values that end with "a", Eg: Neha, Soma
<b>WHERE Name LIKE '%oh%'</b>	Finds any values that have "oh" in any position, Eg: Sohan, Mohan
<b>WHERE Name LIKE '_r %'</b>	Finds any values that have "r" in the second position, Eg: Priya, Priyesh
<b>WHERE Name LIKE 'K_____ %'</b>	Finds any values that start with "K" and are at least 5 characters in length, Eg: Kamal, Karan, Kajal
<b>WHERE Name LIKE 'A__ %'</b>	Find, Eg: s any values that start with "A" and are at least 3 characters in length, Eg: Aki, Anu
<b>WHERE Name LIKE 'A%N'</b>	Finds any values that start with "A" and ends with "N", Eg: Aman, Raman

# SQL NULL Values

- ✓ Attributes can have Null values, if permitted by the schema definition for a table (i.e., no NOT NULL Constraint),
  - ✓ **NULL** represents a missing value, unknown value, non-existent or non-applicable value.
  - ✓ A **NULL value** in a table is a value in a field that appears to be **blank (empty)** i.e. with no value.
- + AB/MY*
- 0 --*
- **Note:** A **NULL value** is different than a zero value or a field that contains spaces.
- A field with a NULL value is one that has been left blank during record creation!

Emp_ID	Name	Age	Address	Salary	Email
1	Rahul	32	Delhi	2000.00	rr@gmail.com
2	Kamal	25	Pune	1500.00	kk@gmail.com
3	Karan	23	Pune	2000.00	NULL ✓
4	Chirag	25	Mumbai	6500.00	cc@gmail.com
5	Harsh	32	Mumbai	8500.00	hh@gmail.com
6	Kajal	22	Bilaspur	4500.00	NULL ✓
7	Mahi	24	Patna	10000.00	mm@gmail.com



SUBSCRIBE

# SQL NULL Values...

- Result of any arithmetic expression involving null is null  $S + \text{null} = \text{null}$
- Result of where clause condition is false if it evaluates to null.
- and, or, not operators handles null as follows:

<u>and</u>	true	false	<u>null</u>
true	true	false	<u>null</u> ✓
<u>null</u>	null	false	<u>null</u> ✓
false	false	false	false ✓

<u>or</u>	true	false	<u>null</u>
true	true	false	true ✓
<u>null</u>	true	null	null ✓
false	true	false	null ✓

<u>not</u>	
true	false ✓
<u>null</u>	null
false	true

# SQL: IS NULL OPERATOR



- The **IS NULL** operator is used to check for **NULL** values (or empty values).
- It allows us to find out the set of records in which value for a particular column is **NULL**.
- ✓ It returns **TRUE** if a **NULL** value is found, otherwise it returns **FALSE**.
- It can be used in a **SELECT**, **UPDATE**, or **DELETE** statement with **Where** Clause.

## ➤ IS NULL Syntax:

```
|SELECT column_names  
|FROM table_name  
WHERE column_name IS NULL;
```



# Example: IS NULL OPERATOR

✓ List employees who have no associated email Ids

```
SELECT *  
FROM Emp  
WHERE Email IS NULL;
```

✓ Emp Table

Emp_ID	Name	Age	Address	Salary	Email
1	Rahul	32	Delhi	2000.00	rr@gmail.com
2	Kamal	25	Pune	1500.00	kk@gmail.com
3	Karan	23	Pune	2000.00	NULL ✓
4	Chirag	25	Mumbai	6500.00	cc@gmail.com
5	Harsh	32	Mumbai	8500.00	hh@gmail.com
6	Kajal	22	Bilaspur	4500.00	NULL ↴
7	Mahi	24	Patna	10000.00	mm@gmail.com

Emp_ID	Name	Age	Address	Salary	Email
3	Karan	23	Pune	2000.00	NULL
6	Kajal	22	Bilaspur	4500.00	NULL



# SQL: IS NOT NULL OPERATOR



- **IS NOT NULL** operator is opposite of **IS NULL** operator
- The **IS NOT NULL** operator is used to check for NOT NULL values (or non-empty values)
- It returns **TRUE** if a **NONE** value is not found, otherwise it returns **FALSE**.
- It can be used in a **SELECT**, **UPDATE**, or **DELETE** statement with Where Clause.
- **IS NOT NULL** Syntax:

**SELECT column\_names**

**FROM table\_name**

**WHERE column\_name IS NOT NULL;**

# SQL: IS NOT NULL OPERATOR



✓  
List employees who have associated email Ids

```
SELECT *
FROM Emp
WHERE Email IS NOT NULL;
```

Emp Table

Emp_ID	Name	Age	Address	Salary	Email
1	Rahul	32	Delhi	2000.00	rr@gmail.com
2	Kamal	25	Pune	1500.00	kk@gmail.com
3	Karan	23	Pune	2000.00	NULL ✗
4	Chirag	25	Mumbai	6500.00	cc@gmail.com
5	Harsh	32	Mumbai	8500.00	hh@gmail.com
6	Kajal	22	Bilaspur	4500.00	NULL ✗
7	Mahi	24	Patna	10000.00	mm@gmail.com

Emp_ID	Name	Age	Address	Salary	Email
1	Rahul	32	Delhi	2000.00	rr@gmail.com
2	Kamal	25	Pune	1500.00	kk@gmail.com
4	Chirag	25	Mumbai	6500.00	cc@gmail.com
5	Harsh	32	Mumbai	8500.00	hh@gmail.com
7	Mahi	24	Patna	10000.00	mm@gmail.com

# SQL Aliases



- SQL **ALIASES** can be used to create a temporary name for columns or tables.
- **SQL Aliases** is used to give an alias name to a table or a column
  - **COLUMN ALIASES** are used to make column headings in result set easier to read.
  - **TABLE ALIASES** are used to shorten SQL query to make it easier to read or when there are more than one table is involved.
- SQL Aliases:
  - Aliases are created to make table or column names easier to read ✓
  - An **alias** only exists for the duration of the query. i.e. The renaming is just a temporary change and the actual table name does not change in the database.
  - Aliases are useful when table or column names are big or not very readable.
  - These are preferred when there are **more than one** table involved in a query.

