

*A day without new  
knowledge is a lost day.*

## *Database Technologies – MySQL*

If A and a, B and b, C and c etc. are treated in the same way then it is case-insensitive.

**MySQL is case-insensitive**

In this module we are going to learn SQL, PL/SQL and NoSQL(MongoDB)

# Introduction

- If anyone who wants to develop a good application then he should have the knowledge three major components.

They are . . . . .

- Presentation Layer [ UI ]
- Application Layer [ Server Application and Client Application ]
- Data Layer [ Data Access Object (DAO) / Data Access Layer (DAL) ] { Flat Files | RDBMS | NoSQL }

## Why do we need databases (Use Case)?

We **need databases** because they organize data in a manner which allows us to **store**, **query**, **sort**, and **manipulate** data in various ways. **Databases allow us to do all this things.**

Many companies collect data from different resource (like Weather data, Geographical data, Finance data, Scientific data, Transport data, Cultural data [ Cultural means: the ideas, customs, and social behaviour of a particular people or society.. ], etc.)

# What is Relation and Relationship?

## Remember:

- A **reference** is a relationship between two tables where the values in one table refer to the values in another table.
- A **referential key** is a column or set of columns in a table that refers to the primary key of another table. It establishes a relationship between two tables, where one table is called the parent table, and the other is called the child table.

# relation and relationship?

**Relation** (*in Relational Algebra "R" stands for relation*): In Database, ( a "relation" refers to a **table** within the database that follows the principles of the relational model **OR** an **entity** in ERD ) than contain attributes.

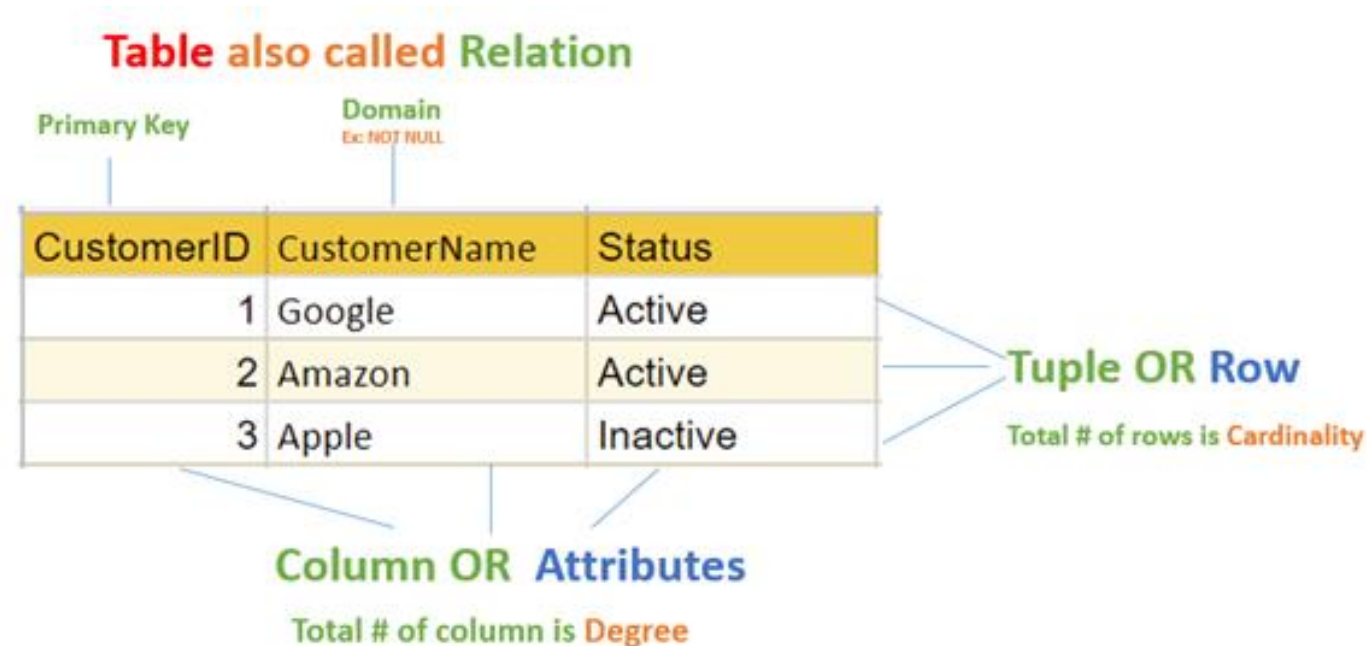
**Relationship:** In database, relationship is that how the two entities are **connected** to each other, i.e. what kind of relationship type they hold between them.

**Primary/Foreign key** is used to specify this relationship.

## Remember:

Foreign Key is also know as

- referential constraint
- referential integrity constraint. (Referential integrity constraint is the state of a database in which all values of all foreign keys are valid.)



File Systems is the traditional way to keep your data organized.

# File System VS DBMS

```
struct Employee {  
    int emp_no;  
    char emp_name[50];  
    int salary;  
} emp[1000];
```

```
struct Employee {  
    int emp_no;  
    char emp_name[50];  
    int salary;  
};  
struct Employee emp[1000];
```

# file-oriented system

## File Anomalies

c:\employee.txt

```
1 suraj 4000  
2 ramesh 6000  
3 rajan 4500  
.  
.  
.  
500 sam 3500  
.  
.  
.  
1000 amit 2300
```

c:\employee.txt

```
1 suraj 4000  
2 ramesh 6000  
3 rajan 4500  
.  
.  
.  
500 sam 3500  
.  
.  
1000 amit 2300  
.  
.  
2000 jerry 4500  
.  
.
```

c:\employee.txt

```
1 suraj 4000  
2 ramesh 6000  
3 rajan 4500  
.  
.  
500 sam 3500  
.  
3 rajan 4500  
.  
500 sam 3500  
.  
.  
1000 amit 2300
```

c:\employee.txt

```
1 suraj 4000  
2 ramesh 6000  
3 rajan 4500  
.  
.  
sam 500 3500  
.  
ram 550 5000  
.  
1000 amit 2300
```

c:\employee.txt

```
1 suraj 4000  
2 ramesh 6000  
3 rajan 4500  
.  
500 sam 3500  
.  
600 neel 4500
```

- Create/Open an existing file
- Reading from file
- Writing to a file
- Closing a file

# *file-oriented system*

## *File Anomalies*

c:\employee.txt

file attributes

file permissions

search empl ID=1

search emp\_name

```
1 suraj 4000
2 ramesh 6000
3 rajan 4500
.
.
.
500 sam 3500
.
.
.
1000 amit 2300
```

- File Name
- Type
- Location

- File permissions
- Share permissions

```
1 suraj 4000
2 ramesh 6000
3 rajan 4500
.
.
.
500 sam 3500
.
.
.
1000 amit 2300
```

```
1 suraj 4000
2 ramesh 6000
3 rajan 4500
.
.
.
500 sam 3500
.
.
.
1000 amit 2300
```

# *advantages of file-oriented system*

The biggest advantage of file-based storage is that anyone can understand the system.

## **Advantage of File-oriented system**

- **Backup:** It is possible to take faster and automatic back-up of database stored in files of computer-based systems.
- **Data retrieval:** It is possible to retrieve data stored in files in easy and efficient way.
- **Flexibility:** File systems provide flexibility in storing various types of data, including text documents, images, audio, video, and more
- **Cost-Effectiveness:** File systems often do not incur licensing costs, making them cost-effective for basic data storage needs.
- **Editing:** It is easy to edit any information stored in computers in form of files.
- **Remote access:** It is possible to access data from remote location.
- **Sharing:** The files stored in systems can be shared among multiple users at a same time.



# *disadvantage of file-oriented system*

The biggest disadvantage of file-based storage is as follows.

## **Disadvantage of File-oriented system**

- **Data redundancy:** It is possible that the same information may be duplicated in different files. This leads to data redundancy results in memory wastage.  
(Suppose a customer having both kind of accounts- saving and current account. In such a situation a customers detail are stored in both the file, saving.txt- file and current.txt- file , which leads to Data Redundancy.)
- **Data inconsistency:** Because of data redundancy, it is possible that data may not be in consistent state.  
(Suppose customer changed his/her address. There might be a possibility that address is changed in only one file (saving.txt) and other (current.txt) remain unchanged.)
- **Limited data sharing:** Data are scattered in various files and also different files may have different formats (for example: .txt, .csv, .tsv and .xml) and these files may be stored in different folders so, due to this it is difficult to share data among different applications.
- **Data Isolation:** Because data are scattered in various files, and files may be in different formats (for example: .txt, .csv, .tsv and .xml), writing new application programs to retrieve the appropriate data is difficult.
- **Data security:** Data should be secured from unauthorized access, for example a account holder in a bank should not be able to see the account details of another account holder, such kind of security constraints are difficult to apply in file processing systems.

**Relation Schema:** A relation schema represents name of the relation with its attributes.

- **e.g.** student (roll\_no int, name varchar, address varchar, phone varchar and age int) is relation schema for STUDENT

# DBMS

- **database:** Is the collection of **related data** which is **organized**, database can store and retrieve large amount of data easily, which is stored in one or more data files by one or more users, it is called as **structured data**.
- **management system:** it is a software, designed to **define, manipulate, retrieve** and **manage** data in a database.

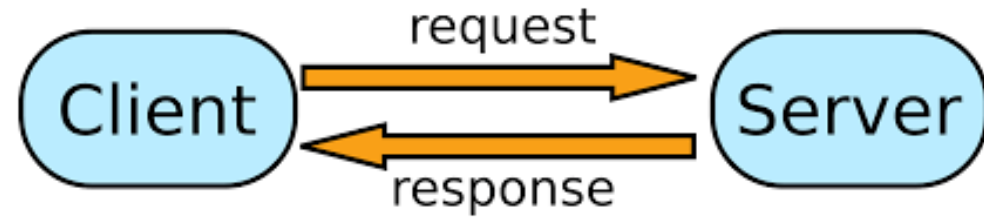


# relational database management system?

A RDBMS is a database management system (DBMS) that is based on the **relational model** introduced by Edgar Frank Codd at IBM in 1970.

RDBMS supports

- *client/server Technology*
- *Highly Secured*
- *Relationship (PK/FK)*



- A server is a computer program or a machine that provides service to another computer program, known as the client.

# *object relational database management system?*

An object database is a database management system in which information is represented in the form of objects.

PostgreSQL is the most popular pure ORDBMS. Some popular databases including Microsoft SQL Server, Oracle, and IBM DB2 also support objects and can be considered as ORDBMS.

## **Advantage of ORDBMS**

- Function/Procedure overloading.
- Extending server functionality with external functions written in C or Java.
- User defined data types.
- Inheritance of tables under other tables.

# relational model concepts and properties of relational table

# relational model concepts

Relational model organizes data into one or more **tables** (or "relations") of **columns** and **rows**. Rows are also called **records** or **tuples**. Columns are also called **attributes**.

- **Tables** – In relational model, relations are saved in the form of Tables. A table has rows and columns.
- **Attribute** – Attributes are the properties that define a relation. **e.g.** (roll\_no, name, address, phone and age)
- **Tuple** – A single row of a table, which contains a single record for that relation is called a tuple.
- **Relation schema** – A relation schema describes the relation name (table name) with its attribute (column) names.  
**e.g.** student(prn, name, address, phone, DoB, age, hobby, email, status) is relation schema for student relation.
- **Attribute domain** – An attribute domain specifies the data type, format, constraints of a column, and defines the range of values that are valid for that column.

---

## Remember:

- In database management systems, **NULL** is used to **represent** **MISSING** or **UNKNOWN** data in a table column.

## properties of relational table

ID	job	firstName	DoB	salary
1	manager	Saleel Bagde	yyyy-mm-dd	●●●●●●
3	salesman	Sharmin	yyyy-mm-dd	●●●●●●
4	accountant	Vrushali	yyyy-mm-dd	●●●●●●
2	salesman	Ruhan	yyyy-mm-dd	●●●●●●
5	9500	manager	yyyy-mm-dd	●●●●●●
5	Salesman	Rahul Patil	yyyy-mm-dd	●●●●●●

### Relational tables have six properties:

- Values are atomic.
- Column values are of the same kind. (Attribute Domain: Every attribute has some pre-defined datatypes, format, constraints of a column, and defines the range of values that are valid for that column known as attribute domain.)
- Each row is unique.
- The sequence of columns is insignificant – (unimportant).
- The sequence of rows is insignificant – (unimportant).
- Each attribute/column must have a unique name.

# What is data?





# what is data?

Data is any facts that can be stored and that can be processed by a computer.

Data can be in the form of **Text** or **Multimedia**

e.g.

- number, characters, or symbol
- images, audio, video, or signal

## Remember:

- A **Binary Large Object ( BLOB )** is a MySQL data type that can store binary data such as multimedia, and PDF files.
- A **Character Large Object(CLOB)** is aa MySQL data type which is used to store large amount of textual data. Using this datatype, you can store data up to 2,147,483,647 characters.
- A number is a mathematical value used to count, measure, and label.



What is Entity Relationship  
Diagram?

# Entity Relationship Diagram (ER Diagram)

Use E-R model to get a high-level graphical view to describe the "**ENTITIES**" and their "**RELATIONSHIP**"

The basic constructs/components of ER Model are **Entity**, **Attributes** and **Relationships**.

An entity can be a **real-world object**.

## What is Entity?

An entity in DBMS is a real-world object that has certain properties called attributes that define the nature of the entity.

In relation to a database , an entity is a

- Person(student, teacher, employee, department, ...)
- Place(classroom, building, ...) --a particular position or area
- Thing(computer, lab equipment, ...) --an object that is not named
- Concept(course, batch, student's attendance, ...) -- an idea,

about which data can be stored. All these entities have some **attributes** or **properties** that give them their **identity**.

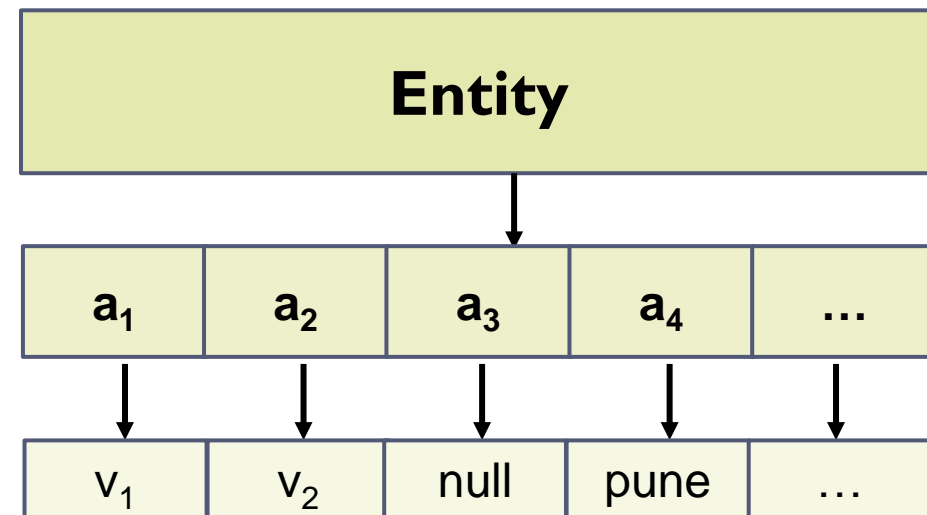
***Every entity has its own characteristics.***

In database management systems, **null** is used to represent **missing** or **unknown** data in a table column.

## What is an Attribute?

Attributes are the properties that define a relation.

e.g. student(ID, firstName, middleName, lastName, city)



In Entity Relationship(ER) Model attributes can be classified into the following types.

- Simple/Atomic and Composite Attribute
- Single Valued and Multi Valued attribute
- Stored and Derived Attributes
- Complex Attribute

## **Remember:**

In SQL, the same name can be used for two (or more) attributes as long as the attributes are in different relations.

# attributes

• <b>Simple / Atomic Attribute</b> (Can't be divided further)	--VS--	<b>Composite Attribute</b> (Can be divided further)
• <b>Single Value Attribute</b> (Only One value)	--VS--	<b>Multi Valued Attribute</b> (Multiple values)
• <b>Stored Attribute</b> (Only One value)	--VS--	<b>Derived Attribute</b> (Virtual)
• <b>Complex Attribute</b> (Composite & Multivalued)		

Employee ID: An employee ID can be a composite attribute, which is composed of sub-attributes such as department code, job code, and employee number.

- **Atomic Attribute:** An attribute that cannot be divided into smaller independent attribute is known as atomic attribute.  
*e.g.* ID's, PRN, age, gender, zip, marital status cannot further divide.
- **Single Value Attribute:** An attribute that has only single value is known as single valued attribute.  
*e.g.* manufactured part can have only one serial number, voter card, blood group, price, quantity, branch can have only one value.
- **Stored Attribute:** The stored attribute are such attributes which are already stored in the database and from which the value of another attribute is derived.  
*e.g.* (HRA, DA...) can be derive from salary, age can be derived from DoB, total marks or average marks of a student can be derived from marks.

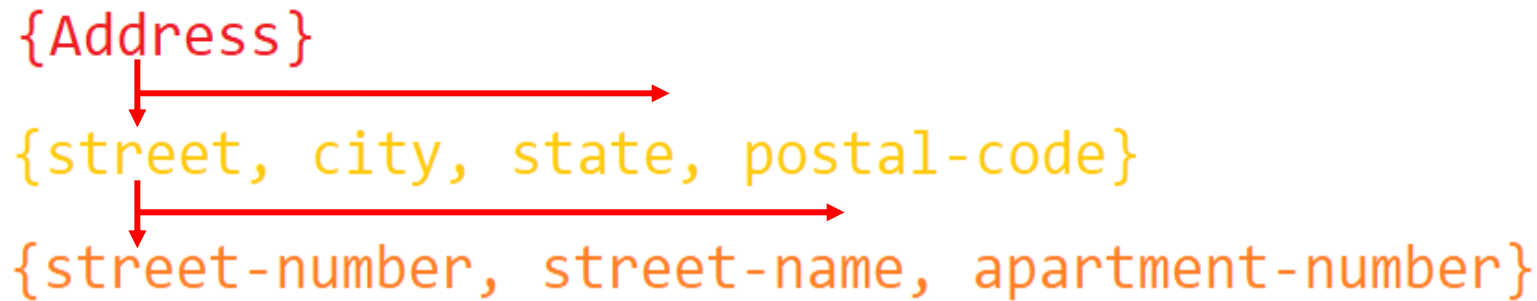


# Composite **VS** Multi Valued Attribute

## Composite Attribute

### Person Entity

- *Name* attribute: ( firstName, middleName, and lastName )
- *PhoneNumber* attribute: ( countryCode, cityCode, and phoneNumber )



## Multi Valued Attribute

### Person Entity

- *Hobbies* attribute: [ reading, hiking, hockey, skiing, photography, ... ]
- *SpokenLanguages* attribute: [ Hindi, Marathi, Gujarati, English, ... ]
- *Degrees* attribute: [ 10<sup>th</sup>, 12<sup>th</sup>, BE, ME, PhD, ... ]
- *emailID* attribute: [ saleel@gmail.com, salil@yahoomail.com, ... ]

# What is an Prime, Non-Prime Attribute?

## **Prime attribute** (*Entity integrity*)

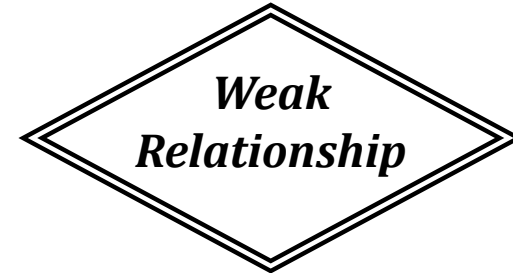
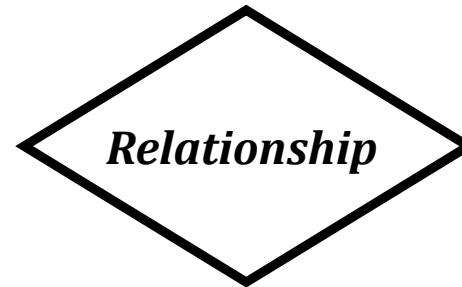
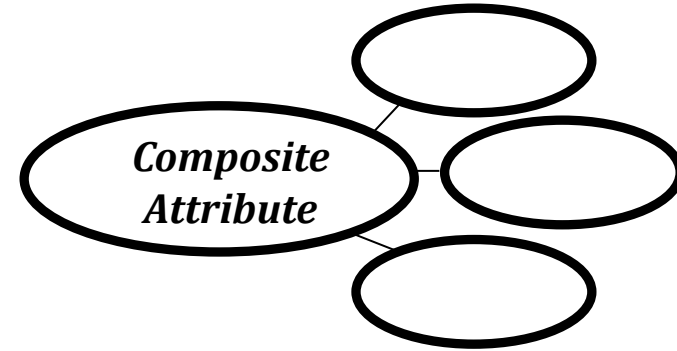
An attribute, which is a **part of the prime-key** (candidate key), is known as a prime attribute.

## **Non-prime attribute**

An attribute, which is **not a part of the prime-key** (candidate key), is said to be a non-prime attribute.

# Entity Relationship Diagram Symbols

# entity relationship diagram symbols



# *strong and weak entity*

An entity may participate in a relation either totally or partially.

**Strong Entity:** A strong entity is not dependent on any other entity in the schema. A strong entity **will always have a primary key**. Strong entities are represented by a single rectangle.

**Weak Entity:** A weak entity is dependent on a strong entity to ensure its existence. Unlike a strong entity, a weak entity **does not have any primary key**. A weak entity is represented by a double rectangle. The relation between one strong and one weak entity is represented by a double diamond. This relationship is also known as identifying relationship.

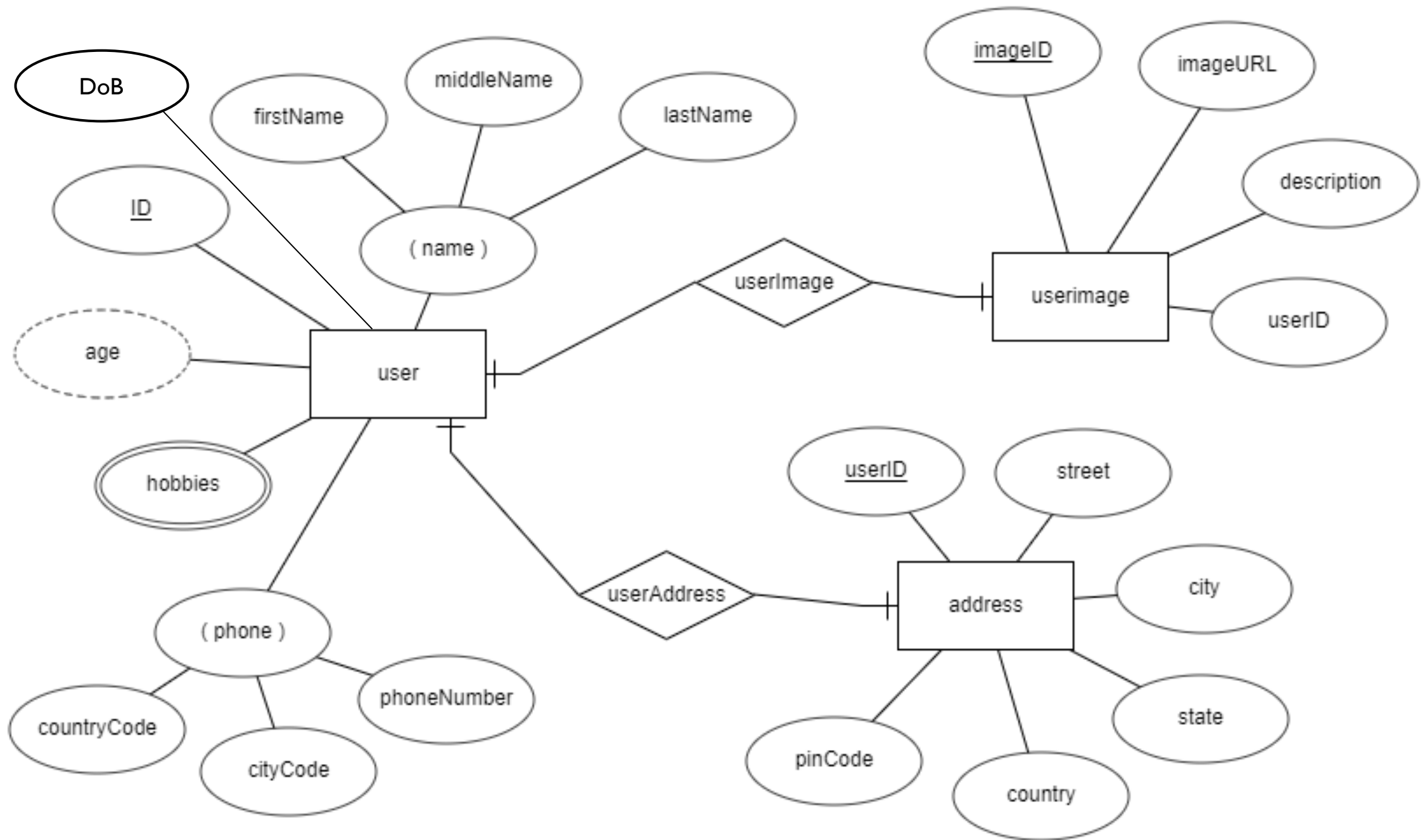
**Example 1** – A loan entity can not be created for a customer if the customer doesn't exist

**Example 2** – A payment entity can not be created for a loan if the loan doesn't exist

**Example 3** – A customer address entity can not be created for the customer if the customer doesn't exist

**Example 4** – A prescription entity can not be created for a patient if the patient doesn't exist

# entity relationship diagram



What is a degree, cardinality, domain and union in database?



# What is a degree, cardinality, domain and union in database?

- **Degree  $d(R)$  / Arity:** Total number of **attributes/columns** present in a relation/table is called **degree of the relation** and is denoted by  **$d(R)$** .
- **Cardinality  $|R|$ :** Total number of **tuples/rows** present in a relation/table, is called **cardinality of a relation** and is denoted by  **$|R|$** .

**Cardinality** is the numerical relationship between rows of one table and rows in another. Common cardinalities include *one-to-one*, *one-to-many*, and *many-to-many*.

- **Domain:** Total range of accepted values for an attribute of the relation is called the **domain of the attribute**. (**Data Type(size)**)
- **Union Compatibility:** Two relations  $R$  and  $S$  are set to be Union Compatible to each other if and only if:
  1. They have the **same degree  $d(R)$** .
  2. Domains of the respective attributes should also be same.

What is domain constraint and types of data integrity constraints?

Data integrity refers to the correctness and completeness of data.

## *A domain constraint and types of data integrity constraints*

- ❖ **Domain Constraint** = data type + Constraints (not null/unique/primary key/foreign key/check/default)  
e.g. custID INT, constraint pk\_custid PRIMARY KEY(custID)

Three types of integrity constraints: **entity integrity**, **referential integrity** and **domain integrity**:

- **Entity integrity:** Entity Integrity Constraint is used to ensure the uniqueness of each record the table. There are primarily two types of integrity constraints that help us in ensuring the uniqueness of each row, namely, UNIQUE constraint and PRIMARY KEY constraint.
- 
- **Referential integrity:** Referential Integrity Constraint ensures that there always exists a valid relationship between two tables. This makes sure that if a foreign key exists in a table relationship then it should always reference a corresponding value in the second table  $t_1[FK] = t_2[PK]$  or it should be null.
- **Domain integrity:** A domain is a set of values of the same type. For example, we can specify if a particular column can hold null values or not, if the values have to be unique or not, the data type or size of values that can be entered in the column, the default values for the column, etc..

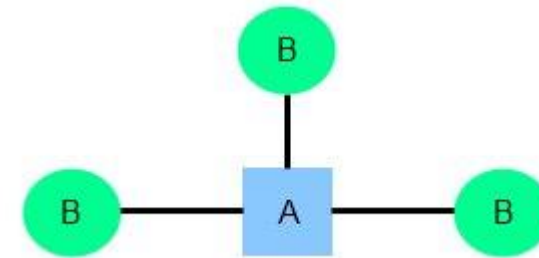
# Common relationships

## Common relationship

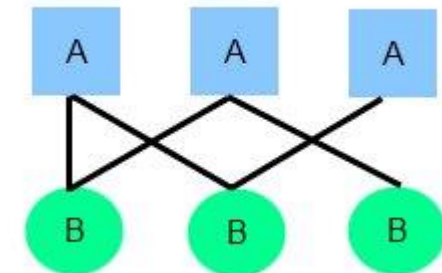
1. one-to-one (1:1)



2. one-to-many (1:M)



3. many-to-many (M:N)

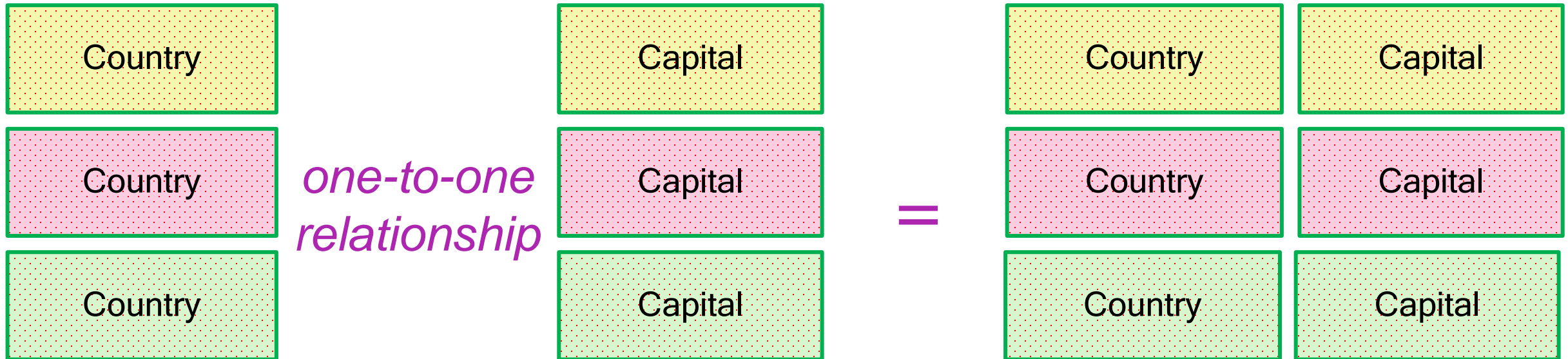


one-to-one relationship

## one-to-one relationship

A *one-to-one* relationship between two tables means that a row in one table can only relate to zero/one row in the table on the other side of their relationship. This is the least common database relationship.

A *one-to-one* relationship is a type of cardinality that refers to the relationship between two entities  $R$  and  $S$  in which one element of entity  $R$  may only be linked to zero/one element of entity  $S$ , and vice versa.



