

A day without new  
knowledge is a lost day.

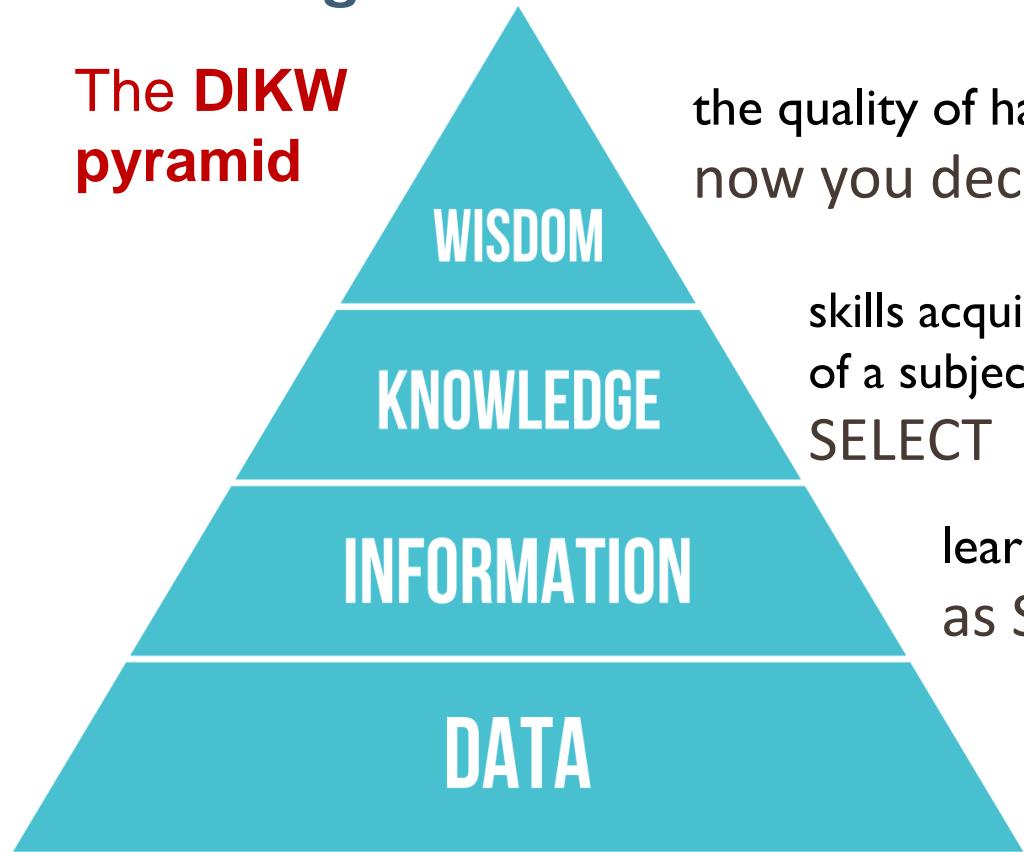
## *Database Technologies – MySQL*

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)

**DIKW Model** describes how the data can be processed and transformed into **information**, **knowledge**, and **wisdom**.

The **DIKW** pyramid



the quality of having experience, knowledge, and good judgement.  
now you decide when to use and how to use(**wisdom**)

skills acquired through education, it can be theoretical or practical understanding  
of a subject.  
SELECT potential (**knowledge**)

learned about something or someone.  
as SELECT (symbol:  $\sigma$ ) (**information**)

Sigma( $\sigma$ ) (**data**)

Let's take another example.

## wisdom

- Our employees Ben, Jo, and Tom are young.
- Our employees Ben, Jo, and Tom are in 20's
- Toy is going to retire next year.
- ....

## knowledge

Age of employees:

$$R = \{ \text{Ben is 25 yrs. Old}, \text{Jo is 29 yrs. Old}, \text{Kim is 45 yrs. Old}, \text{Tom is 23 yrs. Old}, \text{Toy is 60 yrs. Old}, \dots \}$$

## Information

Now we arrange the data and make some sense and create information.

set A is **Age**, set B is **Name**

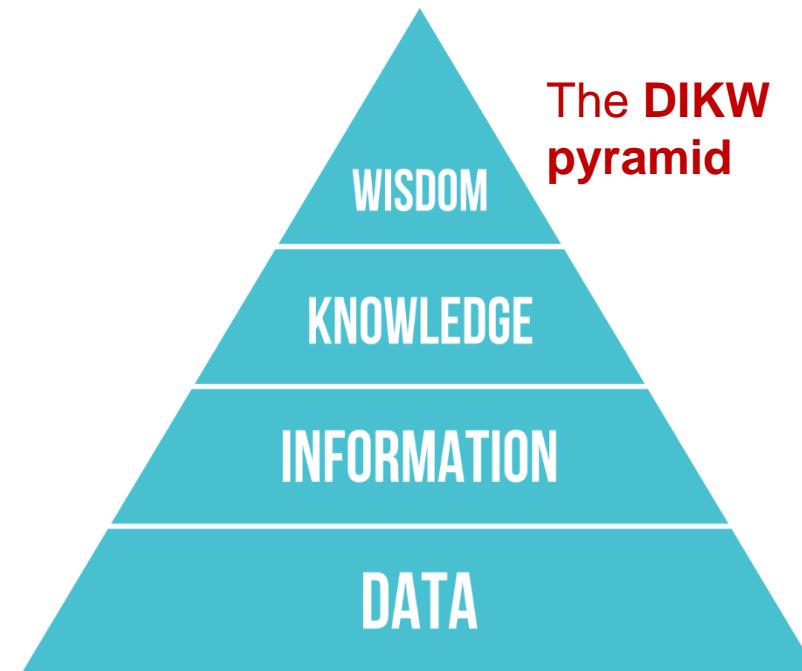
$$\text{Age} - \{ 25, 29, 45, 23, 60, 51, 35, \dots \}$$

$$\text{Name} - \{ \text{Ben}, \text{Jo}, \text{Kim}, \text{Tom}, \text{Toy}, \text{Sam}, \text{Don}, \dots \}$$

## data

$$\text{set A} - \{ 25, 29, 45, 23, 60, 51, 35, \dots \}$$

$$\text{set B} - \{ \text{Ben}, \text{Jo}, \text{Kim}, \text{Tom}, \text{Toy}, \text{Sam}, \text{Don}, \dots \}$$



**Class Room**

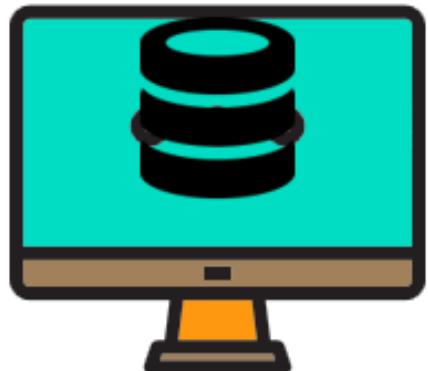
# **Session 1**

# Introduction

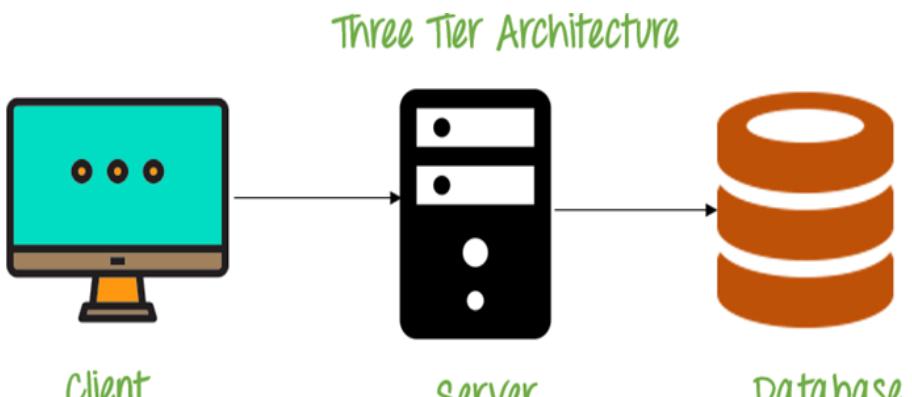
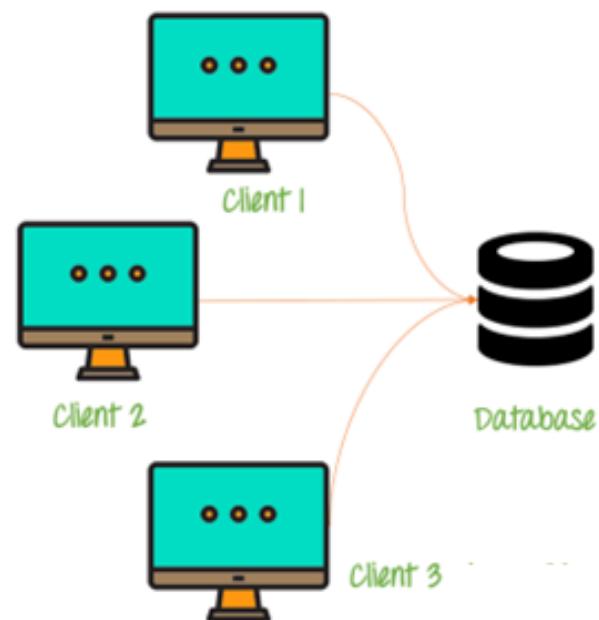
- If anyone who wants to develop a good application then he should have the knowledge three major components.

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 }



Single Tier Architecture



# Introduction

- A layer refers to pieces of software that are logically separated, but typically live within the same process and machine.
- A tier, instead, refers to pieces of software that live in distinct processes or AppDomains or machines.
- A tier refers to physical separation; a layer is about logical separation.

# Introduction

## 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 collects data from different resource (like Weather data, Geographical data, Finance data, Scientific data, Transport data, Cultural data, etc.)

Cultural means: the ideas, customs, and social behaviour of a particular people or society..

# **Introduction**

## **4 Important Roles of Database in Industry.**

- It is needed for data access within the company.
- It is needed to maintain strong relationships between data.
- This system allows newer(latest) and better updates.
- It helps to search data in a better manner.

# 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 represents a **table** or an **entity** than contain attributes.

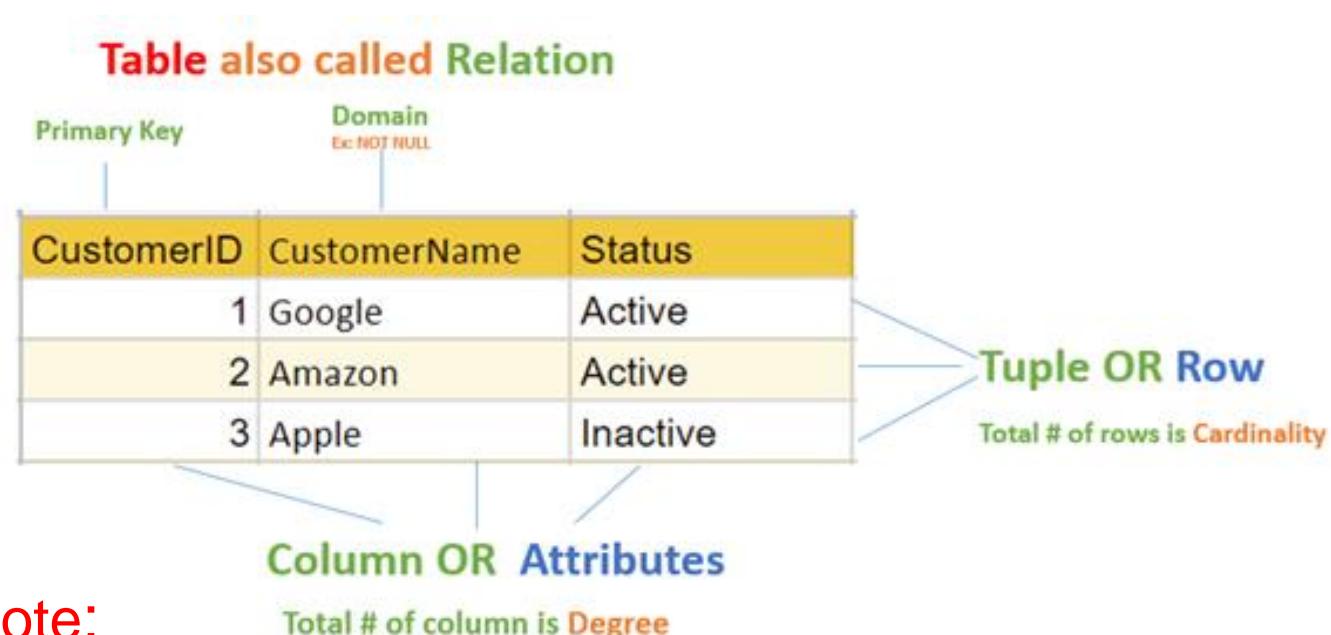
**Relationship:** In database, relationship is that how the two entities are **connected** to each other, i.e. what kind of relationship type they hold between them.

**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.)



**Note:**

- **Table** - The physical instantiation of a relation in the database schema.
- **Relation** - A logical construct that organizes data into rows and columns.

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

```

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

```

file attributes

- File Name
- Type
- Location

file permissions

- File permissions
- Share permissions

search empl ID=1

```

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

```

search emp\_name

```

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

```

# *file-oriented system*

A **flat file** database is a database that stores data in a plain text **file** (e.g. **\*.txt**, **\*.csv** format). Each line of the text **file** holds one record, with fields separated by delimiters, such as **commas** or **tabs**.

1 rajan MG Road Pune MH 34500

2 rahul patil SSG Lane Pune MH 54000

3 suraj raj k Deccan Gymkhana Pune MH 22000

4, S M Kumar, Mg Road Pune MH, 32000

5, S M Kumar, Mg Road, Pune, MH, 32000

1,raj,k,1984-06-12,raj.kumar@gmail.com

2,om,,1969-10-25,om123@gmail.com

3,rajes,kumar,1970-10-25,

4,rahul,patil,1982-10-31,rahul.patil@gmail.com

5,ketan,,,ruhan.bagde@gmail.com

The Zen of Python,

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than \*right\* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

# *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.

A **database application** is a computer program whose primary purpose is entering and retrieving information.

## What is database?

A major purpose of a database system is to provide users with an **abstract view** of the data.

**abstract means** existing in thought or as an idea but not having a physical or concrete existence.



## *what is database?*

A database is a system to **organize**, **store** and **retrieve** large amounts of data easily, which is stored in **one** or **more data files** by **one** or **more users**.

Each database is a collection of tables, which are called **relations**, hence the name "**relational database**".



**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.



## Difference between File System and DBMS

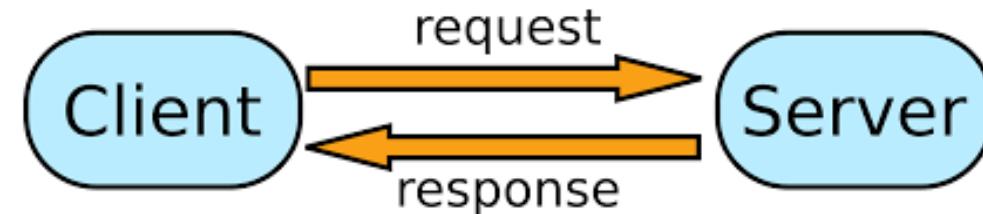
File Management System	Database Management System
• File System is easy-to-use system to store data which require less security and constraints.	• Database Management System is used when security constraints are high.
• Data Redundancy is more in File System.	• Data Redundancy is less in Database Management System.
• Data Inconsistency is more in File System.	• Data Inconsistency is less in Database Management System.
• Centralization is hard to get when it comes to File System.	• Centralization is achieved in Database Management System.
• User locates the physical address of the files to access data in File System.	• In Database Management System, user is unaware of physical address where data is stored.
• Security is low in File System.	• Security is high in Database Management System.
• File System stores unstructured data. "unstructured data" may include documents, audio, video, images, etc.	• Database Management System stores structured data.

# *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 device that provides service to another computer program, also known as the client.
- In the client/server programming model, a server program awaits and fulfills requests from client programs, which might be running in the same, or other computers.

# *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.

# difference between dbms and rdbms

DBMS	RDBMS
<ul style="list-style-type: none"><li>• Data is stored as file.</li></ul>	<ul style="list-style-type: none"><li>• Data is stored as tables.</li></ul>
<ul style="list-style-type: none"><li>• There is no relationship between data in DBMS.</li></ul>	<ul style="list-style-type: none"><li>• Data is present in multiple tables which can be related to each other.</li></ul>
<ul style="list-style-type: none"><li>• DBMS has no support for distributed databases.</li></ul>	<ul style="list-style-type: none"><li>• RDBMS supports distributed databases.</li></ul>
<ul style="list-style-type: none"><li>• Normalization cannot be achieved.</li></ul>	<ul style="list-style-type: none"><li>• Normalization can be achieved.</li></ul>
<ul style="list-style-type: none"><li>• DBMS supports single user at a time.</li></ul>	<ul style="list-style-type: none"><li>• RDBMS supports multiple users at a time.</li></ul>
<ul style="list-style-type: none"><li>• Data Redundancy is common in DBMS.</li></ul>	<ul style="list-style-type: none"><li>• Data Redundancy can be reduced in RDBMS.</li></ul>
<ul style="list-style-type: none"><li>• DBMS provides low level of security during data manipulation.</li></ul>	<ul style="list-style-type: none"><li>• RDBMS has high level of security during data manipulation.</li></ul>

# Codd's Rules

## Rule 1: Information Rule

The data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format.

## Rule 2: Guaranteed Access Rule

Every single data is guaranteed to be accessible with a combination of table-name, primary-key (row value), and attribute-name (column value).

## Rule 3: Systematic Treatment of NULL Values

The NULL values in a database must be given a systematic and uniform treatment. This is a very important rule because a NULL can be interpreted as one the following – data is missing, data is not known, or data is not applicable.

## Rule 4: Active Online Catalog

The structure description of the entire database must be stored in an online catalog, known as **data dictionary**, which can be accessed by authorized users.

## Rule 5: Comprehensive Data Sub-Language Rule

A database can only be accessed using a language having linear syntax that supports data definition, data manipulation, and transaction management operations. This language can be used directly or by means of some application. If the database allows access to data without any help of this language, then it is considered as a violation.

## Rule 6: View Updating Rule

All the views of a database, which can theoretically be updated, must also be updatable by the system.

## Rule 7: High-Level Insert, Update, and Delete Rule

A database must support high-level insertion, updation, and deletion. This must not be limited to a single row, that is, it must also support union, intersection and minus operations to yield sets of data records.

## Rule 8: Physical Data Independence

The data stored in a database must be independent of the applications that access the database. Any change in the physical structure of a database must not have any impact on how the data is being accessed by external applications.

## Rule 9: Logical Data Independence

The logical data in a database must be independent of its user's view (application). Any change in logical data must not affect the applications using it. For example, if two tables are merged or one is split into two different tables, there should be no impact or change on the user application. This is one of the most difficult rule to apply.

## Rule 10: Integrity Independence

A database must be independent of the application that uses it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes a database independent of the front-end application and its interface.

## Rule 11: Distribution Independence

The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. This rule has been regarded as the foundation of distributed database systems.

## Rule 12: Non-Subversion Rule

If a system has an interface that provides access to low-level records, then the interface must not be able to subvert the system and bypass security and integrity constraints.

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 a 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.



ACID

# *ACID properties of transactions*

**Atomicity.** In a transaction involving two or more separate pieces of information, either all of the pieces are committed or none are.

For example, in an application that transfers funds from one account to another, the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.

**Consistency.** A transaction either creates a new and valid state of data, or, if any failure occurs, returns all data to its state before the transaction was started.

For example, in an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.

**Isolation.** A transaction in process and not yet committed must remain isolated from any other transaction.

For example, If multiple users attempt to transfer money at the same time, the transactions should be isolated from one another to prevent conflicts or inconsistencies. Each transaction should be treated as if it is the only transaction being executed at that time.

**Durability.** Committed data is saved by the system such that, even in the event of a failure and system restart, the data is available in its correct state.

For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.

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.***

What is Entity Type?

## *entity type*

The entities that have the **common attributes** is called an **entity type**.

Each entity type in the database is described by a **name** and **a list of attributes**.

**e.g.** an entity Person is an entity type that has *Age*, *Name* and *Address* attributes.

**Eg.**

**Entity TYPE**

*Person (Age, Name, Address, ...)*

**Entity**

*17 , Sharmin, Paud Road, ...*

When you are designing attributes for your entities, **you will sometimes find that an attribute does not have a value**. For example, you might want an attribute for a person's middle name, but you can't require a value because some people have no middle name. For these, you can define the attribute so that it can contain null values.