# Syntax: SQL Aliases



## □ Column Alias Syntax:

**SELECT** column\_name **AS** alias\_name  
**FROM** table\_name;

## □ Table Alias Syntax:

**SELECT** column\_name(s)  
**FROM** table\_name **AS** alias\_name;

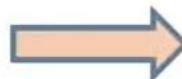
AS

# Example: Column Alias

**Emp**

✓ Emp_Id	Name	Age	Address	Salary	Dept_Id
1	Rahul	32	Delhi	2000.00	3
2	Kamal	25	Pune	1500.00	3
3	Karan	23	Pune	2000.00	3
4	Chirag	25	Mumbai	6500.00	1
5	Harsh	32	Mumbai	8500.00	1
6	Kajal	22	Bilaspur	4500.00	1
7	Mahi	24	Patna	10000.00	1

```
✓ ✓ ✓
SELECT Emp_ID AS ID, Name
FROM Emp;
```



✓ ID	✓ Name
1	Rahul
2	Kamal
3	Karan
4	Chirag
5	Harsh
6	Kajal
7	Mahi

# Example: Table Alias

Emp ✓ ✓

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	3
2	Kamal	25	Pune	1500.00	NULL
3	Karan	23	Pune	2000.00	3
4	Chirag	25	Mumbai	6500.00	1
5	Harsh	32	Mumbai	8500.00	1
6	Kajal	22	Bilaspur	4500.00	NULL
7	Mahi	24	Patna	10000.00	NULL

Dept ✓ ✓

Dept_ID	D_Name
1	Sales
2	HR
3	Finance
4	Marketing

SELECT E.Emp\_ID, E.Name, D.Dept\_ID, D.D\_Name

FROM Emp AS E, Dept AS D

WHERE E.Dept\_ID=D.Dept\_ID

E + D



Emp_ID	Name	Dept_Id	D_Name
1	Rahul	3	Finance
3	Karan	3	Finance
4	Chirag	1	Sales
5	Harsh	1	Sales



SUBSCRIBE



# SQL Functions

- SQL provides many built-in functions to perform operations on data.
- These functions are useful while performing mathematical calculations, string concatenations, sub-strings etc.
- SQL functions are divided into two categories:
  1. **Aggregate Functions**
  2. **Scalar Functions**
- **Aggregate functions and Scalar functions both** return a *single value*.
- **Aggregate functions** operate on *many records* while **Scalar functions** operate on *each record independently*





# Aggregate Functions in SQL

- **Aggregate functions** performs calculations on set of values and **returns a single value.**

- 1. **COUNT()**
- 2. **SUM()**
- 3. **AVG()**
- 4. **MAX()**
- 5. **MIN()**

100  
100  
100

- **Aggregate functions** are often used by **GROUP BY** clause of the **SELECT** statement **to group multiple rows together as input to form a single value output**
- **Aggregate functions** ignore NULL values, except count(\*)





# 1. COUNT() Function

- **COUNT(): Count function** returns the number of rows that matches the specified criterion

- Syntax:

```
SELECT COUNT(column_name)
      FROM table_name
      WHERE condition;
```



# Example: COUNT()

- Find the total number of student records

COUNT(\*) returns total no. of records

```
SELECT COUNT(*)  
FROM Student;
```

COUNT(\*)  
6

- Find the count of entered marks

COUNT(Marks) returns no. of non null values over column Marks

```
SELECT COUNT(Marks)  
FROM Student;
```

COUNT(Marks)  
5



- Find the count of distinct entered marks

COUNT(DISTINCT Marks) returns no. of distinct non null values over column Marks

```
SELECT COUNT(DISTINCT Marks)  
FROM Student;
```

COUNT(DISTINCT Marks)  
4



Student Table

ID	Name	Marks
1	A	90 ✓
2	B	40 ✓
3	C	70 ✓
4	D	60 ✓
5	E	70 ✗
6	F	NULL ✗



SUBSCRIBE



## 2. SUM() Function

- **SUM(): Sum function** returns total sum of a selected **numeric** columns.
- **Syntax:**

**SELECT SUM(column\_name)**

**FROM table\_name**

**WHERE condition;**



# Example: SUM()

- Find the sum of student marks

**SUM(Marks)** sum all non null values of column marks

```
SELECT SUM(Marks)  
FROM Student;
```

→ **SUM(Marks)**  
330 ✓

- Find the distinct sum of student marks

**SUM(DISTINCT Marks)** sum all distinct non null values of column marks

```
SELECT SUM(DISTINCT Marks)  
FROM Student;
```

→ **SUM(DISTINCT Marks)**  
260

Using 'AS' for Renaming the column

```
SELECT SUM(Marks) AS 'Sum Marks'  
FROM Student;
```

→ **Sum Marks**  
330



Student Table

ID	Name	Marks
1	A	90 -
2	B	40 -
3	C	70 -
4	D	60 -
5	E	70
6	F	NULL



### 3. AVG() Function

- **AVG(): Average function** returns the average value of selected **numeric** column.
- **Syntax:**

**SELECT AVG(column\_name)**

**FROM table\_name**

**WHERE condition;**





# Example: AVG()

- Find the average marks of students

```
SELECT AVG(Marks)
FROM Student;
```



AVG(Marks)
66



$$\text{AVG}(\text{Marks}) = \frac{\text{SUM}(\text{Marks})}{\text{COUNT}(\text{Marks})} = \frac{330}{5} = 66$$

- Find the distinct average marks of students

```
SELECT AVG(DISTINCT Marks)
FROM Student;
```