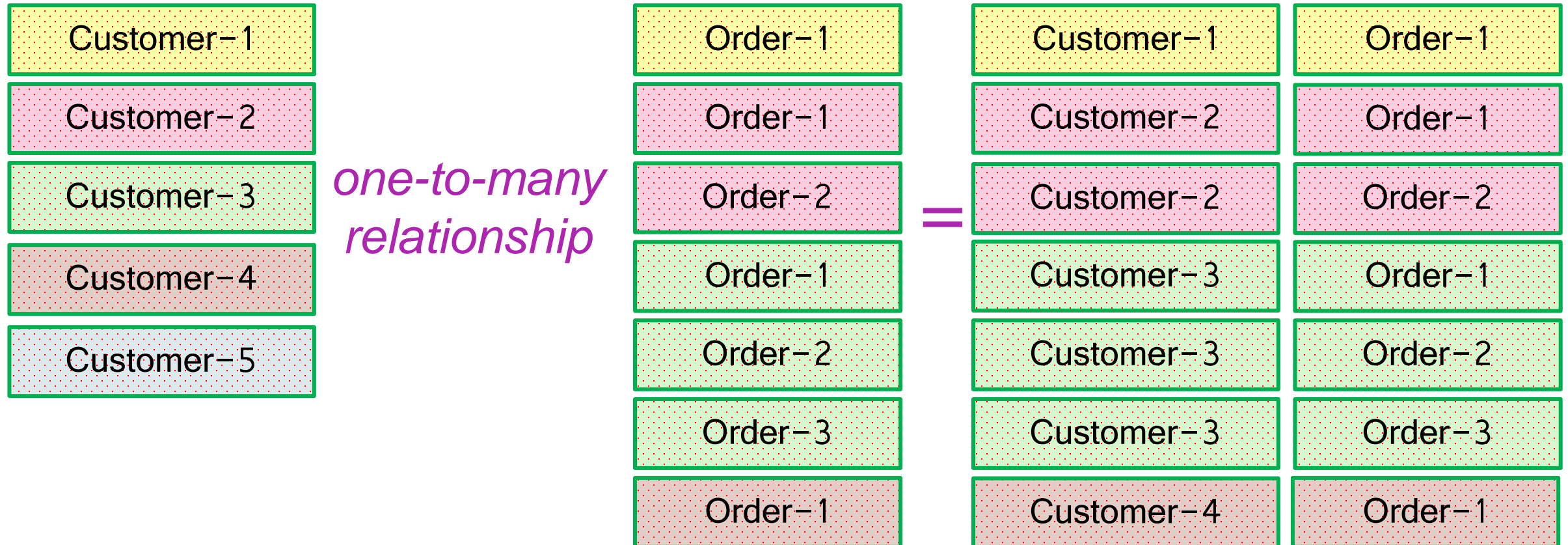
one-to-many relationship



# one-to-many relationship

A *one-to-many* relationship between two tables means that a row in one table can have zero or more row in the table on the other side of their relationship.

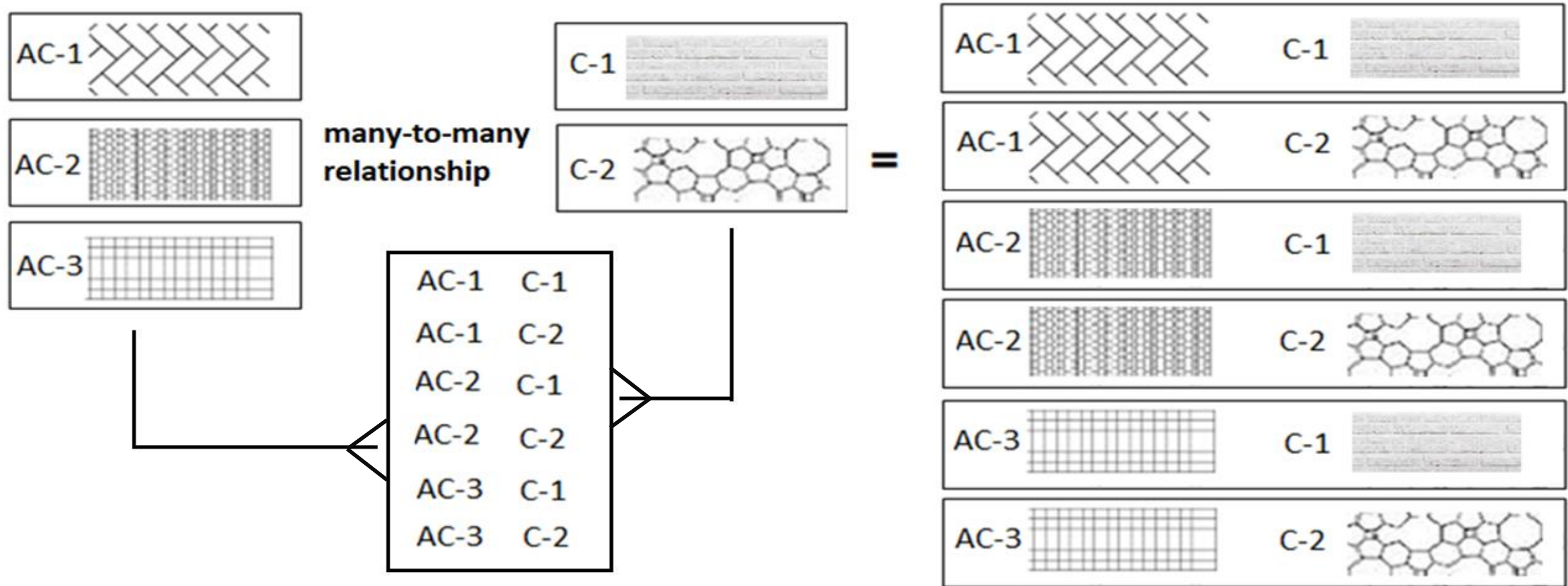
a *one-to-many* relationship is a type of cardinality that refers to the relationship between two entities  $R$  and  $S$  in which an element of  $R$  may be linked to many elements of  $S$ , but a member of  $S$  is linked to only one element of  $R$ .



many-to-many relationship

## many-to-many relationship

A *many-to-many* relationship is a type of cardinality that refers to the relationship between two entities *R* and *S* in which *R* may contain a parent instance for which there are many children in *S* and vice versa.



**Schema:** A schema is a collection of database objects (like table, columns , primary key, foreign key, views, etc.) associated with one particular database username. This username is called the schema owner. You may have one or multiple schemas in a database.

# What is schema and instance

## Instance

- The data stored in database at a particular moment of time is called instance of database.

**For example**, lets say we have a single table student in the database, today the table has 100 records, so today the instance of the database has 100 records. Lets say we are going to add another 100 records in this table by tomorrow so the instance of database tomorrow will have 200 records in table.

An instance of a relation is a set of tuples, also called records

## MySQL is the most popular **Open Source** Relational Database Management System.

MySQL was created by a Swedish company - MySQL AB that was founded in 1995. It was acquired by Sun Microsystems in 2008; Sun was in turn acquired by Oracle Corporation in 2010.

When you use MySQL, you're actually using at least two programmes. One program is the MySQL server (*mysqld.exe*) and other program is MySQL client program (*mysql.exe*) that connects to the database server.



What is SQL?

## Remember:

- **EXPLICIT** or **IMPLICIT** commit will commit the data.

## *what is sql?*

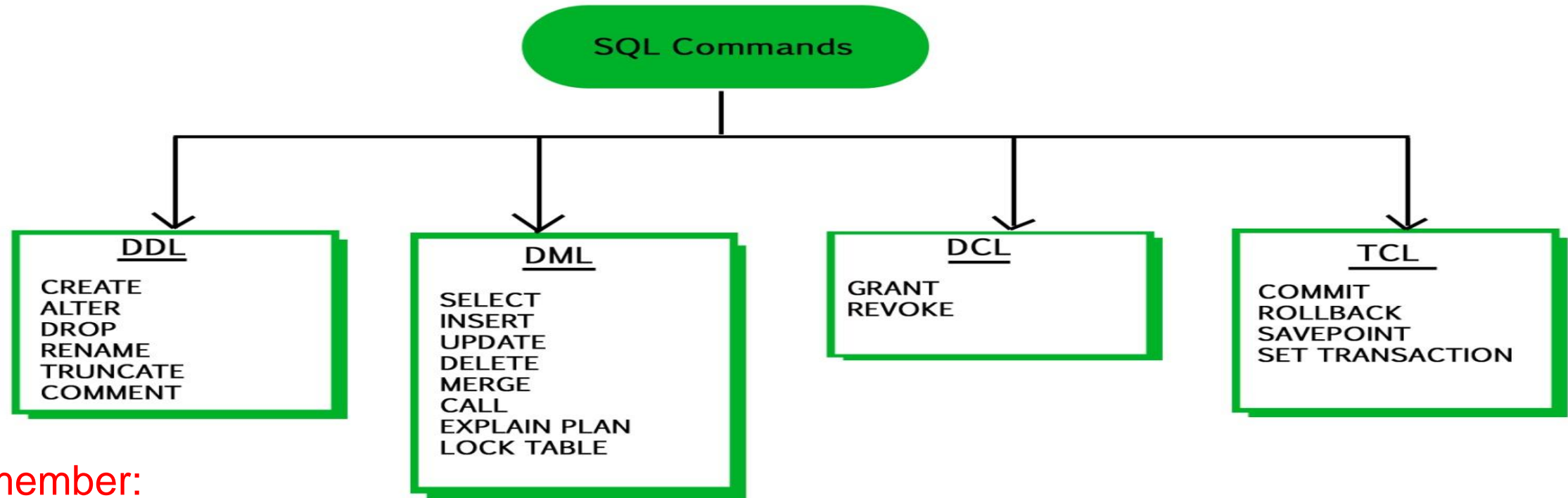
SQL (Structured Query Language) is a database language designed and developed for managing data in relational database management systems (RDBMS). SQL is common language for all Relational Databases.



## Remember:

## what is sql?

- An EXPLICIT commit happens when we execute an SQL "COMMIT" command.
- An IMPLICIT commits occur without running a "COMMIT" command.



## Remember:

- A **NULL** value is not treated as a **blank** or **0**. Null or NULL is a special marker used in Structured Query Language to indicate that a data value does not exist or missing or unknown in the database.
- **Degree  $d(R)$ :** Total no. of attributes/columns present in a relation/table is called degree of the relation and is denoted by  $d(R)$ .
- **Cardinality  $|R|$ :** Total no. of tuples present in a relation or Rows present in a table, is called cardinality of a relation and is denoted by  $|R|$ .



## comments in mysql

- From a **#** character to the end of the line.
- From a **--** sequence to the end of the line.
- From a **/\*** sequence to the following **\*/** sequence.

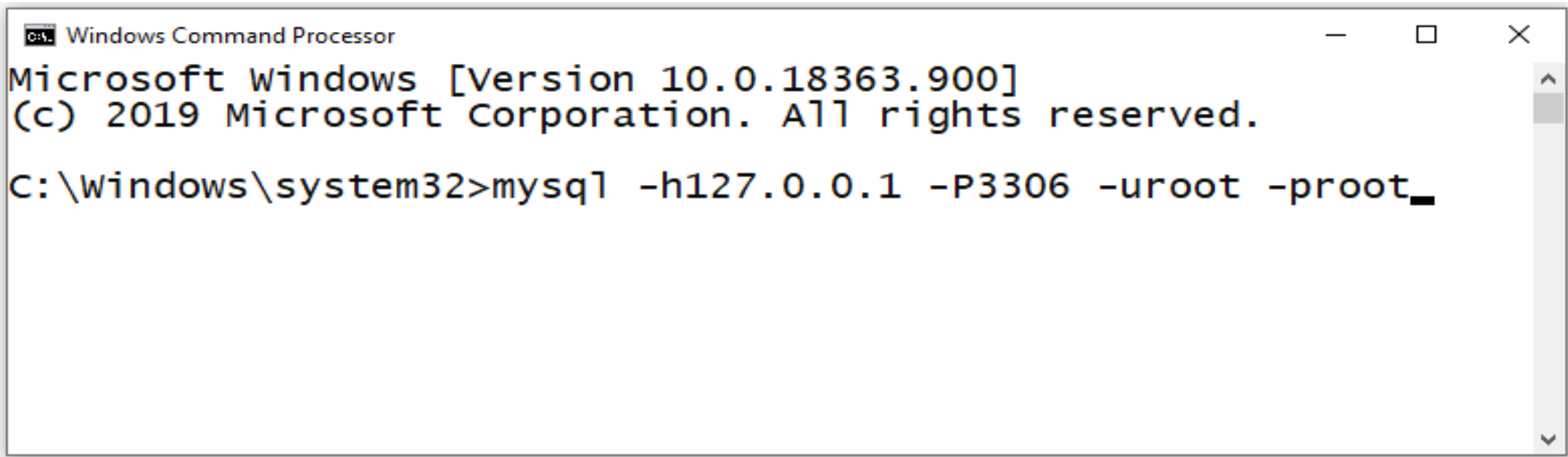
Reconnect to the server	\r
Execute a system shell command	\!
Exit mysql	\q
Change your mysql prompt.	<b>prompt str or \R str</b>

Login to MySQL

## Default port for MySQL Server: 3306

*login*

- C:\> mysql -hlocalhost -P3307 -uroot -p
- C:\> mysql -h127.0.0.1 -P3307 -uroot -p [database\_name]
- C:\> mysql -h192.168.100.14 -P3307 -uroot -psaleel [database\_name]
- C:\> mysql --host localhost --port 3306 --user root --password=ROOT [database\_name]
- C:\> mysql --host=localhost --port=3306 --user=root --password=ROOT [database\_name]

A screenshot of a Windows Command Processor window. The title bar reads "C:\ Windows Command Processor". The window content shows the following text: "Microsoft Windows [Version 10.0.18363.900]", "(c) 2019 Microsoft Corporation. All rights reserved.", and "C:\Windows\system32>mysql -h127.0.0.1 -P3306 -uroot -proot\_". The cursor is at the end of the command line.

```
C:\ Windows Command Processor
Microsoft Windows [Version 10.0.18363.900]
(c) 2019 Microsoft Corporation. All rights reserved.
C:\Windows\system32>mysql -h127.0.0.1 -P3306 -uroot -proot_
```

The **char** is a fixed-length character data type,  
The **varchar** is a variable-length character data type.

*datatypes*

ENAME CHAR (10)	S	A	L	E	E	L					LENGTH -> 6
ENAME VARCHAR(10)	S	A	L	E	E	L					LENGTH -> 6

In MySQL

When CHAR values are retrieved, the trailing spaces are removed  
(unless the **PAD\_CHAR\_TO\_FULL\_LENGTH** SQL mode is enabled)

**Note:**

The BINARY and VARBINARY types are similar to CHAR and VARCHAR, except that they store binary strings rather than nonbinary strings. That is, they store byte strings rather than character strings.

## *datatype - string*

Datatypes	Size	Description
CHAR [(length)]	0-255	
VARCHAR (length)	0 to 65,535	The maximum row size (65,535 bytes, which is shared among all columns.
TINYTEXT [(length)]	$(2^8 - 1)$ bytes	
TEXT [(length)]	$(2^{16} - 1)$ bytes	65,535 bytes ~ 64kb
MEDIUMTEXT [(length)]	$(2^{24} - 1)$ bytes	16,777,215 bytes ~16MB
LONGTEXT [(length)]	$(2^{32} - 1)$ bytes	4,294,967,295 bytes ~4GB
ENUM('value1', 'value2',...)	65,535 members	
SET('value1', 'value2',...)	64 members	
BINARY[(length)]	255	
VARBINARY(length)		

## *datatype - numeric*

Datatypes	Size	Description
TINYINT	1 byte	-128 to +127 <b>(The unsigned range is 0 to 255).</b>
SMALLINT [(length)]	2 bytes	-32768 to 32767. <b>(The unsigned range is 0 to 65535).</b>
MEDIUMINT [(length)]	3 bytes	-8388608 to 8388607. <b>(The unsigned range is 0 to 16777215).</b>
INT, INTEGER [(length)]	4 bytes	-2147483648 to 2147483647. <b>(The unsigned range is 0 to 4294967295).</b>
BIGINT [(length)]	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
FLOAT [(length[,decimals])]	4 bytes	<b>FLOAT(255,30)</b>
DOUBLE [PRECISION] [(length[,decimals])], REAL [(length[,decimals])]	8 bytes	<b>REAL(255,30) / DOUBLE(255,30)</b> REAL will get converted to DOUBLE
DECIMAL [(length[,decimals])], NUMERIC [(length[,decimals])]		<b>DECIMAL(65,30) / NUMERIC(65,30)</b> NUMERIC will get converted in DECIMAL

**UNSIGNED** prohibits negative values.

## *datatype – date and time*

<b>Datatypes</b>	<b>Size</b>	<b>Description</b>
YEAR	1 byte	YYYY
DATE	3 bytes	YYYY-MM-DD
TIME	3 bytes	HH:MM:SS
DATETIME	8 bytes	YYYY-MM-DD hh:mm:ss

Use a CREATE TABLE statement to specify the layout of your table.

### Remember:

- Max 4096 columns per table provided the row size  $\leq$  65,535 Bytes

## create table

Use a **CREATE TABLE** statement to specify the layout of your table.

### Note:

- **USER TABLES:** This is a collection of tables created and maintained by the user. Contain USER information.
- **DATA DICTIONARY:** This is a collection of tables created and maintained by the MySQL Server. It contains database information. All data dictionary tables are owned by the SYS user.



# create table

Use a **CREATE TABLE** statement to specify the layout of your table.

## Remember:

- by default, tables are created in the default database, using the InnoDB storage engine.
- table name should not begin with a number or special symbols.
- table name can start with `_table_name` (underscore) or `$table_name` (dollar sign)
- table name and column name can have max 64 char.
- multiple words as `table_name` is invalid, if you want to give multiple words as `table_name` then give it in ``table_name`` (backtick)
- error occurs if the table exists.
- error occurs if there is no default database.
- error occurs if the database does not exist.

## Note:

- Table names are stored in lowercase on disk. MySQL converts all table names to lowercase on storage. This behavior also applies to database names and table aliases.  
e.g. show variables like 'lower\_case\_table\_names';

# syntax

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name  
    (create_definition, . . .)  
    [table_options]  
    [partition_options]
```

## *create\_definition:*

```
col_name column_definition
```

## *column\_definition:*

```
data_type [NOT NULL | NULL] [DEFAULT default_value]  
    [AUTO_INCREMENT] [UNIQUE [KEY] | [PRIMARY] KEY]  
    [reference_definition]  
| data_type [GENERATED ALWAYS] AS (expression) [VIRTUAL]  
    [VISIBLE | INVISIBLE]
```

table\_options:

```
ENGINE [=] engine_name
```

# create table

e.g.

- CREATE TABLE student (  
 ID INT,  
 firstName VARCHAR(45),  
 lastName VARCHAR(45),  
 DoB DATE,  
 emailID VARCHAR(128)  
 );

```
show engines;
```

```
set default_storage_engine = memory;
```

*col\_name data\_type* DEFAULT value

## default value

The DEFAULT specifies a default value for the column.

- BLOB, TEXT, GEOMETRY or JSON column can't have a default value.  
e.g. CREATE TABLE temp(c1 TEXT DEFAULT('PUNE'));

```
INSERT [IGNORE] [INTO] tbl_name [PARTITION (partition_name [, partition_name] ...)] [ (col_name, . . .) ] {  
VALUES | VALU E } ( { expr | DEFAULT }, . . . ), ( . . . ), . . .
```

## insert rows

**INSERT** is used to add a single or multiple tuple to a relation. We must specify the relation name and a list of values for the tuple. **The values should be listed in the same order in which the corresponding attributes were specified in the CREATE TABLE command.**

You can insert data using following methods:

- INSERT ... VALUES
- INSERT ... SET
- INSERT ... SELECT

Do not use the **\*** operator in your SELECT statements. Instead, use column names. Reason is that in MySQL Server scans for all column names and replaces the **\*** with all the column names of the table(s) in the SELECT statement. Providing column names avoids this search-and-replace, and enhances performance.

## SELECT statement...

```
SELECT what_to_select  
FROM which_table  
WHERE conditions_to_satisfy;
```

# ***SELECT CLAUSE***

The **SELECT** statement retrieves or extracts data from tables in the database.

- You can use one or more tables separated by comma to extract data.
- You can fetch one or more fields/columns in a single **SELECT** command.
- You can specify star (\*) in place of fields. In this case, **SELECT** will return all the fields.
- **SELECT** can also be used to retrieve rows computed without reference to any table e.g. **SELECT 1 + 2;**



# *Capabilities of SELECT Statement*

1. SELECTION
2. PROJECTION
3. JOINING



# Capabilities of *SELECT* Statement

## ➤ *SELECTION*

Selection capability in SQL is to choose the record's/row's/tuple's in a table that you want to return by a query.

***R***

EMPNO	ENAME	JOB	HIREDATE	DEPTNO
1	Saleel	Manager	1995-01-01	10
2	Janhavi	Sales	1994-12-20	20
3	Snehal	Manager	1997-05-21	10
4	Rahul	Account	1997-07-30	10
5	Ketan	Sales	1994-01-01	30





# Capabilities of *SELECT* Statement

## ➤ *PROJECTION*

Projection capability in SQL to choose the column's/attribute's/field's in a table that you want to return by your query.

***R***

EMPNO	ENAME	JOB	HIREDATE	DEPTNO
1	Saleel	Manager	1995-01-01	10
2	Janhavi	Sales	1994-12-20	20
3	Snehal	Manager	1997-05-21	10
4	Rahul	Account	1997-07-30	10
5	Ketan	Sales	1994-01-01	30



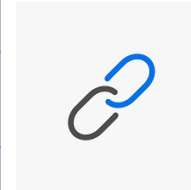
# Capabilities of *SELECT* Statement

## ➤ JOINING

Join capability in SQL to bring together data that is stored in different tables by creating a link between them.

**R**

EMPNO	ENAME	JOB	HIREDATE	DEPTNO
1	Saleel	Manager	1995-01-01	20
2	Janhavi	Sales	1994-12-20	10
3	Snehal	Manager	1997-05-21	10
4	Rahul	Account	1997-07-30	20
5	Ketan	Sales	1994-01-01	30




**S**

DEPTNO	DNAME	LOC
10	HRD	PUNE
20	SALES	BARODA
40	PURCHASE	SURAT



## SELECTION Process

**SELECT**  \* **FROM** <table\_references>  
selection-list | field-list | column-list

### Remember:

- Here, " \* " is known as metacharacter (all columns)

## PROJECTION Process

**SELECT**  column-list **FROM** <table\_references>  
selection-list | field-list | column-list

### Remember:

- Position of columns in SELECT statement will determine the position of columns in the output (as per user requirements)

```
UPDATE tbl_name SET col_name1 = { expr1 | DEFAULT } [, col_name2 = { expr2 | DEFAULT } ] ...  
[WHERE where_condition]
```

In a **SET** statement, **=** is treated identically to **:=**

## single-table update

**UPDATE** is used to change/modify the values of some attributes of one or more selected tuples.

```
DELETE FROM tbl_name  
[WHERE where_condition]
```

## single-table delete

**DELETE** is used to delete tuples from a relation.

# constraints

CONSTRAINT is used to define rules to allow or restrict what values can be stored in columns. The purpose of inducing constraints is to enforce the integrity of a database.

CONSTRAINTS can be classified into two types –

- *Column Level*
- *Table Level*

## Remember:

- **PRI** => primary key
- **UNI** => unique key
- **MUL** => is basically an index that is neither a **primary key** nor a **unique key**. The name comes from "multiple" because multiple occurrences of the same value are allowed.

# constraints

To limit or to restrict or to check or to control.

## Note:

- a table with a foreign key that references another table's primary key is **MUL**.
- If more than one of the Key values applies to a given column of a table, Key displays the one with the highest priority, in the order **PRI**, **UNI**, and **MUL**.
- If a table has a PRIMARY KEY or UNIQUE NOT NULL index that consists of a single column that has an integer type, you can use **\_rowid** to refer to the indexed column in SELECT statements.

## Remember:

- A primary key cannot be NULL.
- A primary key value must be unique.
- A table has only one primary key.
- The primary key values cannot be changed, if it is referred by some other column.
- **An index can consist of 16 columns, at maximum. Since a PRIMARY KEY constraint automatically adds an index, it can't have more than 16 columns.**

*col\_name data\_type* PRIMARY KEY

# PRIMARY KEY constraint

A primary key is a special column (or set of combined columns) in a relational database table, that is used to uniquely identify each record.

## Note:

- Primary key in a relation is always associated with an **INDEX** object.
- If, we give on a column a combination of **NOT NULL & UNIQUE** key then it behaves like a PRIMARY key.
- If, we give on a column a combination of **UNIQUE key & AUTO\_INCREMENT** then also it behaves like a PRIMARY key.



```
ALTER TABLE table_name  
  ADD [ CONSTRAINT constraint_name ]  
    PRIMARY KEY (column1, column2, . . . column_n)
```

## Add / Drop Primary Key

```
ALTER TABLE table_name  
  DROP PRIMARY KEY
```

## Remember:

- A unique key can be NULL.
- A unique key value must be unique.
- A table can have multiple unique key.
- A column can have unique key as well as a primary key.

*col\_name data\_type* **UNIQUE KEY**

# UNIQUE KEY constraint

A **UNIQUE key** constraint is a set of one or more than one fields/columns of a table that uniquely identify a record in a database table.

## Note:

- Unique key in a relation is always associated with an ***INDEX*** object.

```
ALTER TABLE table_name  
  ADD [ CONSTRAINT constraint_name ]  
    UNIQUE (column1, column2, . . . column_n)
```

## Add / Drop Unique Key

```
ALTER TABLE table_name  
  DROP INDEX constraint_name;
```

```
[CONSTRAINT [symbol]] FOREIGN KEY (col_name, ...) REFERENCES tbl_name (col_name,...)  
[ON DELETE reference_option]  
[ON UPDATE reference_option]
```

*reference\_option*: CASCADE | SET NULL

## FOREIGN KEY constraint

- A **FOREIGN KEY** is a **key** used to link two tables together.
- A **FOREIGN KEY** is a field (or collection of fields) in one table that refers to the **PRIMARY KEY** in another table.
- The table containing the **foreign key** is called the child table, and the table containing the candidate **key** is called the referenced or parent table.

# *constraints – foreign key*

## Remember:

- A foreign key can have a different column name from its primary key.
- DataType of primary key and foreign key column must be same.
- It ensures rows in one table have corresponding rows in another.
- Unlike the Primary key, they do not have to be unique.
- Foreign keys can be null even though primary keys can not.

## Note:

- The table containing the FOREIGN KEY is referred to as the child table, and the table containing the PRIMARY KEY (referenced key) is the parent table.
- PARENT and CHILD tables must use the same storage engine,
- and they cannot be defined as temporary tables.

```
ALTER TABLE table_name  
  ADD [ CONSTRAINT constraint_name ]  
    FOREIGN KEY (child_col1, child_col2, . . . child_col_n)  
    REFERENCES parent_table (parent_col1, parent_col2, . . . parent_col_n);
```

## Add / Drop Foreign Key

```
ALTER TABLE table_name  
  DROP FOREIGN KEY constraint_name
```

*col\_name data\_type* CHECK (expr)

# Check Constraint

## CHECK condition expressions must follow some rules.

- Literals, deterministic built-in functions, and operators are permitted.
  - Non-generated and generated columns are permitted, except columns with the `AUTO_INCREMENT` attribute.
  - Sub-queries are not permitted.
  - Environmental variables (such as `CURRENT_USER`, `CURRENT_DATE`, ...) are not permitted.
  - Non-Deterministic built-in functions (such as `AVG`, `COUNT`, `RAND`, `LAST_INSERT_ID`, `FIRST_VALUE`, `LAST_VALUE`, ...) are not permitted.
  - Variables (system variables, user-defined variables, and stored program local variables) are not permitted.
  - Stored functions and user-defined functions are not permitted.
- 

### Note:

Prior to MySQL 8.0.16, `CREATE TABLE` permits only the following limited version of table `CHECK` constraint syntax, which is parsed and ignored.

### Remember:

If you omit the constraint name, MySQL automatically generates a name with the following convention:

- `table_name_chk_n`



```
ALTER TABLE table_name  
  ADD [ CONSTRAINT constraint_name ]  
    CHECK (condition)
```

## Add / Drop Check Constraint

```
ALTER TABLE table_name  
  DROP { CHECK | CONSTRAINT } constraint_name
```

# alter table

ALTER TABLE changes the structure of a table.

## Note:

- you can add or delete columns,
- create or destroy indexes,
- change the type of existing columns, or
- rename columns or the table itself.
- You cannot change the position of columns in table structure. If not, then what? create a new table with **SELECT statement**.

# *syntax*

## *alter table*

ALTER TABLE tbl\_name

[alter\_specification [, alter\_specification] ...

- | ADD [COLUMN] col\_name column\_definition [FIRST | AFTER col\_name ]
- | ADD [COLUMN] (col\_name column\_definition, ...)
- | MODIFY [COLUMN] col\_name column\_definition [FIRST | AFTER col\_name]
- | DROP [COLUMN] col\_name
- | RENAME [TO|AS] new\_tbl\_name
- | RENAME COLUMN old\_col\_name TO new\_col\_name
- | ALTER [COLUMN] col\_name { SET DEFAULT {literal | (expr)} | DROP DEFAULT | SET { VISIBLE | INVISIBLE}}

`DROP [TEMPORARY] TABLE [IF EXISTS] tbl_name [, tbl_name] ...`

## drop table

### Remember:

- DROP and TRUNCATE are DDL commands, whereas DELETE is a DML command.
- DELETE operations can be rolled back (undone), while DROP and TRUNCATE operations cannot be rolled back (DDL statements are auto committed).
- Dropping a TABLE also drops any TRIGGERS for the table.
- Dropping a TABLE also drops any INDEX for the table.
- Dropping a TABLE will not drop any VIEW for the table.
- If you try to drop a PARENT/MASTER TABLE, it will not get dropped.

# create temporary table

## Note:

- it is possible to create, alter, drop, and write (Insert, Update, and Delete rows) to TEMPORARY tables.

# temporary table

## Remember:

- You can use the *TEMPORARY* keyword when creating a table.
- A *TEMPORARY* table is visible only to the current session, and is dropped automatically when the session is closed.
- Use *TEMPORARY* table with the same name as the original can be useful when you want to try some statements that modify the contents of the table, without changing the original table.
- The permanent (original) table becomes hidden (inaccessible) to the client who creates the *TEMPORARY* table with same name as the original.
- If you issue a DROP TABLE statement, the *TEMPORARY* table is removed and the original table reappears, it is possible, only when then original *tbl\_name* and temporary *tbl\_name* are same.
- The original table also reappears if you rename the *TEMPORARY* table.

e.g. ALTER TABLE dept RENAME TO d;

Temporary table\_name

Do not use the \* operator in your SELECT statements. Instead, use column names. Reason is that in MySQL Server scans for all column names and replaces the \* with all the column names of the table(s) in the SELECT statement. Providing column names avoids this search-and-replace, and enhances performance.

continue with SELECT statement...