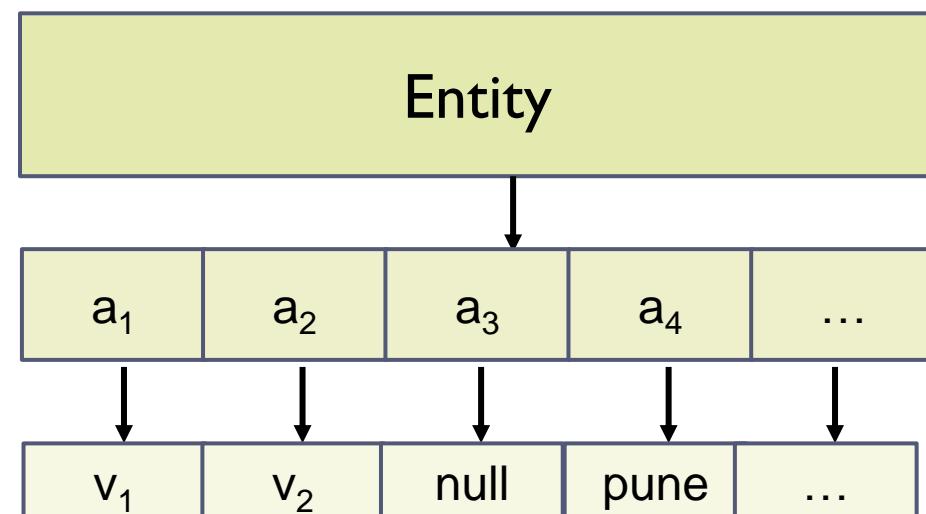
In database management systems, **null** is used to represent missing or unknown data in a table column.

## What is an Attribute?

Attributes are the properties that define a relation.

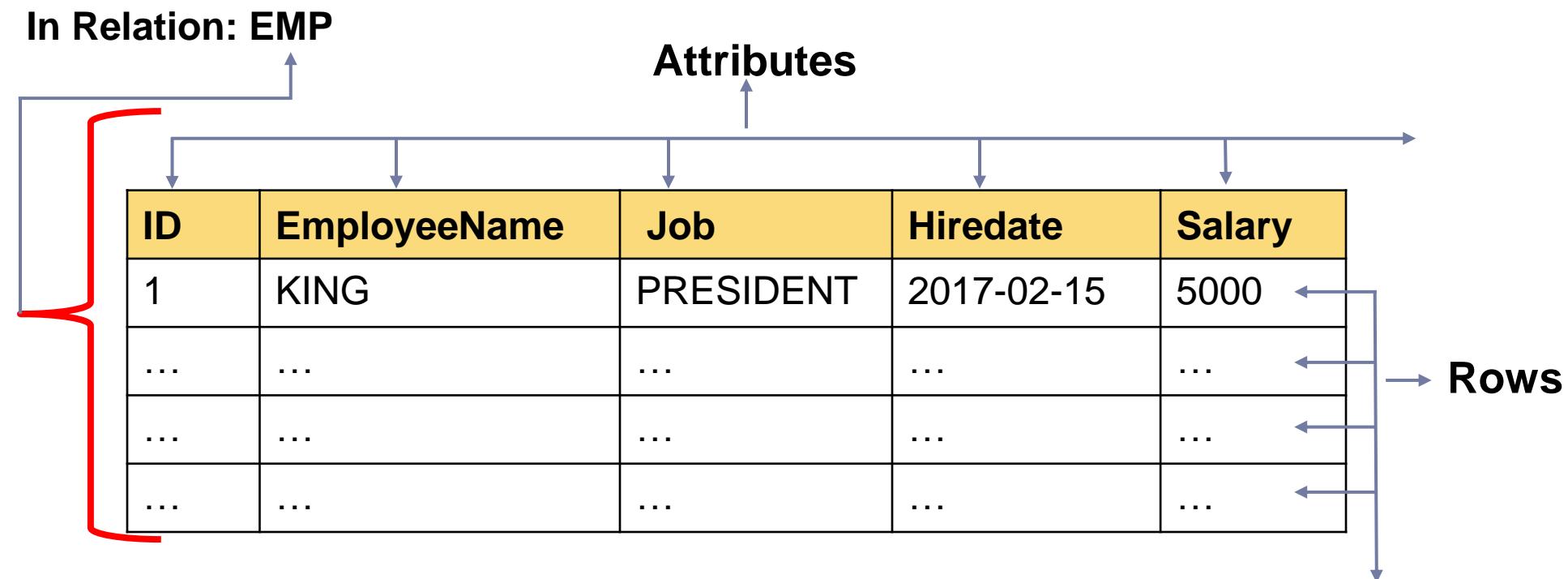
e.g. student(ID, firstName, middleName, lastName, city)

**In some cases, you might not want a specific attribute to contain a null value**, but you don't want to require that the user or program always provide a value. In this case, a default value might be appropriate. **A default value is a value that applies to an attribute if no other valid value is available.**



# A table has rows and columns

In RDBMS, a table organizes data in rows and columns. The **COLUMNS** are known as **ATTRIBUTES / FIELDS** whereas the **ROWS** are known as **RECORDS / TUPLE**.



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)		

- **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.

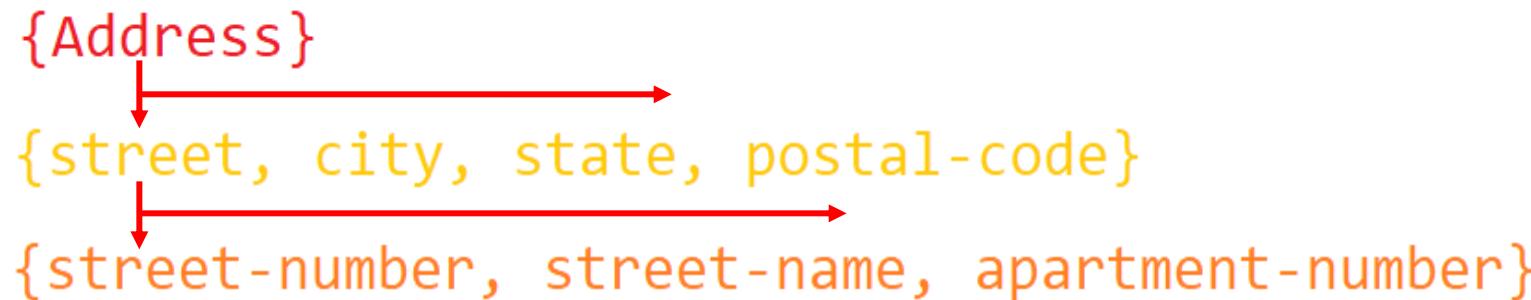
# Composite VS Multi Valued Attribute

# *composite / multi valued attributes*

## 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@yahoo.com, ... ]

What is an Prime, Non-Prime  
Attribute?

## Prime attribute (*Entity integrity*)

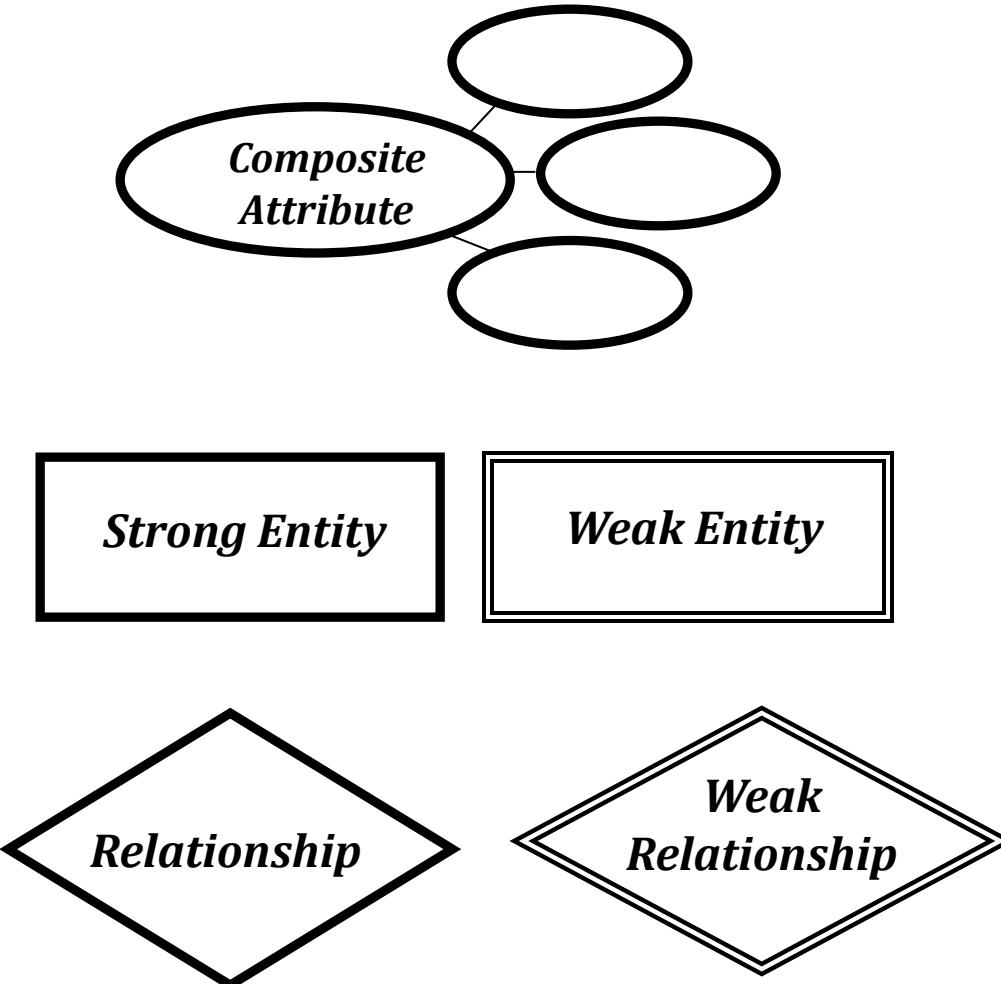
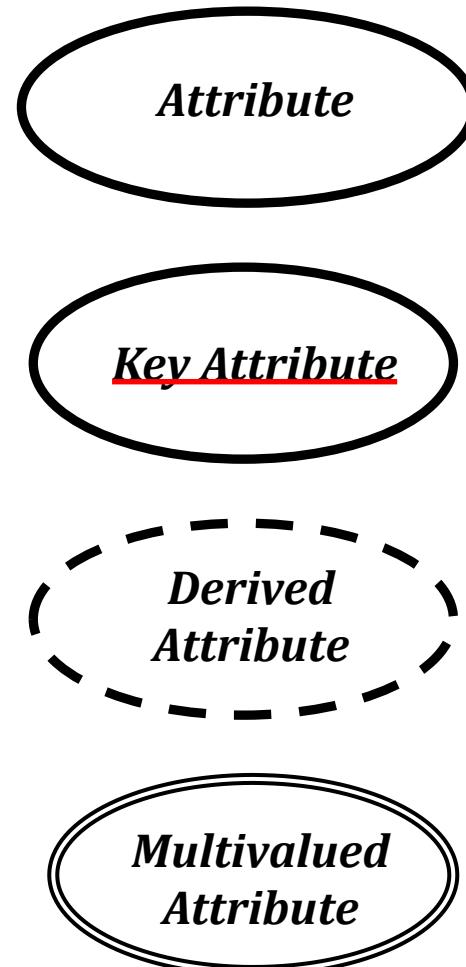
An attribute, which is a **part of the prime-key** (candidate key), is known as a prime attribute.

## 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

## *participation constraints*

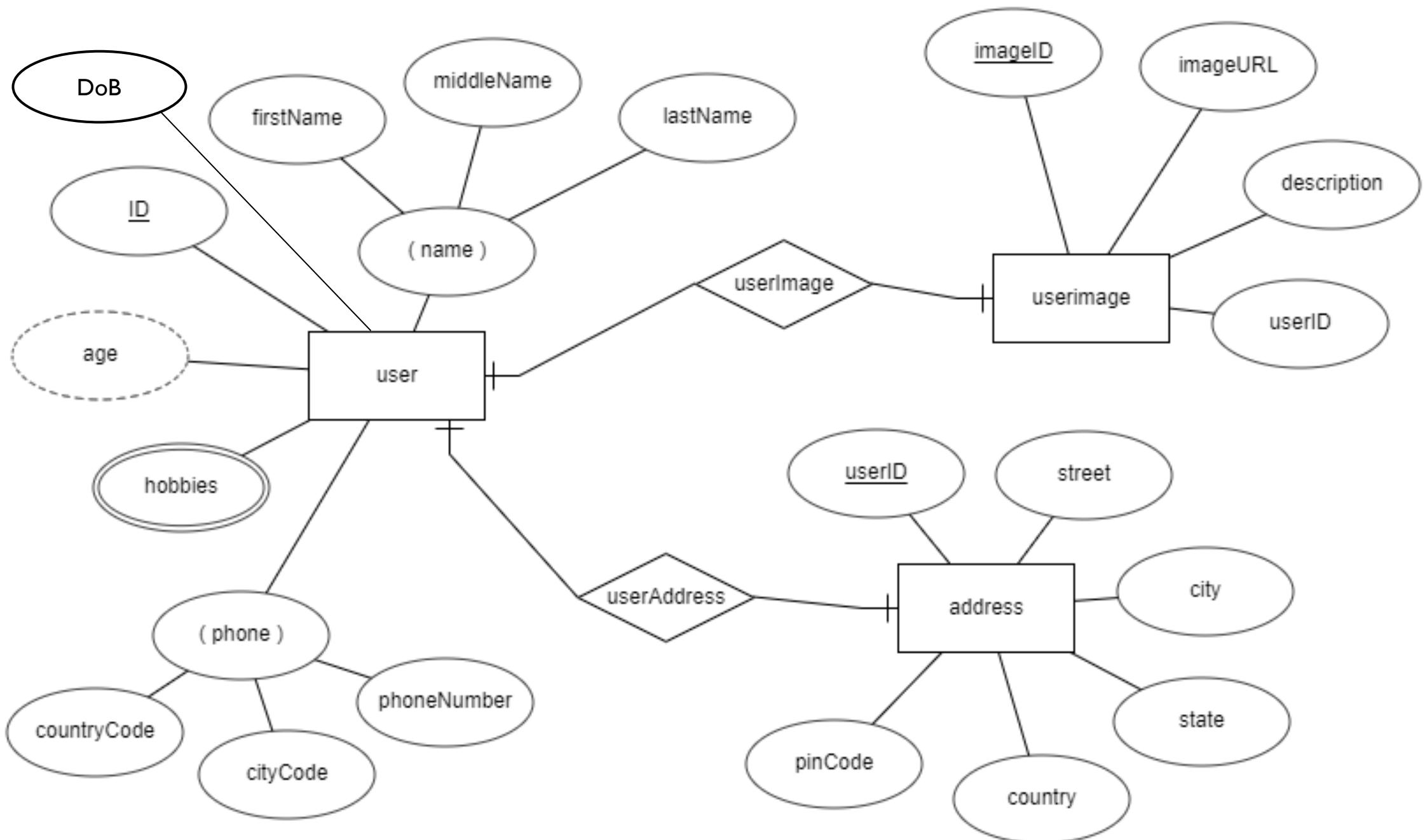
An entity may participate in a relation either totally or partially.

**Total participation** means that every entity is involved in the relationship, e.g., each student must be guided by a professor (there are no students who are not guided by any professor). This kind of relation is represented as a double line.

**Partial participation** means that not all entities are involved in the relationship, e.g., not every professor guides a student (there are professors who don't). A partial participation is represented by a single line.



# *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 in 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  $t_1[\text{FK}] = t_2[\text{PK}]$  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..

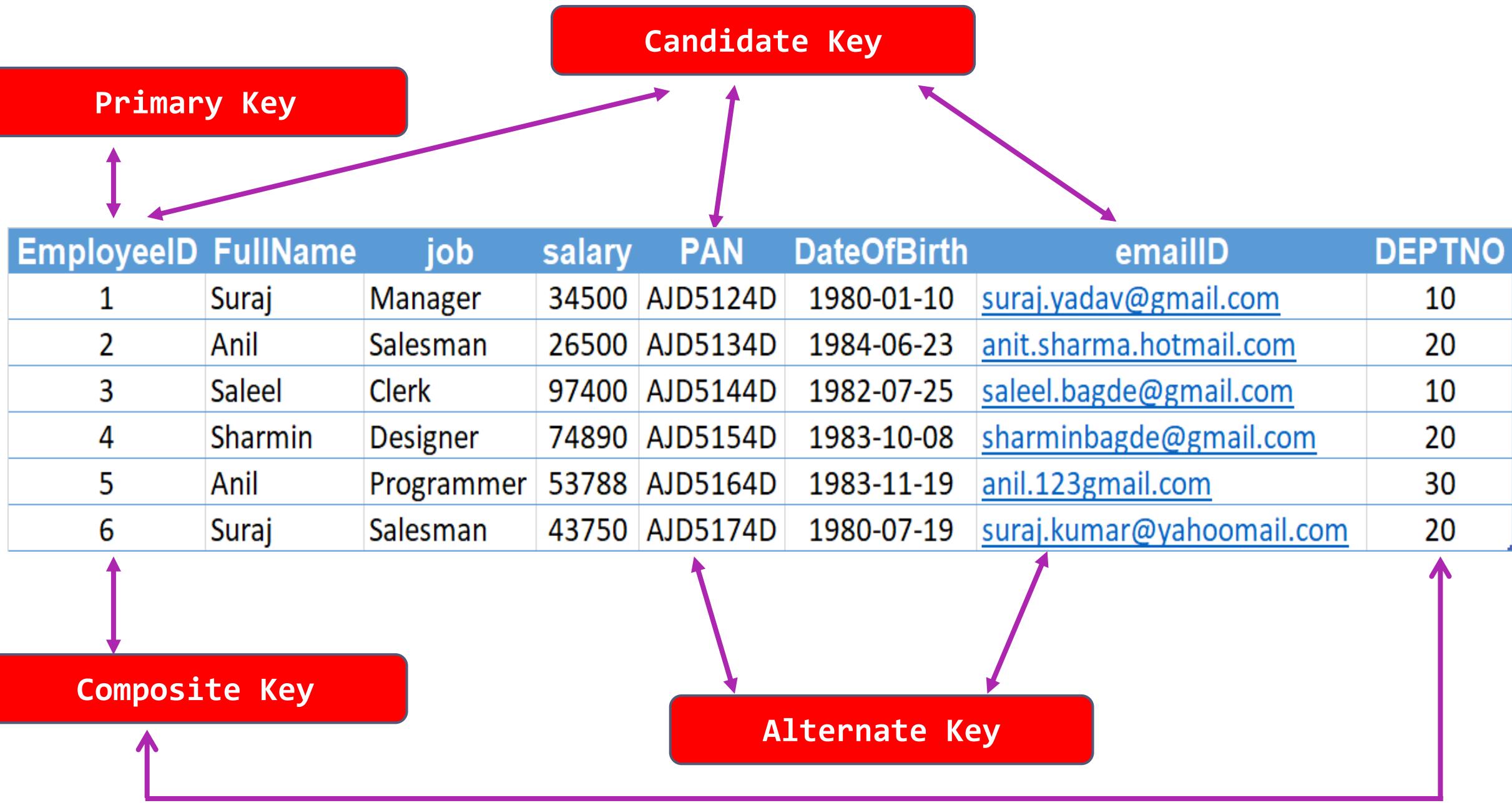
types of Keys?

Keys are used to establish relationships between tables and also to uniquely identify any record in the table.

## *types of Keys?*

$r = \text{Employee}(\text{EmployeeID}, \text{FullName}, \text{job}, \text{salary}, \text{PAN}, \text{DateOfBirth}, \text{emailID}, \text{deptno})$

- **Candidate Key:** are individual columns in a table that qualifies for uniqueness of all the rows. Here in Employee table EmployeeID, PAN or emailID are Candidate keys.
- **Primary Key:** is the columns you choose to maintain uniqueness in a table. Here in Employee table you can choose either EmployeeID, PAN or emailID columns, EmployeeID is preferable choice.
- **Alternate Key:** Candidate column other the primary key column, like if EmployeeID is primary key then , PAN or emailID columns would be the Alternate key.
- **Super Key:** If you add any other column to a primary key then it become a super key, like EmployeeID + FullName is a Super Key.
- **Composite Key:** If a table do not have any single column that qualifies for a Candidate key, then you have to select 2 or more columns to make a row unique. Like if there is no EmployeeID, PAN or emailID columns, then you can make FullName + DateOfBirth as Composite key. But still there can be a narrow chance of duplicate row.



# Degrees of relationship

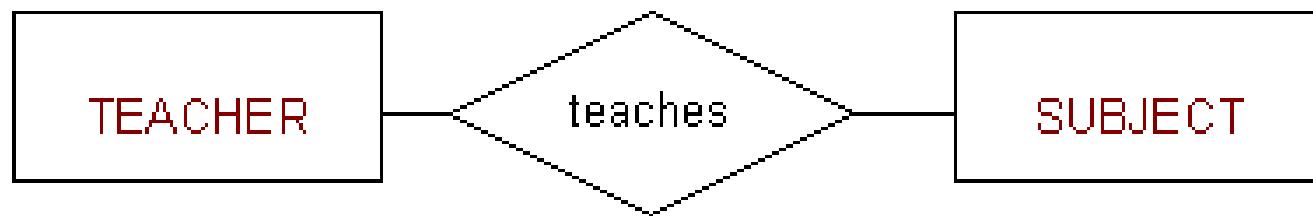
In DBMS, a **degree of relationship** represents the number of entity types that associate in a relationship.