AVG(DISTINCT Marks)
65



$$\text{AVG}(\text{DISTINCT Marks}) = \frac{\text{SUM}(\text{DISTINCT Marks})}{\text{COUNT}(\text{DISTINCT Marks})} = \frac{260}{4} = 65$$

Using 'AS' for Renaming the column

```
SELECT AVG(Marks) AS 'Average Marks'
FROM Student;
```



Average Marks
66



Student Table

ID	Name	Marks
1	A	90
2	B	40
3	C	70
4	D	60
5	E	70
6	F	NULL





# 4. MAX() Function

- MAX(): Maximum function returns maximum value of selected column
- Syntax:

**SELECT MAX(column\_name)**

**FROM table\_name**

**WHERE condition;**



# Example: MAX()

- Find the maximum student marks: MAX(Marks)

```
SELECT MAX(Marks)  
FROM Student;
```



MAX(Marks)

90

Student Table

ID	Name	Marks
1	A	90 ✓
2	B	40
3	C	70
4	D	60
5	E	70
6	F	NULL





# 5. MIN() Function

- **MIN()**: **Minimum function** returns minimum value of selected column.
- **Syntax:**

**SELECT MIN(column\_name)**

**FROM table\_name**

**WHERE condition;**





# Example: MIN()

- Find the minimum students marks: **MIN(Marks)**

```
SELECT MIN(Marks)  
FROM Student;
```



MIN(Marks)
40

Student Table

ID	Name	Marks
1	A	90
2	B	40
3	C	70
4	D	60
5	E	70
6	F	NULL



# Scalar Functions

Suggested: SQL: Aggregate Functions | COUNT,SUM,AVG,MAX,MI...



- Scalar functions return a single value from an input value.

1. **UCASE()**
2. **LCASE()**
3. **MID()**
4. **LENGTH()**
5. **ROUND()**
6. **NOW()**
7. **FORMAT()**

# 1. UCASE() Function

- **UCASE()**: **Upper case function** is used to convert value of string column to uppercase characters.

- **Syntax:**

```
SELECT UCASE(column_name)  
FROM table_name;
```

# Example: UCASE()

- Converting string values of name column to upper case

✓  
**SELECT UCASE(Name)  
FROM Emp;**

UCASE(Name)

RAHUL
KAMAL
KARAN
CHIRAG
HARSH
KAJAL
MAHI

✓  
**Emp Table**

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00



## 2. LCASE() Function

- **LCASE()**: **Lower case function** is used to convert value of string column to lower case characters.

- **Syntax:**

```
SELECT LCASE(column_name)  
FROM table_name;
```

# Example: LCASE()

- Converting string values of name column to lower case

```
SELECT LCASE(Name)  
FROM Emp;
```



LCASE(Name)
rahul
kamal
karan
chirag
harsh
kajal
mahi

Emp Table ✓

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

### 3. MID() Function

- **MID(): MID function** is used to extract substrings from column values of string type in a table.

- **Syntax:**

```
SELECT MID(column_name, start, length)  
FROM table_name;
```

**Note:**

Length is optional

Start signifies the starting position of string

# Example: MID()

- Extract sub-string from name column from second position having length of 3

**SELECT MID(Name,2,3)  
FROM Emp;**



MID( <u>Name</u> , <u>2</u> , <u>3</u> )
ahu
ama
ara
hir
ars
aja
ahi

**Emp Table**

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

# 4. LENGTH() Function



- ✓ **LENGTH(): Length function** returns the length of a string in the column.

- Syntax:** Len( ) -

**SELECT LENGTH(column\_name)**

**FROM table\_name;**

# Example: LENGTH()



*Find the length of string of name column*

```
SELECT LENGTH(Name)
FROM Emp;
```



LEN(Name)
5
5
5
6
5
5
4

**Emp Table**

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00



# 5. ROUND() Function

Suggested: SQL: IN, BETWEEN, LIKE Operators & WILDCARD Char...



- **ROUND():** The **ROUND() function** is used to round a **numeric** column to the number of decimals specified.

- **Syntax:**

```
SELECT ROUND(column_name, decimals)
FROM table_name;
```

## Note:

Decimals represents number of decimals to be fetched.

# Example: ROUND()

- Find the round-off salary of employees

```
SELECT ROUND(Salary)
FROM Emp;
```

or

```
SELECT ROUND(Salary,0)
FROM Emp;
```



ROUND(Salary)

2001

1500

2000

6501

8500

4500

10000

**Emp Table**

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.50
2	Kamal	25	Pune	1500.20
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.80
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00



# Example: ROUND()

Suggested: SQL: AND, OR, NOT Operator | with Examples | DBMS ...



- Find round-off salary of employees (upto 1 decimal place) ✓

```
SELECT ROUND(Salary,1)  
FROM Emp;
```



ROUND(Salary,1)

2001.5

1500.2

2000.0

6501.8

8500.0

4500.0

10000.0

Emp Table

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.50
2	Kamal	25	Pune	1500.20
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.80
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00



# 6. NOW() Function

Suggested: SQL: AND, OR, NOT Operator | with Examples | DBMS ...



- **NOW()**: The **NOW() function** returns the **current system date and time**.
- **Syntax:**

**SELECT NOW()**

**FROM table\_name;**

# Example: NOW()

- Fetching current system date & time

```
SELECT NOW();
```



NOW()

2021-01-25 13:40:25



- Fetching current system date & time in a column

```
SELECT Name, NOW() AS 'Date & Time'  
FROM Emp;
```



Name	Date & Time
Rahul	2021-01-25 13:40:25
Kamal	2021-01-25 13:40:25
Karan	2021-01-25 13:40:25
Chirag	2021-01-25 13:40:25
Harsh	2021-01-25 13:40:25
Kajal	2021-01-25 13:40:25
Mahi	2021-01-25 13:40:25



SUBSCRIBE



# 7. FORMAT() Function

- **FORMAT()**: The **FORMAT()** function is used to format how a column is to be displayed.