```
SELECT what_to_select  
FROM which_table  
WHERE conditions_to_satisfy;
```

The asterisk symbol “ \* ” can be used in the SELECT clause to denote “all attributes.”

# ***SELECT CLAUSE***

The **SELECT** statement retrieves or extracts data from tables in the database.

- You can use one or more tables separated by comma to extract data.
- You can fetch one or more fields/columns in a single **SELECT** command.
- You can specify star (\*) in place of fields. In this case, **SELECT** will return all the fields.
- **SELECT** can also be used to retrieve rows computed without reference to any table e.g. **SELECT 1 + 2;**



# *Capabilities of SELECT Statement*

1. SELECTION
2. PROJECTION
3. JOINING

# Capabilities of *SELECT* Statement

## ➤ *SELECTION*

Selection capability in SQL is to choose the rows in a table that you want to return by a query.

*R*

EMPNO	ENAME	JOB	HIREDATE	DEPTNO
1	Saleel	Manager	1995-01-01	10
2	Janhavi	Sales	1994-12-20	20
3	Snehal	Manager	1997-05-21	10
4	Rahul	Account	1997-07-30	10
5	Ketan	Sales	1994-01-01	30

# Capabilities of *SELECT* Statement

## ➤ *PROJECTION*

Projection capability in SQL to choose the columns in a table that you want to return by your query.

***R***

EMPNO	ENAME	JOB	HIREDATE	DEPTNO
1	Saleel	Manager	1995-01-01	10
2	Janhavi	Sales	1994-12-20	20
3	Snehal	Manager	1997-05-21	10
4	Rahul	Account	1997-07-30	10
5	Ketan	Sales	1994-01-01	30

# Capabilities of *SELECT* Statement

## ➤ JOINING

Join capability in SQL to bring together data that is stored in different tables by creating a link between them.

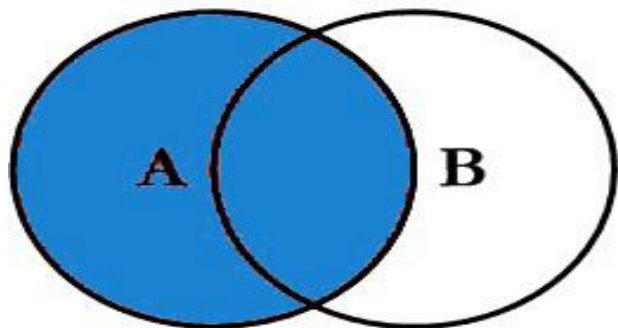
**R**

EMPNO	ENAME	JOB	HIREDATE	DEPTNO
1	Saleel	Manager	1995-01-01	20
2	Janhavi	Sales	1994-12-20	10
3	Snehal	Manager	1997-05-21	10
4	Rahul	Account	1997-07-30	20
5	Ketan	Sales	1994-01-01	30

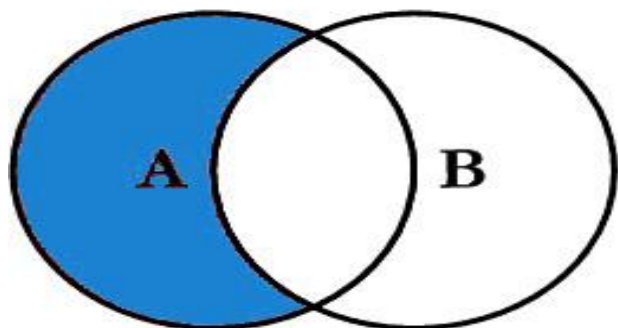
**S**

DEPTNO	DNAME	LOC
10	HRD	PUNE
20	SALES	BARODA
40	PURCHASE	SURAT

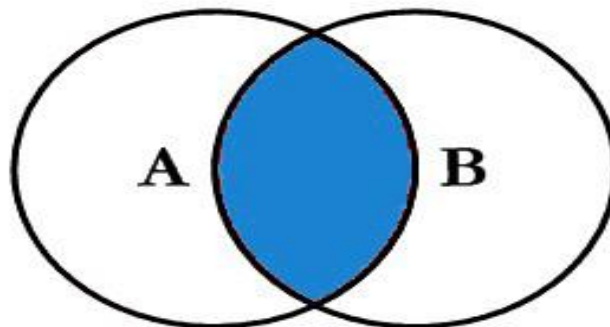
# SQL JOINS



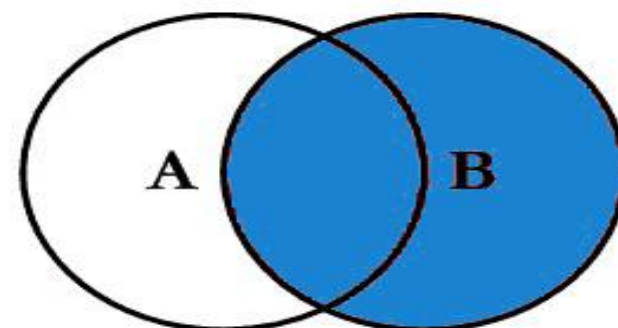
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key
```



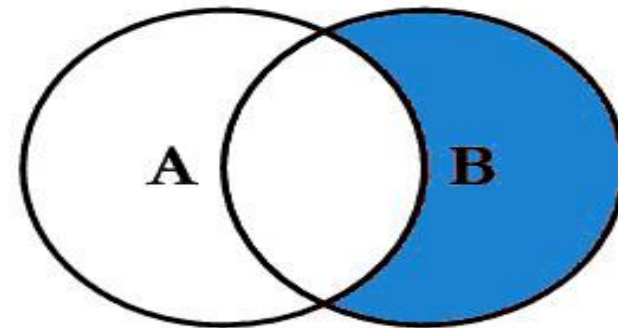
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key  
WHERE B.Key IS NULL
```



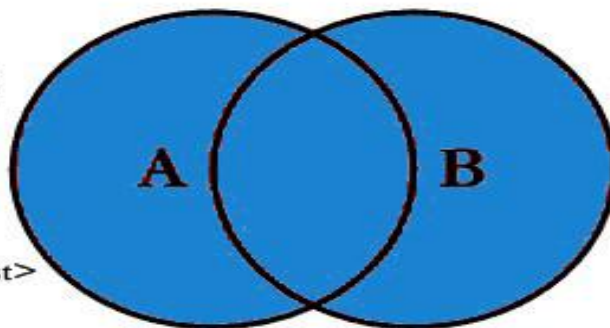
```
SELECT <select_list>  
FROM TableA A  
INNER JOIN TableB B  
ON A.Key = B.Key
```



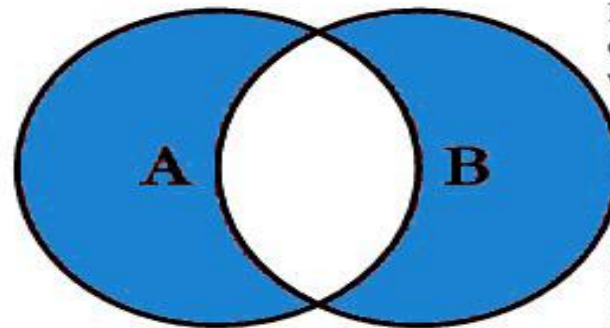
```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL
```




```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL  
OR B.Key IS NULL
```

## SELECTION Process

**SELECT**  **FROM** <table\_references>

selection-list | field-list | column-list

### Remember:

- Here, " \* " is known as metacharacter (all columns)

## PROJECTION Process

**SELECT**  **FROM** <table\_references>

selection-list | field-list | column-list

### Remember:

- Position of columns in SELECT statement will determine the position of columns in the output (as per user requirements)

# column - alias

A programmer can use an alias to temporarily assign another name to a **column** or **table** for the duration of a *SELECT* query.

In the selection-list, a quoted column alias can be specified using identifier ( ` ) or string quote ( ' or " ) characters.

## Note:

- Assigning an alias\_name does not actually rename the column or table.
- You cannot use alias in an expression.

## *select statement - alias*

`SELECT  $A_1$  [ [AS] alias_name],  $A_2$  [ [AS] alias_name], . . . ,  $A_N$  FROM  $r$  [ [AS] alias_name]`

  
column-name as new-name

  
table-name as new-name

### Remember:

- A select\_expr can be given an alias using **AS alias\_name**. The alias is used as the expression's column name and can be used in **GROUP BY**, **HAVING**, or **ORDER BY** clauses.
- The **AS** keyword is optional when aliasing a select\_expr with an identifier.
- Standard SQL **disallows** references to column aliases in a **WHERE** clause.
- A table reference can be aliased using **tbl\_name alias\_name** or **tbl\_name AS alias\_name**
- If the column alias contains spaces, **put it in quotes**.
- Alias name is **max 256 characters**.



*assignment\_operator*

= (assignment), :=

## comparison functions and operator

Comparison operations result in a value of 1 (**TRUE**), 0 (**FALSE**), or **NULL**.

# comparison functions and operator

## 1. *arithmetic\_operators:*

\* | / | DIV | % | MOD | - | +

## 2. *comparison\_operator:*

= | <=> | >= | > | <= | < | <> | !=

## 3. *boolean\_predicate:*

IS [NOT] NULL  
| expr <=> null

## 4. *predicate:*

expr [NOT] LIKE expr [ESCAPE char]  
| expr [NOT] IN (expr1, expr2, ... )  
| expr [NOT] IN (subquery)  
| expr [NOT] BETWEEN expr1 AND expr2

## 5. *logical\_operators*

{ AND | && } | { OR | || }

## 6. *assignment\_operator*

= (assignment), :=

**operand meaning:** the quantity on which an operation is to be done.

- SELECT 23 DIV 6 ;                      #3
- SELECT 23 / 6 ;                      #3 .8333

**Note:**

- AND has higher precedence than OR.

column - expressions

# *select statement - expressions*

## Column EXPRESSIONS

SELECT  $A_1, A_2, A_3, A_4$ , expressions, . . . FROM  $r$

- SELECT 1001 + 1;
  - SELECT 1001 + '1';
  - SELECT '1' + '1';
  - SELECT '1' + 'a1';
  - SELECT '1' + '1a';
  - SELECT 'a1' + 1;
  - SELECT '1a' + 1;
  - SELECT 1 + -1;
  - SELECT 1 + -2;
  - SELECT -1 + -1;
  - SELECT -1 - 1;
  - SELECT -1 - -1;
  - SELECT 123 \* 1;
  - SELECT -123 \* 1;
  - SELECT 123 \* -1;
  - SELECT -123 \* -1;
  - SELECT 2 \* 0;
  - SELECT 2435 / 1;
  - SELECT 2 / 0;
  - SELECT '2435Saleel' / 1;
  - SELECT sal, sal + 1000 AS 'New Salary' FROM emp;
  - SELECT sal, comm, sal + comm FROM emp;
  - SELECT sal, comm, sal + IFNULL(comm, 0) FROM emp;
  - SELECT ename, ename = ename FROM emp;
  - SELECT ename, ename = 'smith' FROM emp;
  - SELECT c1, c1 / 1 R1 FROM numberString;
- Note:**  
If any expression evaluated with NULL, returns NULL.
- SELECT 2 + NULL;
  - SELECT 2 \* NULL;
  - SELECT 2 - NULL;
  - SELECT 2 / NULL;

# identifiers

Certain objects within MySQL, including database, table, index, column, alias, view, stored procedure, stored functions, triggers, partition, tablespace, and other object names are known as **identifiers**.

# *identifiers*

The maximum length for each type of identifiers like (Database, Table, Column, Index, Constraint, View, Stored Program, Compound Statement Label, User-Defined Variable, Tablespace) is **64 characters**, whereas for Alias is **256 characters**.

- You can refer to a table within the default database as
  1. `tbl_name`
  2. `db_name.tbl_name`.
- You can refer to a column as
  1. `col_name`
  2. `tbl_name.col_name`
  3. `db_name.tbl_name.col_name`.

control flow functions

# *control flow functions - ifnull*

## IFNULL function

**MySQL IFNULL()** takes two expressions, if the first expression is not NULL, it returns the first expression. Otherwise, it returns the second expression, **it returns either numeric or string value.**

**IFNULL**(expression1, expression2)

## IF function

If **expr1 is TRUE or expr1 <> 0 or expr1 <> NULL**, then IF() returns expr2, otherwise it returns expr3, **it returns either numeric or string value.**

**IF**(expr1, expr2 , expr3)



# *control flow functions - case*

## CASE function

Returns the result for the first condition that is true. If there was no matching result value, the result after ELSE is returned, or NULL if there is no ELSE part.

**CASE** value **WHEN** [compare\_value] **THEN** result [**WHEN** [compare\_value] **THEN** result . . .] [**ELSE** result] **END**

**CASE WHEN** [condition] **THEN** result [**WHEN** [condition] **THEN** result . . .] [**ELSE** result] **END**

datetime functions

## *sysdate(), now(), curdate(), curtime()*

In MySQL, the **NOW()** function returns a default value for a **DATETIME**.

MySQL inserts the current **date and time** into the column whose default value is NOW().

In MySQL, the **CURDATE()** returns the current date in 'YYYY-MM-DD'. **CURRENT\_DATE()** and **CURRENT\_DATE** are the **synonym of CURDATE()**.

In MySQL, the **CURTIME()** returns the value of current time in 'HH:MM:SS'. **CURRENT\_TIME()** and **CURRENT\_TIME** are the **synonym of CURTIME()**.

## + or - operator

Date arithmetic also can be performed using INTERVAL together with the + or - operator

date + INTERVAL expr unit + INTERVAL expr unit + INTERVAL expr unit + . . .

date - INTERVAL expr unit - INTERVAL expr unit - INTERVAL expr unit - . . .

- SELECT NOW(), NOW() + INTERVAL 1 DAY;
- SELECT NOW(), NOW() + INTERVAL '1-3' YEAR\_MONTH;

unit Value	expr	unit Value	expr
SECOND	SECONDS	DAY_HOUR	'DAYS HOURS' e.g. '1 1'
MINUTE	MINUTES	DAY_MINUTE	'DAYS HOURS:MINUTES' e.g. '1 3:34'
HOUR	HOURS	DAY_SECOND	'DAYS HOURS:MINUTES:SECONDS'
DAY	DAYS	HOUR_MINUTE	'HOURS:MINUTES' e.g. '3:34'
WEEK	WEEKS	HOUR_SECOND	'HOURS:MINUTES:SECONDS'
MONTH	MONTHS	MINUTE_SECOND	'MINUTES:SECONDS' e.g. '27:34'
QUARTER	QUARTERS	YEAR_MONTH	'YEARS-MONTHS' e.g. '1-3'
YEAR	YEARS		

## ADDDATE() is a synonym for DATE\_ADD()

ADDDATE(date, INTERVAL expr unit) / DATE\_ADD (date, INTERVAL expr unit)

- SELECT NOW(), ADDDATE(NOW(), INTERVAL 1 DAY);
- SELECT NOW(), ADDDATE(NOW(), 1);

unit Value	ExpectedexprFormat
SECOND	SECONDS
MINUTE	MINUTES
HOUR	HOURS
DAY	DAYS
WEEK	WEEKS
MONTH	MONTHS
QUARTER	QUARTERS
YEAR	YEARS

## **SUBDATE()** is a synonym for **DATE\_SUB()**

`SUBDATE(date, INTERVAL expr unit)` / `DATE_SUB (date, INTERVAL expr unit)`

- `SELECT NOW(), SUBDATE(NOW(), INTERVAL 1 DAY);`
- `SELECT NOW(), SUBDATE(NOW(), 1);`

unit Value	ExpectedexprFormat
SECOND	SECONDS
MINUTE	MINUTES
HOUR	HOURS
DAY	DAYS
WEEK	WEEKS
MONTH	MONTHS
QUARTER	QUARTERS
YEAR	YEARS

## extract

The EXTRACT() function is used to return a single part of a date/time, such as year, month, day, hour, minute, etc.

EXTRACT(unit FROM date)

Unit Value				
MICROSECOND	SECOND	MINUTE	HOUR	DAY
WEEK	MONTH	QUARTER	YEAR	
MINUTE_SECOND	HOUR_SECOND	DAY_SECOND	DAY_HOUR	
HOUR_MINUTE	DAY_MINUTE	YEAR_MONTH		

### Note:

- There must no space between extract function and ().

e.g.

```
SELECT EXTRACT (MONTH FROM NOW()); # error
```

# *datetime functions*

Syntax	Result
<code>DAY(date)</code>	<code>DAY()</code> is a <b>synonym</b> for <code>DAYOFMONTH()</code> .
<code>DAYNAME(date)</code>	Returns the name of the weekday for date.
<code>DAYOFMONTH(date)</code>	Returns the day of the month for date, in the range 1 to 31
<code>DAYOFWEEK(date)</code>	Returns the weekday index for date (1 = Sunday, 2 = Monday, ..., 7 = Saturday).
<code>DAYOFYEAR(date)</code>	Returns the day of the year for date, in the range 1 to 366
<code>LAST_DAY(date)</code>	Takes a date or datetime value and returns the corresponding value for the last day of the month. Returns NULL if the argument is invalid.
<code>MONTH(date)</code>	Returns the month for date, in the range 1 to 12 for January to December
<code>MONTHNAME(date)</code>	Returns the full name of the month for date.
<code>YEAR(date)</code>	Returns the year in 4 digit

- `SELECT DAYOFWEEK(NOW()), WEEKDAY(NOW());`
- `SELECT DAYOFWEEK(ADDDATE(NOW(), INTERVAL 1 DAY)), WEEKDAY(ADDDATE(NOW(), INTERVAL 1 DAY));`




## *datetime functions*

Syntax	Result
<code>WEEKDAY(date)</code>	Returns the weekday index for date (0 = Monday, 1 = Tuesday, ... 6 = Sunday).
<code>WEEKOFYEAR(date)</code>	Returns the calendar week of the date as a number in the range from 1 to 53.
<code>QUARTER(date)</code>	Returns the quarter of the year for date, in the range 1 to 4.
<code>HOUR(time)</code>	Returns the hour for time. The range of the return value is 0 to 23 for time-of-day values.
<code>MINUTE(time)</code>	Returns the minute for time, in the range 0 to 59.
<code>SECOND(time)</code>	Returns the second for time, in the range 0 to 59.
<code>DATEDIFF(expr1, expr2)</code>	Returns the number of days between two dates or datetimes.
<code>STR_TO_DATE(str, format)</code>	Convert a string to a date.

- `SELECT STR_TO_DATE('24/05/2022', '%d/%m/%Y');`

datetime formats

## *datetime formats*

Formats	Description
%a	Abbreviated weekday name (Sun-Sat)
%b	Abbreviated month name (Jan-Dec)
%c	Month, numeric (1-12)
%D	Day of month with English suffix (0th, 1st, 2nd, 3rd,  )
%d	Day of month, numeric (01-31)
%e	Day of month, numeric (1-31)
%f	Microseconds (000000-999999)
%H	Hour (00-23)
%h	Hour (01-12)

- `SELECT DATE_FORMAT(NOW(), '%a');`

## *datetime formats*

Formats	Description
%I	Hour (01-12)
%i	Minutes, numeric (00-59)
%j	Day of year (001-366)
%k	Hour (0-23)
%l	Hour (1-12)
%M	Month name (January-December)
%m	Month, numeric (01-12)
%p	AM or PM
%r	Time, 12-hour (hh:mm:ss followed by AM or PM)
%S	Seconds (00-59)
%s	Seconds (00-59)

## *datetime formats*

Formats	Description
%T	Time, 24-hour (hh:mm:ss)
%U	Week (00-53) where Sunday is the first day of week
%u	Week (00-53) where Monday is the first day of week
%V	Week (01-53) where Sunday is the first day of week, used with %X
%v	Week (01-53) where Monday is the first day of week, used with %x
%W	Weekday name (Sunday-Saturday)
%w	Day of the week (0=Sunday, 6=Saturday)
%X	Year for the week where Sunday is the first day of week, four digits, used with %V
%x	Year for the week where Monday is the first day of week, four digits, used with %v
%Y	Year, numeric, four digits
%y	Year, numeric, two digits

string functions

## string functions

Syntax	Result
ASCII(str)	Returns the numeric value of the leftmost character of the string str. Returns 0 if str is the empty string. Returns NULL if str is NULL.
CHAR(N, , . . .)	CHAR() interprets each argument N as an integer and returns a string consisting of the characters given by the code values of those integers. <b>NULL values are skipped.</b>
CONCAT(str1, str2, . . .)	Returns the string that results from concatenating the arguments. CONCAT() <b>returns NULL if any argument is NULL.</b>
LEFT(str, len)	Returns the leftmost len characters from the string str, or NULL if any argument is NULL.
RIGHT(str, len)	Returns the rightmost len characters from the string str, or NULL if any argument is NULL.
LTRIM(str)	Returns the string str with leading space characters removed.
RTRIM(str)	Returns the string str with trailing space characters removed.
TRIM(str)	Returns the string str with leading and trailing space characters removed.
BINARY value	Convert a value to a binary string.

# string functions

Syntax	Result
<code>STRCMP(expr1, expr2)</code>	<code>STRCMP()</code> returns 0 if the strings are the same, -1 if the first argument is smaller than the second according to the current sort order, and 1 otherwise.
<code>LCASE(str)</code>	Returns lower case string. <code>LCASE()</code> is a <b>synonym</b> for <code>LOWER()</code> .
<code>UCASE(str)</code>	Returns upper case string. <code>UCASE()</code> is a <b>synonym</b> for <code>UPPER()</code> .
<code>LENGTH(str)</code>	Returns the length of the string.
<code>LPAD(str, len, padstr)</code>	Returns the string <code>str</code> , left-padded with the string <code>padstr</code> to a length of <code>len</code> characters.
<code>RPAD(str, len, padstr)</code>	Returns the string <code>str</code> , right-padded with the string <code>padstr</code> to a length of <code>len</code> characters.
<code>REPEAT(str, count)</code>	Returns a string consisting of the string <code>str</code> repeated <code>count</code> times. If <code>count</code> is less than 1, returns an empty string. Returns NULL if <code>str</code> or <code>count</code> are NULL.
<code>INSTR(str, substr)</code>	Returns the position of the first occurrence of substring <code>substr</code> in string <code>str</code> .
<code>REPLACE(str, from_str, to_str)</code>	Returns the string <code>str</code> with all occurrences of the string <code>from_str</code> replaced by the string <code>to_str</code> . <code>REPLACE()</code> performs a case-sensitive match when searching for <code>from_str</code> .
<code>REVERSE(str)</code>	Returns the string <code>str</code> with the order of the characters reversed.
<code>SUBSTR(str, pos, len)</code>	<b><code>SUBSTR()</code> is a synonym for <code>SUBSTRING()</code>.</b>



mathematical functions

# mathematical functions

Syntax	Result
<code>ABS(x)</code>	Returns the absolute value of X.
<code>CEIL(x)</code>	<code>CEIL()</code> is a synonym for <code>CEILING()</code> .
<code>CEILING(x)</code>	Returns CEIL value.
<code>FLOOR(x)</code>	Returns FLOOR value.
<code>MOD(n, m),</code> <code>n % m,</code> <code>n MOD m</code>	Returns the remainder of N divided by M. <code>MOD(N,0)</code> returns NULL.
<code>POWER(x, y)</code>	This is a synonym for <code>POW()</code> .
<code>RAND()</code>	Returns a random floating-point value
<code>ROUND(x)</code> <code>ROUND(x, d)</code>	Rounds the argument X to D decimal places. The rounding algorithm depends on the data type of X. D defaults to 0 if not specified. D can be negative to cause D digits left of the decimal point of the value X to become zero.
<code>TRUNCATE(x, d)</code>	Returns the number X, truncated to D decimal places. If D is 0, the result has no decimal point or fractional part. D can be negative to cause D digits left of the decimal point of the value X to become zero.

## Note:

- TABLE statement always displays all columns of the table.
- TABLE statement does not support any WHERE clause.
- TABLE statement can be used with temporary tables.

`TABLE tbl_name [ORDER BY col_name] [LIMIT number [OFFSET number]]`

## table statement...

TABLE is a DML statement introduced in MySQL 8.0.19 which returns rows and columns of the named table.

## Remember:

- Here, "\*" is known as metacharacter (all columns)

# select statement... syntax

SELECT is used to retrieve rows selected from one or more tables (using JOINS), and can include UNION statements and SUBQUERIES.



# *syntax*

## modifiers

# *select statement*

**SELECT** [ALL / DISTINCT / DISTINCTROW] *identifier.\* / identifier.A<sub>1</sub>* [ [as] alias\_name], *identifier.A<sub>2</sub>* [ [as] alias\_name], *identifier.A<sub>3</sub>* [ [as] alias\_name], expression1 [ [as] alias\_name], expression2 [ [as] alias\_name] . . .

- [ **FROM** <*identifier.r<sub>1</sub>*> [as] alias\_name], <*identifier.r<sub>2</sub>*> [as] alias\_name], . . . ]
- [ **WHERE** < where\_condition1 > { **and** | **or** } < where\_condition2 > . . . ]
- [ **GROUP BY** < { col\_name | expr | position }, . . . > ]
- [ **HAVING** < having\_condition1 > { **and** | **or** } < having\_condition2 > . . . ]
- [ **ORDER BY** < { col\_name | expr | position } [ ASC | DESC ], . . . > ]
- [ **LIMIT** < { [offset,] row\_count | row\_count OFFSET offset } > ]
- [ { **INTO OUTFILE** '*file\_name*' | **INTO DUMPFILE** '*file\_name*' | **INTO** var\_name [, var\_name], . . . } ]

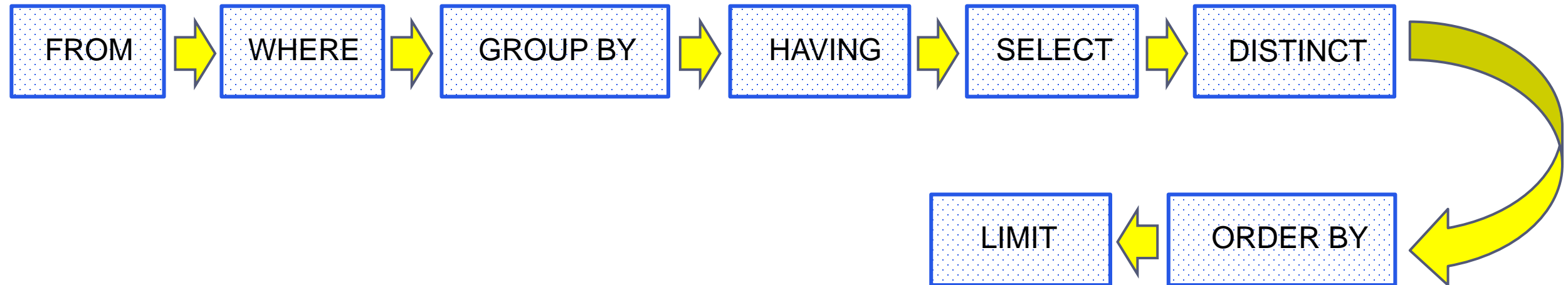
## Remember:

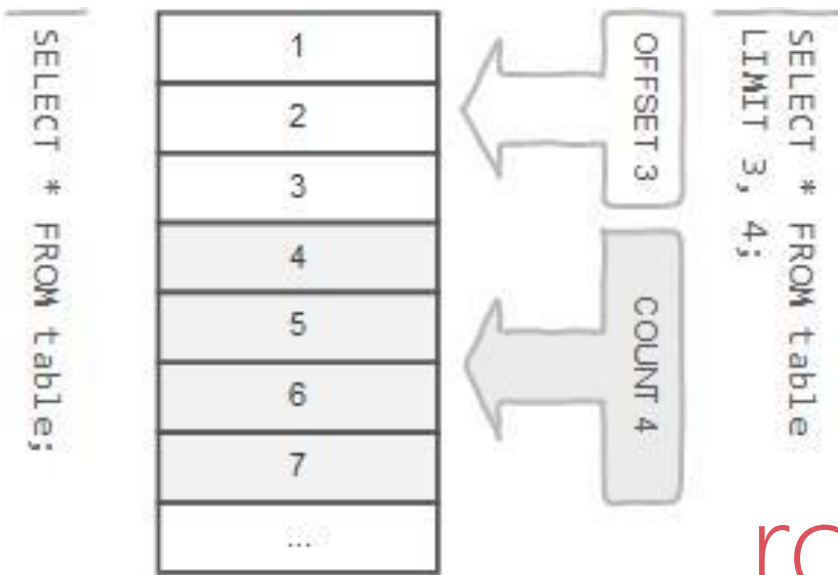
- **ALL** (modifier is default) specifies that all matching rows should be returned, including duplicates.
- **DISTINCT** (modifier) specifies removal of duplicate rows from the result set.
- **DISTINCTROW** (modifier) is a synonym for **DISTINCT**.
- It is an error to specify both modifiers.
- Whenever you use **DISTINCT**, sorting takes place in server.

# sequence of clauses



# select statement... execution





## row limiting clause

LIMIT is applied after HAVING

### Remember:

- LIMIT enables you to pull a section of rows from the middle of a result set. Specify two values: The number of rows to skip at the beginning of the result set, and the number of rows to return.

---

### Note:

- Limit value are **not** to be given within ( . . . )
  - Limit takes one or two numeric arguments, which must both be **non-negative** integer value.
-

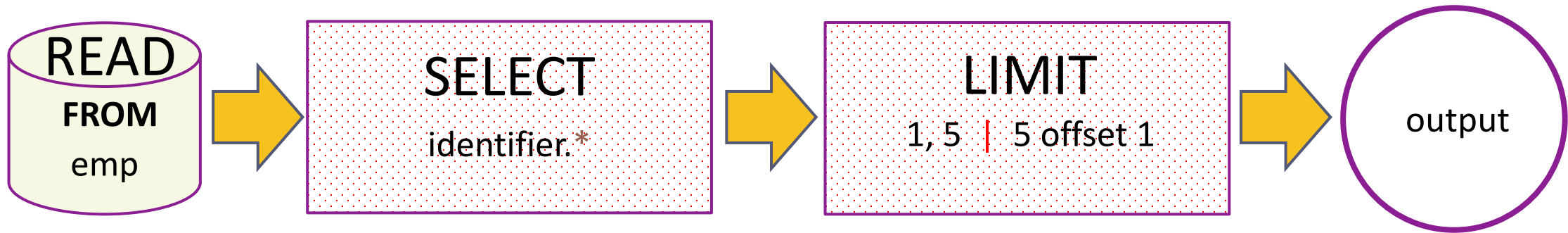
## select - limit

SELECT  $A_1, A_2, A_3, \dots$  FROM  $r$

[ LIMIT { [offset,] row\_count | row\_count OFFSET offset } ]

You can specify an offset using OFFSET from where SELECT will start returning records. By default *offset is zero*.

- SELECT \* FROM emp LIMIT 5 OFFSET 1;



- SELECT \* FROM student LIMIT 5;
- SELECT \* FROM student LIMIT 1, 5;
- SELECT \* FROM student LIMIT 5 offset 1;
- SELECT RAND(), student.\* FROM student ORDER BY 1 LIMIT 1;
- SELECT student.\* FROM student ORDER BY RAND() LIMIT 1;



Nulls by default occur at the top, but you can use *IsNull* to assign default values, that will put it in the position you require. . The *ISNULL()* function tests whether an expression is NULL. If expression is a NULL value, the *ISNULL()* function returns 1. Otherwise, it returns 0.

- `SELECT id AS 'a' FROM tbl_name ORDER BY `a`;`
- `SELECT id AS 'a' FROM tbl_name ORDER BY 'a';`

## order by clause

SQL allows the user to order the tuples in the result of a query by the values of one or more of the attributes that appear in the query result, by using the ORDER BY clause.

### Remember:

- The default sort order is ascending **ASC**, with smallest values first. To sort in descending (reverse) order, add the **DESC** keyword to the name of the column you are sorting by.
- You can sort on multiple columns, and you can sort different columns in different directions.
- If the **ASC** or **DESC** modifier is not provided in the ORDER BY clause, the results will be sorted by expression in **ASC** (ascending) order. This is equivalent to ORDER BY expression ASC.

## *select - order by*

When doing an ORDER BY, NULL values are placed **first** if you do ORDER BY ... ASC and **last** if you do ORDER BY ... DESC.