1. Unary
2. Binary
3. Ternary
4. N-ary

## *relationships - binary*

The three most common relationships in ER models are **Binary**, **Unary**, and **Ternary**

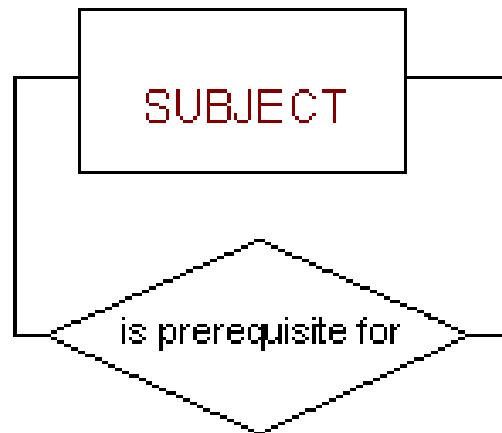
A ***binary relationship*** is when two entities participate and is the most common relationship.



## *relationships - unary*

The three most common relationships in ER models are **Binary**, **Unary**, and **Ternary**

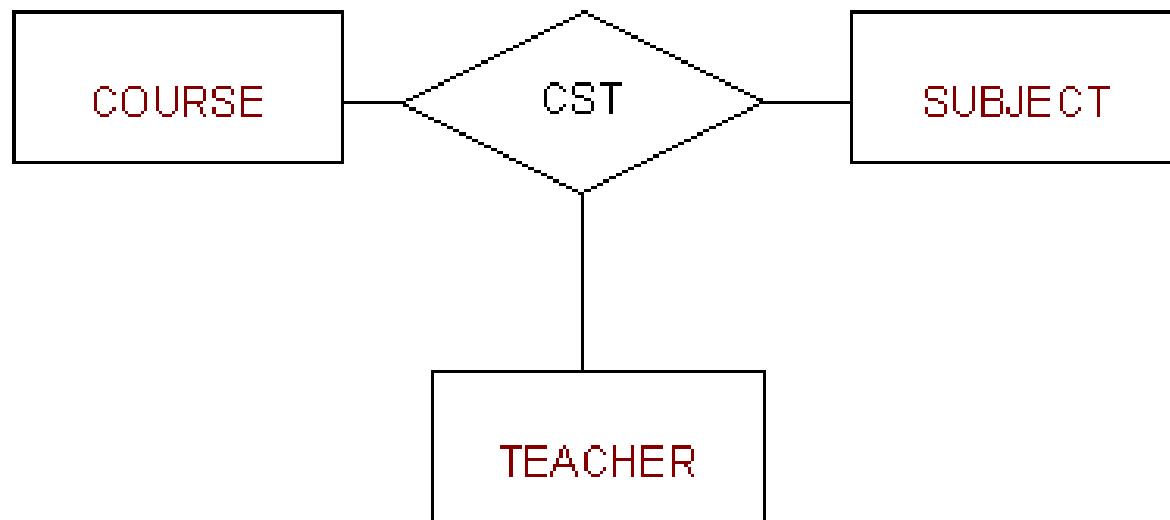
A *unary relationship* is when both participants in the relationship are the same entity.



## *relationships - ternary*

The three most common relationships in ER models are **Binary**, **Unary**, and **Ternary**

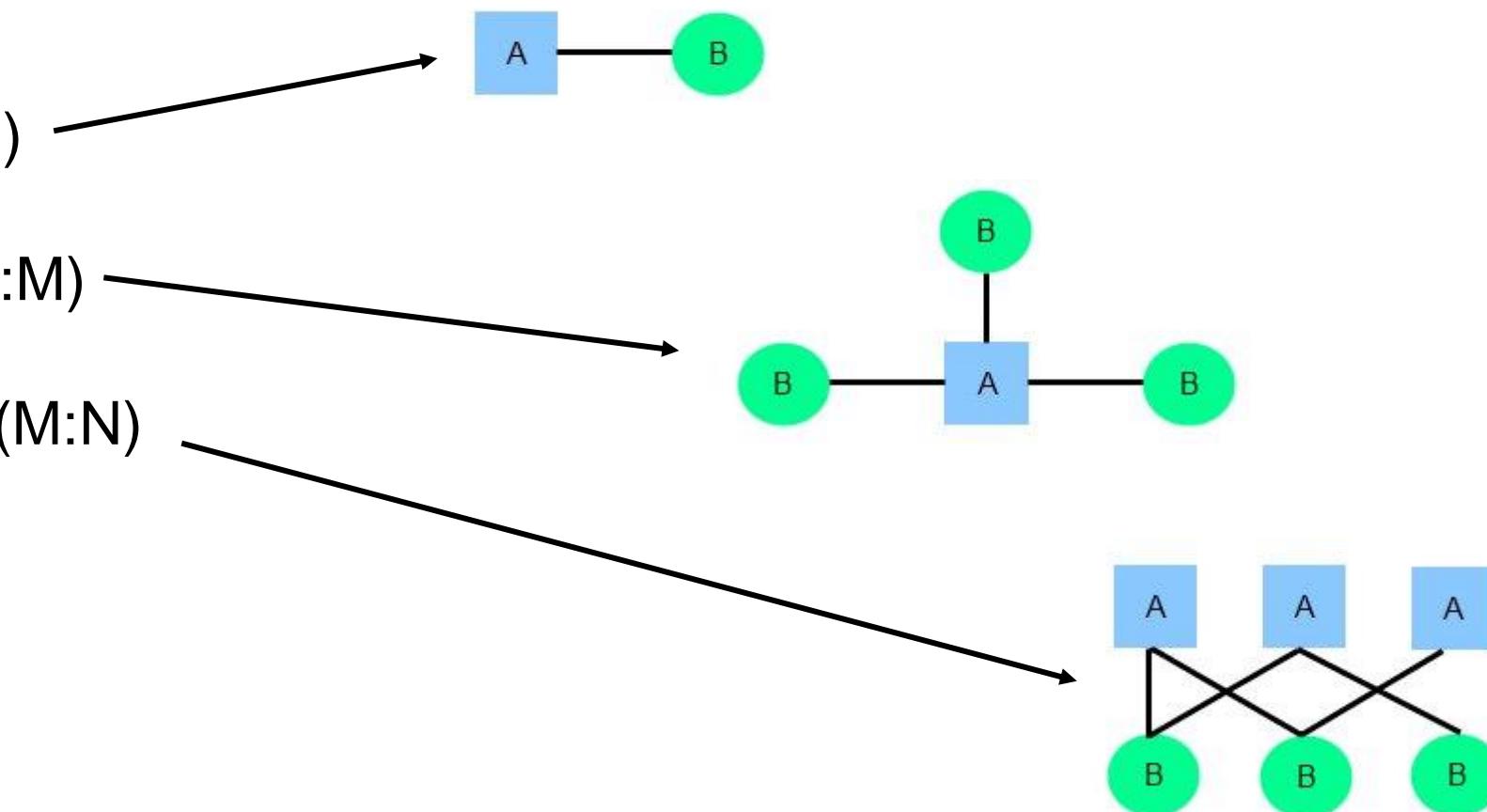
A **ternary relationship** is when three entities participate in the relationship.



# Common relationships

## Common relationship

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



- ONE patient is allocated ONE bed
- ONE bed is occupied by ONLY ONE patient

## *relationships*



One



Many



One (and only one)



Zero or one



One or many

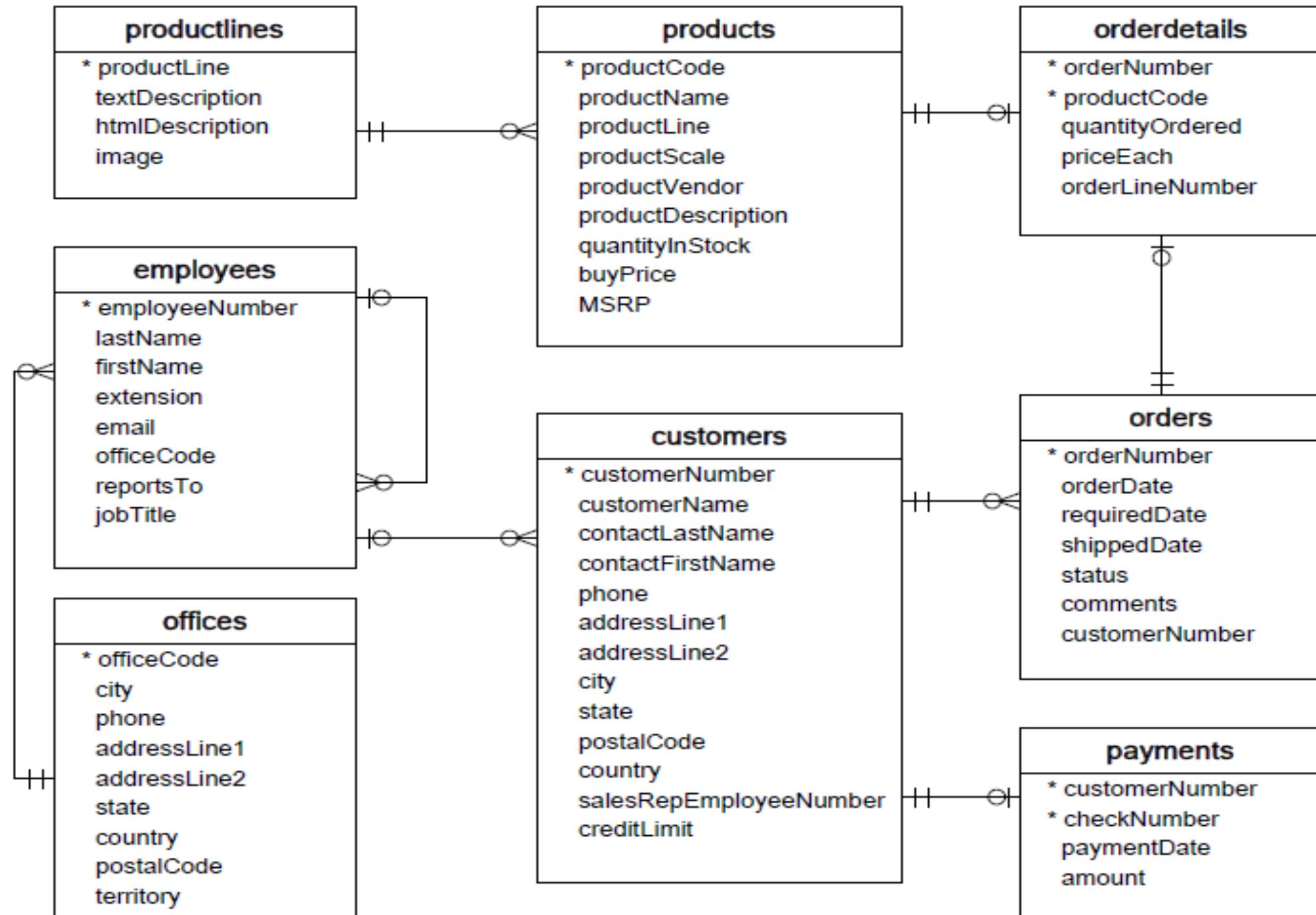


Zero or many

## One and ONLY One (||) Min & Max only One

## *relationships*

Notation	Meaning	Example
—	Relationship	<pre> classDiagram     class Student     class University     Student "Enrolls" --&gt; University   </pre>
+	One	<pre> classDiagram     class Student     class StudentIDNumber     Student "+" --&gt; "Has" StudentIDNumber   </pre>
-->	Many	<pre> classDiagram     class Student     class Class     Student --&gt; "Attends" Class   </pre>
++	One and ONLY One	<pre> classDiagram     class Student     class Chair     Student "++" --&gt; "Uses" Chair   </pre>
—o+	Zero or One	<pre> classDiagram     class Student     class SSN     Student "—o+" --&gt; "+o—" "Has" SSN   </pre>
-->	One or Many	<pre> classDiagram     class Instructor     class Class     Instructor --&gt; "Teaches" Class   </pre>
—o-->	Zero or Many	<pre> classDiagram     class Classroom     class Chair     Classroom "—o--&gt;" --&gt; "&lt;--o—" "Has" Chair   </pre>

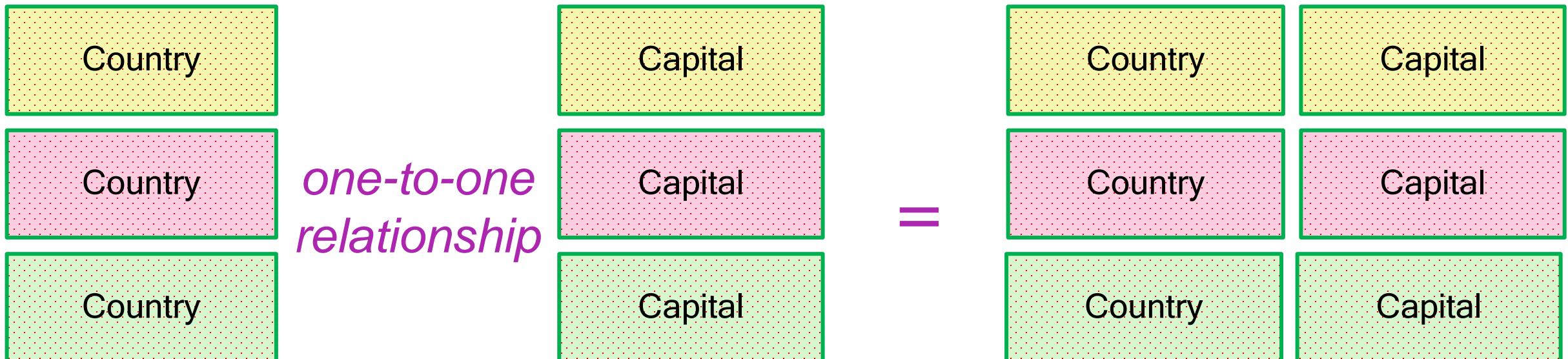


one-to-one relationship

## *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.

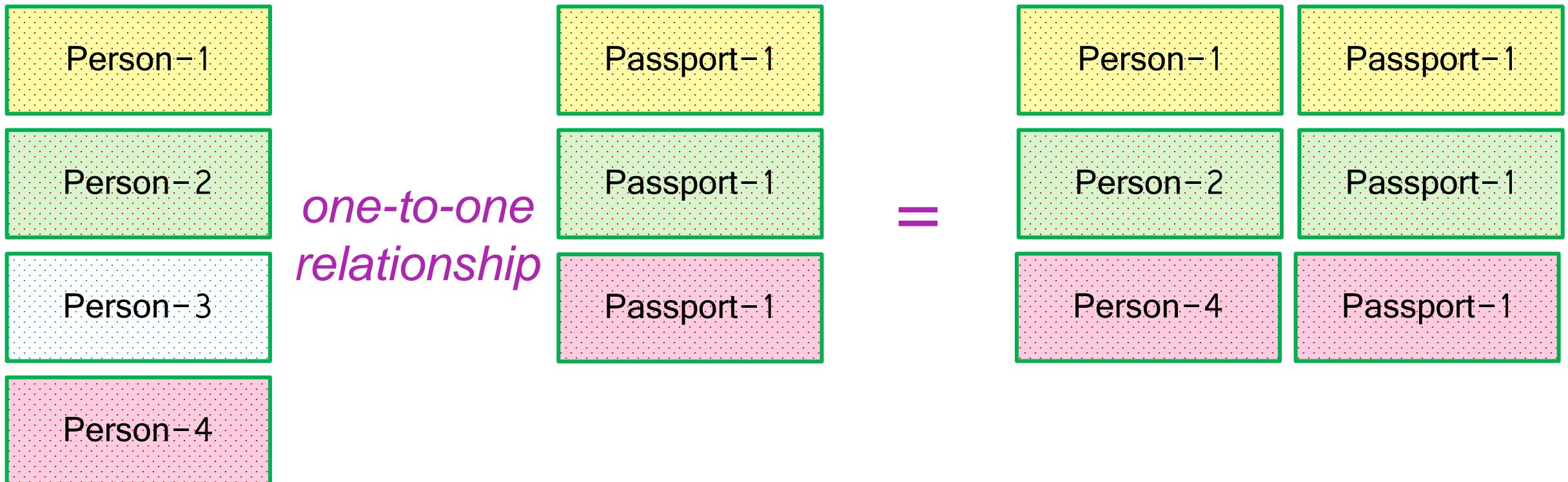
A *one-to-one* relationship is a type of cardinality that refers to the relationship between two entities  $R$  and  $S$  in which one element of entity  $R$  may only be linked to zero/one element of entity  $S$ , and vice versa.



## *one-to-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.

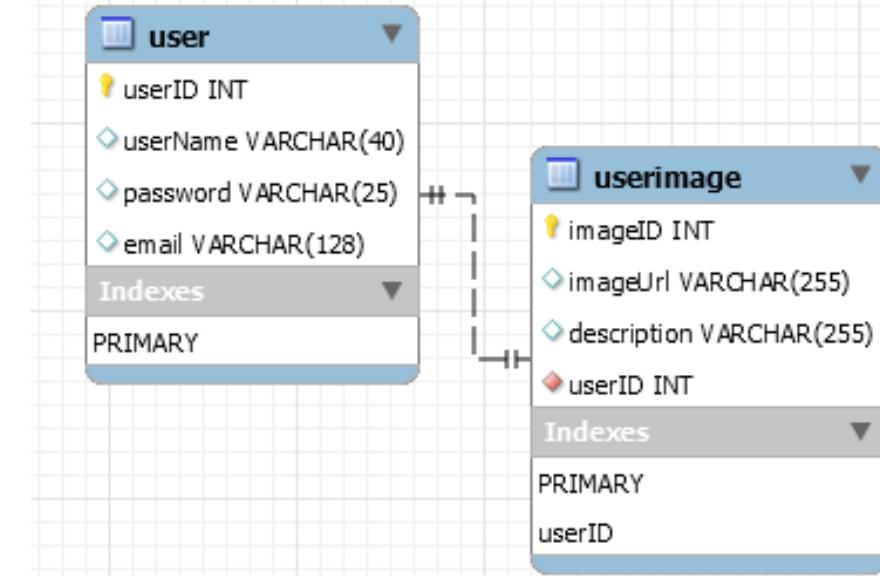


# how to create one-to-one relationship

```
CREATE TABLE user (
    userID INT PRIMARY KEY,
    userName VARCHAR(40),
    password VARCHAR(25),
    email VARCHAR(128)
);
```

```
CREATE TABLE userImage (
    imageID INT PRIMARY KEY,
    imageUrl VARCHAR(255),
    description VARCHAR(255),
    userID INT NOT NULL UNIQUE,
    FOREIGN KEY(userID) REFERENCES user(userID)
);
```

	userID	userName	password	email
▶	1	Ramesh	*****	ramesh@gmail.com
	2	Rajan	*****	rajan.hotmail.com
	3	Kumar	*****	kumer112@yahooomail.com
	4	Suraj	*****	suran@gmail.com
●	NULL	NULL	NULL	NULL

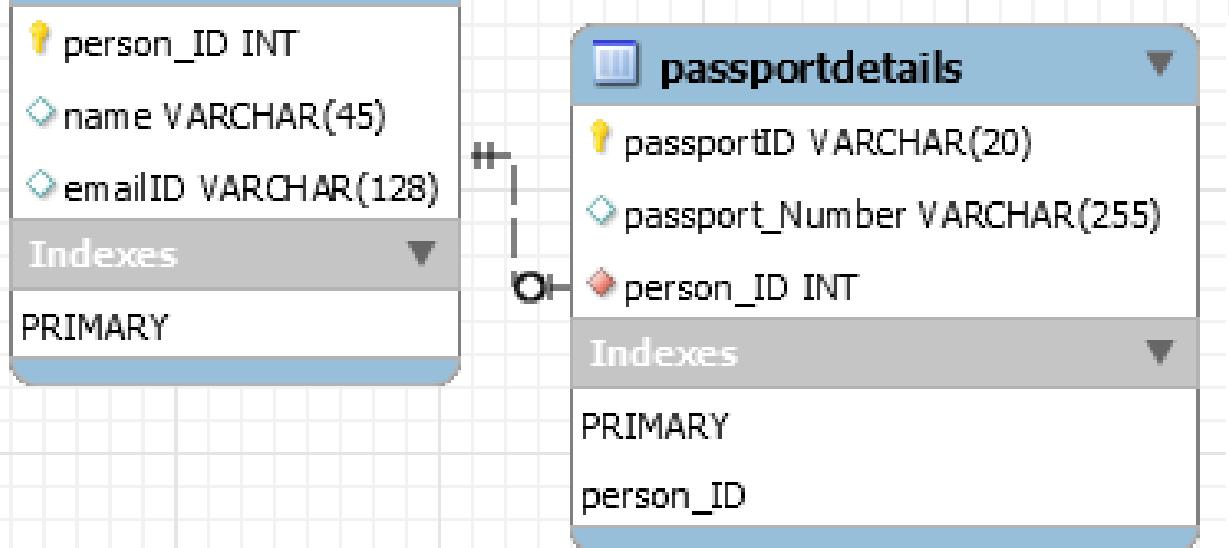
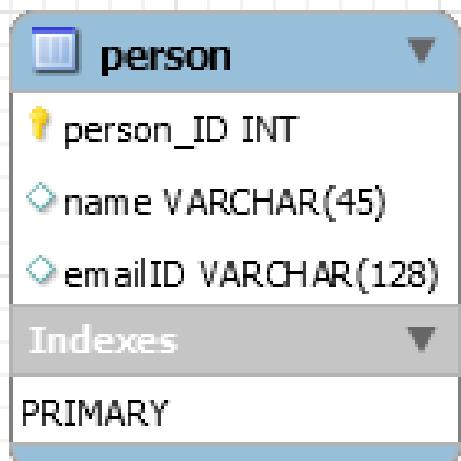


	imageID	imageUrl	description	userID
▶	1001	c:/images/img1.jpeg	For passport	1
	1002	c:/images/img2.jpeg	For Voter Card	2
	1003	c:/images/img3.jpeg	For AADHAR Card	3
	1004	c:/images/img4.jpeg	For Licence	4
●	NULL	NULL	NULL	NULL

# *how to create one-to-one relationship*

```
CREATE TABLE person (
    person_ID INT PRIMARY KEY ,
    name VARCHAR(45),
    emailID VARCHAR(128)
);
```

	person_ID	name	emailID
▶	1	Ramesh	ramesh@gmail.com
	2	Rajan	rajan.hotmail.com
	3	Kumar	kumer112@yahoo.com
●	4	Suraj	suran@gmail.com
	NULL	NULL	NULL



```
CREATE TABLE passportDetails (
    passportID VARCHAR(20) PRIMARY KEY,
    passport_Number VARCHAR(255),
    person_ID INT UNIQUE,
    FOREIGN KEY(person_ID) REFERENCES person(person_ID)
);
```

	passportID	passport_Number	person_ID
▶	IN-zx001	IN-XAJS1028S	1
	IN-zx002	IN-XBDH1738S	2
	IN-zx003	IN-XCKE1933S	3
●	NULL	NULL	NULL

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 rows 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  $R$ .