- **Syntax:**

```
SELECT FORMAT(column_name, format)  
FROM table_name;
```

# Example: **FORMAT()**

- **Formatting current date as 'YYYY-MM-DD'**

```
SELECT Name, FORMAT(NOW(), 'YYYY-MM-DD') AS 'Date'  
FROM Emp;
```



Name	Date
Rahul	2021-01-25
Kamal	2021-01-25
Karan	2021-01-25
Chirag	2021-01-25
Harsh	2021-01-25
Kajal	2021-01-25
Mahi	2021-01-25

# ORDER BY Clause

- ✓ The output of the **SELECT** queries do not have any specific order. The **ORDER BY Clause** **allows us to specify order for the query output.**
- **ORDER BY Clause** is used with **SELECT statement** for arranging retrieved data in sorted order.
- The **ORDER BY Clause** by default sorts the retrieved data in ascending order [ASC].
- To sort the data in descending order, **DESC** keyword is used with **ORDER BY clause**.
- **Syntax:**

```
    | SELECT column-list
    | FROM table_name
    | [WHERE conditions]
    | ORDER BY column1, column2, .. columnN [ASC | DESC];
```

# Example Table: Emp



Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

# Query 1: (Eg: Ascending Order - Default) ✓

- To display details of Emp table in ascending order by salary ✓

**SELECT \***

**FROM Emp**

**ORDER BY Salary;**



Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500.00 ✓
1	Rahul	32	Delhi	2000.00 ✓
3	Karan	23	Pune	2000.00 ✓
6	Kajal	22	Bilaspur	4500.00 ✓
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
7	Mahi	24	Patna	10000.00

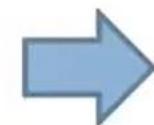
## Query 2: (Eg: Ascending Order with multiple columns)

➤ To sort details of Emp table in ascending order by name and salary

**SELECT \***

**FROM Emp**

**ORDER BY Name, Salary;**



In ORDER BY Clause multiple columns can be used.

In this case:

1<sup>st</sup> column is used as **primary ordering** field,

2<sup>nd</sup> column is used as **secondary ordering** field,

and so on

Emp_ID	Name	Age	Address	Salary
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
7	Mahi	24	Patna	10000.00
1	Rahul	32	Delhi	2000.00

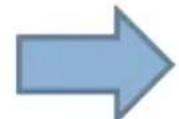
# Query 3: (Eg: Descending Order)

- To sort details of Emp table in descending order by name ✓

**SELECT \***

**FROM Emp**

**ORDER BY Name DESC;**



Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
7	Mahi	24	Patna	10000.00
3	Karan	23	Pune	2000.00
2	Kamal	25	Pune	1500.00
6	Kajal	22	Bilaspur	4500.00
5	Harsh	32	Mumbai	8500.00
4	Chirag	25	Mumbai	6500.00

Settings



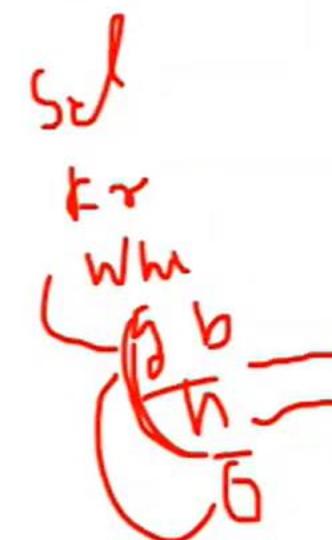
SUBSCRIBE

# GROUP BY Clause

- In SQL, the **GROUP BY** statement is used for organizing similar data into groups.
  - For Eg: "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.
- **Imp Points:**
  - **GROUP BY** clause is used with the **SELECT** statement in the **SQL** query.
  - **GROUP BY** clause is placed after the **WHERE** clause in **SQL**.
  - **GROUP BY** clause is placed before the **ORDER BY** clause in **SQL**.

## ✓ Syntax:

```
SELECT column_name(s), aggregate_function(column_name)
FROM table_name
[WHERE condition]
GROUP BY column_name(s)
[ORDER BY column_name(s)];
```



# Example Table: Emp



Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502



BHAWNA KUTAM

CSE CLASSES



# Query 1:

(GROUP BY with COUNT aggregate function)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the number of employee in each department

```
✓ SELECT Dept_id, COUNT(*)  
✓ FROM Emp  
✓ GROUP BY Dept_id
```



Dept_ID	COUNT(*)
500	3
501	2
502	2

# Query 2:

(GROUP BY with COUNT and ORDER BY)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the number of employee in each department, sorted low to high

```
SELECT Dept_id, COUNT(*)  
FROM Emp  
GROUP BY Dept_id  
ORDER BY COUNT(*);
```



Dept_ID	COUNT(*)
501	2
502	2
500	3



SUBSCRIBE

# Query 3:

(GROUP BY with SUM aggregate function)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the sum of salaries of employees from each department

```
SELECT Dept_id, SUM(Salary) AS 'SumSalary'  
FROM Emp  
GROUP BY Dept_id;
```

Dept_ID	SumSalary
500	5500.00
501	15000.00
502	14500.00

# Query 4:

(GROUP BY with AVG aggregate function)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the average of salaries of employees from each department

```
SELECT Dept_id, ROUND(AVG(Salary))  
FROM Emp  
GROUP BY Dept_id;
```

Dept_ID	AVG(Salary)
500	1833
501	7500
502	7250



BHAWNA KUTAM

CSE CLASSES



SUBSCRIBE

# Query 5:

(GROUP BY with MIN aggregate function)

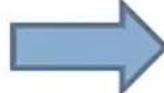
Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the minimum salary of employee from each department

**SELECT Dept\_id, MIN(Salary)**

**FROM Emp**

**GROUP BY Dept\_id;**



Dept_ID	MIN(Salary)
500	1500.00
501	6500.00
502	4500.00

# Query 6:

(GROUP BY with MAX aggregate function)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the maximum salary of employee from each department

```
SELECT Dept_id, MAX(Salary)  
FROM Emp  
GROUP BY Dept_id;
```