```
SELECT  $A_1$ ,  $A_2$ ,  $A_3$ ,  $A_n$  FROM  $r$ 
```

```
[ORDER BY { $A_1$ ,  $A_2$ ,  $A_3$ , ... | expr | position} [ASC | DESC] , ... ]
```

"Ordered by attributes  $A_1$ ,  $A_2$ ,  $A_3$  ..."

- Tuples are sorted by specified attributes
- Results are sorted by  $A_1$  first
- Within each value of  $A_1$ , results are sorted by  $A_2$  then within each value of  $A_2$ , results are sorted by  $A_3$

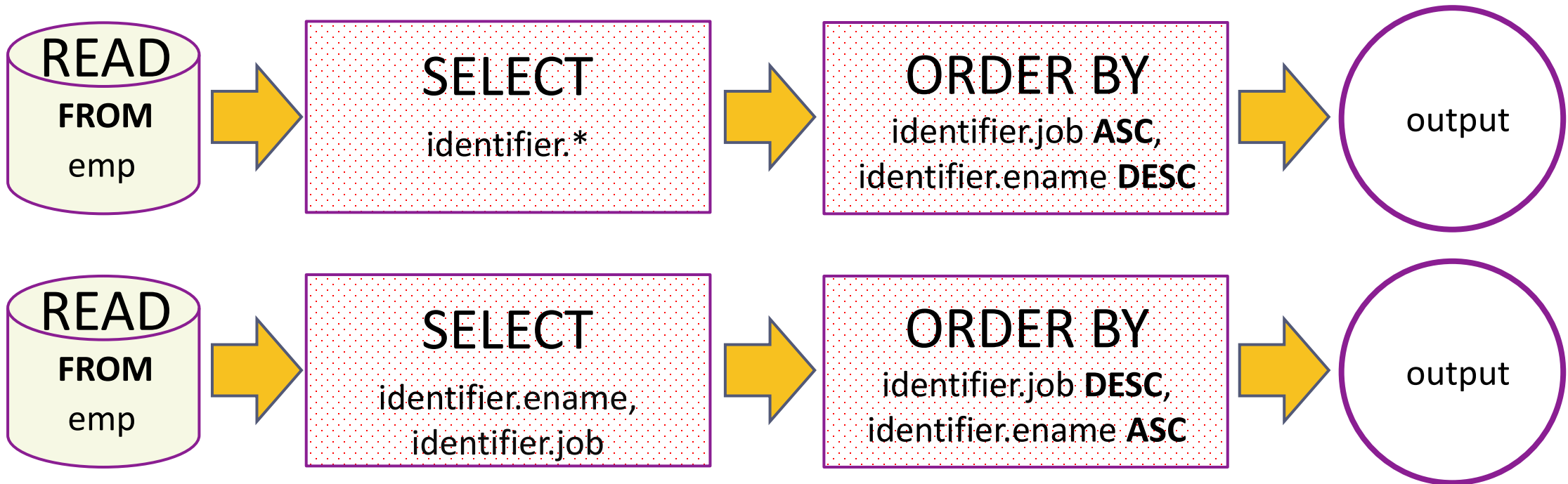
- `SELECT * FROM  $r$  ORDER BY  $key\_part1$ ,  $key\_part2$ ;` `// optimizer does not use the index.`
- `SELECT  $key\_part1$ ,  $key\_part2$  FROM  $r$  ORDER BY  $key\_part1$ ,  $key\_part2$ ;` `// optimizer uses the index.`

## *select - order by*

The **ORDER BY** clause is used to sort the records in your result set.

**SELECT**  $A_1, A_2, A_3, A_n$  **FROM**  $r$

[**ORDER BY** { $A_1, A_2, A_3, \dots$  |  $\text{expr}$  |  $\text{position}$ } [**ASC** | **DESC**] , ... ]



## *select - order by*

When doing an ORDER BY, NULL values are presented **first** if you do ORDER BY ... ASC

- **SELECT \* FROM emp ORDER BY comm ASC;**

[illegible]

## *select - order by*

When doing an ORDER BY, NULL values are presented **last** if you do ORDER BY ... DESC.

- **SELECT \* FROM emp ORDER BY comm DESC;**

[illegible]

SELECT  $A_1, A_2, A_3, A_n$  FROM  $r$

*select - order by*

[ORDER BY { $A_1, A_2, A_3, \dots$  | expr | position} [ASC | DESC] , ... ]

- SELECT \* FROM emp ORDER BY comm;
- SELECT \* FROM emp ORDER BY comm IS NULL ;
- SELECT \* FROM emp ORDER BY comm IS NOT NULL ;
- SELECT \* FROM emp ORDER BY 1 + 1;
- SELECT \* FROM emp ORDER BY True;
- SELECT sal FROM emp ORDER BY -sal;
- SELECT ename, LENGTH(ename) FROM emp ORDER BY LENGTH(ename), ename DESC ;
- SELECT \* FROM emp ORDER BY IF(job = 'manager', 3, IF(job = 'salesman', 2, NULL)) ;
- SELECT \* FROM emp ORDER BY FIELD(job, 'manager', 'salesman') ;
- SELECT \* FROM emp ORDER BY ISNULL(comm), comm ;
- SELECT ename `e` FROM emp ORDER BY `e` ;
- SELECT ename `e` FROM emp ORDER BY e ;
- SELECT ename 'e' FROM emp ORDER BY 'e' ;
- SELECT \* FROM emp ORDER BY CASE WHEN ename='sharmin' THEN 0 ELSE 1 END, ename;

## Remember:

In **WHERE** clause operations can be performed using...

- *CONSTANTS*
- *TABLE columns*
- *FUNCTION calls (PRE-DEFINED / UDF)*

\* In SQL, a logical expression is often called a *predicate*.

# where clause

The WHERE Clause is used when you want to retrieve specific information from a table excluding other irrelevant data.

## Note:

**Expressions in WHERE clause can use.**

- *Arithmetic operators*
- *Comparison operators*
- *Logical operators*

## Note:

- All comparisons return FALSE when either argument is NULL, so no rows are ever selected.

## select - where

We can use a conditional clause called WHERE clause to filter out results. Using WHERE clause, we can specify a selection criteria to select required records from a table.

SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1, r_2, r_3, \dots$  [ WHERE  $P$  ]

- ❖  $r_i$  are the relations (tables)
- ❖  $A_i$  are attributes (columns)
- ❖  $P$  is the selection predicate

SQL permits us to use the notation  $(v_1, v_2, \dots, v_n)$  to denote a tuple of arity (attribute)  $n$  containing values  $v_1, v_2, \dots, v_n$ .

WHERE  $(a_1, a_2) \leq (b_1, b_2)$

WHERE (EMP.DEPTNO, DNAME) = (DEPT.DEPTNO, 'SALES');

### Remember:

- A **predicate** is a condition expression that evaluates to a boolean value, either **true** or **false**.
- **Predicates** can be used as follows: In a SELECT statement's **WHERE** clause or **HAVING** clause to determine which rows are relevant to a particular query.

A value of **zero** is considered **false**. **Nonzero** values are considered **true**.

- SELECT true, false, TRUE, FALSE, True, False;



# select - where

SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1, r_2, r_3, \dots$  [ WHERE  $P$  ]

## 2. comparison\_operator:

= | <=> | >= | > | <= | < | <> | !=

## 5. logical\_operators

{ AND | && } | { OR | || }

What will be the result of the query below?

WHERE state = 'NY' OR 'CA' --Illegal

WHERE salary > 20000 AND < 30000 --Illegal

WHERE state NOT = 'CA' --Illegal

- SELECT 1 = 1;
- SELECT True = 1;
- SELECT True = 2;
- SELECT True = True;
- SELECT 0 = 0;
- SELECT False = False;
- SELECT False = 1;
- SELECT 'a' = 1;
- SELECT 'a' = 0;
- SELECT \* FROM emp WHERE ename = 0;
- SELECT \* FROM emp WHERE ename = 1;
- SELECT \* FROM emp WHERE ename = False;
- SELECT \* FROM emp WHERE ename = True;
- SELECT \* FROM emp WHERE True AND False;
- SELECT \* FROM emp WHERE True OR False;
- SELECT \* FROM emp WHERE True AND 1;
- SELECT \* FROM emp WHERE True OR 0;

**Note:**

**AND** has higher precedence than **OR**.

- EXPLAIN ANALYZE SELECT \* FROM emp WHERE job = 'salesman' OR job = 'manager' AND sal > 2000;

# *select - where*

SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1, r_2, r_3, \dots$  [ WHERE  $P$  ]

Logical Operators		
AND, &&	Logical AND	e.g. SELECT 1 AND 1; / SELECT 1 AND 0; SELECT 0 AND NULL; / SELECT NULL AND 0; SELECT 1 AND NULL; / SELECT NULL AND 1;
OR,	Logical OR	e.g. SELECT 1 OR 1; / SELECT 1 OR 0; SELECT 0 OR NULL; / SELECT NULL OR 0; SELECT 1 OR NULL; / SELECT NULL OR 1;
NOT, !	Negates value	e.g. SELECT NOT 1;

- **Logical AND.** Evaluates to 1 if all operands are non-zero and not NULL, to 0 if one or more operands are 0, otherwise NULL is returned.
- **Logical OR.** When both operands are non-NULL, the result is 1 if any operand is nonzero, and 0 otherwise. With a NULL operand, the result is 1 if the other operand is nonzero, and NULL otherwise. If both operands are NULL, the result is NULL.
- **Logical NOT.** Evaluates to 1 if the operand is 0, to 0 if the operand is nonzero, and NOT NULL returns NULL.

# *select - where*

SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1, r_2, r_3, \dots$  [ WHERE  $P$  ]

## Comparison Functions and Operators

LEAST(value1, value2, ...)	With two or more arguments, returns the smallest argument.
GREATEST(value1, value2, ...)	With two or more arguments, returns the largest argument.
(expr, [expr] ...)	Multiple columns in expr. (sub-query returning multiple columns to compare)
COALESCE(value, ...)	Returns the first non-NULL value in the list, or NULL if there are no non-NULL values.

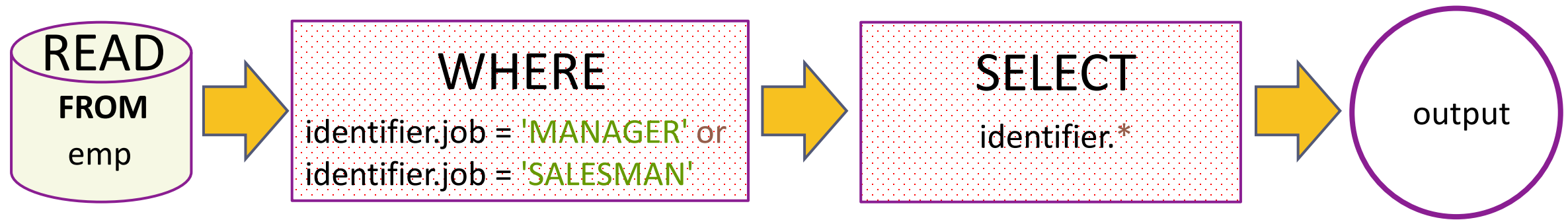
- SELECT GREATEST(10, 20, 30),      # 30  
          LEAST(10, 20, 30);        # 10
- SELECT GREATEST(10, null, 30),    # null  
          LEAST(10, NULL, 30);      # null
- SELECT \* FROM emp WHERE (deptno, pwd) = (SELECT deptno, pwd FROM dept WHERE deptno = 30);

## Remember:

- If any argument is NULL, the both functions return NULLs immediately without doing any comparison..

## select - where

- SELECT \* FROM emp WHERE job = 'MANAGER' OR job = 'SALESMAN';



	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	BONUSID	USER NAME	PWD	phone	isActive
▶	7499	ALLEN	SALESMAN	7698	1981-02-20	1600	300	30	4	ALWAYS TESTE	sales@2017	7032300096	1
	7521	WARD	SALESMAN	7698	1981-02-22	1250	500	30	1	WARD	sales@2017	7132300034	1
	7566	JONES	MANAGER	7839	1981-04-02	2975	NULL	20	4	HONEYCOMB	a12recmpm	7132300039	1
	7654	MARTIN	SALESMAN	7698	1981-09-28	1250	1400	30	6	LIFE RACER	sales@2017	7132300050	1
	7698	BLAKE	MANAGER	7839	1981-05-01	2850	NULL	30	1	BIG BEN	sales@2017	7132300027	1
	7782	CLARK	MANAGER	7839	1981-06-09	2450	NULL	10	3	CLARK	r50mpm	7032300001	1
	7844	TURNER	SALESMAN	7698	1981-09-08	1500	0	30	5	SAND DUST	sales@2017	NULL	1
	7919	HOFFMAN	MANAGER	7566	1982-03-24	4150	NULL	30	3	INTERVAL	sales@2017	NULL	1

# combining and & or - where

## Note:

**AND** has higher precedence than **OR**.

- `SELECT * FROM emp WHERE ename = 'saleel' AND city = 'pune' OR city = 'baroda';`
- `SELECT * FROM emp WHERE ename = 'saleel' AND (city = 'pune' OR city = 'baroda');`
- `SELECT ename, job, comm FROM emp WHERE comm = 0 OR comm IS NULL AND job = 'CLERK';`
- `SELECT ename, job, comm FROM emp WHERE (comm = 0 OR comm IS NULL) AND job = 'CLERK';`
- `EXPLAIN ANALYZE SELECT * FROM emp WHERE job = 'salesman' OR job = 'manager' AND sal > 2000;`

## *select - where*

SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1, r_2, r_3, \dots$  [ WHERE  $P$  ]

What will be the output of the following statement?

- SELECT "Hello" # "World " FROM dual;
- SELECT 10 + 10 as Result FROM dual WHERE False;
- SELECT 10 + 10 as Result FROM dual WHERE True;
- SELECT 10 + 10 as Result FROM dual WHERE 10 - 10;
- SELECT 10 + 10 as Result FROM dual WHERE 10 - 0;
- SELECT 10 + 10 as Result FROM dual WHERE 10 - 30;
- SELECT '5' \* '5' as Result FROM dual;
- SELECT 5 \* 5 - '-5' as Result FROM dual;

- `SELECT * FROM emp WHERE comm IS UNKNOWN;`
- `SELECT * FROM emp WHERE comm IS NOT UNKNOWN;`

- *operand* `IS [NOT] NULL`

### 3. *boolean\_predicate*:

`IS [NOT] NULL`  
| *expr <=> null*

is null / is not null

- "IS NULL" is the keyword that performs the Boolean comparison. It returns true if the supplied value is NULL and false if the supplied value is not NULL.
- "IS NOT NULL" is the keyword that performs the Boolean comparison. It returns true if the supplied value is not NULL and false if the supplied value is null.
- SQL uses a three-valued logic: besides true and false, the result of logical expressions can also be unknown. SQL's three valued logic is a consequence of supporting null to mark absent data.

#### Note:

- `IS UNKNOWN` is synonym of `IS NULL`.
- `IS NOT UNKNOWN` is synonym of `IS NOT NULL`.

## Remember:

`SELECT * FROM emp WHERE comm = NULL;` # will return Empty set

- `SELECT empno, ename, job, sal, comm FROM emp WHERE comm IS NOT NULL;`
- `SELECT empno, ename, job, sal, comm FROM emp WHERE comm IS NOT UNKNOWN;`
- `SELECT empno, ename, job, sal, comm FROM emp WHERE comm is TRUE;`

*is null / is not null*

	empno	ename	job	sal	comm
▶	7499	ALLEN	SALESMAN	1600.00	300.00
	7521	WARD	SALESMAN	1250.00	500.00
	7654	MARTIN	SALESMAN	1250.00	1400.00
	7844	TURNER	SALESMAN	1500.00	0.00
	7920	GRASS	SALESMAN	2575.00	2700.00
	7945	AARUSH	SALESMAN	1350.00	2700.00
	7949	ALEX	MANAGER	1250.00	500.00

	empno	ename	job	sal	comm
▶	7499	ALLEN	SALESMAN	1600.00	300.00
	7521	WARD	SALESMAN	1250.00	500.00
	7654	MARTIN	SALESMAN	1250.00	1400.00
	7920	GRASS	SALESMAN	2575.00	2700.00
	7945	AARUSH	SALESMAN	1350.00	2700.00
	7949	ALEX	MANAGER	1250.00	500.00



## select – boolean

- BOOL and BOOLEAN are synonym of TINYINT(1)

A value of **zero** is considered **false**. **non-zero** values are considered **true**.

`SELECT true, false, TRUE, FALSE, True, False;`

- `SELECT * FROM tasks WHERE completed;` ← - - - - -
- `SELECT * FROM tasks WHERE completed is True;` - - - - -
- `SELECT * FROM tasks WHERE completed = 1;` ← - - - - -
- `SELECT * FROM tasks WHERE completed = True;` - - - - -

	id	title	completed
▶	2	Task2	1
	4	Task4	1
	8	Task8	1
	9	Task9	12
	10	Task10	58
	11	Task11	1
	13	Task13	1
•	NULL	NULL	NULL

	id	title	completed
▶	2	Task2	1
	4	Task4	1
	8	Task8	1
	11	Task11	1
	13	Task13	1
•	NULL	NULL	NULL

- `SELECT * FROM tasks WHERE NOT completed;` ← - - - - -
- `SELECT * FROM tasks WHERE completed is False;` - - - - -
- `SELECT * FROM tasks WHERE completed = 0;` - - - - -
- `SELECT * FROM tasks WHERE completed = False;` ← - - - - -

	id	title	completed
▶	1	Task1	0
	3	Task3	0
	7	Task7	0
	12	Task12	0
•	NULL	NULL	NULL

## *select – boolean*

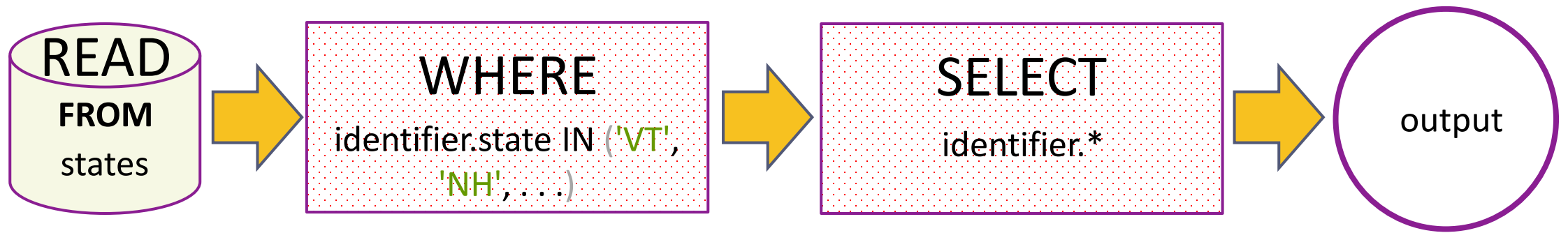
- BOOL and BOOLEAN are synonym of TINYINT(1)

A value of **zero** is considered **false**. **Nonzero** values are considered **true**.

```
SELECT true, false, TRUE, FALSE, True, False;
```

What will be the result of the query below?

- `SELECT * FROM emp WHERE 1;`
- `SELECT * FROM emp WHERE True;`
- `SELECT * FROM emp WHERE 0;`
- `SELECT * FROM emp WHERE False;`
- `SELECT * FROM emp WHERE ename = '' OR 0;`
- `SELECT * FROM emp WHERE ename = '' OR 1;`
- `SELECT * FROM emp WHERE ename = '' OR 1 = 1;`
- `SELECT * FROM emp WHERE ename = 'smith' OR True;`
- `SELECT * FROM emp WHERE ename = 'smith' AND True;`
- `SELECT * FROM emp WHERE ename IN('smith', True);`
- `SELECT * FROM emp WHERE ename = 'smith' OR False;`
- `SELECT * FROM emp WHERE ename = 'smith' AND False;`
- `SELECT * FROM emp WHERE ename IN('smith', False);`



#### 4. *predicate:*

*expr* [NOT] IN (*expr1*, *expr2*, ...) in  
| *expr* [NOT] IN (*subquery*)

The IN statement is used in a WHERE clause to choose items from a set. The IN operator allows you to determine if a specified value matches any value in a set of values or value returned by a subquery.

```
SELECT ... FROM  $r_1$  WHERE (  
  state = 'VT' OR  
  state = 'NH' OR  
  state = 'ME' OR  
  state = 'MA' OR  
  state = 'CT' OR  
  state = 'RI'  
);
```



- SELECT ... FROM  $r_1$  WHERE state IN ('VT', 'NH', 'ME', 'MA', 'CT', 'RI');
- SELECT ... FROM  $r_1$  WHERE state IN (SELECT ... );

A IN (B1, B2, B3, etc.)    A is found in the list (B1, B2, etc.)

## syntax

column | expression IN ( v1, v2, v3, . . . )

column | expression IN (subquery)

## Remember:

- If a value in the column or the expression is equal to any value in the list, the result of the IN operator is TRUE.
- The IN operator is equivalent to multiple **OR** operators.
- To negate the IN operator, you use the **NOT IN** operator.

- 
- **SELECT** empno, ename, job, hiredate, sal, comm, deptno, isactive **FROM** emp **WHERE** job **IN** ('salesman', 'manager');

	empno	ename	job	hiredate	sal	comm	deptno	isactive
▶	7499	ALLEN	SALESMAN	1981-02-20	1600.00	300.00	30	1
	7521	WARD	SALESMAN	1981-02-22	1250.00	500.00	30	1
	7566	JONES	MANAGER	1981-04-02	2975.00	NULL	20	1
	7654	MARTIN	SALESMAN	1981-09-28	1250.00	1400.00	30	1
	7698	BLAKE	MANAGER	1981-05-01	2850.00	NULL	30	1
	7782	CLARK	MANAGER	1981-06-09	2450.00	NULL	10	1
	7844	TURNER	SALESMAN	1981-09-08	1500.00	0.00	30	1
	7919	HOFFMAN	MANAGER	1982-03-24	4150.00	NULL	30	1

## Problem with NOT IN:

*not in*

*a*

c1	c2
1	1
2	1
3	1
4	1
5	1

*b*

c1	c2
1	7
NULL	7
3	7

- `SELECT * FROM a WHERE c1 NOT IN(1, 2, NULL);`
- `SELECT * FROM a WHERE c1 NOT IN(SELECT c1 FROM b );`  
**Empty set (0.00 sec)**

"color NOT IN (Red, Blue, NULL)" This is equivalent to: "`NOT`(color=Red OR color=Blue OR color=NULL)"

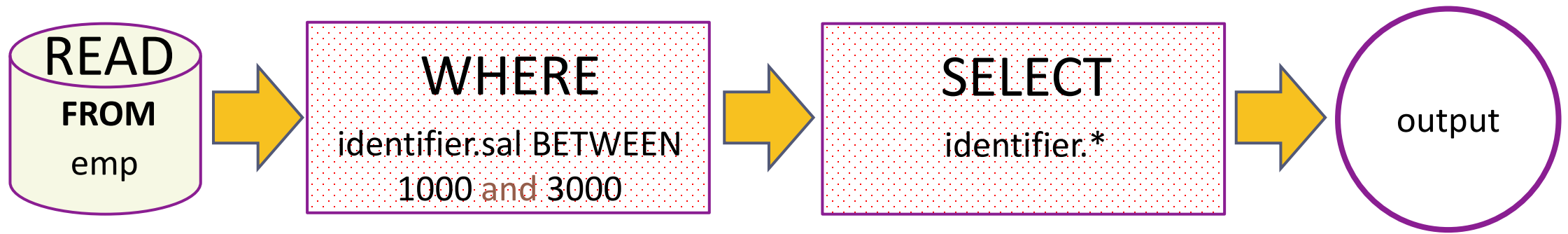
## Remember:

*in*

- On the left side of the IN() predicate, the row constructor contains only column references.
- On the right side of the IN() predicate, there is more than one row constructor.

What will be the result of the query below?

- `SELECT * FROM emp WHERE deptno IN (10);`
- `SELECT * FROM emp WHERE deptno IN (10, 20);`
- `SELECT * FROM emp WHERE False IN (10, 20, 0);`
- `SELECT * FROM emp WHERE True IN (10, 20, 1);`
- `SELECT * FROM emp WHERE 10 IN (10, 20);`
- `SELECT * FROM emp WHERE 7788 IN (empno, mgr);` ←
- `SELECT * FROM emp WHERE 1 IN (10, 20, True, False);`
- `SELECT * FROM emp WHERE deptno IN (10, 20) OR True;`
- `SELECT * FROM emp WHERE deptno IN (10, 20) AND True;`
- `SELECT * FROM emp WHERE deptno IN (SELECT deptno FROM dept);`
- `SELECT * FROM emp WHERE deptno IN (SELECT deptno FROM dept WHERE dname='accounting');`
- `SELECT * FROM emp WHERE deptno IN (TABLE deptno);` # ERROR 1241 (21000): Operand should contain 1 column(s)
- `SELECT * FROM emp WHERE 1 NOT IN (NULL);`
- `SELECT * FROM emp WHERE deptno IN (10, 20);`
- `SELECT * FROM emp WHERE False IN (10, 20, 0);`
- `SELECT * FROM emp WHERE True IN (10, 20, 1);`
- `SELECT * FROM emp WHERE 10 IN (10, 20);`



#### 4. *predicate:*

*expr* [NOT] BETWEEN *expr1* AND *expr2*

between

The BETWEEN operator is a logical operator that allows you to specify a range to test.

A BETWEEN B AND C    A is between B and C

# between

## syntax

WHERE salary BETWEEN ( 20000 AND 30000 ) – Illegal

column | expression BETWEEN start\_expression AND end\_expression

## Remember:

- The BETWEEN operator returns TRUE if the expression to test is greater than or equal to the value of the start\_expression and less than or equal to the value of the end\_expression.
  - You can use the greater than or equal to ( $\geq$ ) and less than or equal to ( $\leq$ ) to substitute the BETWEEN operator.
- 

## Note:

- if any input to the BETWEEN or NOT BETWEEN is NULL, then the result is UNKNOWN.

e.g.

SELET empno, ename, job, hiredate, sal, comm, deptno, isactive FROM emp WHERE sal BETWEEN 1000 AND NULL;

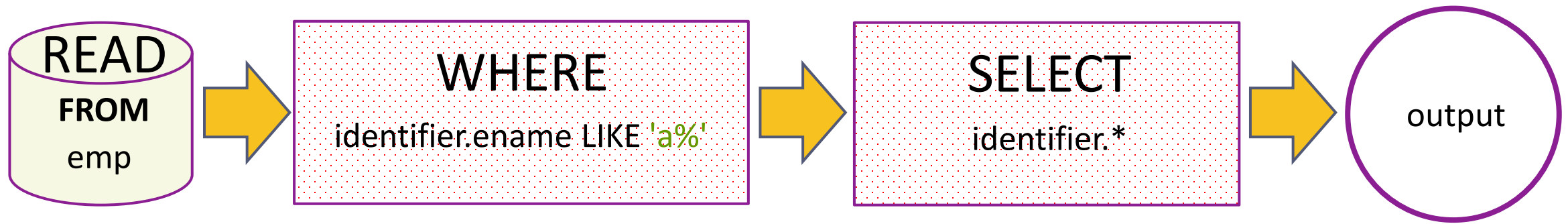
- 
- SELECT \* FROM salespeople WHERE FORMAT(comm, 2) > 0.1 AND FORMAT(comm, 2) < 0.26;



## between

- `SELECT empno, ename, job, hiredate, sal, comm, deptno, isactive FROM emp WHERE sal BETWEEN 1000 AND 3000;`

	empno	ename	job	hiredate	sal	comm	deptno	isactive
▶	7421	THOMAS	CLERK	1981-07-19	1750.00	NULL	10	0
	7499	ALLEN	SALESMAN	1981-02-20	1600.00	300.00	30	1
	7521	WARD	SALESMAN	1981-02-22	1250.00	500.00	30	1
	7566	JONES	MANAGER	1981-04-02	2975.00	NULL	20	1
	7654	MARTIN	SALESMAN	1981-09-28	1250.00	1400.00	30	1
	7698	BLAKE	MANAGER	1981-05-01	2850.00	NULL	30	1
	7782	CLARK	MANAGER	1981-06-09	2450.00	NULL	10	1
	7788	SCOTT	ANALYST	1982-12-09	3000.00	NULL	20	1
	7844	TURNER	SALESMAN	1981-09-08	1500.00	0.00	30	1
	7876	ADAMS	CLERK	1983-01-12	1100.00	NULL	20	1
	7902	FORD	ANALYST	1981-12-03	3000.00	NULL	20	0
	7920	GRASS	SALESMAN	1980-02-14	2575.00	2700.00	30	1



#### 4. *predicate:*

*expr* [NOT] LIKE *expr* [ESCAPE *char*]

like

The LIKE operator is a logical operator that tests whether a string contains a specified pattern or not.

# *like - string comparison functions*

## *syntax*

column | expression LIKE 'pattern' [ESCAPE escape\_character]

## Remember:

- % matches any number of characters, even zero characters.
  - \_ matches exactly one character.
  - If we use default escape character '\', then don't use ESCAPE keyword.
- 

## Note:

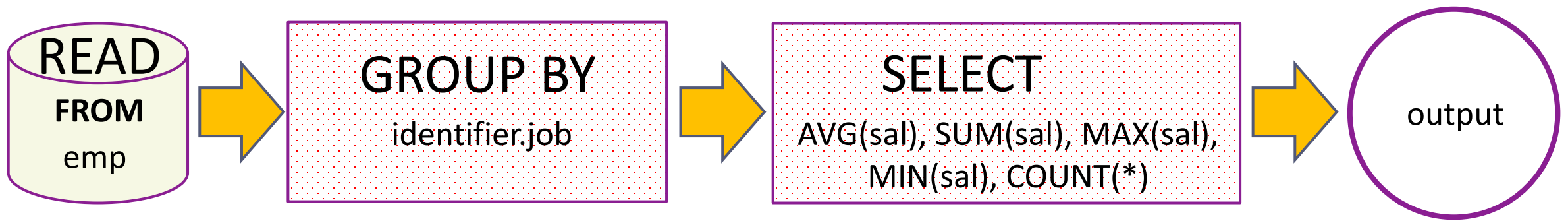
- The ESCAPE keyword is used to escape pattern matching characters such as the (%) percentage and underscore (\_) if they form part of the data.
  - If you do not specify the ESCAPE character, \ is assumed.
-

- `SELECT empno, ename, job, hiredate, sal, comm, deptno, isactive FROM emp WHERE ename LIKE 'a%';`

	empno	ename	job	hiredate	sal	comm	deptno	isactive
▶	7415	AARAV	CLERK	1981-12-31	3350.00	NULL	10	0
	7499	ALLEN	SALESMAN	1981-02-20	1600.00	300.00	30	1
	7876	ADAMS	CLERK	1983-01-12	1100.00	NULL	20	1
	7945	AARUSH	SALESMAN	1980-02-14	1350.00	2700.00	30	0
	7949	ALEX	MANAGER	1982-01-24	1250.00	500.00	30	1

What will be the result of the query below?

- `SELECT * FROM emp WHERE ename LIKE 's%';`
- `SELECT * FROM emp WHERE 'saleel' LIKE 's%';`
- `SELECT * FROM emp WHERE True LIKE '1';`
- `SELECT * FROM emp WHERE True LIKE '1%';`
- `SELECT * FROM emp WHERE True LIKE 001;`
- `SELECT * FROM emp WHERE True LIKE 100;`
- `SELECT * FROM emp WHERE False LIKE 100 OR 0;`
- `SELECT * FROM emp WHERE False LIKE 0 AND 1;`



## aggregate functions

SUM, AVG, MAX, MIN, COUNT, and GROUP\_CONCAT

`SELECT . . . . . FROM table_name WHERE <condition> / GROUP BY column_name`



this is invalid



`SUM(colNM) / AVG(colNM) / MAX(colNM)`  
`MIN(colNM) / COUNT(*) / COUNT(colNM)`

- `SET SQL_MODE = '';`
- `SET SQL_MODE = IGNORE_SPACE;`

### Remember:

None of the below two queries get executed unsuccessfully. The reason is that a condition in a WHERE clause cannot contain any aggregate function (or group function) without a subquery!

- `SELECT empno, ename, sal, deptno FROM emp WHERE sal = MAX(sal); #error`
- `SELECT empno, ename, sal, deptno FROM emp WHERE MAX(sal) = sal; #error`

# aggregate functions

## Remember:

**There are 3 places where aggregate functions can appear in a query.**

- in the **SELECT-LIST/FIELD-LIST** (the items before the FROM clause).
- in the **ORDER BY** clause.
- in the **HAVING** clause.

## Note:

- The aggregate functions allow you to perform the calculation of a set of rows and **return a *single* value**.
- The **WHERE** clause cannot refer to aggregate functions. **e.g. WHERE SUM(sal) = 5000** # Invalid, Error
- The **HAVING** clause can refer to aggregate functions. **e.g. HAVING SUM(sal) = 5000** # Valid, No Error
- Nesting of aggregate functions are not allowed.

**e.g.**

```
SELECT MAX(COUNT(*)) FROM emp GROUP BY deptno;
```

- Blank space between aggregate functions like (**SUM, MIN, MAX, COUNT**) are not allowed.

**e.g.**