Customer-1
Customer-2
Customer-3
Customer-4
Customer-5

*one-to-many  
relationship*

Order-1
Order-1
Order-2
Order-1
Order-2
Order-3
Order-1

Customer-1	Order-1
Customer-2	Order-1
Customer-2	Order-2
Customer-3	Order-1
Customer-3	Order-2
Customer-3	Order-3
Customer-4	Order-1

## *one-to-many relationship*

A *one-to-many* relationship between two tables means that a row in one table can have one or more rows 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  $R$ .

Invoice-1
Invoice-2
Invoice-3
Invoice-4

*one-to-many  
relationship*

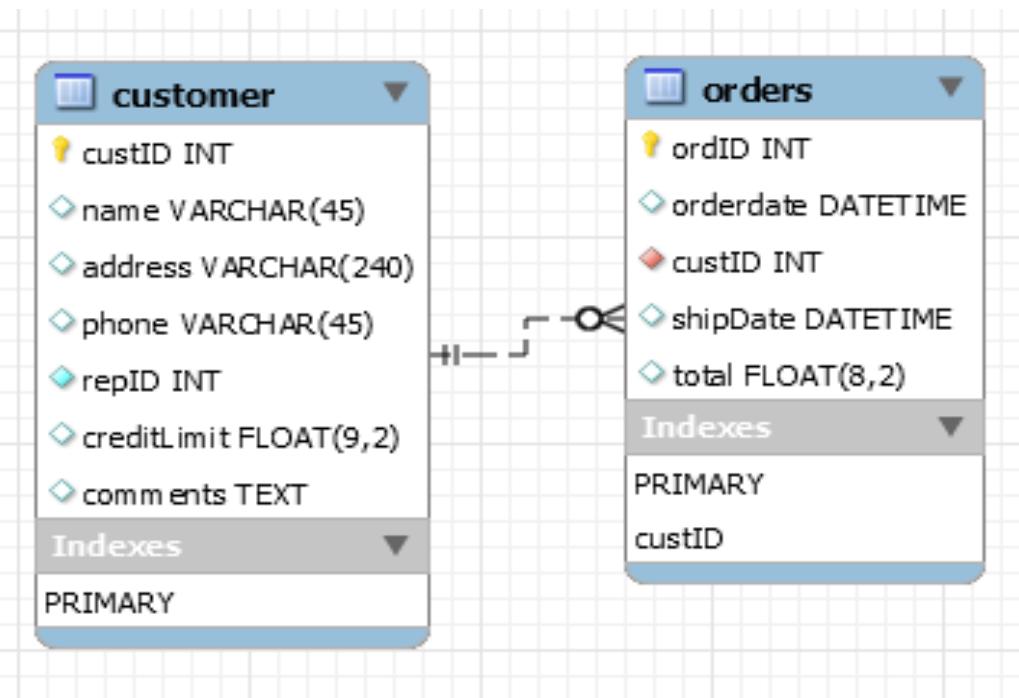
Invoice_Item-1
Invoice_Item-1
Invoice_Item-2
Invoice_Item-1
Invoice_Item-2
Invoice_Item-2
Invoice_Item-1
Invoice_Item-1

Invoice-1	Invoice_Item-1
Invoice-2	Invoice_Item-1
Invoice-2	Invoice_Item-2
Invoice-3	Invoice_Item-1
Invoice-3	Invoice_Item-2
Invoice-3	Invoice_Item-3
Invoice-4	Invoice_Item-1

# *how to create one-to-many relationship*

```
CREATE TABLE customer (
    custID INT PRIMARY KEY,
    name VARCHAR(45),
    address VARCHAR(240),
    phone VARCHAR(45),
    repID INT NOT NULL,
    creditLimit FLOAT(9,2),
    comments TEXT,
    constraint custid_zero CHECK(custID > 0)
);
```

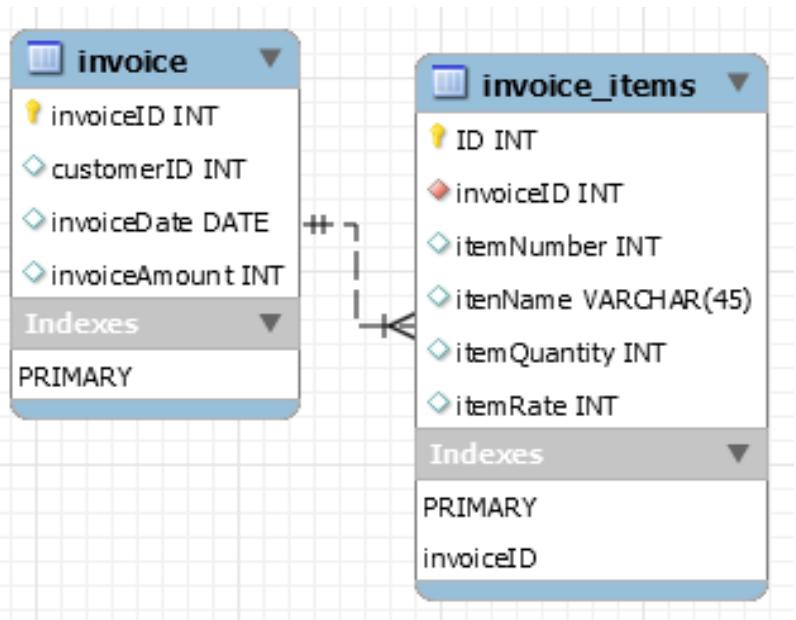
```
CREATE TABLE orders (
    ordID INT PRIMARY KEY,
    orderdate DATETIME,
    custID INT,
    shipDate DATETIME,
    total FLOAT(8,2),
    FOREIGN KEY(custID) REFERENCES customer(custID),
    constraint total_greater_zero CHECK(total >= 0)
);
```



# *how to create one-to-many relationship*

```
CREATE TABLE invoice (
    invoiceID INT PRIMARY KEY,
    customerID INT,
    invoiceDate DATE,
    invoiceAmount INT
);
```

	invoiceID	customerID	invoiceDate	invoiceAmount
▶	1	235	2020-01-13	1750
	2	235	2020-02-28	5000
	3	778	2020-03-10	2000
	4	778	2020-03-16	2300
*	HULL	HULL	HULL	HULL



```
CREATE TABLE invoice_items (
    invoiceID INT,
    itemID INT,
    itenName VARCHAR(45),
    itemQuantity INT,
    itemRate INT,
    PRIMARY KEY(invoiceID, itemID),
    FOREIGN KEY(invoiceID) REFERENCES invoice(invoiceID)
);
```

```
CREATE TABLE invoice_items (
    invoiceID INT NOT NULL,
    itemID INT NOT NULL,
    itenName VARCHAR(45),
    itemQuantity INT,
    itemRate INT,
    UNIQUE(invoiceID, itemID),
    FOREIGN KEY(invoiceID) REFERENCES invoice(invoiceID)
);
```

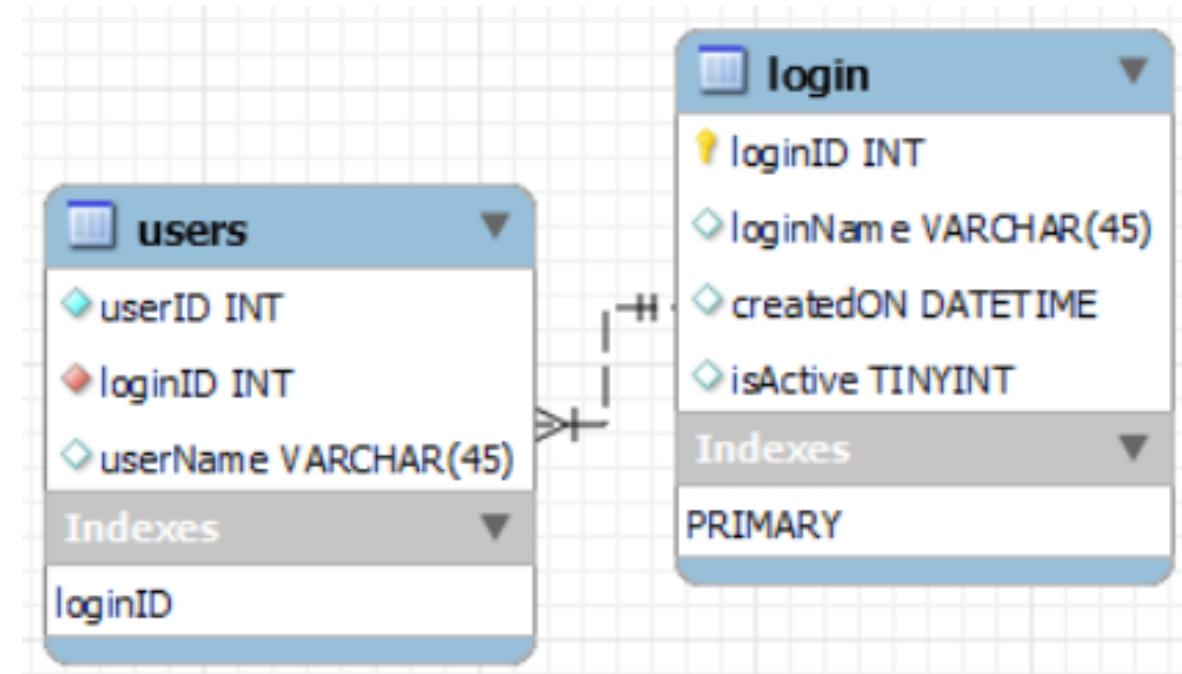
many-to-one relationship

```
CREATE TABLE users (
    userID INT,
    loginID INT,
    userName VARCHAR(45),
    PRIMARY KEY(loginID, userID),
    constraint fk_users_login_loginID1 FOREIGN KEY(loginID)
        REFERENCES login(loginID)
);
```

```
CREATE TABLE users (
    userID INT NOT NULL,
    loginID INT NOT NULL,
    userName VARCHAR(45),
    UNIQUE(loginID, userID),
    constraint fk_users_login_loginID2 FOREIGN KEY(loginID)
        REFERENCES login(loginID)
);
```

```
CREATE TABLE login (
    loginID INT,
    loginName VARCHAR(45),
    createdON DATETIME,
    isActive TINYINT,
    PRIMARY KEY(loginID)
);
```

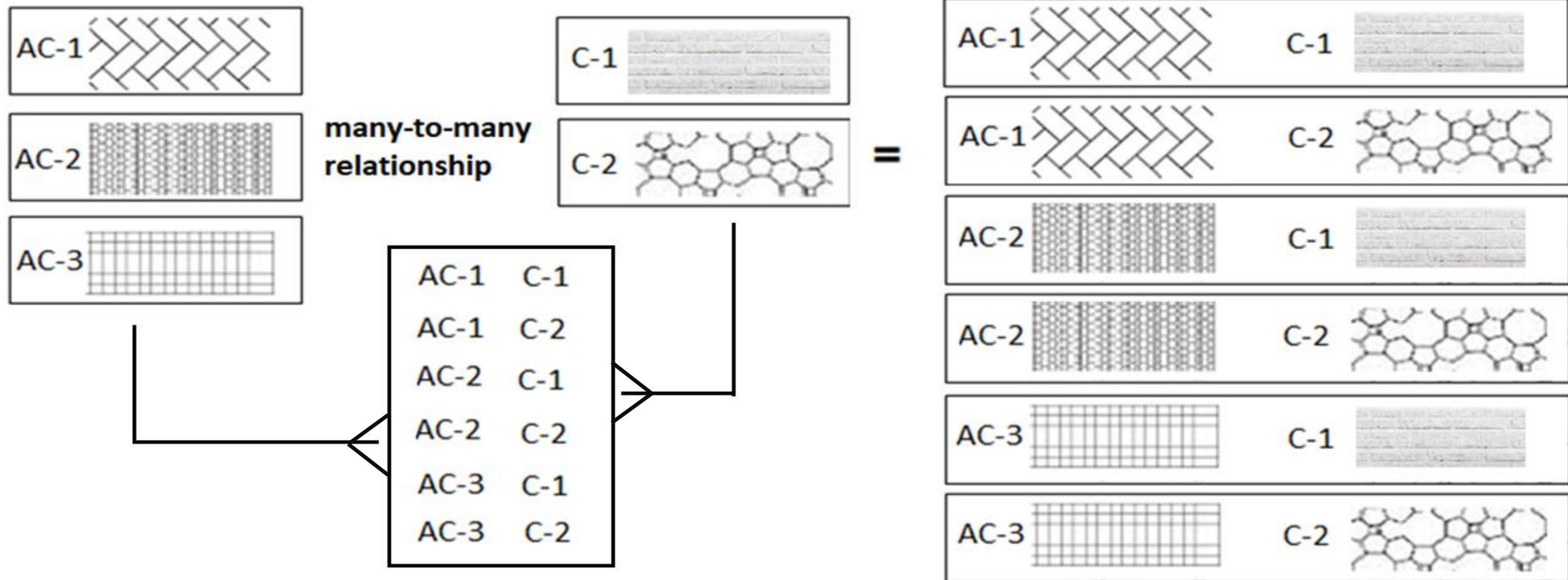
*many-to-one relationship*



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.



# *how to create many-to-many relationship*

```
CREATE TABLE item (
```

```
    ID INT PRIMARY KEY,
```

```
    name VARCHAR(45),
```

```
    description TEXT
```

```
);
```

```
CREATE TABLE orders (
```

```
    ID INT PRIMARY KEY,
```

```
    orderdate DATETIME,
```

```
    custID INT NOT NULL,
```

```
    shipDate DATETIME,
```

```
    total FLOAT(8,2),
```

```
    constraint total_greater_zero CHECK(total >= 0)
```

```
);
```

```
CREATE TABLE orders_has_item (
```

```
    orders_ID INT NOT NULL,
```

```
    item_ID INT NOT NULL,
```

```
    PRIMARY KEY(orders_ID, item_ID),
```

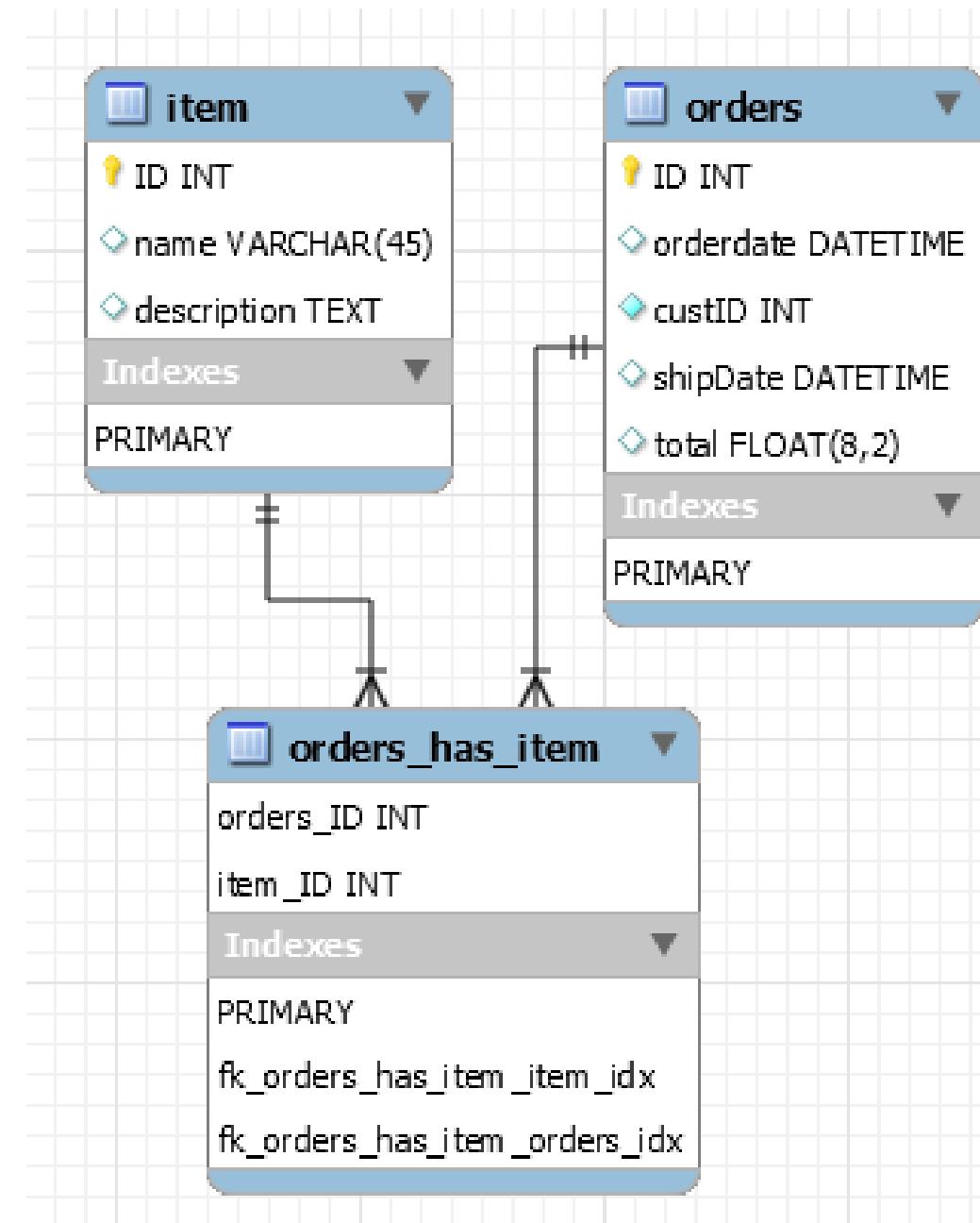
```
    constraint fk_orders_has_item_orders FOREIGN KEY(orders_ID)
```

```
        REFERENCES orders(ID),
```

```
    constraint fk_orders_has_item_item1 FOREIGN KEY(item_ID)
```

```
        REFERENCES item(ID)
```

```
);
```