Dept_ID	MAX(Salary)
500	2000.00
501	8500.00
502	10000.00

# Query 7:

(GROUP BY with MIN & MAX both)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the minimum and maximum salary of employee from each department

```
SELECT Dept_id, MIN(Salary), MAX(Salary)  
FROM Emp  
GROUP BY Dept_id;
```

Dept_ID	MIN(Salary)	MAX(Salary)
500	1500.00	2000.00
501	6500.00	8500.00
502	4500.00	10000.00



# Query 8:

(GROUP BY with multiple columns)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the number of employees in each department location wise

```
SELECT Dept_id, Address, COUNT(*)  
FROM Emp  
GROUP BY Dept_id, Address;
```

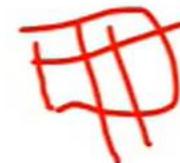
Dept_ID	Address	COUNT(*)
500	Delhi	1
500	Pune	2
501	Mumbai	2
502	Bilaspur	1
502	Patna	1



# HAVING Clause



- ❑  **HAVING** clause is used with GROUP BY clause & filter the groups created by the GROUP BY clause
- ❑ The HAVING clause enables you to specify conditions that filter which group results appear in the results.
- ❑ The **HAVING** clause was added to SQL because the aggregate functions like SUM, AVG, MIN, MAX,COUNT cannot be used with WHERE clause.
- ❑  The WHERE clause places conditions on the selected columns, whereas the **HAVING** clause places conditions on groups created by the GROUP BY clause
- ❑ In SQL query, HAVING clause is placed after the GROUP BY clause



## Syntax:

```
SELECT column_name(s), aggregate_function(column_name)
FROM table_name
[WHERE condition]
GROUP BY column_name(s)
HAVING condition
[ORDER BY column_name(s)];
```



SUBSCRIBE

# Order of Clauses(in SQL):

1. SELECT —
2. FROM —
3. WHERE —
4. GROUP BY —
5. HAVING —
6. ORDER BY —



# Example Table: Emp

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502



# Query 1:

Emp	Emp_ID	Name	Age	Address	Salary	Dept_ID
✓/3	1	Rahul	32	Delhi	2000.00	500
2	2	Kamal	25	Pune	1500.00	500
3	3	Karan	23	Pune	2000.00	500
2	4	Chirag	25	Mumbai	6500.00	501
?	5	Harsh	32	Mumbai	8500.00	501
?	6	Kajal	22	Bilaspur	4500.00	502
?	7	Mahi	24	Patna	10000.00	502

Write a query to display departments where the number of employee is greater than 2

```
SELECT Dept_id, COUNT(*)  
FROM Emp  
GROUP BY Dept_id  
HAVING COUNT(*) > 2;
```



Dept_ID	COUNT(*)
500	3

# Query 2:

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the department for which the sum of salaries of employees is more than 10000

```
SELECT Dept_ID, SUM(Salary)
FROM Emp
GROUP BY Dept_id
HAVING SUM(Salary)>10000;
```



Dept_ID	SUM(Salary)
501	15000.00
502	14500.00



# Query 3:

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the sum of salaries of employee that lives at same location

```
SELECT Address, SUM(Salary)  
FROM Emp  
GROUP BY Address  
HAVING COUNT(Address)>=2;
```

Address	SUM(Salary)
Mumbai	15000.00
Pune	3500.00

# Database Access from Application Programs

- **SQL**: widely used non-procedural, commercial language
- Example: Find the name of the instructor with ID 22222

```
select      name  
from        instructor  
where       instructor.ID = '22222'
```

- **SQL** is NOT a Turing equivalent language which means that everything that need to be computed cannot be computed in SQL
- **SQL** alone is not very powerful; because there are some computations that cannot be obtained by any SQL query. Such computations must be written in a host language, such as C, C++, Java with embedded SQL queries that access the data in the database
  - Application Programs are programs that are used to interact with the database in this fashion. Eg: Banking system are programs that generate: Payroll checks, Debit accounts, Credit Accounts or transfer funds between accounts
  - To access the database, DML statements need to be executed from the host language

**Application Programs generally access the database through one of the 2 ways:**

1. APIs (ODBC/JDBC) which allows SQL Queries to be sent to database
2. Language extensions to allow embedded SQL



# Application Programs generally access the DB through one of the 2 ways

1. By providing an Application Programming Interface i.e API (the set of procedures) that can be used to send DDL & DML statements (SQL Queries) to the database and retrieve the results i.e. APIs (**ODBC/JDBC**) which allows **SQL Queries to be sent to database**

- Eg: 1. ODBC (Open Database Connectivity) standard
  - It is created by Microsoft in 1992 that is used by Windows software applications, using C/C++ language, to access databases via SQL
- 2. JDBC (Java Database Connectivity) standard
  - It is created by Sun Microsystems in 1997 that is used by JAVA application, using only JAVA Language, to access databases via SQL

2. By extending the host language syntax to embed DML calls within the host language program i.e **Language extensions to allow embedded SQL**

- Eg: C,C+, JAVA with embedded SQL queries
- Usually special character (like EXEC) prefaces DML Calls, and a Preprocessor called the DML Pre-Compiler, that converts the DML statements to normal procedure calls in the host language



# Relational Calculus

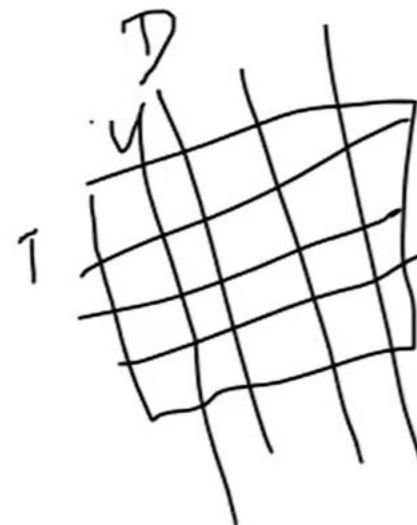
Suggested: Relational Algebra | Relational Algebra Operators | DB...