```
SELECT SUM (sal) FROM emp;
```

- The GROUP BY clause is often used with an aggregate function to perform calculation and **return a single value for each subgroup**.
- To eliminate duplicates before applying the aggregate function is available by including the keyword **DISTINCT**.

## Things to... Remember:

## *aggregate functions*

### TODO

**AVG**([**DISTINCT**] *expr*)

- If there are no matching rows, **AVG()** **returns NULL**.
- **AVG()** may take a numeric argument, and it returns a average of non-NULL values.

e.g.

- **SELECT** **AVG**(1) "R1";
- **SELECT** **AVG**(**NULL**) "R1";
- **SELECT** **AVG**(1) "R1" **WHERE** **True**;
- **SELECT** **AVG**(1) "R1" **WHERE** **False**;
- **SELECT** **AVG**(1) "R1" **FROM** emp;
- **SELECT** **AVG**(sal) "R1" **FROM** emp **WHERE** empno = -1;
- **SELECT** **AVG**(sal) "Avg Salary" **FROM** emp;
- **SELECT** job, **AVG**(sal) "Avg Salary" **FROM** emp **GROUP BY** job;



## Things to... Remember:

## *aggregate functions*

### TODO

`SUM([DISTINCT] expr)`

- If there are no matching rows, SUM() **returns NULL**.
- SUM() may take a numeric argument , and it returns a sum of non-NULL values.

e.g.

- `SELECT SUM(1) "R1";`
- `SELECT SUM(NULL) "R1";`
- `SELECT SUM(2 + 2 * 2);`
- `SELECT SUM(1) "R1" WHERE True;`
- `SELECT SUM(1) "R1" WHERE False;`
- `SELECT SUM(1) "R1" FROM emp;`
- `SELECT SUM(sal) "R1" FROM emp WHERE empno = -1;`
- `SELECT SUM(sal) "Total Salary" FROM emp;`
- `SELECT job, SUM(sal) "Total Salary" FROM emp GROUP BY job;`

## Things to... Remember:

## *aggregate functions*

### TODO

`SUM([DISTINCT] expr)`

- If there are no matching rows, SUM() **returns NULL**.
- SUM() may take a numeric argument , and it returns a sum of non-NULL values.

`r = { -2, 1, 2, -1, 3, -2, 1, 2, 1 }`

- `SELECT SUM(c1) "R1" FROM r;`
- `SELECT SUM(IF(c1 >= 0, c1, NULL)) FROM r;`
- `SELECT SUM(IF(c1 < 0, c1, NULL)) FROM r;`
- `SELECT custId, type, amount, CASE type WHEN 'd' THEN amount WHEN 'c' THEN amount * -1 END amount FROM transactions;`

## Things to... Remember:

## *aggregate functions*

### TODO

`MAX([DISTINCT] expr)`

- If there are no matching rows, MAX() **returns NULL**.
- MAX() may take a string, number, and date argument, and it returns a maximum of non-NULL values.

e.g.

- `SELECT MAX(1) "R1";`
- `SELECT MAX(NULL) "R1";`
- `SELECT MAX('VIKAS');`
- `SELECT MAX(1) "R1" WHERE True;`
- `SELECT MAX(1) "R1" WHERE False;`
- `SELECT MAX(1) "R1" FROM emp;`
- `SELECT MAX(sal) "R1" FROM emp WHERE empno = -1;`
- `SELECT MAX(sal) "Maximum Salary" FROM emp;`
- `SELECT job, MAX(sal) "Maximum Salary" FROM emp GROUP BY job;`

## TODO

`MIN([DISTINCT] expr)`

- If there are no matching rows, MIN() **returns NULL**.
- MIN() may take a string, number, and date argument, and it returns a minimum of non-NULL values.

e.g.

- `SELECT MIN(1) "R1";`
- `SELECT MIN(NULL) "R1";`
- `SELECT MIN(1) "R1" WHERE True;`
- `SELECT MIN(1) "R1" WHERE False;`
- `SELECT MIN(1) "R1" FROM emp;`
- `SELECT MIN(sal) "R1" FROM emp WHERE empno = -1;`
- `SELECT MIN(sal) "Minimum Salary" FROM emp;`
- `SELECT job, MIN(sal) "Minimum Salary" FROM emp GROUP BY job;`

### TODO

**COUNT**([**DISTINCT**] *expr*)

- If there are no matching rows, COUNT() **returns 0**.
- Returns a count of the number of non-NULL values.
- COUNT(\*) is somewhat different in that it returns a count of the number of rows retrieved, whether or not they contain NULL values.
- COUNT (\*) is a special implementation of the COUNT function that returns the count of all the rows in a specified table.
- COUNT (\*) also considers Nulls and duplicates.
- SQL does not allow the use of DISTINCT with COUNT (\*)

### Note:

- **COUNT (\*)**: Returns a number of rows in a table including duplicates rows and rows containing null values in any of the columns.
- **COUNT (EXP)**: Returns the number of non-null values in the column identified by expression.
- **COUNT (DISTINCT EXP)**: Returns the number of unique, non-null values in the column identified by expression.
- **COUNT (DISTINCT \*)**: **is illegal**.

## Things to... Remember:

## *aggregate functions*

### TODO

COUNT([DISTINCT] *expr*)

e.g.

- SELECT COUNT(\*) "R1";
- SELECT COUNT(NULL) "R1";
- SELECT COUNT(\*) "R1" WHERE True;
- SELECT COUNT(\*) "R1" WHERE False;
- SELECT COUNT(0) FROM emp;
- SELECT COUNT(1) FROM emp;
- SELECT COUNT(\*) FROM emp WHERE empno = -1;
- SELECT COUNT(comm) "R1" FROM emp;
- SELECT job, COUNT(\*) "R1" FROM emp GROUP BY job;

TODO

```
GROUP_CONCAT([DISTINCT] expr  
             [ORDER BY {unsigned_integer | col_name | expr} [ASC | DESC] [,col_name . . .]]  
             [SEPARATOR str_val])
```

e.g.

- `SELECT job, GROUP_CONCAT(ename) FROM emp GROUP BY job;`
- `SELECT deptno, GROUP_CONCAT(ename) FROM emp group BY deptno;`
- `SELECT job, CONCAT(GROUP_CONCAT(ename), ' (', COUNT(*), ')') FROM emp GROUP BY job;`
- `SELECT job, CONCAT(GROUP_CONCAT(sal), ' (', MAX(sal), ')') FROM emp GROUP BY job;`
- `SELECT job, CONCAT(GROUP_CONCAT(sal), ' (', SUM(sal), ')') FROM emp GROUP BY job;`

```
SELECT DISTINCT COUNT(JOB) FROM EMP
```

```
SELECT COUNT(DISTINCT JOB) FROM EMP
```



- `SET SQL_MODE = '';`
- `SET SQL_MODE = 'ONLY_FULL_GROUP_BY';`

$$G_{A_1, A_2, \dots, A_n} G_{F_1(A_1), F_2(A_2), \dots, F_m(A_m)} (r)$$

group by clause

### Remember:

- Standard SQL does not allow you to use an ALIAS in the GROUP BY clause, however, MySQL supports this.
- GROUP BY is used in conjunction with aggregating functions to group the results by the unaggregated columns.

### Note:

- DISTINCT (if used outside an aggregation function) that is superfluous.

e.g.

```
SELECT DISTINCT COUNT(ename) FROM emp;
```



## *select - group by*

- Columns selected for output can be referred to in ORDER BY and GROUP BY clauses using column names, column aliases, or column positions. Column positions are integers and begin with 1
- If you use GROUP BY, output rows are sorted according to the GROUP BY columns as if you had an ORDER BY for the same columns. To avoid the overhead of sorting that GROUP BY produces, add ORDER BY NULL.
- If a query includes GROUP BY but you want to avoid the overhead of sorting the result, you can suppress sorting by specifying ORDER BY NULL.

### For example:

- `SELECT job, COUNT(*) FROM emp GROUP BY job ORDER BY NULL;`
- `SELECT * FROM emp ORDER BY FIELD (job, 'MANAGER', 'SALESMAN');`

*This function's will produce a single value for an entire group or a table.*

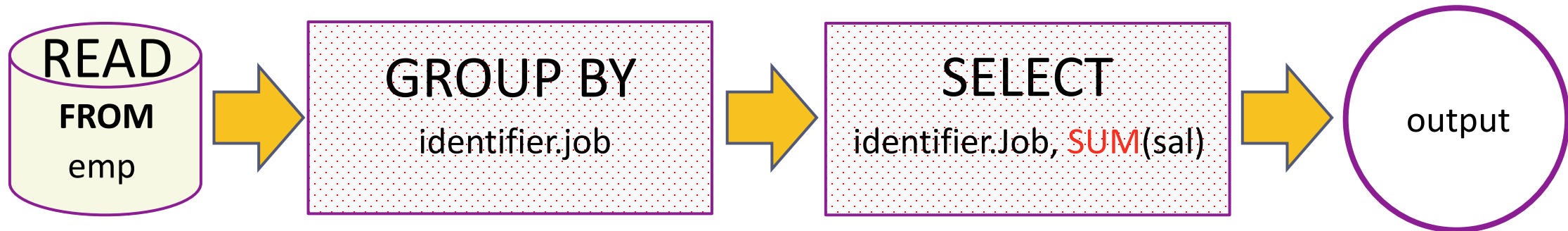
## select - group by

You can use GROUP BY to group values from a column, and, if you wish, perform calculations on that column.

SELECT  $G_1, G_2, \dots, F_1(A_1), F_2(A_2), \dots$  FROM  $r_1, r_2, \dots$

[GROUP BY { $G_1, G_2, \dots$  | expr | position}, ... [WITH ROLLUP]]

- SELECT job, SUM(sal) FROM emp GROUP BY job;
- SELECT job, SUM(sal) FROM emp GROUP BY job WITH ROLLUP;



	job	sum(sal)
▶	CLERK	9250
	SALESMAN	9525
	MANAGER	13675
	ANALYST	6000
	PRESIDENT	5000

	job	sum(sal)
▶	ANALYST	6000
	CLERK	9250
	MANAGER	13675
	PRESIDENT	5000
	SALESMAN	9525
	NULL	43450

## *select - group by*

- `SET SQL_MODE = '';`
- `SET SQL_MODE = 'ONLY_FULL_GROUP_BY';`

### Examples:

- `SELECT job, sal + 1001 FROM emp GROUP BY job;`
- `SELECT job, COUNT(job) FROM emp GROUP BY COUNT(job);` # error
- `SELECT job, sal + 1001 FROM emp GROUP BY sal + 1001;`
- `SELECT LENGTH(ename) R1 FROM emp GROUP BY R1;`
- `SELECT job, SUM(sal) FROM emp GROUP BY job WITH ROLLUP;`
- `SELECT COALESCE (job, 'Total'), SUM(sal) FROM emp GROUP BY job WITH ROLLUP;`

## Remember:

- The **WHERE** clause **cannot refer** to aggregate functions. [ **WHERE SUM**(sal) = 5000 # Error ]
- The **HAVING** clause **can refer** to aggregate functions. [ **HAVING SUM**(sal) = 5000 # No Error ]

# having clause

The MySQL **HAVING** clause is used in the SELECT statement to specify filter conditions for a group of rows. **HAVING** clause is often used with the GROUP BY clause. When using with the GROUP BY clause, we can apply a filter condition to the columns that appear in the GROUP BY clause.

## Note:

- Columns given in **HAVING** clause must be present in selection-list.

e.g.

1. **SELECT COUNT**(\*) **FROM** emp **HAVING** deptno=10; \*

2. **SELECT** deptno, **COUNT**(\*) **FROM** emp **GROUP BY** deptno **HAVING** job='manager'; \*

\* **ERROR: Unknown column '...'**  
in 'having clause'

- **HAVING** is merged with **WHERE** if you do not use GROUP BY or Aggregate Functions (COUNT(), . . .)

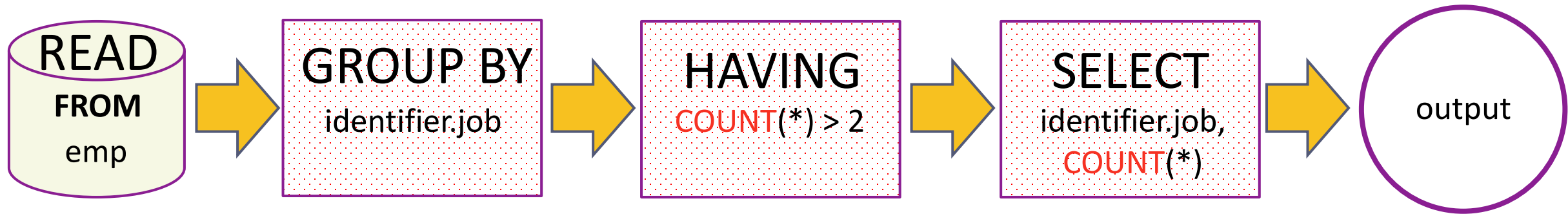
## *select - having*

SELECT  $G_1, G_2, \dots, F_1(A_1), F_2(A_2), \dots$  FROM  $r_1, r_2, \dots$

[GROUP BY { $G_1, G_2, \dots$  | *expr* | *position*}, ... [WITH ROLLUP]]

[HAVING *having\_condition* ]

- SELECT COUNT(\*), job FROM emp GROUP BY job HAVING COUNT(\*) > 2;



	count(*)	job
▶	6	CLERK
	6	SALESMAN
	5	MANAGER

When WHERE and HAVING clause are used together in a SELECT query with aggregate function, WHERE clause is applied first on individual rows and only rows which pass the condition is included for creating groups. Once group is created, HAVING clause is used to filter groups based upon condition specified.

difference between where and  
having clause

# where and having clause

## Remember:

- **WHERE** clause can be used with - **SELECT**, **UPDATE**, and **DELETE** statements, where as **HAVING** clause can only be used with the **SELECT** statement.
  - **WHERE** clause filters rows before aggregation (GROUPING), where as, **HAVING** clause filters groups, after the aggregations are performed.
  - **WHERE** is used before the 'GROUP BY' clause if required and **HAVING** is used after the 'GROUP BY' clause.
  - Aggregate functions (**SUM**, **MIN**, **MAX**, **AVG** and **COUNT**) cannot be used in the **WHERE** clause, unless it is in a sub query contained in a **WHERE** or **HAVING** clause, whereas, aggregate functions can be used in **HAVING** clause.
- 

## Note:

- The **WHERE** clause acts as a pre-filter where as **HAVING** clause acts as a post-filter.

**WHERE**

**Vs**

**HAVING**

*where vs having*

WHERE	HAVING
Implemented in row operations.	Implemented in column operations.
Single row	Summarized row or group or rows.
It only fetches the data from particular rows or table according to the condition.	It only fetches the data from grouped data according to the condition.
Aggregate Functions cannot appear in WHERE clause.	Aggregate Functions Can appear in HAVING clause.
Used with SELECT and other statements such as UPDATE, DELETE or either one of them.	Used with SELECT statement only.
Pre-filter	Post-filter
GROUP BY Comes after WHERE.	GROUP BY Comes before HAVING.



window function

# window function

Use **ORDER BY** *expr* with **PARTITION BY** *expr* to see the effect of **PARTITION BY** *expr*.

- **RANK()** **OVER**(**PARTITION BY** *expr* [, *expr*] . . . **ORDER BY** *expr* [ASC|DESC] [, *expr* [ASC|DESC]] . . . )
- **DENSE\_RANK()** **OVER**(**PARTITION BY** *expr* [, *expr*] . . . **ORDER BY** *expr* [ASC|DESC] [, *expr* [ASC|DESC]] . . . )
- **ROW\_NUMBER()** **OVER**([ **PARTITION BY** *expr* [, *expr*] . . . **ORDER BY** *expr* [ASC|DESC] [, *expr* [ASC|DESC]] . . . ] )

## Note:

MySQL does not support these window function features.

- DISTINCT syntax for aggregate functions.
- Nested window functions
- Window function cannot be the part of **WHERE** condition

- `SELECT ROW_NUMBER() OVER() R1, emp.* FROM emp;`
- `SELECT RANK() OVER(PARTITION BY job ORDER BY sal) R1, ename, sal, job FROM emp;`
- `SELECT DENSE_RANK() OVER(PARTITION BY job ORDER BY sal) R1, ename, sal, job FROM emp;`
- `SELECT ordid, total, SUM(total) OVER(ORDER BY ordid) FROM ord;`
- `SELECT * FROM (SELECT ROW_NUMBER() OVER() R1, emp.* FROM emp) d WHERE R1 > (SELECT COUNT(*) - 2 FROM emp);`
  
- `SELECT custid, type, amount, CASE type WHEN 'd' THEN amount WHEN 'c' THEN amount * -1 END amount FROM transactions;`
- `SELECT year, quarter, amount, SUM(amount) OVER(PARTITION BY year ORDER BY quarter) R1 FROM quarter_revenue;`
- `SELECT custid, type, amount, SUM(CASE type WHEN 'd' THEN amount WHEN 'c' THEN amount * -1 END) OVER(PARTITION BY custid ORDER BY _id) amount FROM transactions;`

user-defined variables

# *user-defined variables*

## TODO

### Remember:

- A user variable name can contain other characters if you quote it as a string or identifier (for example, `@'my-var'`, `@"my-var"`, or `@`my-var``).
- User-defined variables are session specific. A user variable defined by one client cannot be seen or used by other clients.
- All variables for a given client session are automatically freed when that client exits.
- User variable names are not case sensitive. Names have a maximum length of 64 characters.
- If the value of a user variable is selected in a result set, it is returned to the client as a string.
- If you refer to a variable that has not been initialized, it has a value of NULL and a type of string.

e.g. `SELECT @variable_name;`

---

# user-defined variables

You can store a value in a user-defined variable in one statement and refer to it later in another statement. This enables you to pass values from one statement to another.

`SET @variable_name = expr [, @variable_name = expr] . . .`

## Remember:

- for SET, either `=` or `:=` can be used as the assignment operator.
- You can also assign a value to a user variable in statements (SELECT, ...) other than SET. In this case, the assignment operator must be `:=` and **not** `=` because latter is treated as the **comparison operator** `=`.
- `set @v1 = 1001, @v2 := 2, @v3 = 'Saleel';`
- `set @v1 = 1001, @v2 = 2, @v3 := @v1 + @v2;`
- `SELECT @v1 := MIN(sal), @v2 := MAX(SAL) FROM emp;`
- `SELECT @v1, @v2, @v3;`

```
SELECT VARIABLE_NAME, VARIABLE_VALUE FROM  
PERFORMANCE_SCHEMA.USER_VARIABLES_BY_THREAD;
```

## *user-defined variables*

## Note:

- User variables are intended to provide data values. They cannot be used directly in an SQL statement as an identifier or as part of an identifier.

e.g.

```
SET @v1 = 'ENAME';
```

## #WHERE ENAME IS COLUMN NAME.

```
SELECT @v1 FROM emp;
```

[illegible]

# rownum

```
mysql> SELECT * FROM (SELECT @cnt := @cnt + 1 "R1", emp.* FROM emp, (SELECT @cnt := 0) T1) T2 WHERE "R1" > @cnt - 7;
```



## select - rownum

```
mysql> SET @rank = 0;
```

```
mysql> SELECT @row := @row + 1 as rownum , emp.* FROM emp;
```

```
mysql> SELECT @row := @row + 1 as rownum , emp.* FROM emp, (SELECT @row := 0) as E;
```