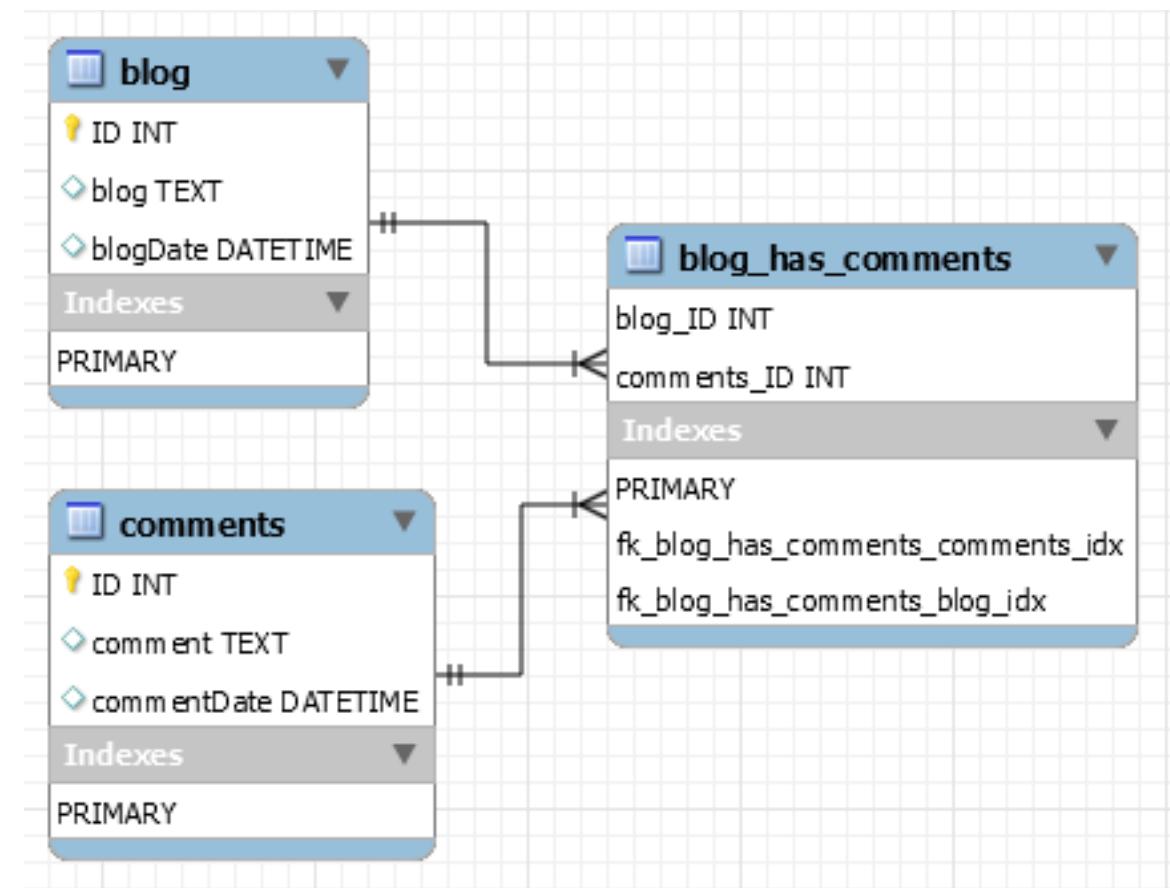


# how to create many-to-many relationship

```
CREATE TABLE blog (
    ID INT PRIMARY KEY,
    blog TEXT,
    blogDate DATETIME
);
```

```
CREATE TABLE comments (
    ID INT PRIMARY KEY,
    comment TEXT,
    commentDate DATETIME
);
```

```
CREATE TABLE blog_has_comments (
    blog_ID INT,
    comments_ID INT,
    PRIMARY KEY(blog_ID, comments_ID),
    constraint fk_blog_has_comments_blog FOREIGN KEY(blog_ID) REFERENCES blog(ID),
    constraint fk_blog_has_comments_comments FOREIGN KEY(comments_ID) REFERENCES comments(ID)
);
```



- ▶ Branch { branch-ID, branch-name, branch-city, assets }
- ▶ Customer { customer-id, customer-name, customer-street, customer-only }
- ▶ Account { account-number, branch-ID, balance }
- ▶ Loan { loan-number, branch-ID, amount }
- ▶ Depositor { customer-id, account-number }
- ▶ Borrower { customer-id, loan-number }

## Relational Algebra

$\wedge$  (and),  $\vee$  (or),  $\neg$  (not)

# *Relational Algebra*

- Select (sigma)               $(\sigma)$
  - Project (pi)                 $(\Pi)$
  - Union                         $(\cup)$
  - Set different                 $(-)$
  - Cartesian product           $(X)$
  - Rename                       $(\rho)$
- and                          $\wedge$
  - or                             $\vee$
  - not                          $\neg$

SELECT

Select ( $\sigma$ ) Notation –  $\sigma_p(r)$

$\sigma_{condition}(r)$

$\sigma$  would represent the SELECT command

$p$  would represent the condition for selection.

( $r$ ) would represent the Relation or the Table from which we are making a selection of the tuples.

- Select the EMPLOYEE whose employee number is 7, or those whose date of birth is '01-Jan-1980'

( $r$ ) =  $\sigma_{empno=7}$  (EMPLOYEE)

( $r$ ) =  $\sigma_{dob \leq '01-Jan-1980'}$  (EMPLOYEE)

( $r$ ) =  $\sigma_{empno=7 \vee dob='01-Jan-1980'}$  (EMPLOYEE)

Returns the result to new relation.

SELECT

Select ( $\sigma$ ) Notation –  $\sigma_p(r)$

$\sigma_{condition}(r)$

- Select the EMPLOYEE tuples whose employee number is  $> 7$  and department ID is 10

$(r) = \sigma_{deptno=10} [\sigma_{empno>7} (\text{EMPLOYEE})]$

$(r) = \sigma_{empno>7 \wedge deptno=10} (\text{EMPLOYEE})$

# PROJECT

Project ( $\Pi$ ) Notation –  $\Pi_{A_1, A_2, A_n} (r)$

$\Pi_{<\text{attribute-list}>} (r)$

$\Pi$  would represent the PROJECT.

$<\text{attribute list}>$  would represent the attributes(columns) we want from a relational.

( $r$ ) would represent the Relation or the Table from which we are making a selection of the tuples.

Select Date of Birth (dob) and Employee Number (empno) from the relation EMPLOYEE or Select Date of Birth (dob) and Employee Number (empno) from the relation EMPLOYEE whose date of birth is before 1980

$(r) = \Pi_{\text{dob, empno}} (\text{EMPLOYEE})$

$(r) = \Pi_{\text{dob, empno}} [\sigma_{\text{dob} < '01-Jan-1980'} (\text{EMPLOYEE})]$

# UNION

Union (U) Notation –  $r \cup s$

$\Pi_{\langle \text{attribute-list} \rangle}(r) \cup \Pi_{\langle \text{attribute-list} \rangle}(s)$

- $r$ , and  $s$  must have the same number of attributes.
- Attribute domains must be compatible.
- Duplicate tuples are automatically eliminated.

Selects the customer name who have either purchased a book or a computer or both.

$(r) = \Pi_{cName} (\text{BOOK}) \cup \Pi_{cName} (\text{COMPUTER})$

# INTERSECTION

Intersection ( $\cap$ ) Notation –  $r \cap s$

$\Pi_{\langle \text{attribute-list} \rangle}(r) \cap \Pi_{\langle \text{attribute-list} \rangle}(s)$

- $r$ , and  $s$  must have the same number of attributes.
- Attribute domains must be compatible.
- Duplicate tuples are automatically eliminated.

Projects the customer name who have purchased a book and a computer both.

$(r) = \Pi_{cName} (\text{BOOK}) \cap \Pi_{cName} (\text{COMPUTER})$

# MINUS

Minus (-) Notation – r - s

$$\Pi_{\langle \text{attribute-list} \rangle}(r) - \Pi_{\langle \text{attribute-list} \rangle}(s)$$

- **r**, and **s** must have the same number of attributes.
- Attribute domains must be compatible.
- Duplicate tuples are automatically eliminated.
- Projects the customer name who have purchased a BOOK but not a COMPUTER.
- Projects the customer name who have purchased a COMPUTER but not a BOOK.

$$(r) = \Pi_{cName} (\text{BOOK}) - \Pi_{cName} (\text{COMPUTER})$$

$$(r) = \Pi_{cName} (\text{COMPUTER}) - \Pi_{cName} (\text{BOOK})$$

## CARTESIAN PRODUCT

Cartesian product (X)      Notation – r X s

$\Pi_{\langle \text{attribute-list} \rangle}(r) \times \Pi_{\langle \text{attribute-list} \rangle}(s)$

## Examples Queries

Find all loans of over \$1200

$(r) = \sigma_{amount > 1200} (loan)$

Find the loan number for each loan of an amount greater than \$120

$(r) = \prod_{loan-number} (\sigma_{amount > 1200} (loan))$

## Examples Queries

Find the names of all customers who have a loan, an account, or both, from the bank

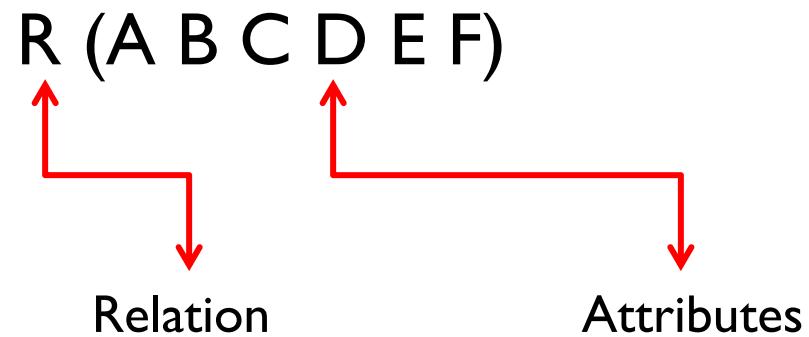
$$\Pi_{customer-name} (borrower) \cup \Pi_{customer-name} (depositor)$$

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

$$\Pi_{customer-name} (borrower) \cap \Pi_{customer-name} (depositor)$$

# Closure of Attributes

# Closure of attributes



**Schema:** A schema is a collection of database objects (like table, foreign key, primary key, views, columns, data types, stored procedure, etc.) associated with one particular database username. This username is called the schema owner, or the owner of the related group of objects. 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



# What is Data Model

Data modeling is the process of creating a data model for the data to be stored in a Database.  
This data model is a conceptual representation of

- Data objects
- The associations between different data objects
- The rules.

# *data modeling*

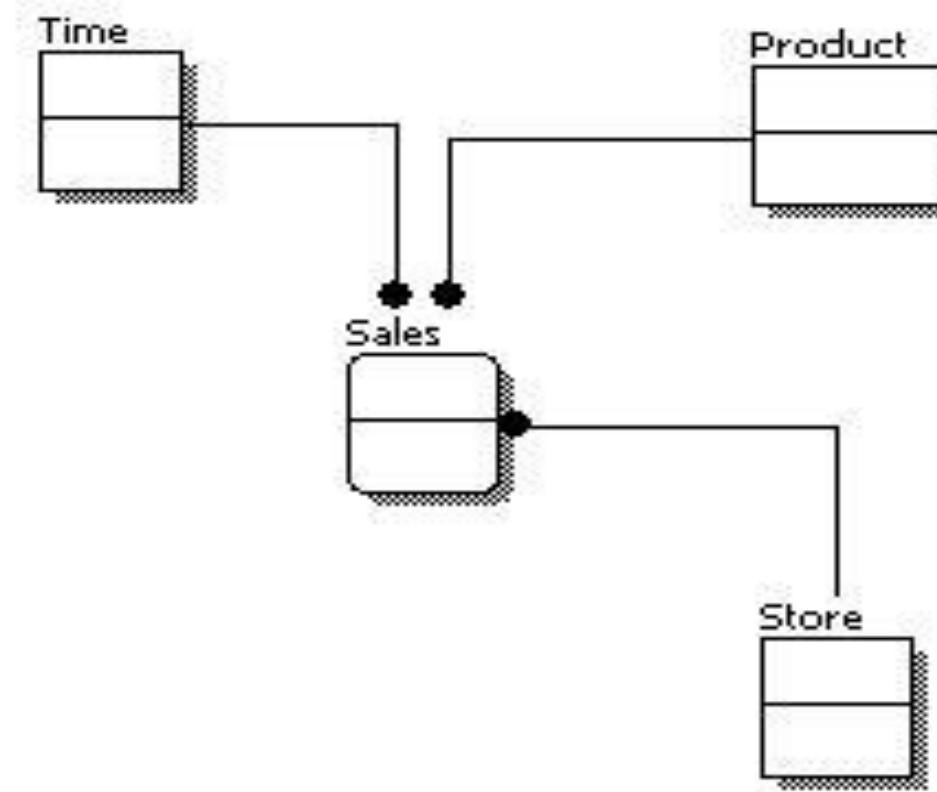
**Data models** define how data is connected to each other and how they are processed and stored inside the system.

**Data modeling** involves a progression from conceptual model to logical model to **physical object**.

# *conceptual data model*

Features of conceptual data model include:

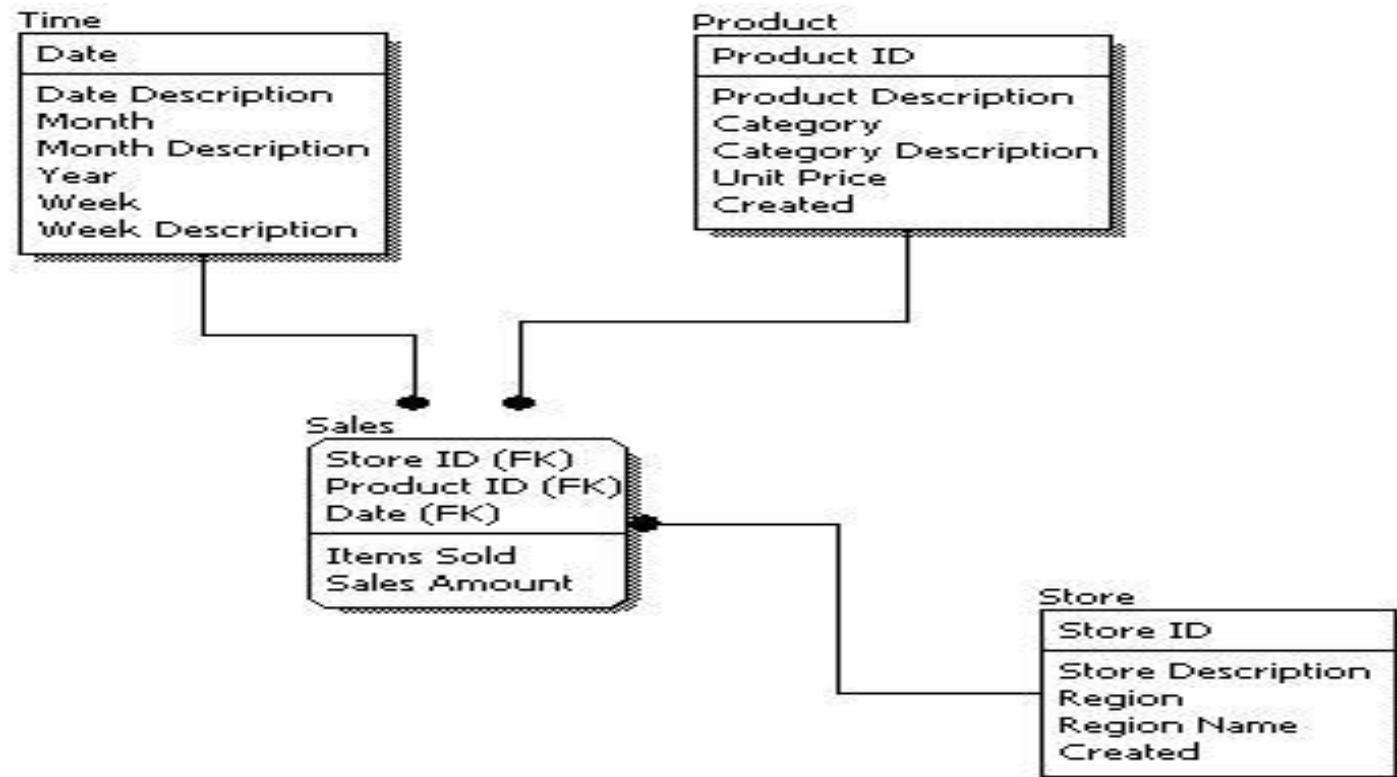
- Includes the important entities and the relationships among them.
- No attribute is specified.
- No primary key is specified.



# *logical data model*

Features of logical data model include:

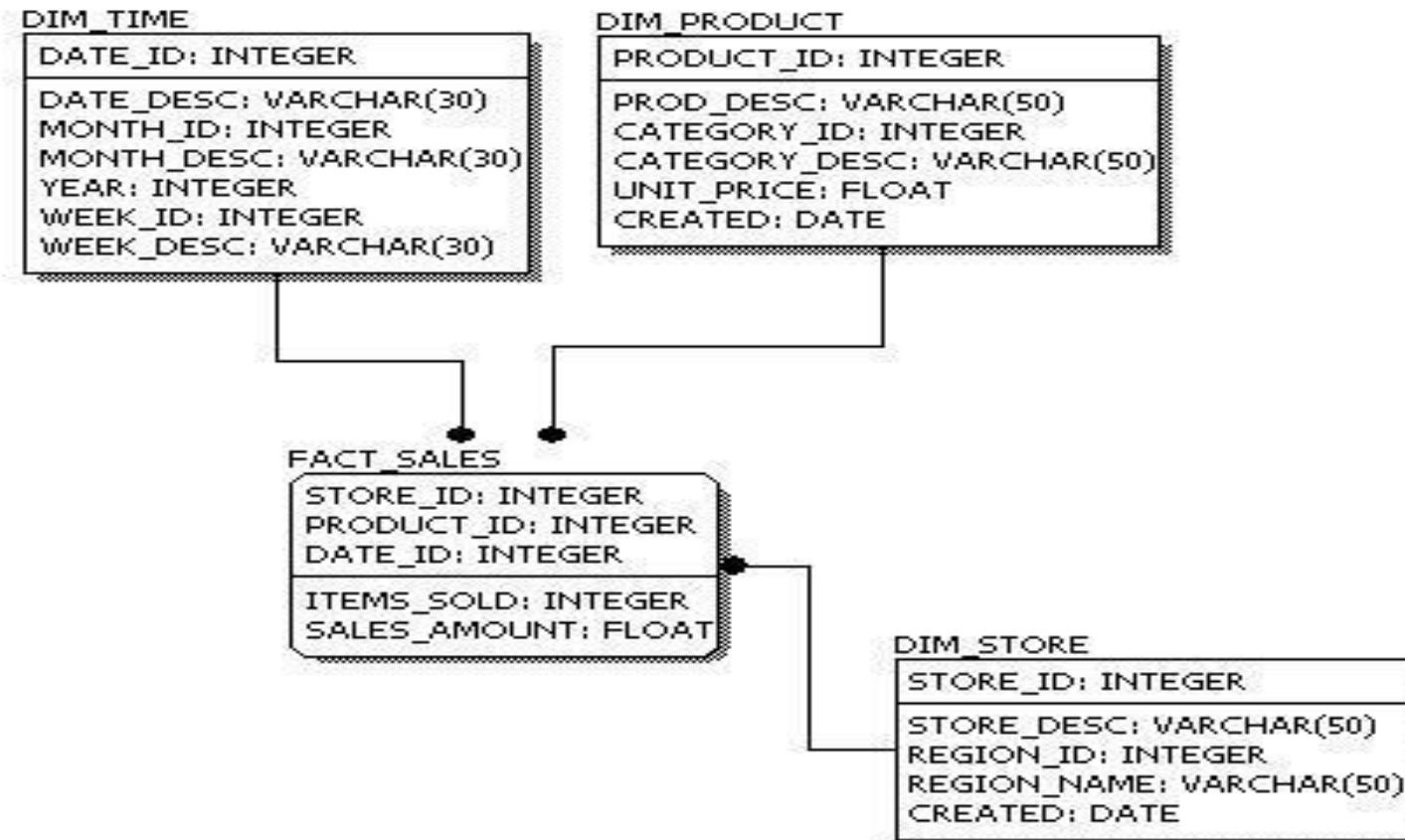
- Includes all entities and relationships among them.
- All attributes for each entity are specified.
- The primary key for each entity is specified.
- Foreign keys are specified.



# *physical data model*

Features of physical data model include:

- Convert entities into tables.
- Convert relationships into foreign keys.
- Convert attributes into columns.