- **Relational Calculus** is a non-procedural query language (or declarative language). It uses mathematical predicate calculus (or first-order logic) instead of algebra.
- **Relational Calculus** tells what to do but never explains how to do.
- **Relational Calculus** provides description about the query to get the result where as **Relational Algebra** gives the method to get the result.
- When applied to database, it comes in **two flavors**:

## 1. **Tuple Relational Calculus (TRC):**

- Proposed by Codd in the year 1972
- Works on tuples (or rows) like SQL



## 2. **Domain Relational Calculus (DRC):**

- Proposed by Lacroix & piotte in the year 1977
- Works on domain of attributes (or Columns) like QBE (Query-By-Example)



SUBSCRIBE

# Relational Calculus ...

- **Calculus** has variables, constants, comparison operator, logical connectives and quantifiers.
  - ✓  $L > L \leq - -$
  - ✓  $\exists$
  - ✓  $\wedge \vee \neg$
- ✓ **TRC**: Variables range over tuples.
  - ✓ Like SQL
- ✓ **DRC**: Variables range over domain elements.
  - ✓ Like Query-By-Example (QBE)
- Expressions in the calculus are called formulas.
- **Resulting tuple** is an assignment of constants to variables that make the formula evaluate to true.

$R \in \{ \cdot \}$

# 1. Tuple Relational Calculus (TRC)

- **Tuple relational calculus** is a non-procedural query language
- **Tuple relational calculus** is used for selecting the tuples in a relation that satisfy the given condition (or predicate). The result of the relation can have one or more tuples.
- A query in **TRC** is expressed as:

$$\{ \overset{\checkmark}{t} \mid \overset{\checkmark}{P(t)} \}$$

Where  $t$  denotes resulting tuple and  $P(t)$  denotes predicate (or condition) used to fetch tuple  $t$

- **Result of Query:** It is the set of all tuples  $t$  such that predicate  $P$  is true for  $t$

- **Notations** used:



- $t$  is a tuple variable,
- $t[A]$  denotes the value of tuple  $t$  on attribute  $A$
- $t \in r$  denotes that tuple  $t$  is in relation  $r$
- $P$  is a **formula** similar to that of the **predicate calculus**



# Predicate Calculus Formula



1. Set of attributes and constants

2. Set of comparison operators: e.g.,  $<$ ,  $\leq$ ,  $=$ ,  $\neq$ ,  $>$ ,  $\geq$

3. Set of connectives: and ( $\wedge$ ), or ( $\vee$ ), not ( $\neg$ )

4. Implication ( $\Rightarrow$ ):  $x \Rightarrow y$ , if  $x$  is true, then  $y$  is true  $x \Rightarrow y \equiv \neg x \vee y$

5. Quantifiers: **Existential Quantifiers ( $\exists$ ) and Universal Quantifier ( $\forall$ ).**

✓  $\exists \underline{t} \in r (\underline{Q(t)}) \equiv$  “there exists” a tuple  $\underline{t}$  in relation  $r$  such that predicate  $\underline{Q(t)}$  is true

◻  $\forall \underline{t} \in r (\underline{Q(t)}) \equiv Q$  is true “for all” tuples  $\underline{t}$  in relation  $r$

✓ Free and Bound variables: ✓

◻ The use of **quantifiers**  $\exists X$  and  $\forall Y$  in a formula is said to **bind**  $X$  in the formula.

◻ A variable that is **not bound** is **free**.

◻ Let us revisit the definition of a **query**:  $\{t \mid P(t)\}$

◻ **There is an important restriction**

- the variable  $t$  that appears to the left of ‘|’ must be the **only free variable** in the formula  $P(t)$ .
- in other words, **all other tuple variables** must be **bound** using a **quantifier**

# Example Queries: TRC

- Find the loan-number, branch-name, and amount for all loans of over \$1200.

$\{ t \mid t \in \text{loan} \wedge t[\text{amount}] > 1200 \}$  ✓ (Selection)

It selects all tuples  $t$  from relation loan such that the resulting loan tuples will have amount greater than \$1200

- Find the loan number for each loan of an amount greater than \$1200

$\{ t \mid \exists s \in \text{loan} \ (t[\text{loan-number}] = s[\text{loan-number}] \wedge s[\text{amount}] > 1200) \}$  ✓ (Projection)

It selects the set of tuples  $t$  such that there exists a tuple  $s$  in relation loan for which the values of  $t$  &  $s$  for the loan-number attribute are equal and the value of  $s$  for the amount attribute is greater than \$1200

✓  
**branch** (branch-name, branch-city, assets)  
**customer** (customer-name, customer-street, customer-city)  
**account** (account-number, branch-name, balance)  
**loan** (loan-number, branch-name, amount) ✓  
**depositor** (customer-name, account-number)  
**borrower** (customer-name, loan-number)

# Example Queries: TRC...

branch (branch-name, branch-city, assets)  
customer (customer-name, customer-street, customer-city)  
account (account-number, branch-name, balance)  
loan (loan-number, branch-name, amount)  
depositor (customer-name, account-number)  
borrower (customer-name, loan-number)



- Find the names of all customers having a loan at the Perryridge branch

{  $t \mid \exists s \in \underline{\text{borrower}} (t[\text{customer-name}] = s[\text{customer-name}] \wedge \exists u \in \underline{\text{loan}} (u[\text{branch-name}] = \text{"Perryridge"} \wedge u[\text{loan-number}] = s[\text{loan-number}]))$  } *Join*