```
mysql> SELECT @row := @row + 1 ,E.* FROM (SELECT job, sal FROM emp GROUP BY job ORDER BY SAL DESC) E , (SELECT @row := 0) EE;
```

ROWNUM	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	BONUSID	USER NAME	PWD
1	7839	KING	PRESIDENT	NULL	1981-11-17 00:00:00	5000.00	NULL	10	1	KING	r50m0m
2	7698	BLAKE	MANAGER	7839	1981-05-01 00:00:00	2850.00	NULL	30	1	BLAKE	sales@2017
3	7782	CLARK	MANAGER	7839	1981-06-09 00:00:00	2450.00	NULL	10	3	CLARK	r50m0m
4	7566	JONES	MANAGER	7839	1981-04-02 00:00:00	2975.00	NULL	20	4	JONES	a12recm0m
5	7654	MARTIN	SALESMAN	7698	1981-09-28 00:00:00	1250.00	1400.00	30	6	MARTIN	sales@2017
6	7499	ALLEN	SALESMAN	7698	1981-02-20 00:00:00	1600.00	300.00	30	4	ALLEN	sales@2017
7	7844	TURNER	SALESMAN	7698	1981-09-08 00:00:00	1500.00	0.00	30	5	TURNER	sales@2017
8	7900	JAMES	CLERK	7698	1981-12-03 00:00:00	950.00	NULL	30	2	JAMES	sales@2017
9	7521	WARD	SALESMAN	7698	1981-02-22 00:00:00	1250.00	500.00	30	1	WARD	sales@2017
10	7902	FORD	ANALYST	7566	1981-12-03 00:00:00	3000.00	NULL	20	4	FORD	a12recm0m

## Examples:

# common sql statements mistakes

- `SELECT` `ename`, `job`, `sal`, `comm` `FROM` `emp` `WHERE` `comm` = `NULL`; #using comparison operator to check NULL
- `SELECT` `job`, `COUNT`(`job`) `FROM` `emp`; #not giving group by clause
- `SELECT` `job`, `COUNT`(`job`) `FROM` `emp` `WHERE` `COUNT`(`job`) > 4; #use of aggregate function in where clause
- `SELECT` `job`, `deptno`, `COUNT`(`job`) `FROM` `emp` `GROUP BY` `job`; #not giving all the columns in group by clause
- `SELECT` `ename`, `COUNT`(`job`) `FROM` `emp` `GROUP BY` `ename`; #grouping by a unique key
- `SELECT` `ename`, `sal`, `sal` + 1000 `R1` `FROM` `emp` `WHERE` `R1` > 2400; #use of alias name in where clause
- `SELECT` `ename`, `sal` `FROM` `emp` `WHERE` `sal` `BETWEEN` (1000 and 4000); #use of () in between comparison operator

***r1 = { col1, col2, col3 }***

- `INSERT INTO` ***r1*** `VALUE`(10, 10); #number of values are less than the number of columns in the table
- `INSERT INTO` ***r1*** `VALUE`(10, 10, 10, 10); #number of values are more than the number of columns in the table

IS NULL  
NULL

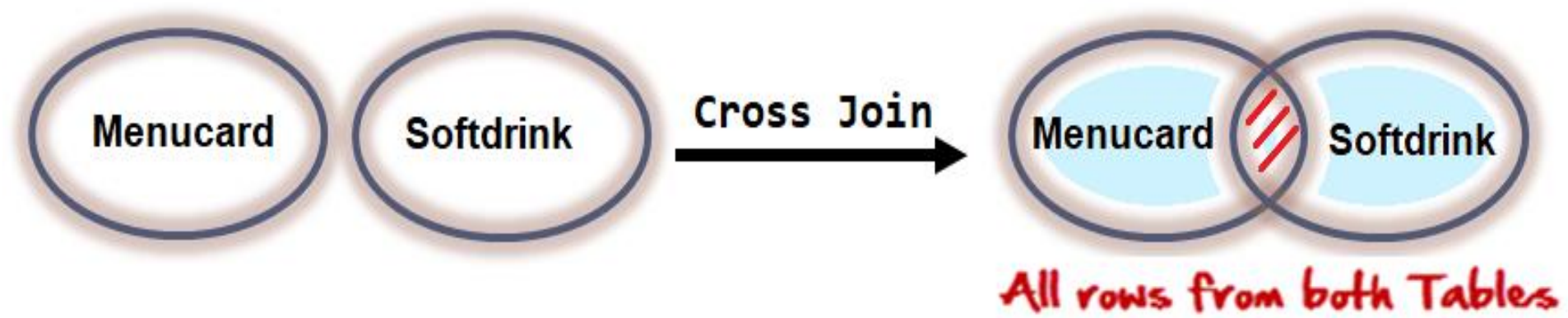
joins

**JOINS** are used to **retrieve data from multiple tables.**

**JOIN** is performed whenever **two or more tables** are joined in a SQL statement.

## *Type of JOINS*

- Cartesian or Product Join – Cross Join
- Equijoin – Inner Join
- Natural Join
- Simple Join
- Outer Join – Right Outer Join, Left Outer Join
- Self Join



## cartesian or product join

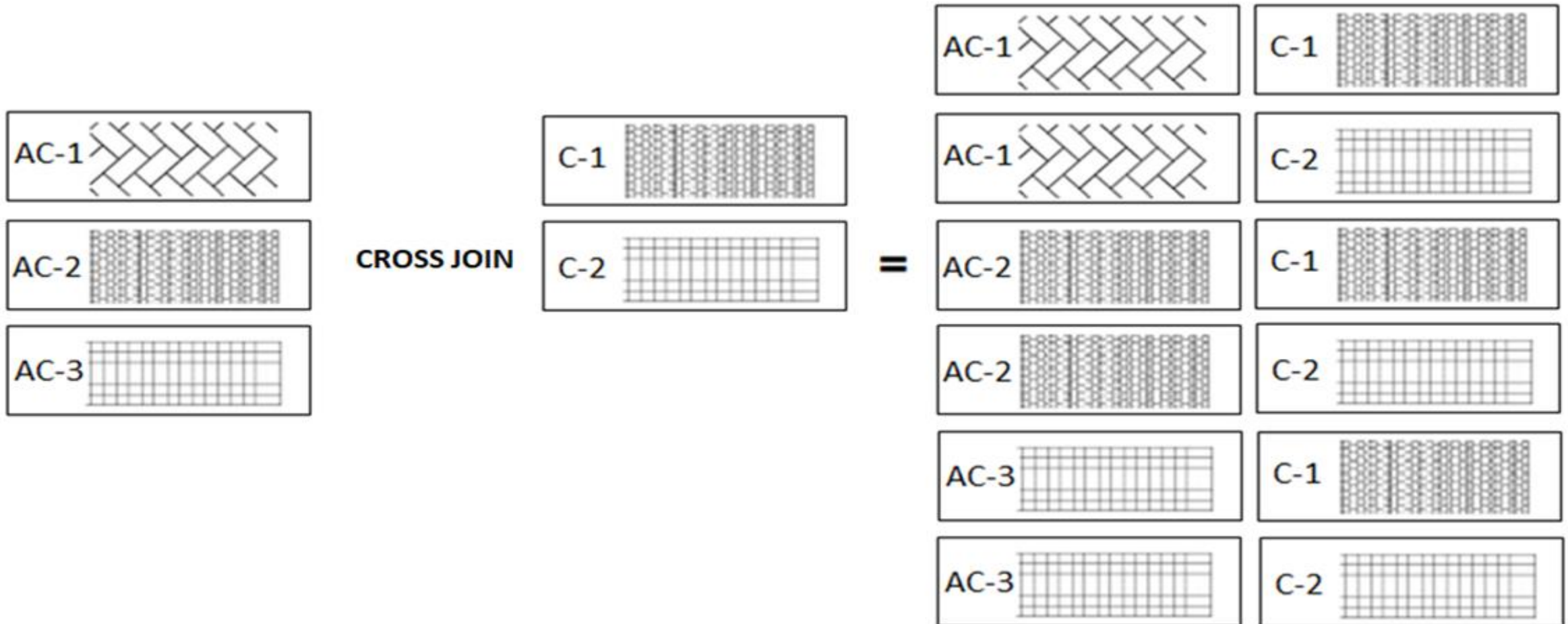
- **Cartesian/Product means** Number of Rows present in Table 1 Multiplied by Number of Rows present in Table 2.
- **Cross Join in MySQL** does not require any common column to **join** two table.

The result of  $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)$  is a relation  $Q$  with degree  $n + m$  attributes  $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$ , in that order.

$$\text{Degree } d(R \times S) = d(R) + d(S).$$
$$\text{Cardinality } |R \times S| = |R| * |S|$$

# joins - cartesian or product

The CROSS JOIN gets a row from the first table ( $r_1$ ) and then creates a new row for every row in the second table ( $r_2$ ). It then does the same for the next row for in the first table ( $r_1$ ) and so on.



# joins - cartesian or product

Cartesian or Product joins are joins without a join condition. Each row of one table is combined with each row of another table. The result is referred to as a Cartesian product.

`SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1, r_2, \dots$`

- Warehouse/product
- Product/sales\_channel
- Cards

$r_1 = \{1, 2\}$

$r_2 = \{a, b, c\}$

$r_1 \times r_2$

$R = \{ (1, a),$   
     $(2, a),$   
     $(1, b),$   
     $(2, b),$   
     $(1, c),$   
     $(2, c) \}$

$r_1 = \{1, 2, 3\}$

$r_2 = \{a, b\}$

$r_1 \times r_2$

$R = \{ (1, a),$   
     $(1, b),$   
     $(2, a),$   
     $(2, b),$   
     $(3, a),$   
     $(4, b) \}$

$r_1 = \{1, 2\}$

$r_2 = \{a, b, \text{null}\}$

$r_1 \times r_2$







$R = \{ (1, a),$   
     $(2, a),$   
     $(1, b),$   
     $(2, b),$   
     $(1, \text{null}),$   
     $(2, \text{null}) \}$












# joins - cartesian or product









Cartesian or Product joins are joins without a join condition. Each row of one table is combined with each row of another table. The result is referred to as a Cartesian product.

`SELECT  $A_1$ ,  $A_2$ ,  $A_3$ , ... FROM  $r_1$ ,  $r_2$ , ...`

customer	supplier
 CUSTOMER_ID INT	 SUPPLIER_ID INT
 FIRST_NAME VARCHAR(45)	 FIRST_NAME VARCHAR(45)
 LAST_NAME VARCHAR(45)	 LAST_NAME VARCHAR(45)
Indexes	Indexes
PRIMARY	PRIMARY

- Warehouse/product
- Product/sales\_channel
- Person/Cards

products	warehouses
 product_id INT	 warehouse_id INT
 product_name VARCHAR(255)	 warehouse_name VARCHAR(255)
 description VARCHAR(2000)	 location_id INT
 standard_cost INT	
 list_price INT	
 category_id INT	
Indexes	Indexes
	PRIMARY

products	sales_channel
 product_id INT	 channel_id INT
 product_name VARCHAR(255)	 channel VARCHAR(255)
 description VARCHAR(2000)	
 standard_cost INT	
 list_price INT	
 category_id INT	
Indexes	Indexes
PRIMARY	PRIMARY

# joins - cartesian or product

SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1, r_2, \dots$

- SELECT \* FROM menucard, softdrink;

	ID	NAME	RATE
►	1	Extra Long Cheeseburger	100
	2	Double Stacker	125
	3	Double Cheeseburger	100
	4	Hamburger	85
	5	Classic Grilled Dog	95
	6	Chili Cheese Grilled Dog	115
	7	Flame Grilled Chicken Burger	135
	8	Original Chicken Sandwich	55
	9	McALLO TIKKI	45
	10	Veg Maharaja Mac	75
	11	Big Spicy Chicken Wrap	100
	12	McVeggie Schezwan	85

	ID	NAME	RATE
►	1	Coca-Cola	45
	2	Mello Yello	75
	3	Diet Coke	60
	4	Frozen Fanta Cherry	65
	5	Iced Tea	35

	ID	NAME	RATE	ID	NAME	RATE
►	1	Extra Long Cheeseburger	100	1	Coca-Cola	45
	1	Extra Long Cheeseburger	100	2	Mello Yello	75
	1	Extra Long Cheeseburger	100	3	Diet Coke	60
	1	Extra Long Cheeseburger	100	4	Frozen Fanta Cherry	65
	1	Extra Long Cheeseburger	100	5	Iced Tea	35
	2	Double Stacker	125	1	Coca-Cola	45
	2	Double Stacker	125	2	Mello Yello	75
	2	Double Stacker	125	3	Diet Coke	60
	2	Double Stacker	125	4	Frozen Fanta Cherry	65
	2	Double Stacker	125	5	Iced Tea	35
	3	Double Cheeseburger	100	1	Coca-Cola	45
	3	Double Cheeseburger	100	2	Mello Yello	75
	3	Double Cheeseburger	100	3	Diet Coke	60
	3	Double Cheeseburger	100	4	Frozen Fanta Cherry	65
	3	Double Cheeseburger	100	5	Iced Tea	35
	4	Hamburger	85	1	Coca-Cola	45
	4	Hamburger	85	2	Mello Yello	75
	4	Hamburger	85	3	Diet Coke	60
	4	Hamburger	85	4	Frozen Fanta Cherry	65
	4	Hamburger	85	5	Iced Tea	35

# joins - cartesian or product

SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1, r_2, \dots$

- SELECT mnu.name, sftdrink.name, mnu.rate, sftdrink.rate, mnu.rate + sftdrink.rate as "Total" FROM menucard mnu, softdrink sftdrink;

	ID	NAME	RATE
▶	1	Extra Long Cheeseburger	100
	2	Double Stacker	125
	3	Double Cheeseburger	100
	4	Hamburger	85
	5	Classic Grilled Dog	95
	6	Chili Cheese Grilled Dog	115
	7	Flame Grilled Chicken Burger	135
	8	Original Chicken Sandwich	55
	9	McALLO TIKKI	45
	10	Veg Maharaja Mac	75
	11	Big Spicy Chicken Wrap	100
	12	McVeggie Schezwan	85

	ID	NAME	RATE
▶	1	Coca-Cola	45
	2	Mello Yello	75
	3	Diet Coke	60
	4	Frozen Fanta Cherry	65
	5	Iced Tea	35

	name	name	rate	rate	Total
▶	Extra Long Cheeseburger	Coca-Cola	100	45	145
	Extra Long Cheeseburger	Mello Yello	100	75	175
	Extra Long Cheeseburger	Diet Coke	100	60	160
	Extra Long Cheeseburger	Frozen Fanta Cherry	100	65	165
	Extra Long Cheeseburger	Iced Tea	100	35	135
	Double Stacker	Coca-Cola	125	45	170
	Double Stacker	Mello Yello	125	75	200
	Double Stacker	Diet Coke	125	60	185
	Double Stacker	Frozen Fanta Cherry	125	65	190
	Double Stacker	Iced Tea	125	35	160
	Double Cheeseburger	Coca-Cola	100	45	145
	Double Cheeseburger	Mello Yello	100	75	175
	Double Cheeseburger	Diet Coke	100	60	160
	Double Cheeseburger	Frozen Fanta Cherry	100	65	165
	Double Cheeseburger	Iced Tea	100	35	135
	Hamburger	Coca-Cola	85	45	130
	Hamburger	Mello Yello	85	75	160
	Hamburger	Diet Coke	85	60	145
	Hamburger	Frozen Fanta Cherry	85	65	150
	Hamburger	Iced Tea	85	35	120
	Classic Grilled Dog	Coca-Cola	95	45	140
	Classic Grilled Dog	Mello Yello	95	75	170
	Classic Grilled Dog	Diet Coke	95	60	155

## joins - cartesian or product

- `SELECT name, COUNT(*) "Total Employees", rate * COUNT(*) "Total Cost" FROM emp, softdrink GROUP BY name;`

	NAME	TOTAL EMPLOYEES	TOTAL COST
	Coca-Cola	30	1350
	Diet Coke	30	1800
	Frozen Fanta Cherrv	30	1950
	Iced Tea	30	1050
	Mello Yello	30	2250

## joins – cross join

The CROSS JOIN produced a result set which is the product of rows of two associated tables when no WHERE clause is used with CROSS JOIN. In this join, the result set appeared by multiplying each row of the first table with all rows in the second table if no condition introduced with CROSS JOIN.

**SELECT**  $A_1, A_2, A_3, \dots$  **FROM**  $r_1$  **CROSS JOIN**  $r_2, \dots$

envelope **Table**

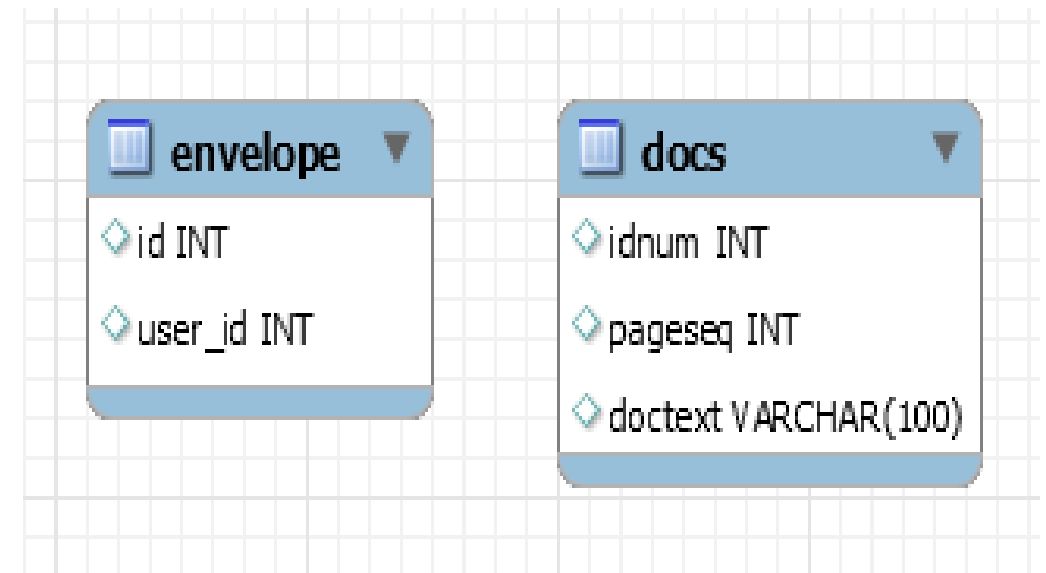
	id	user_id
▶	1	1
	2	2
	3	3

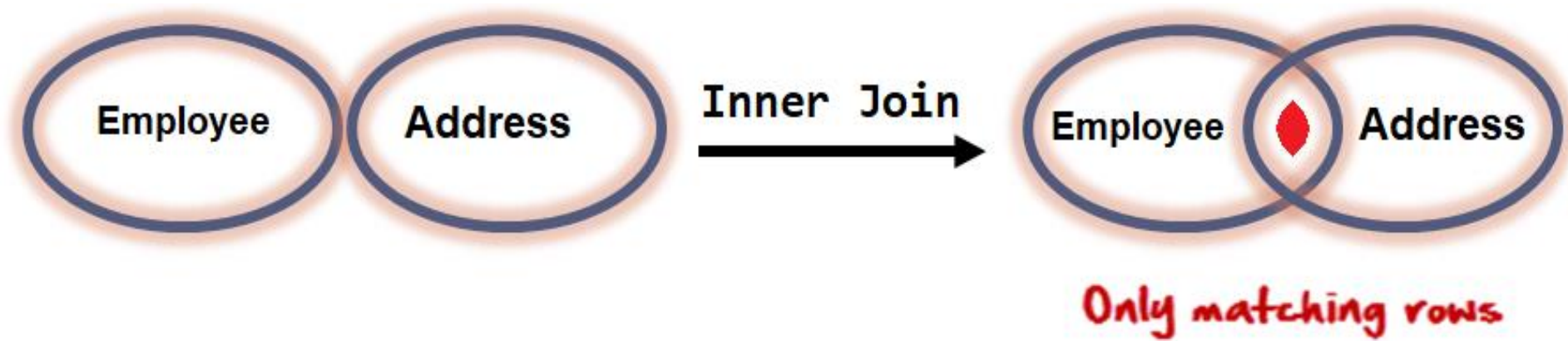
docs **Table**

	idnum	pageseq	doctext
▶	1	5	NULL
	2	6	NULL
	NULL	0	NULL

- **SELECT** \* **FROM** envelope **CROSS JOIN** docs;

	id	user_id	idnum	pageseq	doctext
▶	1	1	1	5	NULL
	2	2	1	5	NULL
	3	3	1	5	NULL
	1	1	2	6	NULL
	2	2	2	6	NULL
	3	3	2	6	NULL
	1	1	NULL	0	NULL
	2	2	NULL	0	NULL
	3	3	NULL	0	NULL



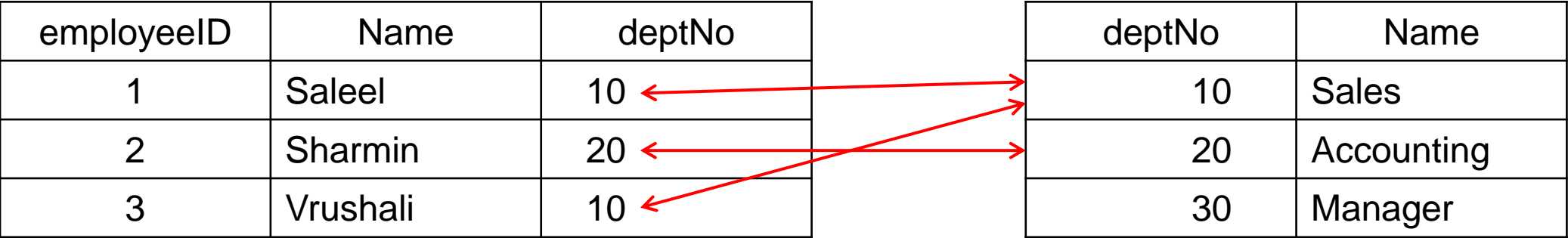


## equi join

An **equi join** / **Inner Join** is a join with a join condition containing an equality operator.

An equijoin returns only those rows that have equivalent values for the specified columns. Rows that match remain in the result, those that don't are rejected. The match condition is commonly called the **join condition**. **equi join** / **Inner Join** returns rows when there is at least one match in both tables.

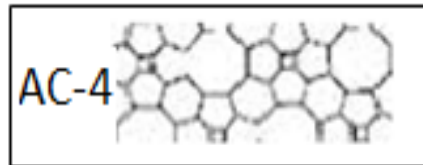
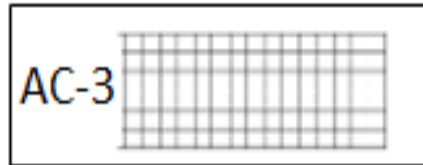
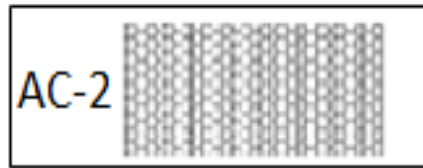
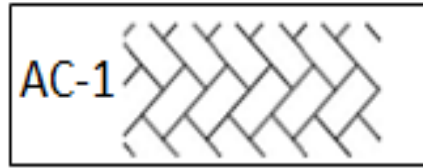
The result of  $R(A_1, A_2, \dots, A_n) \bowtie_{\langle \text{join condition} \rangle} S(B_1, B_2, \dots, B_m)$  is a relation  $Q$  with degree  $n + m$  attributes  $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$ , **in that order**.  $Q$  has one tuple for each combination of tuples—one from  $R$  and one from  $S$ —whenever the combination satisfies the join condition.



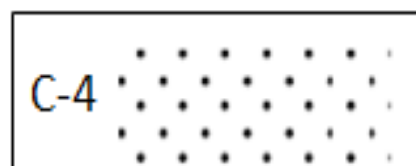
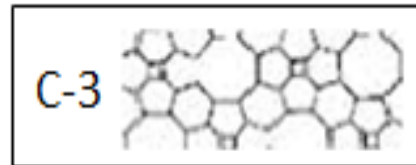
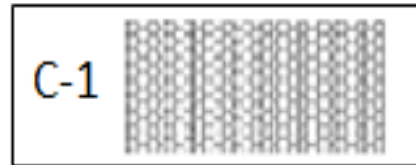
employeeID	Name	deptNo	deptNo	Name
1	Saleel	10	10	Sales
2	Sharmin	20	20	Accounting
3	Vrushali	10	10	Sales

## equi join example

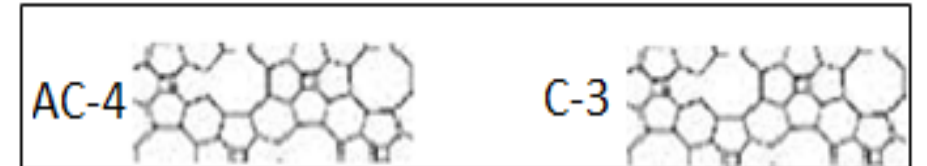
The following table illustrates the inner join of two tables  $r_1$ (AC-1, AC-2, AC-3, AC-4, AC-5) and  $r_2$ (C-1, C-2, C-3, C-4). The result includes rows: (2,A), (3,B), and (4,C) as they have the same patterns.



**INNER JOIN**



**=**





## joins – equi join

EQUI JOIN performs a JOIN against equality or matching column(s) values of the associated tables. An equal sign (=) is used as comparison operator in the where clause to refer equality.

SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1, r_2$  WHERE  $r_1.A_1 = r_2.A_1$

$r_1 = \{ 1, 2, 3, 4 \}$

$r_2 = \{ (1, a), (2, b), (1, c), (3, d), (2, e), (1, f) \}$

$r_1 = r_2$

$R = \{(1,1,a),$   
     $(2,2,b),$   
     $(1,1,c),$   
     $(3,3,d),$   
     $(2,2,e),$   
     $(1,1,f)\}$

**Remember:**

A general join condition is of the form  $\langle \text{condition} \rangle \text{ AND } \langle \text{condition} \rangle \text{ AND } \dots \text{ AND } \langle \text{condition} \rangle$ , where each  $\langle \text{condition} \rangle$  is of the form  $A_i \theta B_j$ ,  **$A_i$  is an attribute of R,  $B_j$  is an attribute of S.**

# joins – equi join

- `SELECT * FROM emp , dept WHERE emp.deptno = dept.deptno;`

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	BONUSID	USER NAME	PWD	isActive	DEPTNO	DNAME	LOC	PWD
▶	7369	SMITH	CLERK	7902	1980-12-17	800.00	NULL	20	2	SMITH	a12recmpm	0	20	RESEARCH	DALLAS	a12recmpm
	7415	AARAV	CLERK	7902	1981-12-31	3350.00	NULL	10	NULL	AARAV	NULL	0	10	ACCOUNTING	NEW YORK	r50mpm
	7421	THOMAS	CLERK	7920	1981-07-19	1750.00	NULL	10	1	THOMAS	r50mpm	0	10	ACCOUNTING	NEW YORK	r50mpm
	7499	ALLEN	SALESMAN	7698	1981-02-20	1600.00	300.00	30	4	ALLEN	sales@2017	1	30	SALES	CHICAGO	sales@2017
	7521	WARD	SALESMAN	7698	1981-02-22	1250.00	500.00	30	1	WARD	sales@2017	1	30	SALES	CHICAGO	sales@2017
	7566	JONES	MANAGER	7839	1981-04-02	2975.00	NULL	20	4	JONES	a12recmpm	1	20	RESEARCH	DALLAS	a12recmpm
	7654	MARTIN	SALESMAN	7698	1981-09-28	1250.00	1400.00	30	6	MARTIN	sales@2017	1	30	SALES	CHICAGO	sales@2017
	7698	BLAKE	MANAGER	7839	1981-05-01	2850.00	NULL	30	1	BLAKE	sales@2017	1	30	SALES	CHICAGO	sales@2017
	7782	CLARK	MANAGER	7839	1981-06-09	2450.00	NULL	10	3	CLARK	r50mpm	1	10	ACCOUNTING	NEW YORK	r50mpm
	7788	SCOTT	ANALYST	7566	1982-12-09	3000.00	NULL	20	3	SCOTT	a12recmpm	1	20	RESEARCH	DALLAS	a12recmpm
	7839	KING	PRESIDENT	NULL	1981-11-17	5000.00	NULL	10	1	KING	r50mpm	1	10	ACCOUNTING	NEW YORK	r50mpm
	7844	TURNER	SALESMAN	7698	1981-09-08	1500.00	0.00	30	5	TURNER	sales@2017	1	30	SALES	CHICAGO	sales@2017
	7876	ADAMS	CLERK	7788	1983-01-12	1100.00	NULL	20	1	ADAMS	a12recmpm	1	20	RESEARCH	DALLAS	a12recmpm
	7900	JAMES	CLERK	7698	1981-12-03	950.00	NULL	30	2	JAMES	sales@2017	1	30	SALES	CHICAGO	sales@2017
	7902	FORD	ANALYST	7566	1981-12-03	3000.00	NULL	20	4	FORD	a12recmpm	0	20	RESEARCH	DALLAS	a12recmpm
	7919	HOFFMAN	MANAGER	7566	1982-03-24	4150.00	NULL	30	3	HOFFMAN	sales@2017	1	30	SALES	CHICAGO	sales@2017
	7920	GRASS	SALESMAN	7919	1980-02-14	2575.00	2700.00	30	5	GRASS	sales@2017	1	30	SALES	CHICAGO	sales@2017
	7934	MILLER	CLERK	7919	1982-01-23	1300.00	NULL	10	2	MILLER	r50mpm	0	10	ACCOUNTING	NEW YORK	r50mpm

## Remember:

- `SELECT * FROM emp, dept WHERE emp.deptno = dept.deptno AND dname = 'accounting';`
- `SELECT * FROM emp, dept WHERE (emp.deptno, dname) = (dept.deptno, 'accounting');`

tableA Table

	id	name
▶	5	aa
	1	a
	2	b
	3	y
	NULL	d
	5	NULL
	1	NULL
	1	b
	8	a

tableB Table

	id	name
▶	1	a
	2	x
	4	b
	NULL	c
	6	NULL
	NULL	NULL
	7	z
	2	NULL
	5	z
	9	u

*joins – equi join*

tablea
id INT
name VARCHAR(10)

tableb
id INT
name VARCHAR(10)

- `SELECT * FROM tableA , tableB WHERE tableA.id = tableB.id;`

	id	name	id	name
▶	1	a	1	a
	1	NULL	1	a
	1	b	1	a
	2	b	2	x
	2	b	2	NULL
	5	aa	5	z
	5	NULL	5	z

- `SELECT * FROM tableA , tableB WHERE tableA.name = tableB.name;`

	id	name	id	name
▶	1	a	1	a
	8	a	1	a
	2	b	4	b
	1	b	4	b

## ON Contrition

- When this join condition gets applied none of the columns of the relation will get eliminated in the result set.
- In order to apply this join condition, on any two tables they need not to have any common column.

# on condition and using attribute

## USING Attribute Contrition

- When all the common columns are used in the join predicate then the result would be same as Natural join.
- In the result set of the join the duplicates of the columns used in the predicate gets eliminated.
- It should not have a qualifier(table name or Alias) in the referenced columns.

### Note:

- **ON** clause is optional, If not given then *INNER JOIN* works like *CROSS JOIN*.

# *joins – on & using clause example*

## The ON clause

The ON clause is used to join tables where the column names don't match in both tables.

```
SELECT * FROM EMP
INNER JOIN DEPT
ON EMP.DEPTNO = DEPT.ID
```

JOINING CONDITION



## The USING clause

The USING clause is used if several columns share the same name but you don't want to join using all of these common columns. The columns listed in the USING clause can't have any qualifiers in the statement.

```
SELECT * FROM EMP
INNER JOIN DEPT
USING(DEPTNO)
```

JOINING CONDITION



# inner join

The inner join is one of the most commonly used joins in SQL. The inner join clause allows you to query data from two or more related tables.

INNER JOIN returns rows when there is at least one match in both tables.

## joins – inner join

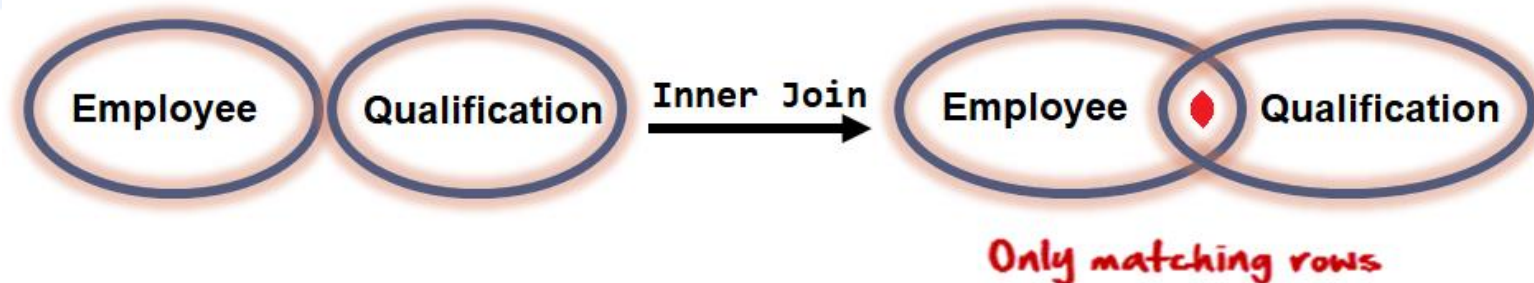
The INNER JOIN selects all rows from both participating tables as long as there is a match between the columns. An SQL INNER JOIN is same as JOIN clause, combining rows from two or more tables.

$\text{SELECT } A_1, A_2, A_3, \dots \text{ FROM } r_1 \text{ [INNER] JOIN } r_2 \text{ ON } r_1.A_1 = r_2.A_1$

- $\text{SELECT } * \text{ FROM employee emp INNER JOIN qualification quali ON emp.id = quali.employeeid;}$

	ID	FIRSTNAME	LASTNAME	GENDER	HIREDATE	ID	EMPLOYEEID	NAME	Stream	ADDMISSIONYEAR	INSTITUTE	UNIVERSITY	YEAROFPASSING	PERCENTAGE	GRADE
	1	Denis	Murphy	M	1964-06-12	1	1	10	General	1957-08-02	Alabama	Stanford University	1958	62.00	D
	1	Denis	Murphy	M	1964-06-12	2	1	12	Science	1959-06-22	Alaska	Harvard University	1960	56.00	D
	1	Denis	Murphy	M	1964-06-12	3	1	BE	IT	1960-06-12	Arizona	Harvard University	1964	75.00	B
	2	Jenny	Ross	F	1964-10-25	4	2	10	General	1957-01-19	Alaska	University of Chicago	1958	67.00	C
	2	Jenny	Ross	F	1964-10-25	5	2	12	Commerce	1959-10-23	New York	Yale University	1960	67.00	C
	2	Jenny	Ross	F	1964-10-25	6	2	B.Com	Accounting	1960-06-12	Arkansas	Yale University	1964	69.00	C
	3	David	Ross	M	1964-10-25	7	3	10	General	1957-11-25	Arizona	Yale University	1958	86.00	A
	3	David	Ross	M	1964-10-25	8	3	12	Science	1959-02-17	California	California University	1960	57.00	D
	3	David	Ross	M	1964-10-25	9	3	BE	IT	1960-06-12	Florida	University of Florida	1964	85.00	A
	4	Fred	NULL	M	1965-10-31	10	4	10	General	1958-03-19	Idaho	Pennsylvania University	1959	89.00	A
	4	Fred	NULL	M	1965-10-31	11	4	12	Commerce	1960-05-21	New Ham...	Yale University	1961	96.00	A+
	4	Fred	NULL	M	1965-10-31	12	4								
	5	Helen	Taylor	F	1965-01-12	13	5								

ON clause is optional, If not given then INNER JOIN works like CROSS JOIN





INNER JOIN returns rows when there is at least one match in both tables.

## joins – inner join

The INNER JOIN selects all rows from both participating tables as long as there is a match between the columns. An SQL INNER JOIN is same as JOIN clause, combining rows from two or more tables.

$\text{SELECT } A_1, A_2, A_3, \dots \text{ FROM } r_1 [\text{INNER}] \text{ JOIN } r_2 \text{ ON } r_1.A_1 = r_2.A_1$

- SELECT \* FROM customer INNER JOIN ord USING (custid);

	CUSTID	NAME	ADDRESS	CITY	STATE	ZIP	AREA	PHONE	REPID	CREDITLIMIT	COMMENTS	ORDID	ORDERDATE	COMPLAN	SHIPDATE	STA
▶	106	SHAPE UP	908 SEQUOIA	PALO ALTO	CA	94301	415	364-9777	7521	6000.00	Support intensive. Orders small a...	601	1986-05-01 00:00:00	A	1986-05-30 00:00:00	In Pr
	102	VOLLYRITE	9722 HAMILTON	BURLINGAME	CA	95133	415	644-3341	7654	7000.00	Company doing heavy promotion ...	602	1986-06-05 00:00:00	B	1986-06-20 00:00:00	On H
	102	VOLLYRITE	9722 HAMILTON	BURLINGAME	CA	95133	415	644-3341	7654	7000.00	Company doing heavy promotion ...	603	1986-06-05 00:00:00	NULL	1986-06-05 00:00:00	Canc
	106	SHAPE UP	908 SEQUOIA	PALO ALTO	CA	94301	415	364-9777	7521	6000.00	Support intensive. Orders small a...	604	1986-06-15 00:00:00	A	1986-06-30 00:00:00	Resol
	106	SHAPE UP	908 SEQUOIA	PALO ALTO	CA	94301	415	364-9777	7521	6000.00	Support intensive. Orders small a...	605	1986-07-14 00:00:00	A	1986-07-30 00:00:00	Dispu
	100	JOCKSPORTS	345 VIEWRIDGE	BELMONT	CA	96711	415	598-6609	7844	5000.00	Very friendly people to work with ...	606	1986-07-14 00:00:00	A	1986-07-30 00:00:00	Shipp
	104	EVERY MOUNTAIN	574 SURRY RD.	CUPERTINO	CA	93301	408	996-2323	7499	10000.00	Customer with high market share ...	607	1986-07-18 00:00:00	C	1986-07-18 00:00:00	In Pr
	104	EVERY MOUNTAIN	574 SURRY RD.	CUPERTINO	CA	93301	408	996-2323	7499	10000.00	Customer with high market share ...	608	1986-07-25 00:00:00	C	1986-07-25 00:00:00	Shipp
	100	JOCKSPORTS	345 VIEWRIDGE	BELMONT	CA	96711	415	598-6609	7844	5000.00	Very friendly people to work with ...	609	1986-08-01 00:00:00	B	1986-08-15 00:00:00	On H
	101	TKB SPORT SHOP	490 BOLI RD.	REDWOOD CITY	CA	94061	415	368-1223	7521	10000.00	Rep called 5/8 about change in or...	610	1987-01-07 00:00:00	A	1987-01-08 00:00:00	In Pr
	102	VOLLYRITE	9722 HAMILTON	BURLINGAME	CA	95133	415	644-3341	7654	7000.00	Company doing heavy promotion ...	611	1987-01-11 00:00:00	B	1987-01-11 00:00:00	Shipp
	104	EVERY MOUNTAIN	574 SURRY RD.	CUPERTINO	CA	93301	408	996-2323	7499	10000.00	Customer with high market share ...	612	1987-01-15 00:00:00	C	1987-01-20 00:00:00	Canc
	108	NORTH WOODS ...	98 LONE PINE ...	HIBBING	MN	55649	612	566-9123	7844	8000.00	NULL	613	1987-02-01 00:00:00	NULL	1987-02-01 00:00:00	Shipp
	102	VOLLYRITE	9722 HAMILTON	BURLINGAME	CA	95133	415	644-3341	7654	7000.00	Company doing heavy promotion ...	614	1987-02-01 00:00:00	NULL	1987-02-05 00:00:00	In Pr
	107	WOMENS SPORTS	VALCO VILLAGE	SUNNYVALE	CA	93301	408	967-4398	7499	10000.00	First sporting goods store geared ...	615	1987-02-01 00:00:00	NULL	1987-02-06 00:00:00	In Pr
	103	JUST TENNIS	HILLVIEW MALL	BURLINGAME	CA	97544	415	677-9312	7521	3000.00	Contact rep about new line of ten...	616	1987-02-03 00:00:00	NULL	1987-02-10 00:00:00	Resol
	105	K + T SPORTS	3476 EL PASEO	SANTA CLARA	CA	91003	408	376-9966	7844	5000.00	Tends to order large amounts of ...	617	1987-02-05 00:00:00	NULL	1987-03-03 00:00:00	Shipp
	102	VOLLYRITE	9722 HAMILTON	BURLINGAME	CA	95133	415	644-3341	7654	7000.00	Company doing heavy promotion ...	618	1987-02-15 00:00:00	A	1987-03-06 00:00:00	On H
	104	EVERY MOUNTAIN	574 SURRY RD.	CUPERTINO	CA	93301	408	996-2323	7499	10000.00	Customer with high market share ...	619	1987-02-22 00:00:00	NULL	1987-02-04 00:00:00	In Pr
	100	JOCKSPORTS	345 VIEWRIDGE	BELMONT	CA	96711	415	598-6609	7844	5000.00	Very friendly people to work with ...	620	1987-03-12 00:00:00	NULL	1987-03-12 00:00:00	Canc
	100	JOCKSPORTS	345 VIEWRIDGE	BELMONT	CA	96711	415	598-6609	7844	5000.00	Very friendly people to work with ...	621	1987-03-15 00:00:00	A	1987-01-01 00:00:00	Shipp

ON clause is optional, If not given then  
INNER JOIN works like CROSS JOIN





In general, the join condition for NATURAL JOIN is constructed by equating each pair of join attributes that have the same name in the two relations and combining these conditions with AND.

## natural join

The NATURAL JOIN is such a join that performs the same task as an INNER JOIN. NATURAL JOIN does not use any comparison operator. We can perform a NATURAL JOIN only if there is at least one common attribute that exists between two relations. In addition, the attributes must have the same name. When this join condition gets applied always the duplicates of the common columns get eliminated from the result.

### Remember:

The standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have the same name in both relations. **If this is not the case, a renaming operation is applied first.**

**e.g.**

- `SELECT * FROM r NATURAL JOIN (SELECT a1 AS c1, a2 FROM s) t1;`

## joins – natural join

Joins two tables based on common column names. Hence one must confirm the common columns before using a NATURAL JOIN

The **NATURAL JOIN** is such a join that performs the same task as an **INNER JOIN**.

`SELECT  $A_1$ ,  $A_2$ ,  $A_3$ , ... FROM  $r_1$  NATURAL [INNER] JOIN  $r_2$  NATURAL [INNER] JOIN  $r_3$  ...`

- `SELECT * FROM emp NATURAL JOIN dept;`
- The associated tables have one or more pairs of identically column-names.
- The columns must be of the same name.
- The columns datatype may differ.
- Don't use ON / USING clause in a NATURAL JOIN.
- When this join condition gets applied always the duplicates of the common columns get eliminated from the result.

A **NATURAL JOIN** can be used with a **LEFT OUTER** join, or a **RIGHT OUTER** join.

If the column-names are not same, then NATURAL JOIN will work as CROSS JOIN.

```
SELECT * FROM EMP
NATURAL JOIN DEPT
```

**INNER**

Vs

**NATURAL**

## *inner join vs natural join*

### INNER JOIN

Inner Join joins two table on the basis of the column which is explicitly specified in the ON clause.

In Inner Join, The resulting table will contain all the attribute of both the tables including duplicate columns also

In Inner Join, only those records will return which exists in both the tables

SYNTAX:

- `SELECT * FROM r1 INNER JOIN r2 ON r1.A1 = r2.A1;`
- `SELECT * FROM r1 INNER JOIN r2 USING(A1, [A2]);`

### NATURAL JOIN

Natural Join joins two tables based on same attribute name.

In Natural Join, The resulting table will contain all the attributes of both the tables but keep only one copy of each common column

Same as Inner Join

SYNTAX:

- `SELECT * FROM r1 NATURAL JOIN r2;`

simple join

TODO

## joins – simple join

The **SIMPLE JOIN** is such a join that performs the same task as an **INNER JOIN**.

```
SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1$  SIMPLE JOIN  $r_2$  USING ( $A_1, \dots$ )
```

- `SELECT * FROM emp SIMPLE JOIN dept USING(deptno)`

```
SELECT * FROM EMP  
SIMPLE JOIN DEPT  
USING(DEPTNO)
```

↑  
JOINING CONDITION

The ON clause is required for a left or right outer join.

The LEFT OUTER JOIN operation keeps every tuple in the first, or left, relation R in  $R \bowtie S$ , if no matching tuple is found in S, then the attributes of S in the join result are filled or padded with NULL values.

The RIGHT OUTER JOIN keeps every tuple in the second, or right, relation S in the result of  $R \bowtie S$ , if no matching tuple is found in R, then the attributes of R in the join result are filled or padded with NULL values.

## outer joins

In an outer join, along with rows that satisfy the matching criteria, we also include some or all rows that do not match the criteria.