# *data modeling*

Feature	Conceptual	Logical	Physical
Entity Names	✓	✓	
Entity Relationships	✓	✓	
Attributes		✓	
Primary Keys		✓	✓
Foreign Keys		✓	✓
Table Names			✓
Column Names			✓
Column Data Types			✓

- Hierarchical databases
- Network databases
- Object-oriented databases
- Relational databases
- NoSQL databases

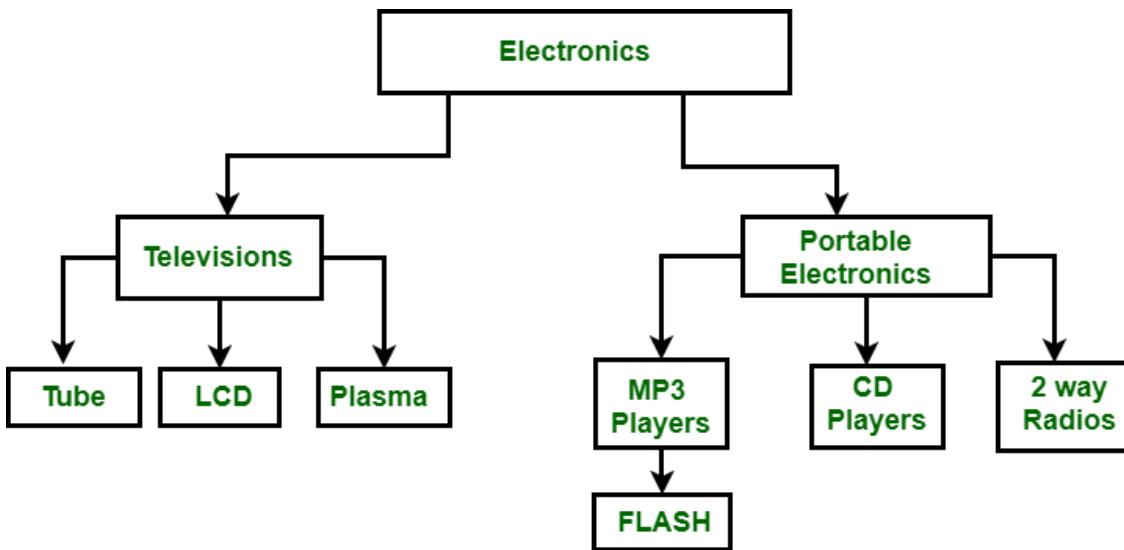


## Types of data modeling

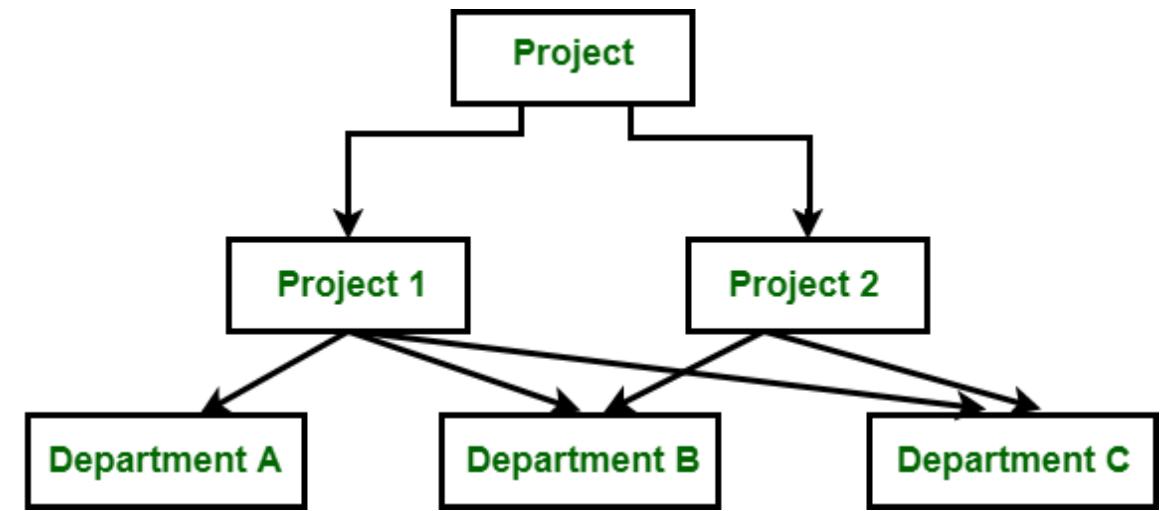
There are three main types of DBMS data models: relational, hierarchical and network.

# *types of data modeling*

hierarchical data models



network data model



Problems that can occur in un-normalized database where all the data is stored in one table.

## Anomalies in DBMS

Something that deviates from what is standard, normal, or expected.

There are three types of anomalies that occur when the database is not normalized. These are –

- **Insertion** anomaly
- **Updation** anomaly
- **Deletion** anomaly.

# *anomalies in dbms - example*

student table.

<b>Rollno (PK)</b>	<b>name</b>	<b>branch</b>	<b>hod</b>	<b>office_tel</b>
401	Amit	CSE	Mr. X	53337
402	Rajan	CSE	Mr. X	53337
403	Bhavin	CSE	Mr. X	53337
404	Pankaj	CSE	Mr. X	53337
NULL	NULL	BCA	Mr.Y	77127

# *insert, update, and delete anomaly*

student table.

RollNo (PK)	name	branch	hod	office_tel
401	Amit	CSE	Mr. X	53337
402	Rajan	CSE	Mr. X	53337
403	Bhavin	CSE	Mr. X	53337
404	Pankaj	CSE	Mr. X	53337
NULL	NULL	BCA	Mr.Y	77127

- **Insert anomaly:** What if we want to add new branch in the student table? In this case as the student Rollno is a primary key, it will not allow you to insert new branch. Also, if we have to insert data of 100 students of same branch, then the branch information will be repeated for all those 100 students. This is **Insertion anomalies**.
- **Update anomaly:** What if Mr. X telephone number changes? In that case all the student records will have to be updated, and if by mistake we miss any record, it will lead to data inconsistency. This is **Updation anomaly**.
- **Delete anomaly:** In our **Student** table, two different information are kept together, Student information and Branch information. Hence, at the end of the academic year, if student records are deleted, we will also lose the branch information. This is **Deletion anomaly**.

Movie_ID	MovieName	Genres	Film_certificate
RollNo	CourseName	Course_fees	

If  $P \rightarrow Q$  and  $Q \rightarrow R$  is true, then  $P \rightarrow R$  is a *transitive dependency*.

- Movie\_ID  $\rightarrow$  Genres
- Genres  $\rightarrow$  Film\_certificate
- Therefore, Movie\_ID  $\rightarrow$  Film\_certificate has *transitive dependency*.

## Functional & Transitivity Dependency

A functional dependency (FD denoted by an arrow “ $\rightarrow$ ”) is a relationship between two attributes, typically between the prime attribute (PK) and other non-prime attributes within a table.

**If  $X \rightarrow Y$  , the value of X uniquely determines the value of Y, then Y is functionally dependent on attribute X (usually the PK) .**

**Note:**

The left side is called the **determinant**, and the right side is the **dependent**.

- Movie\_ID  $\rightarrow$  MovieName

Therefore, MovieName is *functional dependent* on attribute Movie\_ID

## Partial Dependency

A partial dependency would occur whenever a non-prime attribute depends functionally on a part of the given candidate key.

R(X, Y, A, B, C)

FD: {XY -> A, A -> B, Y -> C }. Find partial dependency.

BranchID	AccountNo	holderName	Pin code	Area
BranchID	AccountNo	holderName	PincodeID	
			Pincode	Area

Where:

- XY is the only candidate key.
- key attributes =X, Y and
- non-key attributes = A, B, C.

{studentID } -> studentName // partial dependency  
{assignmentNo } -> assignment // partial dependency

Note:

**Key attributes** are also called **prime attributes** and **non-key attributes** are also called **non-prime attributes**.

XY -> A is not a partial dependency // it is full dependency.

A -> B is not a partial dependency //it is full dependency.

Y -> C is a partial dependency, since Y is a part of the candidate key and C is a non-key attribute.

# Normalization in DBMS

Normalization is the process of removing redundant data from the database by splitting the table into two or more tables and defining relationships between these tables.

# **Problem**

To understand normalization in database with example tables, let's assume that we are supposed to store the details of **courses** and **instructors** in a university.

Course code	Course venue	Instructor Name	Instructor's phone number
CS101	Lecture Hall 20	Prof. Ronald	+1 6514821924
CS152	Lecture Hall 21	Prof. John	+1 6519272918
CS154	CS Auditorium	Prof. Ronald	+1 6514821924

At first, this design seems to be good. However, issues start to develop once we need to modify information. For instance, suppose, **if Prof. Ronald changed his mobile number. In such a situation, we will have to make edits in 2 places**. What if someone just edited the mobile number against **CS101**, but forgot to edit it for **CS154**? This will lead to stale/wrong information in the database.

# **Solution**

This problem, however, can be easily tackled by dividing our table into 2 simpler tables.

**Table 1 (Instructor)**

Instructor's ID	Instructor's Name	Instructor's Number
1	Prof. Ronald	+1 6514821924
2	Prof. John	+1 6519272918
3	Prof. Smith	+1 6519272919

**Table 2 (Course)**

Course Code	Course Venue	Instructor ID
CS101	Lecture Hall 20	1
CS152	Lecture Hall 21	2
CS154	CS Auditorium	1

# *First Normal Form (1NF)*

**For a table to be in the First Normal Form, it should follow the following 4 rules:**

1. It should only have single(atomic) valued attributes/columns.
2. Values stored in a column should be of the same domain
3. All the columns in a table should have unique names.
4. And the order in which data is stored, does not matter.
5. The table should have no “repeating groups” of fields.

<b>OrderID</b>	<b>CustomerName</b>	<b>Product1</b>	<b>Product2</b>	<b>Product3</b>
1	Rajesh Singh	Apple	Orange	Banana
2	Rajat Kumar	Apple	NULL	NULL
3	Ramesh	Orange	Grape	NULL

## *First Normal Form (1NF)*

The First normal form simply says that each cell of a table should contain exactly one value.

Instructor's Name	Course Code
Prof. Ronald	(CS101, CS154)
Prof. John	(CS152)
Prof. Smith	(CS189, CS107)

Here, the issue is that in the first row, we are storing 2 courses against Prof. Ronald. A better method would be to store the courses separately. This way, if we want to edit some information related to **CS101**, we do not have to touch the data corresponding to **CS154**.

Instructor's Name	Course Code
Prof. Ronald	CS101
Prof. Ronald	CS154
Prof. John	CS152
Prof. Smith	CS189
Prof. Smith	CS107

# *Second Normal Form (2NF)*

**For a table to be in the Second Normal Form**

1. It should be in the First Normal form.
2. And, it should not have **Partial Dependency**. (i.e In the second normal form, all **non-key attributes** are **fully functional dependent** on the primary key)

# Second Normal Form (2NF)

For a table to be in second normal form, the following 2 conditions are to be met:

- The table should be in the first normal form.
- The primary key of the table should compose of exactly 1 column (a compound key, if necessary).

Let us take another example of storing student enrollment in various courses. Each student may enroll in multiple courses. Similarly, each course may have multiple enrollments.

A sample table may look like this (**student name and course code**)

## Enrollments

Student Name	Phone	City	Course Code	Course Name
Rahul	•••••	Pune	CS152	Computer_Course_1
Rajat	•••••	Mumbai	CS101	Computer_Course_2
Rahul	•••••	Pune	CS154	Computer_Course_3
Raman	•••••	Surat	CS101	Computer_Course_2
Rajat	•••••	Mumbai	CS189	Computer_Course_5

{ Student Name }  $\rightarrow$  Course Name or { Course Code }  $\rightarrow$  Course Name , Course Name is a partial dependency  
{ Student Name, Course Code }  $\rightarrow$  Course Name, so Course Name is fully functional dependency

# *Second Normal Form (2NF)*

**Student**

ID(PK)	Student Name	Phone	City
1	Rahul	•••••	Pune
2	Rajat	•••••	Mumbai
3	Raman	•••••	Surat
4	Pankaj	•••••	Baroda

**Courses**

Course Code(PK)	Course Name
CS101	Computer_Course_2
CS152	Computer_Course_1
CS154	Computer_Course_3
CS107	Computer_Course_4
CS189	Computer_Course_5

**Student\_Enrollment**

ID(PK)	StudentID(FK)	Course Code(FK)
1	1	CS152
2	1	CS154
3	2	CS101
4	3	CS101
5	2	CS189

# *Third Normal Form (3NF)*

**A table is said to be in the Third Normal Form when,**

1. It is in the Second Normal form.
  2. And, it doesn't have Transitive Dependency.
- 
- **Transitive dependency** is expressing the dependency of A on C when A depends on B and B depends on C.
  - **A functional dependency** is an association between two attributes of the same relational database table. One of the attributes is called the determinant and the other attribute is called the determined. For each value of the determinant there is associated one and only one value of the determined.

## *Third Normal Form (3NF)*

Column A is said to be functionally dependent on column B if changing the value of A may require a change in the value of B.

Here, the department column is dependent on the professor name column. This is because if in a particular row, we change the name of the professor, we will also have to change the department value.

<b>Course Code</b>	<b>Course venue</b>	<b>Instructor's Name</b>	<b>Department</b>
CS101	Lecture Hall 18	Prof. Ronald	Mathematics Department
CS152	Auditorium building	Prof. John	Electronics Department
CS154	Lecture Hall 19	Prof. Ronald	Mathematics Department
CS189	Auditorium building #1	Prof. Smith	Computer Science Department
CS107	Lecture Hall #4	Prof. Smith	Computer Science Department

## *Third Normal Form (3NF)*

Here, when we changed the name of the professor, we also had to change the department column. This is not desirable since someone who is updating the database may remember to change the name of the professor, but may forget updating the department value. This can cause inconsistency in the database.

Third normal form avoids this by breaking this into separate tables

Course Code	Course Venue	Instructor's ID
CS101	Lecture Hall 18	1
CS152	Auditorium building,	2
CS154	Lecture Hall 19	1
CS189	Auditorium building #1	3
CS107	Lecture Hall #4	3

Here, the first column is the ID of the professor who's taking the course.

Instructor's ID	Instructor's Name	Instructor's Number	Department
1	Prof. Ronald	+1 6514821924	Mathematics Department
2	Prof. John	+1 6519272918	Electronics Department
3	Prof. Smith	+1 6519272919	Computer Science Department

## *Third Normal Form (3NF)*

Therefore, in the third normal form, the following conditions are required:

- The table should be in the second normal form.
- There should not be any functional dependency.

If this is the data the convert in in 3NF

Business Rules.

1. A salesman can sale one or many products.
2. A salesman can sale a product in one or many regions.
3. A product can be sold in one or may regions.

Salesman Name	Region1	Region2	Region3	Region4
Saleel	Car, Bus	Car, Bus, Cycle	Cycle, Motorcycle	Car
Rahul	Shirt, Pant, Gloves	Gloves, Book, Pencil	Pencil, Notebook	Keyboard, Mouse
Omkar	Keyboard, Mouse	Pen Drives, CD	Monitors, Scanners	Scanner, CD
Ninad	Photos, Photo Frames	Shirt, Pants, Gloves	Cycle, Motorcycle	Bus, Cycle, Car

## 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:

*what is sql?*

- EXPLICIT or IMPLICIT commit will commit the data.

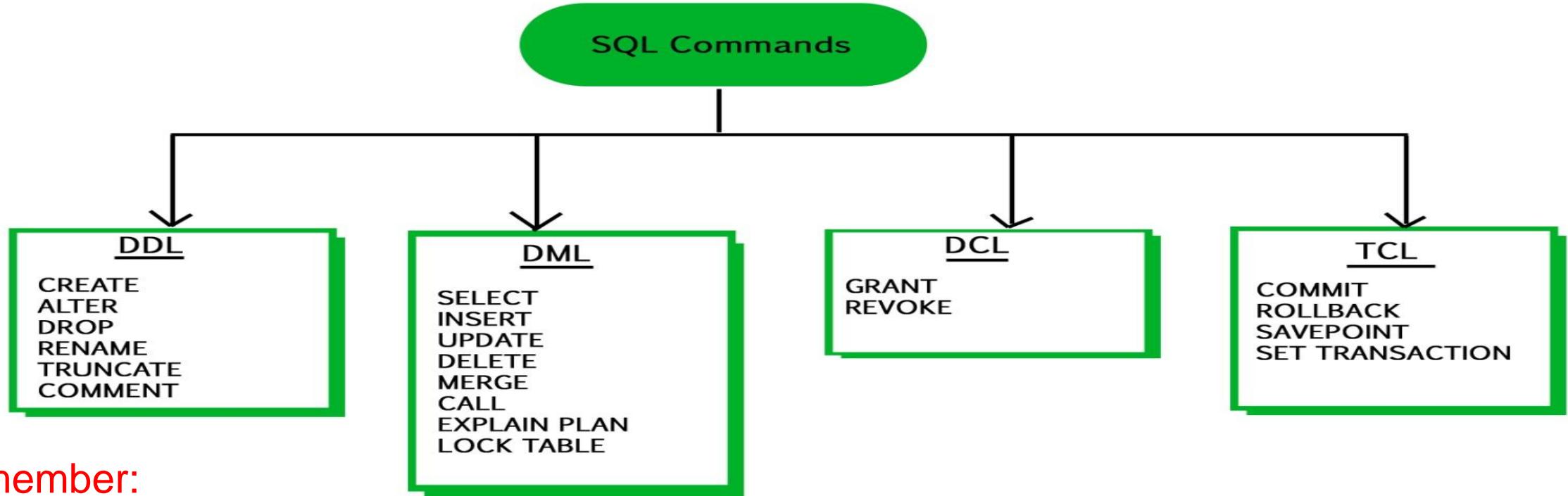
SQL (Structured Query Language) is a database language designed and developed for managing data in relational database management systems (RDBMS). SQL is common language for all Relational Databases.



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	<code>\r</code>
Execute a system shell command	<code>!</code>
Exit mysql	<code>\q</code>
Change your mysql prompt.	<code>prompt str or \R str</code>

Tablespace, Control files, Data  
files

# *database storage structures*

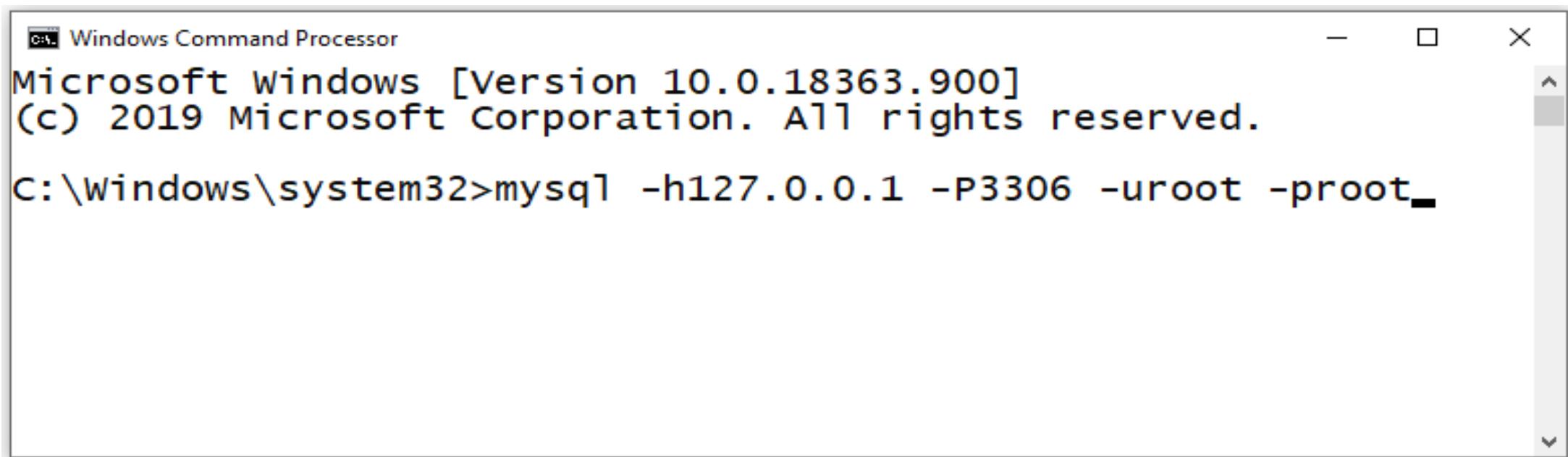
```
SHOW VARIABLES LIKE 'datadir';
```

# 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 "Windows Command Processor". The window displays the following text:

```
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_-
```

- Before MySQL version 5.5, MyISAM is the default storage engine.
- From version 5.5, MySQL uses InnoDB as the default storage engine.

# STORAGE ENGINES

A storage engine is a software module that a database management system uses to create, read, update data from a database. There are two types of storage engines in MySQL: **transactional** and **non-transactional**.

Storage Engine	File on disk
MEMORY	Data is not stored on the disk
InnoDB	.ibd (data and index)
MyISAM	MYD (data), .MYI (index)
CSV	.CSV (data), CSM (metadata)

# SHOW ENGINES Syntax

For MySQL 5.5 and later, the default storage engine is *InnoDB*. The default storage engine for MySQL prior to version 5.5 was *MyISAM*.

## SHOW [STORAGE] ENGINES

SHOW ENGINES displays status information about the server's storage engines. This is particularly useful for checking whether a storage engine is supported, or to see what the default engine is.

### INFORMATION\_SCHEMA.ENGINES

- `show engines;`
- `show STORAGE engines;`

**INFORMATION\_SCHEMA** provides access to database metadata, information about the MySQL server such as the name of a database or table, the data type of a column, or access privileges.



# ENGINES

- `show engines;`
- `show STORAGE engines;`

```
mysql> show engines;
```

Engine	Support	Comment	Transactions	XA	Savepoints
MEMORY	YES	Hash based, stored in memory, useful for temporary tables	NO	NO	NO
MRG_MYISAM	YES	Collection of identical MyISAM tables	NO	NO	NO
CSV	YES	CSV storage engine	NO	NO	NO
FEDERATED	NO	Federated MySQL storage engine	NULL	NULL	NULL
PERFORMANCE_SCHEMA	YES	Performance Schema	NO	NO	NO
MyISAM	YES	MyISAM storage engine	NO	NO	NO
InnoDB	DEFAULT	Supports transactions, row-level locking, and foreign keys	YES	YES	YES
BLACKHOLE	YES	/dev/null storage engine (anything you write to it disappears)	NO	NO	NO
ARCHIVE	YES	Archive storage engine	NO	NO	NO

```
9 rows in set (0.00 sec)
```

```
mysql> ■
```

- `SET DEFAULT_STORAGE_ENGINE = MyISAM;`

**MEMORY** tables are visible to another client, but  
**TEMPORARY** tables are not visible to another client.

# ENGINES

- **InnoDB** is the most widely used storage engine with transaction support. It is the only engine which provides foreign key referential integrity constraint. Oracle recommends using InnoDB for tables except for specialized use cases.
- **MyISAM** is the original storage engine. It is a fast storage engine. It does not support transactions. MyISAM provides table-level locking. It is used mostly in Web and data warehousing.
- **Memory** storage engine creates tables in memory. It is the fastest engine. It provides table-level locking. It does not support transactions. Memory storage engine is ideal for creating temporary tables or quick lookups. The data is lost when the database is re-started.
- **CSV** stores data in CSV files. It provides great flexibility because data in this format is easily integrated into other applications.