- Find the names of all customers having a loan, an account, or both at the bank

{  $t \mid \exists s \in \underline{\text{borrower}} (t[\text{customer-name}] = s[\text{customer-name}]) \vee \exists u \in \underline{\text{depositor}} (t[\text{customer-name}] = u[\text{customer-name}])$  } *Union*

- Find the names of all customers who have a loan and an account at the bank

{  $t \mid \exists s \in \underline{\text{borrower}} (t[\text{customer-name}] = s[\text{customer-name}]) \wedge \exists u \in \underline{\text{depositor}} (t[\text{customer-name}] = u[\text{customer-name}])$  } *Intersection*

## 2. Domain Relational Calculus (DRC)



- **Domain Relational Calculus** is a non-procedural query language.
- In **Domain Relational Calculus** the records are filtered based on the domains.
- DRC uses list of attributes to be selected from relation based on the condition (or predicate).
- DRC is same as TRC but differs by selecting the attributes rather than selecting whole tuples
- In DRC, each query is an expression of the form:  
$$\{ \langle \underline{a_1, a_2, \dots, a_n} \rangle \mid P(a_1, a_2, \dots, a_n) \}$$

$a_1, a_2, \dots, a_n$  represent domain variables

$P$  represents a **predicate** similar to that of the predicate calculus
- Result of Query: It is the set of all tuples  $a_1, a_2, \dots, a_n$  such that predicate  $P$  is true for  $a_1, a_2, \dots, a_n$  tuples



# Predicate Calculus Formula



## □ Notations used:

- $\langle \underline{a_1}, \underline{a_2}, \dots, \underline{a_n} \rangle \in \underline{r}$ , where  $\underline{r}$  is relation on  $n$  attributes and  $\underline{a_1}, \underline{a_2}, \dots, \underline{a_n}$  are domain variables or domain constants
- $\underline{P}$  is a **formula** similar to that of the **predicate calculus**

## □ Formula:

### 1. Set of domain variables and constants

### 2. Set of comparison operators: e.g., $<$ , $\leq$ , $=$ , $\neq$ , $>$ , $\geq$

### 3. Set of connectives: and ( $\wedge$ ), or ( $\vee$ ), not ( $\neg$ )

### 4. Implication ( $\Rightarrow$ ): $x \Rightarrow y$ , if $x$ is true, then $y$ is true $x \Rightarrow y \equiv \neg x \vee y$

### 5. Quantifiers:

**Existential Quantifiers ( $\exists$ ) and Universal Quantifier( $\forall$ ).**

$\exists \underline{x} (\underline{P(x)})$  and  $\forall \underline{x} (\underline{P(x)})$

x is Free domain variable

# Example Queries: DRC

- Find the loan-number, branch-name, and amount for loans of over \$1200:

$$\{ \underline{l}, \underline{b}, \underline{a} | \underline{l}, \underline{b}, \underline{a} \in \text{loan} \wedge \underline{a} > 1200 \}$$

(Selection)

- Find the loan number for each loan of an amount greater than \$1200:

$$\{ \underline{l} | \exists \underline{b}, \underline{a} (\underline{l}, \underline{b}, \underline{a} \in \text{loan} \wedge \underline{a} > 1200) \}$$

(Selection then Projection)

**branch** (branch-name, branch-city, assets)

**customer** (customer-name, customer-street, customer-city)

**account** (account-number, branch-name, balance)

**loan** (loan-number, branch-name, amount)

**depositor** (customer-name, account-number)

**borrower** (customer-name, loan-number)

# Example Queries: DRC ...

- Find the names of all customers who have a loan of over \$1200:

$$\{ \underline{\underline{c}} | \exists \underline{l}, \underline{b}, \underline{a} (\underline{\underline{c}}, \underline{l}) \in \underline{\underline{\text{borrower}}} \wedge (\underline{\underline{l}}, \underline{b}, \underline{a}) \in \underline{\underline{\text{loan}}} \wedge \underline{a} > 1200) \}$$

1 mark

(Join)

- Find the names of all customers having a loan at the Perryridge branch and find the loan amount:

$$\{ \underline{\underline{c}}, \underline{a} | \exists \underline{l} (\underline{\underline{c}}, \underline{l}) \in \underline{\underline{\text{borrower}}} \wedge \exists \underline{b} (\underline{\underline{l}}, \underline{b}, \underline{a}) \in \underline{\underline{\text{loan}}} \wedge \underline{b} = \text{"Perryridge"} ) \}$$

1 mark

(Join)

branch (branch-name, branch-city, assets)

customer (customer-name, customer-street, customer-city)

account (account-number, branch-name, balance)

loan (loan-number, branch-name, amount)

depositor (customer-name, account-number)

borrower (customer-name, loan-number)



# Database Users

- **Database Users** are the persons who interact with the database and take the benefits of database.
- Users are differentiated by the way they expect to interact with the system.
- Four types of DB users:
  1. **Naive users/Native users/End users**
  2. **Application programmers**
  3. **Sophisticated users**
  - Specialized users**



# Types of Database Users

Suggested: View of Data in DBMS | Data Abstraction| Schema & I...



1. **Naive Users/Native Users/End Users:** They are the unsophisticated users who use the existing application to interact with the database.

- Example: People accessing database over the web, bank tellers, clerical staffs, etc.

2. **Application Programmers:** They are the computer professionals who write the application programs. They interact with system through DML queries.

- For example: Writing a C program to generate the report of employees who are working in particular department, will involve a query to fetch the data from database. It will include an embedded SQL query in the C Program.