```
CREATE TABLE r1 (id INT, c1 VARCHAR(10));
```

```
CREATE TABLE r2 (id INT, c1 VARCHAR(10));
```

```
INSERT INTO r1 VALUES (4,'AC-1'), (1,'AC-2'),(2,'AC-3'),(3,'AC-4'),(5,'AC-5');
```

```
INSERT INTO r2 VALUES (1,'C-1'), (2,'C-2'),(3,'C-3'),(7,'C-4');
```

Suppose, we want to join two tables: r1 and r2. SQL left outer join returns all rows in the left table (r1) and all the matching rows found in the right table (r2). It means the result of the SQL left join always contains the rows in the left table. . **If no matching rows found in the right table, NULL are displayed.**

## left outer joins

$r_1 = \{1, 2, 3, 4\}$

$r_2 = \{(1, a), (2, b), (1, c), (3, d), (2, e), (1, f), (5, z)\}$

$r_1$  left join  $r_2$

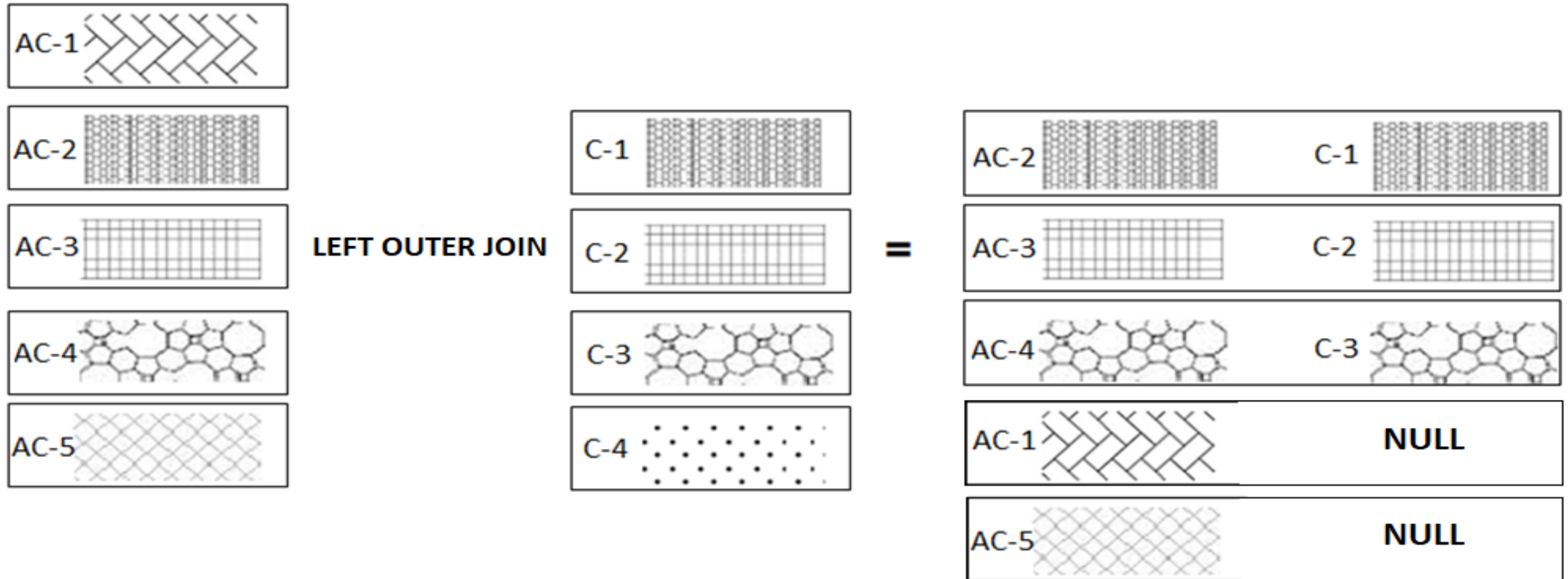
$R = \{(1, 1, a),$   
     $(2, 2, b),$   
     $(1, 1, c),$   
     $(3, 3, d),$   
     $(2, 2, e),$   
     $(1, 1, f),$   
     $(4, \text{NULL}, \text{NULL})\}$

```
SELECT * FROM r1 LEFT JOIN r2 ON r1.c1 = r2.c1;
```

	c1	c1	c2
▶	1	1	a
	2	2	b
	1	1	c
	3	3	d
	2	2	e
	1	1	f
	4	NULL	NULL

## joins – left outer join

The following example the LEFT JOIN of two tables  $r_1$ (AC-1, AC-2, AC-3, AC-4, AC-5) and  $r_2$ (C-1, C-2, C-3, C-4). The LEFT JOIN will match rows from the  $r_1$  table with the rows from  $r_2$  table using patterns:





## joins – left outer join

The **LEFT JOIN** keyword returns all rows from the left table ( $r_1$ ), with the matching rows in the right table ( $r_2$ ). The result is **NULL** in the right side when there is no match.

`SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1$  LEFT [OUTER] JOIN  $r_2$  ON  $r_1.A_1 = r_2.A_1$`

`SELECT * FROM orders ord LEFT OUTER JOIN employee emp ON emp.id = ord.employeeid;`



## joins – left outer join

- SELECT \* FROM student LEFT OUTER JOIN student\_order ON student.id = student\_order.studentid ;

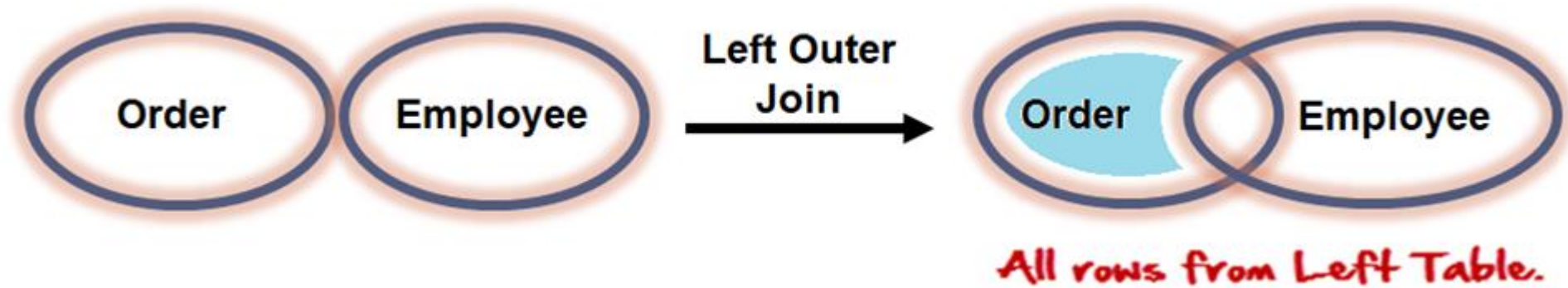
	ID	namefirst	namelast	DOB	emailID	ID	studentID	orderdate	amount
	6	lala	prasad	1980-12-01	lala.prasad@gmail.com	26	6	2019-02-02	280
	6	lala	prasad	1980-12-01	lala.prasad@gmail.com	30	6	2019-07-10	750
	7	sharmin	bagde	1986-12-14	sharmin.bagde@gmail.com	10	7	2019-10-10	2500
	7	sharmin	bagde	1986-12-14	sharmin.bagde@gmail.com	33	7	2019-06-23	945
	8	vrushali	bagde	1984-12-29	vrushali.bagde@gmail.com	21	8	2019-01-12	4500
	8	vrushali	bagde	1984-12-29	vrushali.bagde@gmail.com	40	8	2019-01-12	650
	9	vasant	khande	1992-10-26	vasant.khande@gmail.com	NULL	NULL	NULL	NULL
	10	nitish	patil	1990-10-26	nitish.patil@gmail.com	11	10	2019-11-11	150
	10	nitish	patil	1990-10-26	nitish.patil@gmail.com	22	10	2019-11-02	650
	10	nitish	patil	1990-10-26	nitish.patil@gmail.com	34	10	2019-01-19	225
	11	neel	save	1975-10-30	neel.save@gmail.com	NULL	NULL	NULL	NULL
	12	deep	save	1986-11-30	deep.save@gmail.com	5	12	2019-05-03	655
	12	deep	save	1986-11-30	deep.save@gmail.com	6	12	2019-05-04	1000
	12	deep	save	1986-11-30	deep.save@gmail.com	28	12	2019-02-02	45
	12	deep	save	1986-11-30	deep.save@gmail.com	29	12	2019-01-12	190
	13	nrupali	save	1981-12-01	nrupali.save@gmail.com	13	13	2019-11-02	655
	13	nrupali	save	1981-12-01	nrupali.save@gmail.com	36	13	2019-01-12	180
	14	supriya	karnik	1983-12-15	supriya.karnik@gmail.com	12	14	2019-07-21	340
	14	supriya	karnik	1983-12-15	supriya.karnik@gmail.com	35	14	2019-10-10	325
	15	bandish	karnik	1987-12-30	bandish.karnik@gmail.com	NULL	NULL	NULL	NULL
	16	sangita	karnik	1990-12-01	sangita.karnik@gmail.com	NULL	NULL	NULL	NULL
	17	sangita	menon	1989-10-26	sangita.menon@gmail.com	NULL	NULL	NULL	NULL
	18	rahul	shah	1982-06-12	rahul.shah@gmail.com	NULL	NULL	NULL	NULL

## joins – left outer join

The **LEFT JOIN** keyword returns all rows from the left table ( $r_1$ ), with the matching rows in the right table ( $r_2$ ). The result is **NULL** in the right side table when there is no match.

```
SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1$  LEFT [OUTER] JOIN  $r_2$  ON  $r_1.A_1 = r_2.A_1$  WHERE  $r_2.A_1$  IS NULL
```

```
SELECT * FROM orders ord LEFT OUTER JOIN employee emp ON emp.id = ord.employeeid WHERE emp.id IS NULL;
```



## joins – left outer join

- `SELECT * FROM student LEFT OUTER JOIN student_order ON student.id = student_order.studentid WHERE student_order.studentID IS NULL;`

	ID	namefirst	namelast	DOB	emailID	ID	studentID	orderdate	amount
▶	3	ulka	joshi	1970-10-25	ulka.joshi@gmail.com	NULL	NULL	NULL	NULL
	9	vasant	khande	1992-10-26	vasant.khande@gmail.com	NULL	NULL	NULL	NULL
	11	neel	save	1975-10-30	neel.save@gmail.com	NULL	NULL	NULL	NULL
	15	bandish	karnik	1987-12-30	bandish.karnik@gmail.com	NULL	NULL	NULL	NULL
	16	sangita	karnik	1990-12-01	sangita.karnik@gmail.com	NULL	NULL	NULL	NULL
	17	sangita	menon	1989-10-26	sangita.menon@gmail.com	NULL	NULL	NULL	NULL
	18	rahul	shah	1982-06-12	rahul.shah@gmail.com	NULL	NULL	NULL	NULL
	19	bhavin	patel	1983-11-13	bhavin.patel@gmail.com	NULL	NULL	NULL	NULL
	20	kaushal	patil	1982-07-30	kaushal.patil@gmail.com	NULL	NULL	NULL	NULL
	21	pankaj	gandhi	1982-07-30	pankaj.gandhi@gmail.com	NULL	NULL	NULL	NULL
	22	rajan	patel	1982-07-30	rajan.patel@gmail.com	NULL	NULL	NULL	NULL
	23	bhavin	patel	1982-07-30	bhavin.patel@gmail.com	NULL	NULL	NULL	NULL
	24	mukesh	bhavsar	1982-07-30	mukesh.bhavsar@gmail.com	NULL	NULL	NULL	NULL
	25	dilu	khande	1982-07-30	dilu.khande@gmail.com	NULL	NULL	NULL	NULL
	26	sonam	khan	1972-05-13	sonam.khan@gmail.com	NULL	NULL	NULL	NULL
	27	rohit	patil	1976-12-31	rohit.patil@gmail.com	NULL	NULL	NULL	NULL
	28	raj	bubber	1982-02-28	raj.bubber@gmail.com	NULL	NULL	NULL	NULL
	29	sharmin	patil	1999-11-10	sharmin.patil@gmail.com	NULL	NULL	NULL	NULL

## joins – left outer join

The **LEFT JOIN** keyword returns all rows from the left table ( $r_1$ ), with the matching rows in the right table ( $r_2$ ).  
**The result is NULL in the right side table when there is no match.**

SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1$  LEFT [OUTER] JOIN  $r_2$  USING ( $A_1, \dots$ )

- SELECT \* FROM emp LEFT OUTER JOIN dept USING(deptno);

SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1$  NATURAL LEFT [OUTER ] JOIN  $r_2$

- SELECT \* FROM emp NATURAL LEFT OUTER JOIN dept;

Suppose, we want to join two tables: r1 and r2. Right outer join returns all rows in the right table (r1) and all the matching rows found in the left table (r2). It means the result of the SQL right join always contains the rows in the right table. . If no matching rows found in the left table, NULL are displayed.

## right outer joins

$r_1 = \{1, 2, 3, 4\}$

$r_2 = \{(1, a), (2, b), (1, c), (3, d), (2, e), (1, f), (5, z)\}$

$r_1$  right join  $r_2$

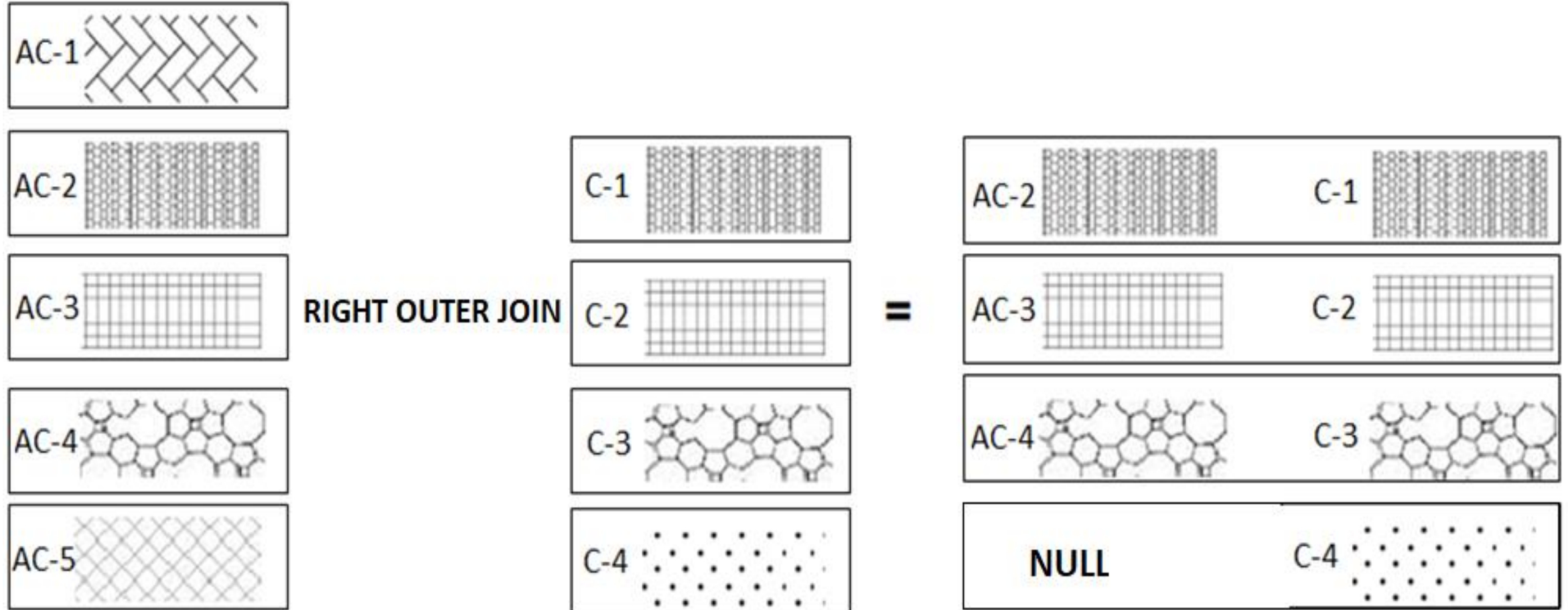
$R = \{(1, 1, a),$   
     $(1, 1, c),$   
     $(1, 1, f),$   
     $(2, 2, b),$   
     $(2, 2, e),$   
     $(3, 3, d),$   
     $(\text{NULL}, 5, z)\}$

`SELECT * FROM r1 RIGHT JOIN r2 ON r1.c1 = r2.c1;`

	c1	c1	c2
▶	1	1	a
	1	1	c
	1	1	f
	2	2	b
	2	2	e
	3	3	d
	NULL	5	z

## joins – right outer join

The following example the RIGHT OUTER JOIN of two tables  $r_1$ (AC-1, AC-2, AC-3, AC-4, AC-5) and  $r_2$ (C-1, C-2, C-3, C-4). The RIGHT JOIN will match rows from the  $r_1$  table with the rows from  $r_2$  table using patterns:

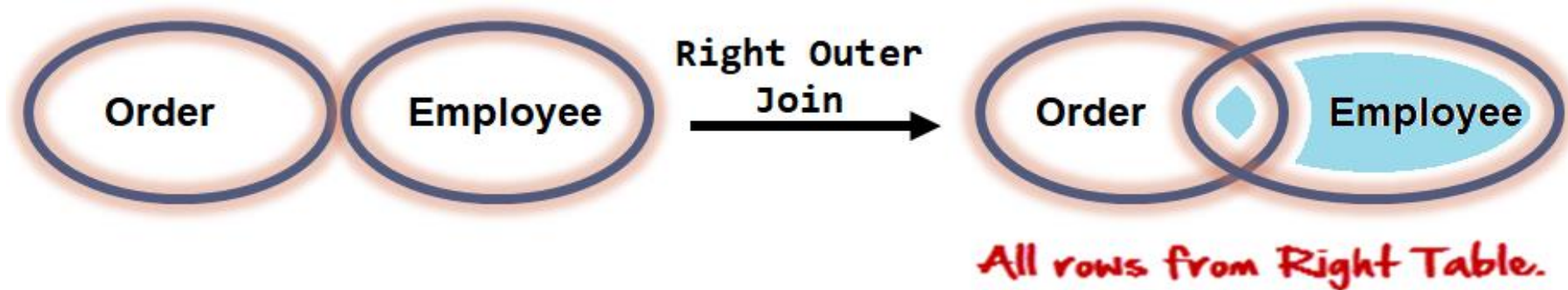


## joins – right outer join

The **RIGHT JOIN** keyword returns all rows from the right table ( $r_2$ ), with the matching rows in the left table ( $r_1$ ). The result is **NULL** in the left side table when there is no match.

```
SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1$  RIGHT [OUTER] JOIN  $r_2$  ON  $r_1.A_1 = r_2.A_1$ 
```

```
SELECT * FROM orders ord RIGHT OUTER JOIN employee emp ON emp.id = ord.employeeid;
```





## joins – right outer join

- `SELECT * FROM student RIGHT OUTER JOIN student_order ON student.id = student_order.studentid;`

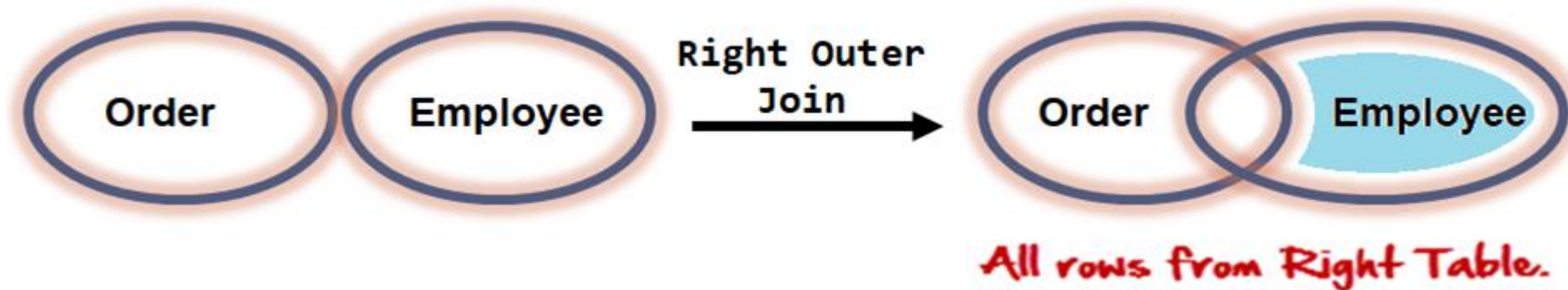
	ID	namefirst	namelast	DOB	emailID	ID	studentID	orderdate	amount
	6	lala	prasad	1980-12-01	lala.prasad@gmail.com	7	6	2019-11-11	4000
	1	saleel	bagde	1986-12-14	saleel.bagde@gmail.com	8	1	2019-07-19	1270
	5	ruhan	bagde	1984-01-12	ruhan.bagde@gmail.com	9	5	2019-04-07	2000
	7	sharmin	bagde	1986-12-14	sharmin.bagde@gmail.com	10	7	2019-10-10	2500
	10	nitish	patil	1990-10-26	nitish.patil@gmail.com	11	10	2019-11-11	150
	14	supriya	karnik	1983-12-15	supriya.karnik@gmail.com	12	14	2019-07-21	340
	13	nrupali	save	1981-12-01	nrupali.save@gmail.com	13	13	2019-11-02	655
	4	rahul	patil	1982-10-31	rahul.patil@gmail.com	14	4	2019-01-12	1000
	NULL	NULL	NULL	NULL	NULL	15	NULL	2019-04-07	4000
	NULL	NULL	NULL	NULL	NULL	16	NULL	2019-10-10	1270
	NULL	NULL	NULL	NULL	NULL	17	NULL	2019-11-11	4588
	NULL	NULL	NULL	NULL	NULL	18	NULL	2019-07-21	1200
	NULL	NULL	NULL	NULL	NULL	19	NULL	2019-11-02	125
	NULL	NULL	NULL	NULL	NULL	20	NULL	2019-01-12	350
	8	vrushali	bagde	1984-12-29	vrushali.bagde@gmail.com	21	8	2019-01-12	4500
	10	nitish	patil	1990-10-26	nitish.patil@gmail.com	22	10	2019-11-02	650
	4	rahul	patil	1982-10-31	rahul.patil@gmail.com	23	4	2019-10-19	700

## joins – right outer join

The **RIGHT JOIN** keyword returns all rows from the right table ( $r_2$ ), with the matching rows in the left table ( $r_1$ ). The result is **NULL** in the left side table when there is no match.

```
SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1$  RIGHT [OUTER] JOIN  $r_2$  ON  $r_1.A_1 = r_2.A_1$  WHERE  $r_1.A_1$  IS NULL
```

```
SELECT * FROM orders ord RIGHT OUTER JOIN employee emp ON emp.id = ord.employeeid WHERE ord.employeeid IS NULL;
```



## joins – right outer join

- `SELECT * FROM student RIGHT OUTER JOIN student_order ON student.id = student_order.studentid WHERE student.ID IS NULL;`

	ID	namefirst	namelast	DOB	emailID	ID	studentID	orderdate	amount
▶	NULL	NULL	NULL	NULL	NULL	15	NULL	2019-04-07	4000
	NULL	NULL	NULL	NULL	NULL	16	NULL	2019-10-10	1270
	NULL	NULL	NULL	NULL	NULL	17	NULL	2019-11-11	4588
	NULL	NULL	NULL	NULL	NULL	18	NULL	2019-07-21	1200
	NULL	NULL	NULL	NULL	NULL	19	NULL	2019-11-02	125
	NULL	NULL	NULL	NULL	NULL	20	NULL	2019-01-12	350

## *joins – right outer join*

The **RIGHT JOIN** keyword returns all rows from the right table ( $r_2$ ), with the matching rows in the left table ( $r_1$ ).  
**The result is NULL in the left side table when there is no match.**

```
SELECT  $A_1$ ,  $A_2$ ,  $A_3$ , . . . FROM  $r_1$  RIGHT [OUTER] JOIN  $r_2$  USING ( $A_1$ , . . .)
```

```
SELECT * FROM emp RIGHT OUTER JOIN dept USING(deptno);
```

```
SELECT  $A_1$ ,  $A_2$ ,  $A_3$ , . . . FROM  $r_1$  NATURAL RIGHT [OUTER ] JOIN  $r_2$ 
```

```
SELECT * FROM emp NATURAL RIGHT OUTER JOIN dept;
```

**LEFT JOIN**

Vs

**RIGHT JOIN**

## *left join vs right join*

LEFT OUTER JOIN	RIGHT OUTER JOIN	FULL OUTER JOIN
All the tuples of the left table remain in the result.	All the tuples of the right table remain in the result.	All the tuples from left as well as right table remain in the result.
The tuples of left table that does not have a matching tuple in right table are extended with NULL value for attributes of the right table.	The tuples of right table that does not have a matching tuple in left table are extended with NULL value for attributes of the left table.	The tuples of left as well as the right table that does not have the matching tuples in the right and left table respectively are extended with NULL value for attributes of the right and left tables.

TODO

self joins

TODO

## joins – self join

A **SELF JOIN** is a join in which a table is joined with itself (which is also called Unary relationships), especially when the table has a FOREIGN KEY which references its own PRIMARY KEY.

```
SELECT  $r_x.A_1, r_x.A_2, r_y.A_1, r_y.A_2, \dots$  FROM  $r_1 r_x, r_1 r_y$  WHERE  $r_x.A_1 = r_y.A_1$ 
```

```
SELECT distinct e1.* FROM emp e1 , emp e2 WHERE e1.sal = e2.sal AND e1.empno != e2.empno ORDER BY e1.sal;
```

## Remember:

- A subquery must be enclosed in parentheses.
- Use single-row operators with single-row subqueries, and use multiple-row operators with multiple-row subqueries.
- If a subquery (inner query) returns a null value to the outer query, the outer query will not return any rows when using certain comparison operators in a **WHERE** clause.
- If **ORDER BY** occurs within a subquery and also is applied in the outer query, the outermost **ORDER BY** takes precedence.
- If **LIMIT** occurs within a subquery and also is applied in the outer query, the outermost **LIMIT** takes precedence.

## sub-queries

A subquery is a **SELECT** statement within another statement.

## Note:

- You may use comparison operators such as **<>**, **<**, **>**, **<=**, and **>=** with a single row subquery.
- Multiple row subquery returns one or more rows to the outer SQL statement. You may use the **IN**, **ANY**, or **ALL** operator in outer query to handle a subquery that returns multiple rows.



A subquery is a **SELECT** statement within another statement.

## subqueries

### Remember:

A subquery may occur in:

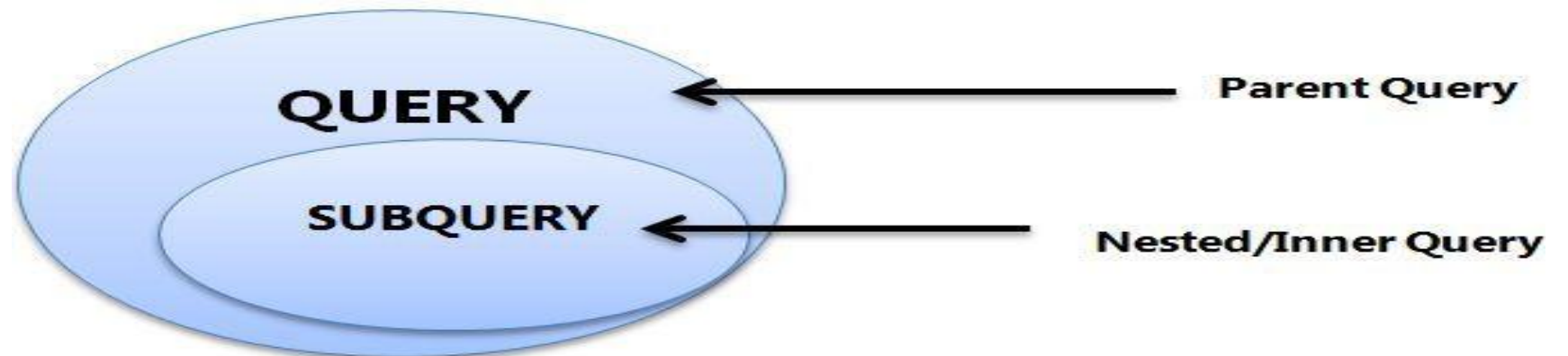
- a **SELECT** clause
- a **FROM** clause
- a **WHERE** clause
- a **HAVING** clause

### Note:

A subquery's outer statement can be any one of:

- **SELECT**
- **INSERT**
- **UPDATE**
- **DELETE**
- **CREATE**

- **INSERT ... SELECT ...**
- **UPDATE ... SELECT ...**
- **DELETE ... SELECT ...**
- **CREATE TABLE ... AS SELECT ...**
- **CREATE VIEW ... AS SELECT ...**
- **DECLARE CURSOR ... AS SELECT ...**
- **EXPLAIN SELECT ...**



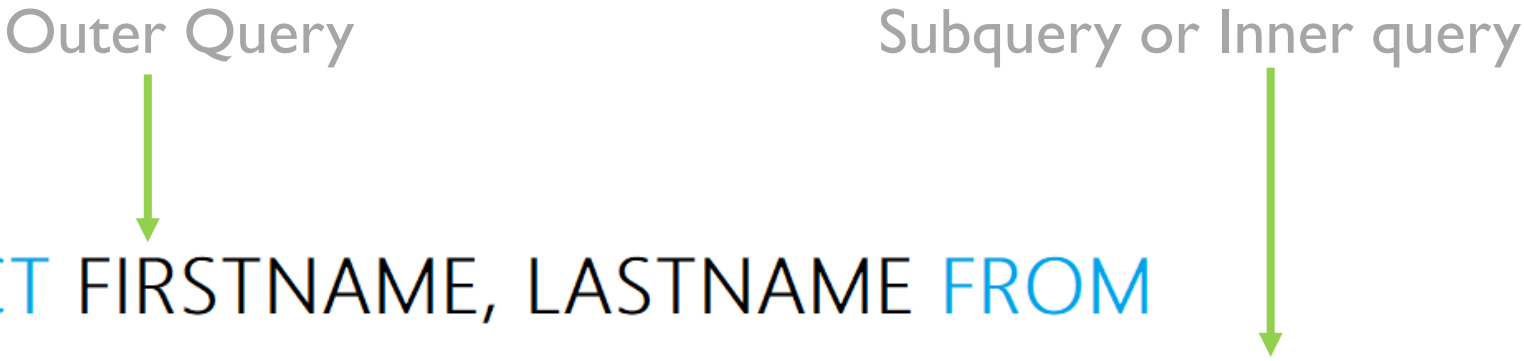
## single row subqueries

A single row subquery returns **zero or one row** to the outer SQL statement. You can place a subquery in a **WHERE** clause, a **HAVING** clause, or a **FROM** clause of a SELECT statement.

Outer Query

Subquery or Inner query

```
SELECT FIRSTNAME, LASTNAME FROM  
EMP WHERE DEPTNO = (SELECT DEPTNO FROM EMP  
                     WHERE DNAME = 'SALES')
```



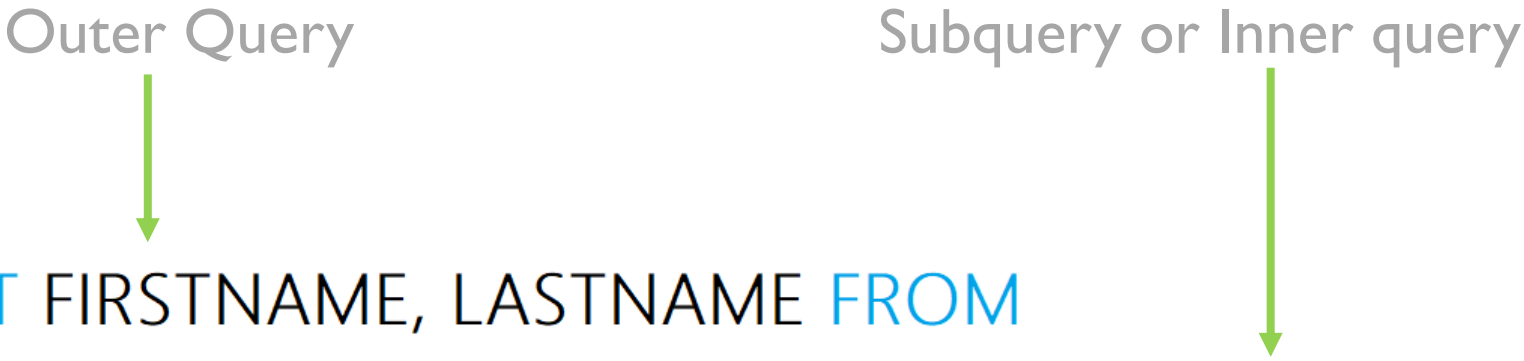
## *multiple row and column subqueries*

A multiple row subquery returns **one or more rows** to the outer SQL statement. You may use the **IN**, **ANY**, or **ALL** operator in outer query to handle a subquery that returns multiple rows.

Outer Query

Subquery or Inner query