# ENGINES

- MEMORY tables are visible to another client, but
- TEMPORARY tables are not visible to another client.

In the case of InnoDB, it is used to store the tables in tablespace whereas, in the case of MyISAM, it stores each MyISAM table in a separate file.

In my.ini do this changes.

```
[mysqld]
innodb_file_per_table = 1
```

Tablespace: A data file that can hold data for one or more InnoDB tables and associated indexes.

- a. System tablespace
- b. File per tablespace
- c. General tablespace

Note:- By default, InnoDB contains only one tablespace called the System tablespace whose identifier is 0

SHOW DATABASES

# SHOW DATABASES Syntax

SHOW {DATABASES | SCHEMAS} [LIKE 'pattern' | WHERE *expr*]

SHOW SCHEMAS is a synonym for SHOW DATABASES.

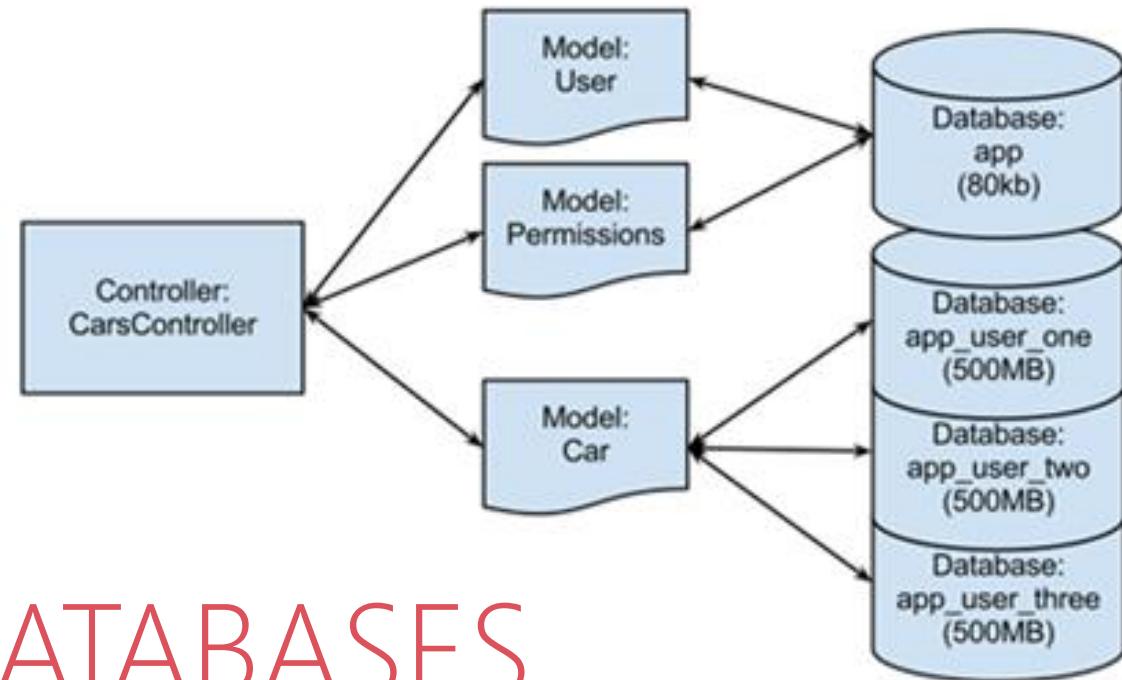
SHOW DATABASES;

SHOW SCHEMAS;

SHOW DATABASES LIKE 'U%';

SHOW SCHEMAS LIKE 'U%';

**NULL** means “no database is selected”. Issue the **USE dbName** command to select the database.



# USE DATABASES

The **USE db\_name** statement tells MySQL to use the db\_name database as the default (current) database for subsequent statements. The database remains the default until the end of the session or another **USE** statement is issued.

# USE DATABASES Syntax

`USE db_name`

`\U db_name`

## Note:

- USE, does not require a semicolon.
- USE must be followed by a database name.

`USE db1`

`\U db1`

CREATE DATABASE  
ALTER DATABASE

# *create / alter database*

CREATE DATABASE creates a database with the given name. To use this statement, you need the CREATE privilege for the database.

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name  
ALTER {DATABASE | SCHEMA} [db_name] READ ONLY [=] { 0 | 1}
```

**CREATE SCHEMA** is a synonym for **CREATE DATABASE**.

- `CREATE DATABASE db1;`
- `CREATE DATABASE IF NOT EXISTS db1;`
  
- `ALTER DATABASE db1 READ ONLY = 0; // is in read write mode.`
- `ALTER DATABASE db1 READ ONLY = 1; // is in read only mode.`

## Note:

- It is **not** possible to Create, Alter, Drop any object, and Write (Insert, Update, and Delete rows) in a read-only database.
- TEMPORARY tables; it is possible to create, alter, drop, and write (Insert, Update, and Delete rows) to TEMPORARY tables in a read-only database.

# DROP DATABASE

If the default database is dropped, the default database is unset (the DATABASE() function returns NULL).

# *drop database*

DROP DATABASE drops all tables in the database and deletes the database. Be very careful with this statement! To use DROP DATABASE, you need the DROP privilege on the database.

`DROP {DATABASE | SCHEMA} [IF EXISTS] db_name`

**DROP SCHEMA** is a synonym for **DROP DATABASE**.

`DROP DATABASE db1;`

`DROP DATABASE IF EXISTS db1;`

# Information Functions

Use these function with SELECT statement

# *information function*

- **FOUND\_ROWS()** - For a SELECT with a LIMIT clause, the number of rows that would be returned were there no LIMIT clause

e.g.

- `SELECT * FROM emp;`
- `SELECT FOUND_ROWS();`

- **LAST\_INSERT\_ID()** - Value of the AUTOINCREMENT column for the last INSERT
- **ROW\_COUNT()** - The number of rows updated.

e.g.

- `SELECT * FROM emp;`
- `SELECT ROW_COUNT();`

- **DATABASE()** - Return the default (current) database name

e.g.

- `SELECT DATABASE();`

- **SCHEMA()** - Synonym for DATABASE()

# *information function*

- **CURRENT\_USER()**, **CURRENT\_USER** - The authenticated user name and host name
- **USER()** - The user name and host name provided by the client

e.g.

- `SELECT USER();`

- **SESSION\_USER()** - Synonym for **USER()**
- **SYSTEM\_USER()** - Synonym for **USER()**
- **VERSION()** - Return a string that indicates the MySQL server version

e.g.

- `SELECT VERSION();`

- **CONNECTION\_ID()** - Return the connection ID (thread ID) for the connection

```
mysqlshow [options] [db_name [tbl_name [col_name]]]
```

## mysqlshow.exe

The mysqlshow client can be used to quickly see which databases exist, their tables, or a table's columns or indexes

# *mysqlshow.exe*

## **mysqlshow** – Get Quick Info On MySQL DB, Table, Column and Indexes

[Display available databases]

- C:\> mysqlshow -hlocalhost -P3306 -uroot -proot

[Display all tables in a database]

- C:\> mysqlshow -hlocalhost -P3306 -uroot -proot db1

[Display tables along with number of columns in a database]

- C:\> mysqlshow -hlocalhost -P3306 -uroot -proot db1 -v

[Display total number of columns and rows of all tables in a database]

- C:\> mysqlshow -hlocalhost -P3306 -uroot -proot db1 -v -v

# *mysqlshow.exe*

## **mysqlshow** – Get Quick Info On MySQL DB, Table, Column and Indexes

[Display all columns of a table]

- C:\> mysqlshow -hlocalhost -P3306 -uroot -proot db1 emp

[Display details about a specific column from a table]

- C:\> mysqlshow -hlocalhost -P3306 -uroot -proot db1 emp empno

[Display both indexes and columns of a table]

- C:\> mysqlshow -hlocalhost -P3306 -uroot -proot db1 emp -k

# Source Command

## *source command*

You can execute an SQL script file using the **source command** or **\. command**

`\. file_name`

`source file_name`

- `\. 'D:\mysqldemobld7.sql'`
- `SOURCE 'D:\mysqldemobld7.sql'`
- `SOURCE //infoserver1/infodomain1/Everyone/DBT/mysqldemobld7.sql`

SHOW COLUMNS

# EMP & DEPT Table structure

```
mysql> show columns from emp;
```

Field	Type	Null	Key	Default	Extra
EMPNO	int	NO	PRI	NULL	
ENAME	varchar(12)	YES		NULL	
GENDER	char(1)	YES		NULL	
JOB	varchar(20)	YES	MUL	NULL	
MGR	int	YES	MUL	NULL	
HIREDATE	date	YES		NULL	
SAL	int	YES		NULL	
COMM	int	YES		NULL	
DEPTNO	int	NO	MUL	NULL	
BONUSID	int	YES		NULL	
USER NAME	varchar(20)	YES		NULL	
PWD	varchar(20)	YES		NULL	
PHONE	varchar(45)	YES		NULL	
isActive	tinyint(1)	YES		NULL	

```
14 rows in set (0.00 sec)
```

```
mysql> show columns from dept;
```

Field	Type	Null	Key	Default	Extra
DEPTNO	int	NO	PRI	NULL	
DNAME	varchar(12)	YES		NULL	
LOC	varchar(10)	YES		NULL	
PWD	varchar(20)	YES		NULL	
STARTEDON	varchar(10)	YES		NULL	

```
5 rows in set (0.00 sec)
```

# SHOW COLUMNS Syntax

```
SHOW [FULL] {COLUMNS | FIELDS} {FROM | IN} tbl_name [{FROM | IN}  
db_name] [LIKE 'pattern' | WHERE expr]
```

- SHOW COLUMNS FROM emp;
- SHOW COLUMNS IN emp;
- SHOW FULL COLUMNS FROM emp; # WITH PRIVILEGES
- SHOW COLUMNS FROM emp FROM dbName;
- SHOW COLUMNS FROM user01.emp;
- SHOW COLUMNS FROM emp LIKE 'E%'; # STARTING WITH E
- SHOW COLUMNS FROM emp WHERE FIELD IN ('ename'); # ONLY ENAME COLUMN

SHOW TABLES

# SHOW TABLES Syntax

```
SHOW [FULL] TABLES [{FROM | IN} db_name] [LIKE 'pattern' | WHERE expr]
```

- SHOW TABLES;
- SHOW FULL TABLES; // WITH TABLE TYPE
- SHOW TABLES FROM USER01;
- SHOW TABLES WHERE TABLES\_IN\_USER01 LIKE 'E%' OR TABLES\_IN\_USER01 LIKE 'B%';
- SHOW TABLES WHERE TABLES\_IN\_USER01 IN ('EMP');

SHOW TABLES STATUS

# SHOW TABLES STATUS Syntax

SHOW TABLE STATUS [{FROM | IN} *db\_name*] [LIKE '*pattern*' | WHERE *expr*]

- SHOW TABLE STATUS;
- SHOW TABLE STATUS FROM user01;
- SHOW TABLE STATUS IN user01;
- SHOW TABLE STATUS LIKE 'emp';

# SHOW VARIABLES

shows the values of MySQL system variables.

# SHOW VARIABLES Syntax

SHOW [GLOBAL | SESSION] VARIABLES [LIKE '*pattern*' | WHERE *expr*]

```
SET SQL_SAFE_UPDATES = 0;  
SET SQL_SAFE_UPDATES = false;
```

Enable/Disable :- 1 (true) / 0 (false)

explain analyze

# *explain*

The DESCRIBE and EXPLAIN statements are synonyms. In practice, the DESCRIBE keyword is more often used to obtain information about table structure, whereas EXPLAIN is used to obtain a query execution plan.

```
{EXPLAIN | DESCRIBE | DESC}  
tbl_name [col_name]
```

- `DESC emp;`
- `DESC emp ename;`
- `DESCRIBE emp;`
- `EXPLAIN emp;`
- `EXPLAIN emp empno;`

**EXPLAIN works with SELECT, DELETE, INSERT, REPLACE, and UPDATE statements.**

# *explain*

The DESCRIBE and EXPLAIN statements are synonyms. In practice, the DESCRIBE keyword is more often used to obtain information about table structure, whereas EXPLAIN is used to obtain a query execution plan.

```
{EXPLAIN | DESCRIBE | DESC}  
{explainable_stmt}
```

```
explainable_stmt: {  
    SELECT statement  
    | DELETE statement  
    | INSERT statement  
    | REPLACE statement  
    | UPDATE statement  
}
```

- EXPLAIN ANALYZE SELECT \* FROM emp;
- EXPLAIN ANALYZE SELECT \* FROM emp **where** empno = 7788;
- EXPLAIN ANALYZE SELECT \* FROM emp INNER JOIN dept USING(deptno);

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

```
CREATE TABLE temp (c1 CHAR(10), c2 VARCHAR(10));  
INSERT INTO temp VALUES('SALEEL', 'SALEEL');  
SELECT * FROM temp WHERE c1 LIKE 'SALEEL';
```

## datatypes

ENAME CHAR (10)	S	A	L	E	E	L					LENGTH -> 10
ENAME VARCHAR2(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)

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

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)		

By default, trailing spaces are trimmed from CHAR column values on retrieval. If **PAD\_CHAR\_TO\_FULL\_LENGTH** is enabled, trimming does not occur and retrieved CHAR values are padded to their full length.

- *SET sql\_mode = '';*
- *SET sql\_mode = 'PAD\_CHAR\_TO\_FULL\_LENGTH';*

# *example of char and varchar*

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).

## Try Out

- `CREATE TABLE x (x1 CHAR(4), x2 VARCHAR(4));`
- `INSERT INTO x VALUE("", "");`
- `INSERT INTO x VALUE('ab', 'ab');`
- `INSERT INTO x VALUE('abcd', 'abcd');`
- `SELECT x1, LENGTH(x1), x2, LENGTH(x2) FROM x;`
- `SET sql_mode = 'PAD_CHAR_TO_FULL_LENGTH';`
- `SELECT x1, LENGTH(x1), x2, LENGTH(x2) FROM x;`
- `SET sql_mode = "";`
- `SELECT x1, LENGTH(x1), x2, LENGTH(x2) FROM x;`

\* In CHAR, if a table contains value 'a', an attempt to store 'a ' causes a duplicate-key error.

- `CREATE TABLE x (x1 CHAR(4) PRIMARY KEY, x2 VARCHAR(4));`
- `INSERT INTO x VALUE('a', 'a');`
- `INSERT INTO x VALUE('a ', 'a ');`

- 
- `CREATE TABLE x (x1 CHAR(4), x2 VARCHAR(4) PRIMARY KEY);`
  - `INSERT INTO x VALUE('a', 'a');`
  - `INSERT INTO x VALUE('a ', 'a ');`

## *example of text, blob, and longblob*

### Try Out

- `CREATE TABLE movies( _id INT AUTO_INCREMENT PRIMARY KEY, description TEXT, img1 BLOB, img2 LONGBLOB);`
- `INSERT INTO movies VALUE(default, load_file("d:/movie1.txt") , load_file("d:/img1.jpg") , load_file("d:/img2.jpg") );`
- `INSERT INTO movies VALUE(default, load_file("d:/movie2.txt") , load_file("d:/img1.jpg") , load_file("d:/img2.jpg") );`
- `INSERT INTO movies VALUE(default, load_file("d:\\movie3.txt") , load_file("d:\\\\img1.jpg") , load_file("d:\\\\img2.jpg") );`
- `SELECT * FROM movies;`

# *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

For: `float(M,D)`, `double(M,D)` or `decimal(M,D)`, M must be  $\geq D$

Here, **(M,D)** means than values can be stored with up to **M** digits in total, of which **D** digits may be after the decimal point.

**UNSIGNED** prohibits negative values.

## *datatype – date and time*

Datatypes	Size	Description
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

# datatype – boolean

```
CREATE TABLE temp (col1 INT ,col2 BOOL, col3 BOOLEAN);
```

```
CREATE TABLE tasks ( id INT AUTO_INCREMENT PRIMARY KEY, title VARCHAR(255) NOT NULL, completed BOOLEAN);
```

- INSERT INTO tasks VALUE(default, 'Task1', 0);
- INSERT INTO tasks VALUE(default, 'Task2', 1);
- INSERT INTO tasks VALUE(default, 'Task3', False);
- INSERT INTO tasks VALUE(default, 'Task4', True);
- INSERT INTO tasks VALUE(default, 'Task5', null);
- INSERT INTO tasks VALUE(default, 'Task6', default);
- INSERT INTO tasks VALUE(default, 'Task7', 1 > 2);
- INSERT INTO tasks VALUE(default, 'Task8', 1 < 2);
- INSERT INTO tasks VALUE(default, 'Task9', 12);
- INSERT INTO tasks VALUE(default, 'Task10', 58);
- INSERT INTO tasks VALUE(default, 'Task11', .75);
- INSERT INTO tasks VALUE(default, 'Task12', .15);
- INSERT INTO tasks VALUE(default, 'Task13', 'a' = 'a');

	id	title	completed
▶	1	Task1	0
	2	Task2	1
	3	Task3	0
	4	Task4	1
	5	Task5	NULL
	6	Task6	NULL
	7	Task7	0
	8	Task8	1
	9	Task9	12
	10	Task10	58
	11	Task11	1
	12	Task12	0
	13	Task13	1
●	NULL	NULL	NULL

## Note:

- BOOL and BOOLEAN are synonym of TINYINT(1)

- An ENUM column can have a maximum of **65,535** distinct elements.

## *datatype – enum*

- ENUM values are sorted based on their index numbers, which depend on the order in which the enumeration members were listed in the column specification.
- Default value, NULL if the column can be NULL, first enumeration value if NOT NULL

- `CREATE TABLE temp (col1 INT, COL2 ENUM('A','B','C'));`
- `INSERT INTO temp (col1, col2) VALUES(1, 1);`
- `INSERT INTO temp(col1) VALUES (1); // NULL`
- `CREATE TABLE temp (col1 INT, col2 ENUM('A','B','C') NOT NULL);`
- `INSERT INTO temp(col1) VALUES (1); // First element from the ENUM datatype`
- `CREATE TABLE temp (col1 INT, col2 ENUM("") NOT NULL);`
- `INSERT INTO temp (col1, col2) VALUES (1,'This is the test'); // NULL`
- `CREATE TABLE temp (col1 INT, COL2 ENUM('A','B','C') default 'D' ); // Invalid default value for 'COL2'`

### IMP:

- MySQL maps [ membership `ENUM('Silver', 'Gold', 'Diamond', 'Platinum')` ] these enumeration member to a numeric index where Silver=1, Gold=2, Diamond=3, Platinum=4 respectively.

- An ENUM column can have a maximum of **65,535** distinct elements.

## *datatype – enum*

size `ENUM('small', 'medium', 'large', 'x-large')`

membership `ENUM('Silver', 'Gold', 'Diamond', 'Platinum')`

interest `ENUM('Movie', 'Music', 'Concert')`

zone `ENUM('North', 'South', 'East', 'West')`

season `ENUM('Winter', 'Summer', 'Monsoon', 'Autumn')`

sortby `ENUM('Popularity', 'Price -- Low to High', 'Price -- High to Low', 'Newest First')`

status `ENUM('active', 'inactive', 'pending', 'expired', 'shipped', 'in-process', 'resolved', 'on-hold', 'cancelled', 'disputed')`

### Note:

- You cannot use user variable as an enumeration value. This pair of statements do not work:

```
SET @mysize = 'medium';
```

```
CREATE TABLE sizes ( size ENUM('small', @mysize, 'large')); // error
```

## *datatype – set*

- A SET column can have a maximum of **64** distinct members.
- A SET is a string object that can have zero or more values, each of which must be chosen from a list of permitted values specified when the table is created.
- SET column values that consist of multiple set members are specified with members separated by commas (,) without leaving a spaces.

`CREATE TABLE clients(`

```
id INT AUTO_INCREMENT PRIMARY KEY,  
name VARCHAR(10),  
membership ENUM('Silver', 'Gold', 'Premium', 'Diamond'),  
interest SET('Movie', 'Music', 'Concert'));
```

`INSERT INTO clients (name, membership, interest) VALUES('Saleel', 'Gold', 'Music');`

`INSERT INTO clients (name, membership, interest) VALUES('Saleel', 'Premium', 'Movie, Concert');`

**IMP:**

- The SET data type allows you to specify a list of values to be inserted in the column, like ENUM. But, unlike the ENUM data type, which lets you choose only one value, the SET data type allows you to choose multiple values from the list of specified values.

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