SHANU KUTTAN  
CSE CLASSES



SUBSCRIBE

# Types of Database Users ...

3. **Sophisticated Users:** They interact with the system by writing SQL queries directly through the query processor (like SQL) without writing application programs.
  - Example: Analysts, who submits SQL queries to explore data in the DBMS.
4. **Specialized Users:** They are also sophisticated users who write specialized database applications that do not fit into the traditional data processing framework. They are the developers who develop the complex programs to the requirement.
  - Example: Computer-Aided Design (CAD) Systems, Expert Systems, Knowledge Based System, etc. that store complex data types (graphics and audio data) & environment modelling systems



SHANTU KUTTAN

CSE CLASSES



SUBSCRIBE

# Database Administrators (DBAs)

- DBA is a person or group that is responsible for supervising both the database and the use of the DBMS.
- Database Administrators (DBAs) coordinate all the activities of the database system.
- They use specialized software to store and organize data
- They have all the permissions



# DBAs Tasks

- Schema definition
- Storage structure and access method definition
- Schema and physical organization modification
- Specifying integrity constraints
- Granting user authority to access database
- Monitoring performance & responding to changes in requirements
- Routine Maintenance
- Acting as liaison with users
- Backing up and restoring databases



SHANU KUTTAN  
CSE CLASSES



# Transaction Management

Suggested: Database Management System | DBMS | Introduction ...



Press Esc to exit full screen

- **What if system fails?**
- **What if more than one user is concurrently updating the same data?**
- A **Transaction** is a collection of operations that performs a single logical function in a database application
  - Each **transaction** is a unit of both atomicity and consistency. Thus, we require that transaction do not violate any database consistency constraints
  - For example: Lets say your account is A and your friend's account is B and you are transferring 10000 from A to B, the set of operations or the steps of the transaction are
    1. Read your account balance R(A);
    2. Deduct the amount from your balance  $A = A - 10000;$
    3. Write the remaining balance to your account W(A);
    4. Read your friend's account balance R(B);
    5. Add the amount to his account balance  $B = B + 10000;$
    6. Write the new updated balance to his account W(B);
  - This whole set of operations can be called a **transaction**. R is Read and W is Write



SHANU KUTTAN  
CSE CLASSES



# Transaction Management...

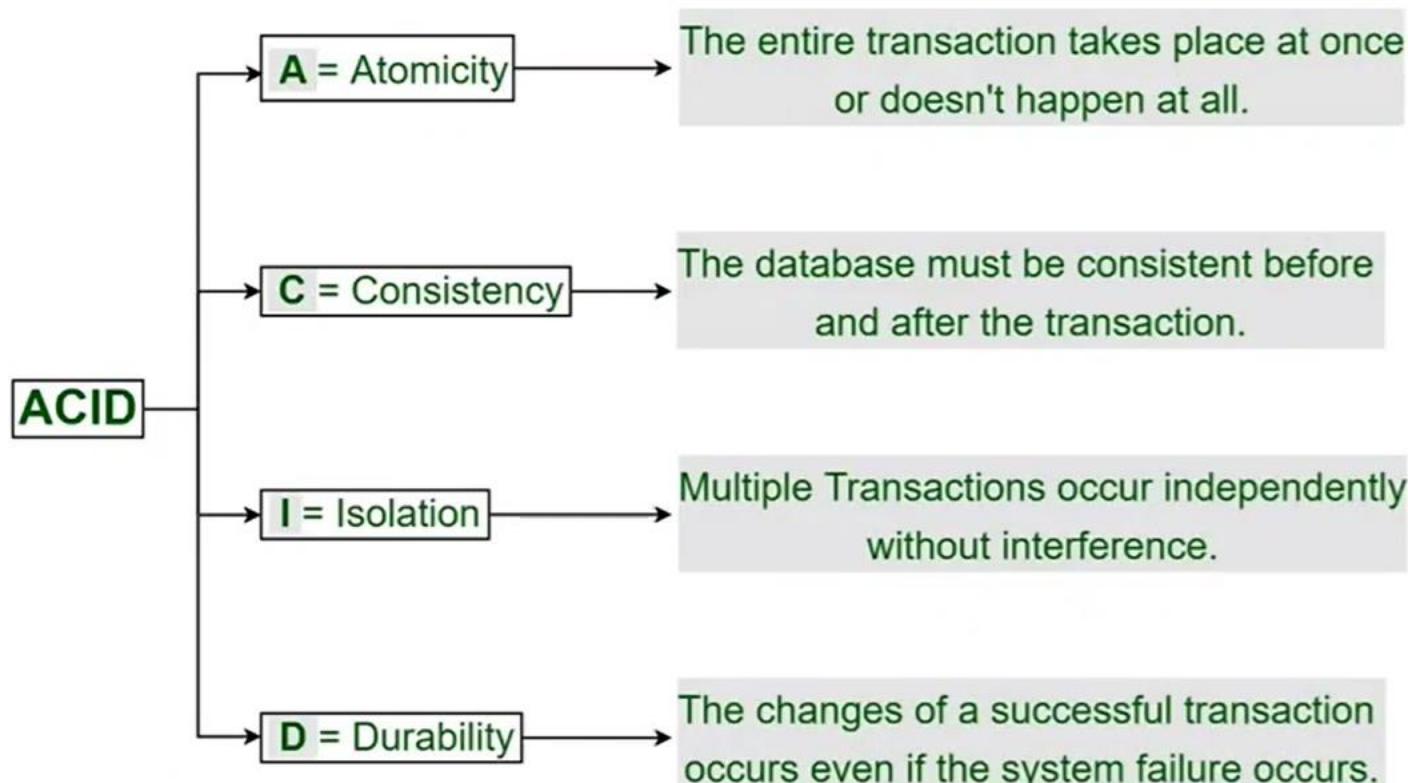
- **Transaction-management component** ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.
- **Concurrency-control manager** controls the interaction among the concurrent transactions, to ensure the consistency of the database.



# Transaction Management...

- In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties

## ACID Properties in DBMS



# Storage Management

- **Storage manager** is a program module that provides the interface between the *low-level data stored in the database and the application programs and queries submitted to the system.*
- The storage manager is responsible to the following tasks:
  - Interaction with the OS file manager
  - Efficient storing, retrieving and updating of data
- Issues:
  - Storage access
  - File organization
  - Indexing and hashing