```
SELECT FIRSTNAME, LASTNAME FROM  
EMP WHERE DEPTNO IN (SELECT DEPTNO FROM DEPT  
                     WHERE DNAME = 'RESEARCH' OR  
                     DNAME = 'SALES')
```



## *subqueries with insert*

TODO

```
INSERT INTO table_name [(column1 [, column2 ])] (subquery)
```

```
INSERT INTO EMPLOYEE SELECT * FROM EMP
```

```
INSERT INTO EMPLOYEE (FIRSTNAME, LASTNAME)  
SELECT FIRSTNAME, LASTNAME FROM EMP
```

# *subqueries with update*

TODO

```
UPDATE table_name SET column_name = (subquery) [WHERE (subquery)]
```

# *subqueries with delete*

TODO

```
DELETE FROM table_name [ WHERE (subquery) ]
```

# types of subqueries

- The Subquery as Scalar Operand – SELECT clause
- Comparisons using Subqueries – WHERE / HAVING clause (*Single row subquery*)
- Subqueries in the FROM Clause – INLINE VIEWS (*Derived Tables*)
- Subqueries with ALL, ANY, IN, or SOME – WHERE / HAVING clause (*Multiple row subquery*)
- Subqueries with EXISTS or NOT EXISTS

WITH var(param) as (SELECT) [CET] Common Table Expressions

WITH a(p1, p2, p3, p4) AS (SELECT \* FROM dept) SELECT p1, p2, p3, p4 FROM a;

the subquery as scalar operand



# *the subquery as scalar operand*

TODO

SELECT  $A_1$ ,  $A_2$ ,  $A_3$ , (*subquery*) as  $A_4$ , . . . FROM  $r$

## Remember:

- A scalar subquery is a subquery that returns **exactly one column value from one row**.
- A scalar subquery is a simple operand, and you can use it almost anywhere a single column value is legal.

## Note:

- If the subquery returns 0 rows then the value of scalar subquery expression is **null**.
- if the subquery returns more than one row then MySQL returns an **error**.

## Think:

- SELECT (SELECT 1, 2); #error
- SELECT (SELECT ename, sal FROM emp); #error
- SELECT (SELECT \* FROM emp); #error
- SELECT (SELECT NULL + 1);
- SELECT ename, (SELECT dname FROM dept WHERE emp.deptno = dept.deptno) R1 FROM emp ;

## *the subquery as scalar operand*

e.g.

- `SELECT (SELECT stdprice FROM price WHERE prodid = 100890 AND enddate IS NOT NULL) AS "Standard Price",  
(SELECT minprice FROM price WHERE prodid = 100890 AND enddate IS NOT NULL) AS "Minimum Price";`

	Standard Price	Minimum Price
	54.00	40.50

- `SELECT (SELECT stdprice FROM price WHERE prodid = 100890 AND enddate IS NOT NULL) - (SELECT minprice FROM price WHERE prodid = 100890 AND enddate IS NOT NULL) AS "Price Difference";`

	Price Difference
	13.50

subquery in the from clause

# subqueries in the from clause

TODO

SELECT  $A_1, A_2, A_3, (subquery) \text{ as } A_4, \dots$  FROM  $(subquery) [AS] \text{ name}, \dots$

## Note:

- Every table in a FROM clause must have a name, therefore the [AS] name clause is mandatory.
- SET @x := 0;  
SELECT \* FROM (SELECT @x := @x + 1 as R1, emp.\* FROM emp) DT WHERE R1 = 5;
- SELECT \* FROM (SELECT @cnt := @cnt + 1 R1, MOD(@cnt,2) R2, emp.\* FROM emp, (SELECT @cnt:=0) DT1 ) DT2 WHERE R2 = 0;
- SELECT MIN(R1) FROM (SELECT COUNT(job) R1 FROM emp GROUP BY job) DT;
- SELECT MAX(R1) FROM (SELECT COUNT(\*) R1 FROM actor\_movie GROUP BY actorid) DT

	MAX(R1)
	5

- SELECT AVG(SUM(column1)) FROM t1 GROUP BY column1; //ERROR
- SELECT AVG(sum\_column1)  
FROM (SELECT SUM(column1) AS sum\_column1  
FROM t1 GROUP BY column1) AS t1;

comparisons using subquery

# comparisons using subqueries

TODO

Comparison Operators like : =, !=/<>, >, >=, <, <= ,<=>

SELECT  $A_1$ ,  $A_2$ ,  $A_3$ , (*subquery*) as  $A_4$ , . . . FROM  $r$  WHERE  $p$  = (*subquery*)

## Remember:

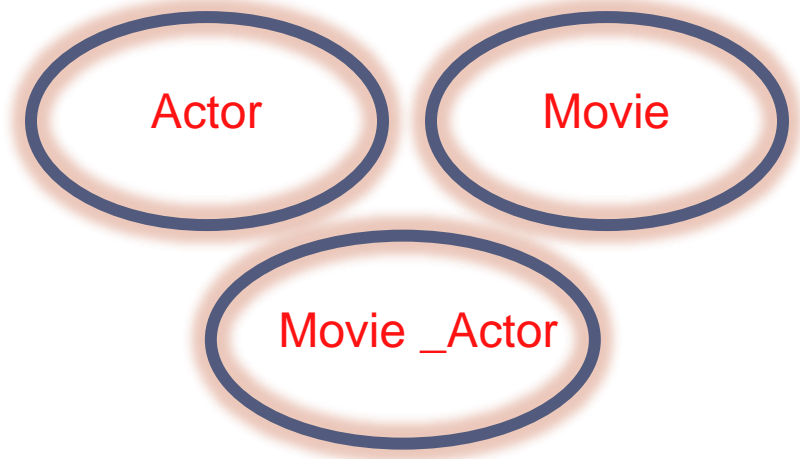
- A subquery can be used before or after any of the comparison operators.
- The subquery can return **at most one value**.
- The value can be the result of an arithmetic expression or a function.
  
- SELECT \* FROM emp WHERE deptno = (SELECT 5 + 5);
- SELECT \* FROM emp WHERE sal = (SELECT MAX(sal) FROM emp);
- SELECT MAX(sal) FROM emp WHERE sal < (SELECT MAX(sal) FROM emp);
- SELECT \* FROM emp WHERE sal > (SELECT sal FROM emp GROUP BY sal ORDER BY sal DESC limit 3, 1) ORDER BY sal DESC;

## Following statements will raise an error.

- SELECT \* FROM emp WHERE deptno = (SELECT deptno FROM dept WHERE deptno IN(10, 20));

# comparisons using subqueries

movie : (movieid, name, release\_date)  
actor : (actorid, name)  
actor\_movie : (actorid, movieid)



- `SELECT a.actorid, a.name FROM actor a, actor_movie am WHERE a.actorid = am.actorid GROUP BY am.actorid HAVING COUNT(*) = (SELECT MAX(R1) FROM (SELECT COUNT(*) "R1" FROM actor_movie GROUP BY actorid) M);`

	actorid	name
	2	Akshav Kumar
	3	Salman Khan
	7	Madhuri Dixit

## Remember:

- When used with a subquery, the word IN is an alias for =ANY
- NOT IN is not an alias for <>ANY, but for <>ALL

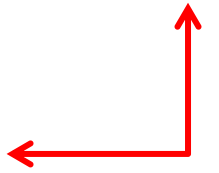
subquery with in, all, any, some



# subqueries with in, all, any, some

The word SOME is an alias for ANY.

- *operand comparison\_operator* **ANY** (*subquery*)
  - *operand* **IN** (*subquery*)
  - *operand comparison\_operator* **SOME** (*subquery*)
  - *operand comparison\_operator* **ALL** (*subquery*)
- 
- The **ANY** keyword, which must follow a comparison operator, means return TRUE if the comparison is TRUE for ANY of the values in the column that the subquery returns.
  - The word **ALL**, which must follow a comparison operator, means return TRUE if the comparison is TRUE for ALL of the values in the column that the subquery returns.
  - **IN** and **=ANY** are **not synonyms** when used with an expression list. **IN** can take an expression list, but **= ANY** cannot.
- 
- `SELECT * FROM emp WHERE deptno IN (5 + 5, 10 + 10);`
  - `SELECT * FROM emp WHERE job =ANY (SELECT job FROM emp WHERE deptno IN(10, 20)) ;`
  - `SELECT * FROM emp WHERE deptno =ANY (10, 20); //error`



# subqueries with in, all, any, some

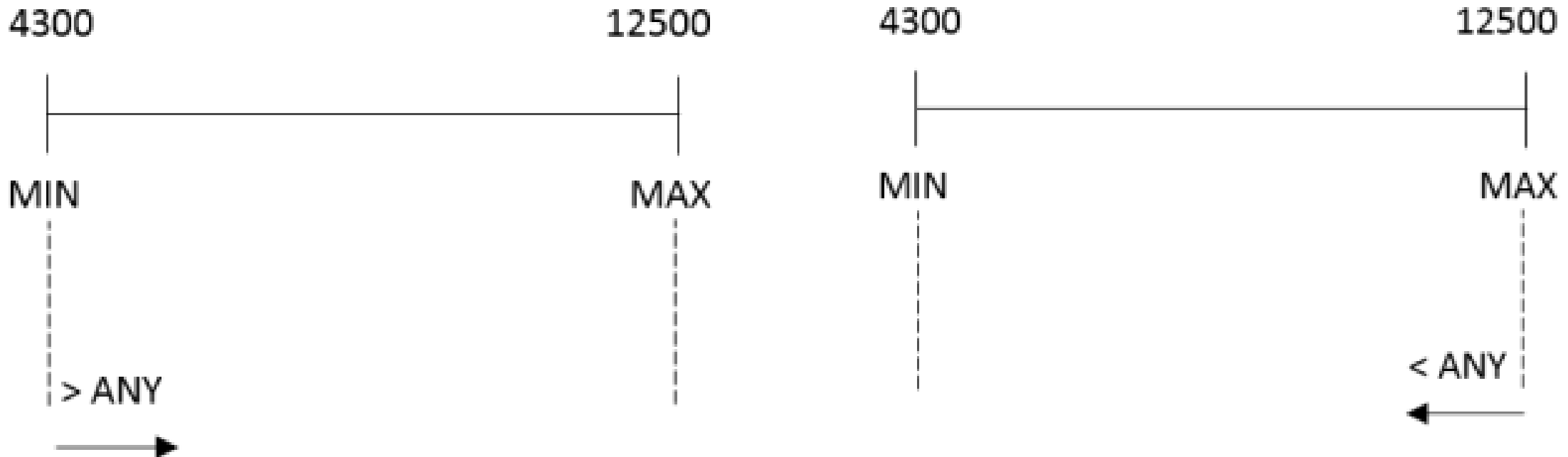
TODO

```
SELECT  $A_1$ ,  $A_2$ ,  $A_3$ , (subquery) as  $A_4$ , . . . FROM (subquery) [AS] Alias WHERE  $p$  IN (subquery)
SELECT  $A_1$ ,  $A_2$ ,  $A_3$ , (subquery) as  $A_4$ , . . . FROM (subquery) [AS] Alias WHERE  $p$  ANY (subquery)
SELECT  $A_1$ ,  $A_2$ ,  $A_3$ , (subquery) as  $A_4$ , . . . FROM (subquery) [AS] Alias WHERE  $p$  ALL (subquery)
```

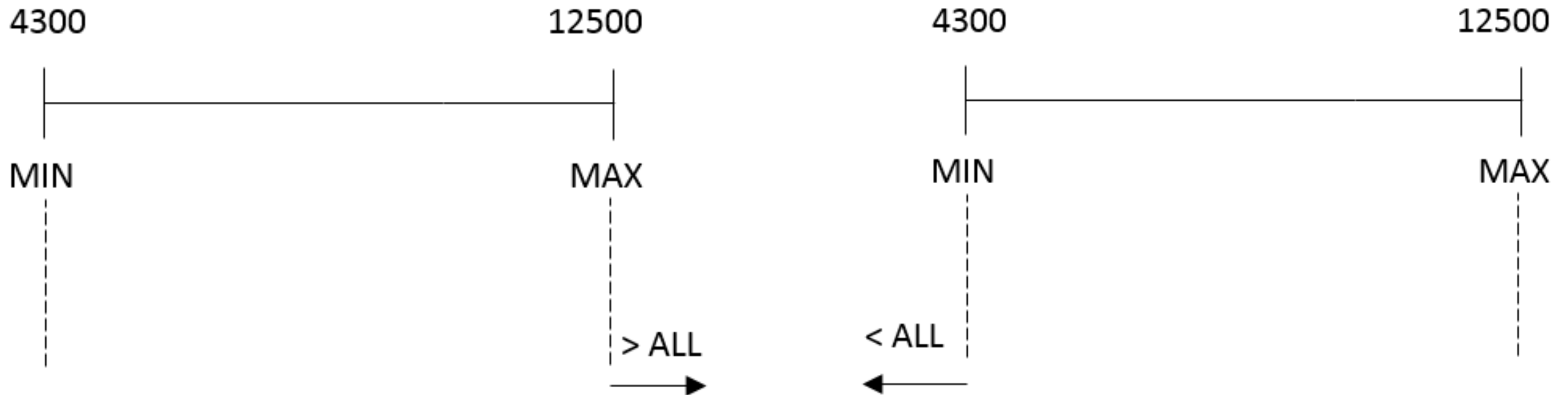
- Remember:
- `SELECT * FROM emp WHERE deptno = SELECT deptno FROM dept WHERE dname = 'SALES';` // error
- `SELECT * FROM emp WHERE deptno IN SELECT deptno FROM dept WHERE dname = 'SALES';` // error
- `SELECT * FROM emp WHERE deptno IN SELECT * FROM dept WHERE dname = 'SALES';` // error

## *any / some*

- "x = ANY (...)": The value must match one or more values in the list to evaluate to TRUE.
- "x != ANY (...)": The value must not match one or more values in the list to evaluate to TRUE.
- "x > ANY (...)": The value must be greater than the smallest value in the list to evaluate to TRUE.
- "x < ANY (...)": The value must be smaller than the biggest value in the list to evaluate to TRUE.
- "x >= ANY (...)": The value must be greater than or equal to the smallest value in the list to evaluate to TRUE.
- "x <= ANY (...)": The value must be smaller than or equal to the biggest value in the list to evaluate to TRUE.



- "x = ALL (...)": The value must match all the values in the list to evaluate to TRUE.
- "x != ALL (...)": The value must not match any values in the list to evaluate to TRUE.
- "x > ALL (...)": The value must be greater than the biggest value in the list to evaluate to TRUE.
- "x < ALL (...)": The value must be smaller than the smallest value in the list to evaluate to TRUE.
- "x >= ALL (...)": The value must be greater than or equal to the biggest value in the list to evaluate to TRUE.
- "x <= ALL (...)": The value must be smaller than or equal to the smallest value in the list to evaluate to TRUE.



*all*

The expression is TRUE, if table T2 is empty.

```
SELECT * FROM EMP WHERE DEPTNO >ALL (SELECT C1 FROM T2) //
```



This statement will return all rows from EMP table.

# *subqueries with in, all, any, some*

You can use a subquery after a comparison operator, followed by the keyword IN, ALL, ANY, or SOME.

- `SELECT * FROM emp WHERE deptno IN (SELECT deptno FROM emp WHERE deptno = 10 OR deptno = 20);`
- `SELECT * FROM emp WHERE sal >ALL (SELECT sal FROM emp WHERE ename = 'ADAMS' OR ename = 'TURNER');`
- `SELECT * FROM emp WHERE sal >ANY (SELECT sal FROM emp WHERE ename = 'ADAMS' OR ename = 'TURNER');`
- `SELECT * FROM emp WHERE sal >SOME (SELECT sal FROM emp WHERE ename = 'ADAMS' OR ename = 'TURNER');`
- `SELECT * FROM server WHERE id =ANY (SELECT id FROM runningserver);`
- `SELECT * FROM server WHERE id <ALL (SELECT id FROM runningserver);`
- `SELECT * FROM server WHERE id <ANY (SELECT id FROM runningserver);`

```
SELECT * FROM dI WHERE cI not in (SELECT min(cI) FROM dI GROUP BY deptno, dname, loc, walletid) ORDER BY deptno;
```

- `SELECT * FROM emp WHERE EXISTS (SELECT 1);`

subquery with exists or not exists

# *subqueries with exists or not exists*

The EXISTS operator tests for the existence of rows in the results set of the subquery. If a subquery row value is found, EXISTS subquery returns TRUE and in this case NOT EXISTS subquery will return FALSE.

`SELECT  $A_1$ ,  $A_2$ ,  $A_3$ ,  $A_4$ , . . . FROM  $r$  WHERE [NOT] EXISTS (subquery)`

The records will be displayed from outer SELECT statement....

- `SELECT * FROM emp WHERE EXISTS (SELECT * FROM dept WHERE emp.deptno = dept.deptno);`
- `SELECT * FROM dept WHERE NOT EXISTS (SELECT deptno FROM emp WHERE emp.deptno = dept.deptno);`
- `SELECT * FROM emp m WHERE EXISTS (SELECT * FROM emp e WHERE e.mgr = m.empno);`
- `SELECT * FROM emp m WHERE NOT EXISTS (SELECT * FROM emp e WHERE e.mgr = m.empno);`
- `SELECT * FROM dept WHERE deptno IN (SELECT * FROM emp WHERE emp.deptno = dept.deptno);`
- `SELECT * FROM dept WHERE deptno NOT IN (SELECT * FROM emp WHERE emp.deptno = dept.deptno);`
- `SELECT * FROM emp f WHERE NOT EXISTS (SELECT * FROM emp m WHERE f.deptno = m.deptno AND gender = 'm');`
- `SELECT * FROM emp m WHERE NOT EXISTS (SELECT true FROM emp f WHERE m.deptno = f.deptno AND f.gender = 'f');`



correlated subquery

# correlated subqueries

A correlated subquery (**also known as a synchronized subquery**) is a subquery that uses values from the outer query. The subquery is evaluated once for each row processed by the outer query.

Following query find all employees who earn more than the average salary in their department.

- `SELECT * FROM emp e WHERE sal > (SELECT AVG(sal) FROM emp WHERE e.deptno = emp.deptno) ORDER BY deptno;`
- `SELECT ename, sal, job FROM emp WHERE sal > (SELECT AVG(sal) FROM emp E WHERE e.job = emp.job);`
- `SELECT job, MAX(sal) FROM emp WHERE sal < (SELECT MAX(sal) FROM emp E WHERE emp.job = e.job GROUP BY e.job) GROUP BY job;`
- `SELECT DISTINCTROW deptno FROM emp WHERE EXISTS (SELECT deptno FROM dept WHERE emp.deptno = dept.deptno); (Intersect)`
- `SELECT DISTINCTROW deptno FROM DEPT WHERE NOT EXISTS (SELECT deptno FROM emp WHERE emp.deptno = dept.deptno);`

Write query to find all employees who have same salary.

- `SELECT e1.* FROM emp e1 WHERE sal IN (SELECT e2.sal FROM emp e2 WHERE e1.sal = e2.sal AND e1.empno <> e2.empno);`

# truncate table

## Remember:

- DROP and TRUNCATE are DDL commands, whereas DELETE is a DML command.
  - DELETE operations can be rolled back, while DROP and TRUNCATE operations cannot be rolled back.
  - The TRUNCATE TABLE statement removes all the data/rows of a table and resets the auto-increment value to zero.
-

# *truncate table*

Logically, TRUNCATE TABLE is similar to a DELETE statement that deletes all rows, or a sequence of DROP TABLE and CREATE TABLE statements.

TRUNCATE [TABLE] tbl\_name

## Remember:

- Truncate operations drop and re-create the table, which is much faster than deleting rows one by one.
  - Truncate operations cause an implicit commit, and so cannot be rolled back.
  - Truncate does not cause ON DELETE triggers to fire.
  - Truncate cannot be performed for parent-child foreign key relationships.
  - Truncate retain Identity and reset to the seed (start value) value.
  - Cannot truncate a table referenced in a foreign key constraint.
  - Any AUTO\_INCREMENT value is reset to its start value.
-

**DELETE**

Vs

**TRUNCATE**

## *delete vs truncate*

DELETE	TRUNCATE
You can specify the tuple that you want to delete.	It deletes all the tuples from a relation.
DELETE is a Data Manipulation Language command.	TRUNCATE is a Data Definition Language command.
DELETE command can have WHERE clause.	TRUNCATE command do not have WHERE clause.
DELETE command activates the trigger applied on the table and causes them to fire.	TRUNCATE command does not activate the triggers to fire.
DELETE command eliminate the tuples one-by-one.	TRUNCATE delete the entire data page containing the tuples.
DELETE command lock the row/tuple before deleteing it.	TRUNCATE command lock data page before deleting table data.
DELETE command acts slower as compared to TRUNCATE.	TRUNCATE is faster as compared to DELETE.
DELETE records transaction log for each deleted tuple.	TRUNCATE record transaction log for each deleted data page.
DELETE command can be followed either by COMMIT or ROLLBACK.	TRUNCATE command can't be ROLLBACK.

**DROP**

**Vs**

**TRUNCATE**

## *drop vs truncate*

### **DROP**

The DROP command is used to remove table definition and its contents.

In the DROP command, VIEW of table does not exist.

In the DROP command, integrity constraints will be removed.

In the DROP command, INDEX associated to the table will be removed.

In the DROP command, TRIGGER associated to the table will execute.

In the DROP command, TRIGGER associated to the table will be removed.

### **TRUNCATE**

Whereas the TRUNCATE command is used to delete all the rows from the table.

In the TRUNCATE command, VIEW of table exist.

In the TRUNCATE command, integrity constraints will not be removed.

In the TRUNCATE command, INDEX associated to the table will not be removed.

In the TRUNCATE command, TRIGGER associated to the table will not execute.

In the TRUNCATE command, TRIGGER associated to the table will be not removed.

# rename table

Change the table name.

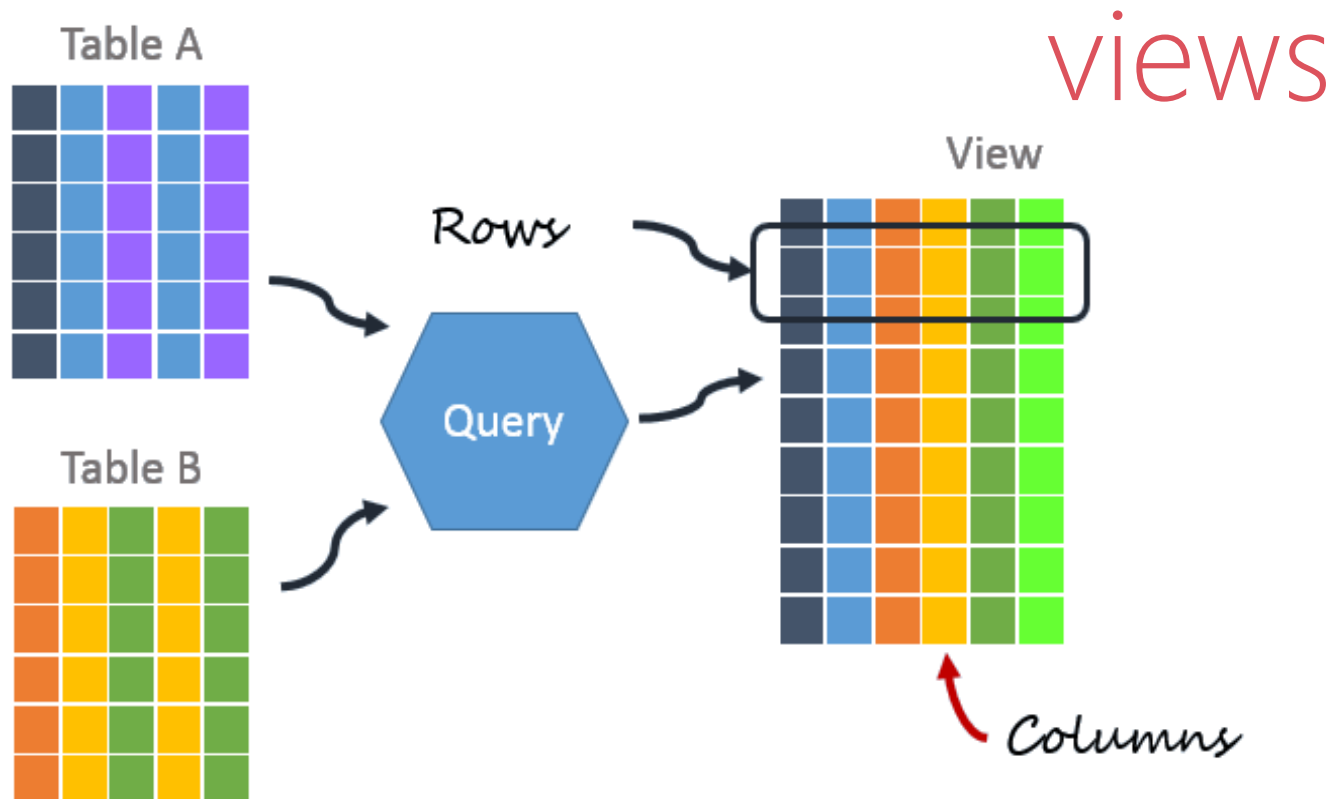
```
RENAME TABLE old_tbl_name TO new_tbl_name
```

- `RENAME TABLE emp TO employee;`

A **VIEW** in SQL as a logical subset of data from one or more tables. Views are used to restrict data access. A **VIEW** contains no data of its own but its like window through which data from tables can be viewed or changed. The table on which a View is based are called BASE Tables.

There are 2 types of Views in SQL:

- **Simple View** : Simple views can only contain a single base table.
- **Complex View** : Complex views can be constructed on more than one base table. In particular, complex views can contain: join conditions, a group by clause, a order by clause.







## Remember:

## view

- it can be described as **virtual/derived** table which derived its data from one or more than one base table.
- View names may appear in a query in any place where a relation name may appear.
- it is stored in the database.
- it can be created using tables of same database or different database.
- it is used to implement the security mechanism in the SQL.
- It can have max 64 char (view name).

## Rules:

- If a VIEW is defined as SELECT \* on a table, new columns added to the base table later do not become part of the VIEW, and columns dropped from the base table will result in an error when selecting from the VIEW.
- A VIEW must have unique column names with no duplicates, just like a base table. By default, the names of the columns retrieved by the SELECT statement are used for the VIEW column names.
- The VIEW definition cannot refer to a TEMPORARY table, and you cannot create a TEMPORARY VIEW.
- You cannot associate a TRIGGER with a VIEW.

## Note:

- If we drop the BASE TABLE, the VIEW will not be dropped.
- A VIEW definition (structure) is not permanently stored as part of the database.

- DESC dept;
- CREATE VIEW v1 AS SELECT \* FROM dept;
- DROP TABLE dept;
- DESC v1;

Try ↑ this

Uses of a View: A good database should contain views due to the following reasons:

- **Restricting data access** – Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.
- **Hiding data complexity** – A view can hide the complexity that exists in multiple tables join.
- **Simplify commands for the user** – Views allow the user to select information from multiple tables without requiring the users to actually know how to perform a join.
- **Store complex queries** – Views can be used to store complex queries.
- **Rename Columns** – Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of columns specified in select statement. Thus, renaming helps to hide the names of the columns of the base tables.
- **Multiple view facility** – Different views can be created on the same table for different users.

Views are not updatable in the following cases:

- A table in the FROM clause is reference by a subquery in the WHERE statement.
- There is a subquery in the SELECT clause.
- The SQL statement defining the view joins tables.
- One of the tables in the FROM clause is a non-updatable view.
- The SELECT statement of the view contains an aggregate function such as SUM(), COUNT(), MAX(), MIN(), and so on.
- The keywords DISTINCT, GROUP BY, HAVING clause, LIMIT clause, UNION, or UNION ALL appear in the defining SQL statement.

## *difference between simple and complex view*

SIMPLE VIEW	COMPLEX VIEW
Contains only one base table or is created from only one table, it may include WHERE clause and ORDER BY clause.	Contains more than one base tables or is created from more than one tables.
We cannot use group functions like MAX(), COUNT(), etc.	We can use group functions.
Does not contain groups of data.	It can contain groups of data.
DML operations could be performed through a simple view.	DML operations could not always be performed through a complex view.
INSERT, DELETE and UPDATE are directly possible on a simple view.	We cannot apply INSERT, DELETE and UPDATE on complex view directly.
Simple view does not contain group by, , having clause, limit clause, distinct, pseudo column like rownum, columns defined by expressions.	It can contain group by, having clause, limit clause, distinct, pseudocolumn like rownum, columns defined by expressions.
Does not include NOT NULL columns from base tables.	NOT NULL columns that are not selected by simple view can be included in complex view.

## create view

The select\_statement is a SELECT statement that provides the definition of the view. The select\_statement can select from base tables or other views.

```
CREATE [ OR REPLACE ] VIEW view_name [ (column_list) ]  
    AS select_statement [ WITH CHECK OPTION ]
```

UPDATE AND DELETE on VIEW (with check option given on view) will work only when the DATA MATCHES IN WHERE CLAUSE.

- `CREATE VIEW v1 as SELECT * FROM dept;`
- `CREATE VIEW v1(A1, A2) as SELECT deptno, dname FROM dept;`
- `CREATE or REPLACE VIEW v1 as SELECT * FROM dept WITH CHECK OPTION;`

```
desc INFORMATION_SCHEMA.VIEWS;
```

# licence table

create view

	ID	customerID	licenceClass	licenceType	licenceNumber	ValidFrom	ValidTo	agentID
▶	1	1	LMV	Light motor vehicles including motorcars, jeeps, taxis, delivery vans	RQFY8GZND9E	*****	*****	1
	2	2	LMV	Light motor vehicles including motorcars, jeeps, taxis, delivery vans	ZFVHYH4OAHGS	*****	*****	2
	3	1	HMV	Heavy Motor Vehicles	20CMZQ6ERXJ4	*****	*****	3
	4	2	HMV	Heavy Motor Vehicles	DNIE1K9R5BG6	*****	*****	1
	5	3	LMV	Light motor vehicles including motorcars, jeeps, taxis, delivery vans	Y3CNWQ6MKPCG	*****	*****	2
	6	3	HMV	Heavy Motor Vehicles	Y6DC2IRD2EWZ	*****	*****	1
	7	4	HGMV	Heavy Goods Motor Vehicle	2EFSXY04OAW	*****	*****	3
	8	5	HMV	Heavy Motor Vehicles	GC1BMWKCTVQ4	*****	*****	2
	9	6	LMV	Light motor vehicles including motorcars, jeeps, taxis, delivery vans	2148PC50IR3P	*****	*****	1
	10	7	HPMV	Heavy passenger motor vehicle/Heavy transport vehicle	N4NODNCFJENP	*****	*****	2
	11	8	HGMV	Heavy Goods Motor Vehicle	EBPW9M34FKOI	*****	*****	2
	12	9	HPMV	Heavy passenger motor vehicle/Heavy transport vehicle	940MGEL4HO00	*****	*****	3
	13	10	LMV	Light motor vehicles including motorcars, jeeps, taxis, delivery vans	JWRDAETHDU5Z	*****	*****	1
	14	11	HGMV	Heavy Goods Motor Vehicle	Z95XCS7W9NAU	*****	*****	2
	15	12	HGMV	Heavy Goods Motor Vehicle	1YJ12NQX8HAJ	*****	*****	1
	16	13	LMV	Light motor vehicles including motorcars, jeeps, taxis, delivery vans	1XMTETH8E6JS	*****	*****	1
	17	14	HPMV	Heavy passenger motor vehicle/Heavy transport vehicle	0WEMMTPVTBPM	*****	*****	2
✱	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

## # all are Simple View

- CREATE or REPLACE VIEW agent1\_view AS SELECT \* FROM licence WHERE agentID = 1 ORDER BY validTo;
- CREATE or REPLACE VIEW agent2\_view AS SELECT \* FROM licence WHERE agentID = 2;
- CREATE or REPLACE VIEW agent3\_view AS SELECT \* FROM licence WHERE agentID = 3;