```
CREATE TABLE `123` (c1 INT, c2 VARCHAR(10));
```

### Remember:

- Max 4096 columns per table provided the row size <= 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';

# *create table*

## **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

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;

# 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'));`

# *default value*

*col\_name data\_type* **DEFAULT** value

The **DEFAULT** specifies a **default** value for the column.

- `CREATE TABLE posts (  
 postID INT,  
 postTitle VARCHAR(255),  
 postDate DATETIME DEFAULT NOW(),  
 deleted INT  
);`

# version 8.0 and above.

- `CREATE TABLE empl (  
 ID INT PRIMARY KEY,  
 firstName VARCHAR(45),  
 phone INT,  
 city VARCHAR(10) DEFAULT 'PUNE',  
 salary INT,  
 comm INT,  
 total INT DEFAULT(salary + comm)  
);`

	Field	Type	Null	Key	Default	Extra
▶	postID	int	YES		NULL	
▶	postTitle	varchar(255)	YES		NULL	
▶	postDate	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED
▶	deleted	int	YES		NULL	

	Field	Type	Null	Key	Default	Extra
▶	ID	int	NO	PRI	NULL	
▶	firstName	varchar(45)	YES		NULL	
▶	phone	int	YES		NULL	
▶	city	varchar(10)	YES		PUNE	
▶	salary	int	YES		NULL	
▶	comm	int	YES		NULL	
▶	total	int	YES		(`salary` + `comm`)	DEFAULT_GENERATED

## *default value - insert*

The **DEFAULT** example.

- `CREATE TABLE t (`  
    `c1 INT,`  
    `c2 INT DEFAULT 1,`  
    `c3 INT DEFAULT 3,`  
    `);`
- `INSERT INTO t VALUES();`
- `INSERT INTO t VALUES(-1, DEFAULT, DEFAULT);`
- `INSERT INTO t VALUES(-2, DEFAULT(c2), DEFAULT(c3));`
- `INSERT INTO t VALUES(-3, DEFAULT(c3), DEFAULT(c2));`

	Field	Type	Null	Key	Default	Extra
▶	c1	int	YES		NUL	
	c2	int	YES		1	
	c3	int	YES		3	

## *default value - update*

The **DEFAULT** example.

- `CREATE TABLE temp ( c1 INT, c2 INT, c3 INT DEFAULT(c1 + c2), c4 INT DEFAULT(c1 * c2 ) );`
- `INSERT INTO temp(c1, c2, c3, c4) VALUES(1, 1, 1, 1);`
- `INSERT INTO temp(c1, c2, c3, c4) VALUES(2, 2, 2, 2);`
- `UPDATE temp SET c3 = DEFAULT;`
- `UPDATE temp SET c4 = 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**

***INSERT** can violate for any of the four types of constraints.*

**Important:**

- If an attribute value is not of the appropriate data type.
- Entity integrity can be violated if a key value in the new tuple  $t$  already exists in another tuple in the relation  $r(R)$ .
- Entity integrity can be violated if any part of the primary key of the new tuple  $t$  is NULL.
- Referential integrity can be violated if the value of any foreign key in  $t$  refers to a tuple that does not exist in the referenced relation.

***INSERT** will also fail in following cases.*

**Important :**

- Your database table has **X** columns, Where as the **VALUES** you are passing are for **(X-1)** or **(X+1)**. This mismatch of column-values will give you the error.
- Inserting a string into a string column that exceeds the column maximum length. Data too long for column error will be raised.
- Inserting data into a column that does not exist, then Unknown column error will raise.

- **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.**
- A second form of the **INSERT** statement allows the user to specify explicit attribute names that correspond to the values provided in the **INSERT** command. This is useful if a relation has many attributes but only a few of those attributes are assigned values in the new tuple. However, the values must include all attributes with **NOT NULL** specification and no default value. Attributes with **NULL** allowed or **DEFAULT** values are the ones that can be left out.

insert rows using values

## *dml- insert ... values*

INSERT inserts new row(s) into an existing table. The INSERT ... VALUES

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

The affected-rows value for an INSERT can be obtained using the ROW\_COUNT() function.

**INSERT INTO DEPT VALUES (1, 'HRD', 'Pune')**

Column Values

**INSERT INTO DEPT(ID, NAME, LOC) VALUES (1, 'HRD', 'Pune')**

Column List

**INSERT INTO DEPT(ID, NAME, LOC) VALUES (1, 'HRD', 'Baroda'),  
(2,'Sales','Surat'), (3,'Purchase','Pune'), (4,'Account','Mumbai')**

Inserting multiple rows

## *dml- insert ... values*

INSERT inserts new rows into an existing table. The INSERT ... VALUES

```
INSERT [IGNORE] [INTO] tbl_name [PARTITION (partition_name [, partition_name] ...)] [ (col_name, ...) ] {  
VALUES | VALUE } [ROW] ( { expr | DEFAULT }, ... ), [ROW] ( ... ), [ROW] ... [ ON DUPLICATE KEY UPDATE  
assignment_list ]
```

```
CREATE TABLE student (  
    ID INT PRIMARY KEY,  
    nameFirst VARCHAR(45),  
    nameLast VARCHAR(45),  
    DoB DATE ,  
    emailID VARCHAR(128)  
);
```

e.g.

- `INSERT INTO student VALUES (29, 'sharmin', 'patil', '1999-11-10', 'sharmin.patil@gmail.com');`
- `INSERT INTO student (ID, nameFirst, nameLast, DOB, emailID) VALUES (30, 'john', 'thomas', '1983-11-10', 'john.thomas@gmail.com');`
- `INSERT INTO student (ID, nameFirst, emailID) VALUES (31, 'jack', 'jack.thorn@gmail.com');`
- `INSERT INTO student (ID, nameFirst) VALUES (32, 'james'), (33, 'jr. james'), (34, 'sr. james');`

insert multiple rows

## *dml- insert ... values*

INSERT inserts new rows into an existing table. The INSERT ... VALUES

`INSERT [INTO] tbl_name { VALUES | VALUE } [ROW] ( { expr | DEFAULT }, . . . ), [ROW] ( . . . ), [ROW] ( . . . )`

```
CREATE TABLE student (
    ID INT PRIMARY KEY,
    nameFirst VARCHAR(45),
    nameLast VARCHAR(45),
    DoB DATE ,
    emailID VARCHAR(128)
);
```

e.g.

- `INSERT INTO student (ID, nameFirst) VALUES (32, 'james'), (33, 'jr. james'), (34, 'sr. james');`
- `INSERT INTO student (ID, nameFirst) VALUES ROW (32, 'james'), ROW(33, 'jr. james'), ROW(34, 'sr. james');`

encoding/decoding

## *aes\_encrypt / aes\_decrypt*

AES\_ENCRYPT() and AES\_DECRYPT() implement encryption and decryption of data using the official AES (Advanced Encryption Standard) algorithm, previously known as "Rijndael".

- `AES_ENCRYPT(str, key_str)`
- `AES_DECRYPT(str, key_str)`
- `CREATE TABLE pwd (`  
    `ID INT PRIMARY KEY,`  
    `name VARCHAR(45),`  
    `pwd VARBINARY(45)`  
`);`
- `INSERT INTO pwd VALUES(1, 'sharmin', AES_ENCRYPT('sharmin', 'sharmin1'));`
- `INSERT INTO pwd VALUES(2, 'saleel', AES_ENCRYPT('saleel', 'sharmin1'));`
- `INSERT INTO pwd VALUES(3, 'kaushal', AES_ENCRYPT('kaushal', 'kaushal1'));`
- `INSERT INTO pwd VALUES(4, 'ruhan', AES_ENCRYPT('ruhan', 'ruhan1'));`
- `SELECT * FROM pwd;`
- `SELECT ID, name, AES_DECRYPT(pwd, 'sharmin1') FROM pwd;`
- `SELECT ID, name, CAST(AES_DECRYPT(pwd, 'sharmin1') AS CHAR) FROM pwd;`

*md5()*

TODO

insert rows using select

# *insert ... select*

With INSERT ... SELECT, you can quickly insert many rows into a table from one or many tables.

`INSERT [INTO] tbl_name [(col_name, ...)] SELECT ...`

- `INSERT INTO dept(deptno) SELECT 1 + 1;`
- `INSERT INTO dept(deptno) SELECT deptno FROM dept;`
- `INSERT INTO dept SELECT * FROM dept;`
- `INSERT INTO dept(SELECT MAX(deptno) + 1, 'HRD', 'BARODA', 'r57px33px' FROM dept);`
- `set @x := 40;`
- `INSERT INTO dept VALUES (@x := @x + 1 , 'HRD', 'BARODA', 'r57px33px');`
- `set @x := (SELECT MAX(deptno) FROM dept);`
- `INSERT INTO dept VALUES (@x := @x + 1 , 'HRD', 'BARODA', 'r57px33px');`

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



**Table DEPARTMENTS**

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
60	IT	103	1400
90	Executive	100	1700

Projection  
Selection

**Table EMPLOYEES**

EMPLOYEE_ID	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	MANAGER_ID	DEPARTMENT_ID
100	King	SKING		AD_PRES	90	
101	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	100	90
102	De Hann	LDEHANN	13-JAN-93	AD_VP	100	90
103	Hunold	AHUNOLD		IT_PROG	102	60

# **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

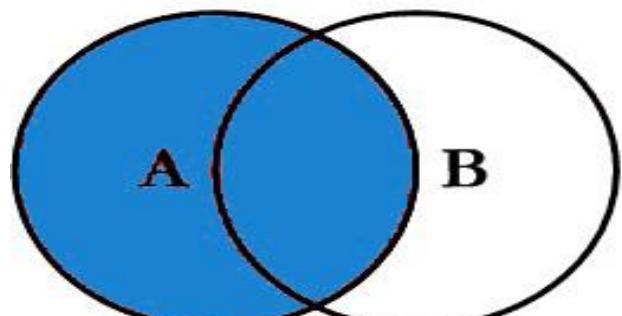


**S**

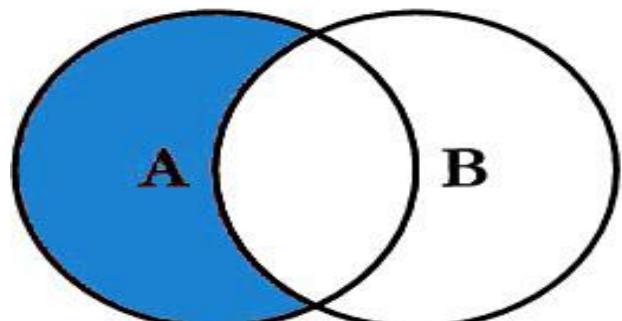
DEPTNO	DNAME	LOC
10	HRD	PUNE
20	SALES	BARODA
40	PURCHASE	SURAT



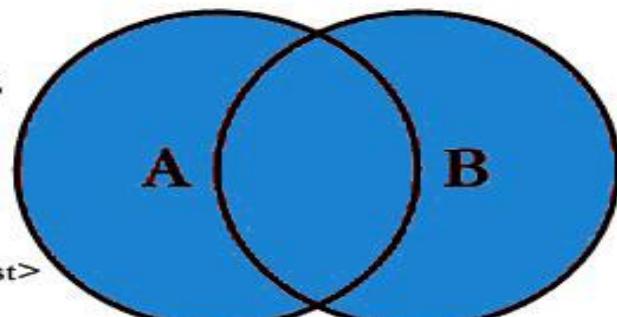
# SQL JOINS



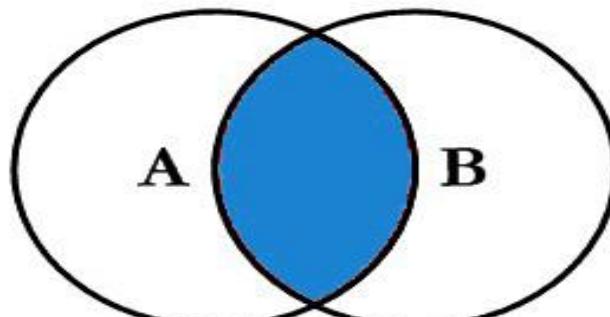
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key
```



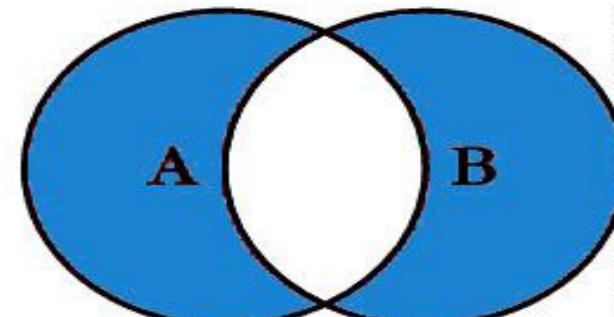
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key  
WHERE B.Key IS NULL
```



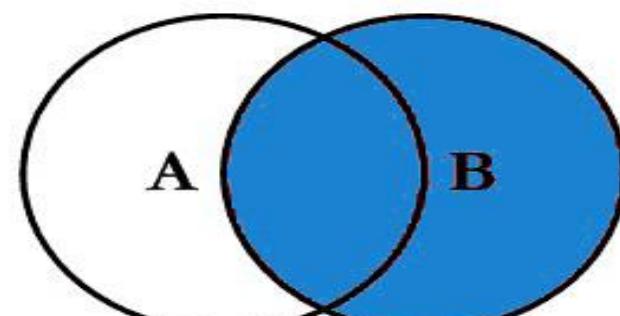
```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key
```



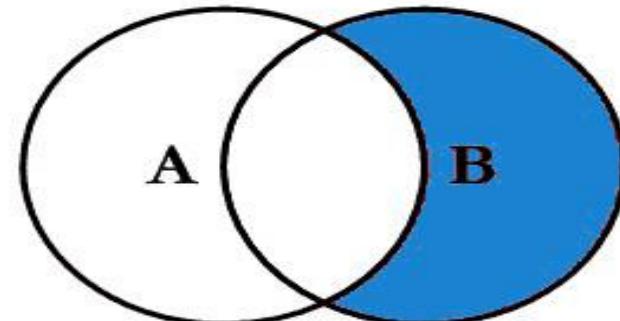
```
SELECT <select_list>  
FROM TableA A  
INNER JOIN TableB B  
ON A.Key = B.Key
```



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



```
SELECT <select_list>  
FROM TableA A  
RIGHT 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
```

# *select statement*

## 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)

ORDER BY in UPDATE: if the table contains two values 1 and 2 in the id column and 1 is updated to 2 before 2 is updated to 3, an error occurs. To avoid this problem, add an ORDER BY clause to cause the rows with larger id values to be updated before those with smaller values.

Note:

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

Here c1 column is a Primary Key

- `UPDATE temp SET c1 = c1 - 1 ORDER BY c1 ASC;` # In case of decrement
- `UPDATE temp SET c1 = c1 + 1 ORDER BY c1 DESC;` # In case of increment

## single-table update

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

- `SET @x := 0;`
- `UPDATE emp SET id = @x := @x + 1;`
- `UPDATE t, (SELECT isactive, COUNT(isactive) r1 FROM emp GROUP BY isactive) a SET t.c2 = a.r1 WHERE t.c1 = a.isactive;`

mysql> `SELECT * FROM t;`

c1	c2	c1	c2
0	NULL	0	6
1	NULL	1	14

e.g.

1. Update top 2 rows.
2. Update UnitPrice for the top 5 most expensive products.

*UPDATE can violate.*

**Important:**

- TODO

## *single-table update*

The UPDATE statement updates columns of existing rows in the named table with new values. The SET clause indicates which columns to modify and the values they should be given. The WHERE clause, if given, specifies the conditions that identify which rows to update. With no WHERE clause, all rows are updated. If the ORDER BY clause is specified, the rows are updated in the order that is specified. The LIMIT clause places a limit on the number of rows that can be updated.

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

- `UPDATE temp SET dname = 'new_value' LIMIT 2;`
- `UPDATE temp SET c1 = 'new_value' ORDER BY loc LIMIT 2;`
- `UPDATE temp SET c1 := 'new_value' WHERE deptno < 50;`
- `UPDATE temp SET c1 := 'new_value' WHERE deptno < 50 LIMIT 2;`
- `ALTER TABLE dept ADD SUMSALARY INT;`
- `UPDATE dept SET sumsalary = (SELECT SUM(sal) FROM emp WHERE emp.deptno = dept.deptno GROUP BY emp.deptno);`
- `UPDATE candidate SET totalvotes = (SELECT COUNT(*) FROM votes WHERE candidate.id = votes.candidateID GROUP BY votes.candidateID);`
- `UPDATE duplicate SET id = ( SELECT @cnt := @cnt + 1 );`

# single-table delete

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

*DELETE can violate only in referential integrity.*

**Important:**

- The **DELETE** operation can violate only referential integrity. This occurs if the tuple  $t$  being deleted is referenced by foreign keys from other tuple  $t$  in the database.

# *single-table delete*

The DELETE statement deletes rows from `tbl_name` and returns the number of deleted rows. To check the number of deleted rows, call the `ROW_COUNT()` function. The optional WHERE clause identify which rows to delete. With no WHERE clause, all rows are deleted. If the ORDER BY clause is specified, the rows are deleted in the order that is specified. The LIMIT clause places a limit on the number of rows that can be deleted.

`DELETE FROM tbl_name`

[`PARTITION` (`partition_name [, partition_name] ...`)]

[`WHERE` `where_condition`]

[`ORDER BY` ...]

[`LIMIT` `row_count`]

## Note:

- LIMIT clauses apply to single-table deletes, but not multi-table deletes.
- `DELETE FROM temp;`
- `DELETE FROM temp ORDER BY loc LIMIT 2;`
- `DELETE FROM temp WHERE deptno < 50;`
- `DELETE FROM temp WHERE deptno < 50 LIMIT 2;`

# auto\_increment column

The **AUTO\_INCREMENT** attribute can be used to generate a unique number/identity for new rows.

# *auto\_increment*

*IDENTITY* is a synonym to the *LAST\_INSERT\_ID* variable.

*col\_name data\_type AUTO\_INCREMENT [UNIQUE [KEY] | [PRIMARY] KEY]*

## Remember:

- There can be only one AUTO\_INCREMENT column per table.
- it must be indexed.
- it cannot have a DEFAULT value.
- it works properly only if it contains only positive values.
- It applies only to integer and floating-point types.
- when you insert a value of NULL or 0 into AUTO\_INCREMENT column, it generates next value.
- use *LAST\_INSERT\_ID()* function to find the row that contains the most recent AUTO\_INCREMENT value.

- 
- `SELECT @@IDENTITY`
  - `SELECT LAST_INSERT_ID()`
  - `SET INSERT_ID = 7`
  - `CREATE TABLE posts (`  
 `c1 INT UNIQUE KEY AUTO_INCREMENT,`  
 `c2 VARCHAR(20)`  
`) AUTO_INCREMENT = 2; // auto_number will start with value 2.`

# *auto\_increment*

The **auto\_increment** specifies a **auto\_increment** value for the column.

- `CREATE TABLE posts (postID INT AUTO_INCREMENT UNIQUE KEY, postTitle VARCHAR(255), postDate DATETIME DEFAULT NOW(), deleted INT);`

	Field	Type	Null	Key	Default	Extra
▶	postID	int	NO	PRI	NULL	auto_increment
	postTitle	varchar(255)	YES		NULL	
	postDate	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED
	deleted	int	YES		NULL	

- `CREATE TABLE comments (commentID INT AUTO_INCREMENT PRIMARY KEY, comment TEXT, commentDate DATETIME DEFAULT NOW(), deleted INT);`

	Field	Type	Null	Key	Default	Extra
▶	commentID	int	NO	PRI	NULL	auto_increment
	comment	text	YES		NULL	
	commentDate	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED
	deleted	int	YES		NULL	

- `CREATE TABLE animals(id INT NOT NULL AUTO_INCREMENT, breed INT, PRIMARY KEY (id, breed)); //valid`
- `CREATE TABLE animals(id NT NOT NULL, breed INT AUTO_INCREMENT, PRIMARY KEY (id, breed)); //invalid`

# *auto\_increment*

- **auto\_increment\_increment** – is the incremental value, controls the interval between successive column values.
  - **auto\_increment\_offset** – determines the starting value for the AUTO\_INCREMENT field; this value is used for the first record inserted into the table.
- 
- `SET @@auto_increment_offset=5;`
  - `SET @@auto_increment_increment=10;`
  - `ALTER TABLE temp AUTO_INCREMENT = 0;`

## Remember:

- If value of `auto_increment_offset` set is greater than that of `auto_increment_increment`, the value of `auto_increment_offset` is ignored.

## *auto\_increment*

*NO\_AUTO\_VALUE\_ON\_ZERO* affects handling of *AUTO\_INCREMENT* columns. Normally, you generate the next sequence number for the column by inserting either NULL or 0 into it.

*NO\_AUTO\_VALUE\_ON\_ZERO* suppresses this behaviour for 0 so that only NULL generates the next sequence number.

- *SET sql\_mode = '';*
- *SET sql\_mode = 'NO\_AUTO\_VALUE\_ON\_ZERO';*

- **CREATE TABLE ... LIKE ...**, the destination table *preserves generated column information* from the original table.
- **CREATE TABLE ... SELECT ...**, the destination table *does not preserves generated column information* from the original table.

## generated column

### Remember:

- Stored functions and user-defined functions are not permitted.
- Stored procedure and function parameters are not permitted.
- Variables (system variables, user-defined variables, and stored program local variables) are not permitted.
- Subqueries are not permitted.
- The AUTO\_INCREMENT attribute cannot be used in a generated column definition.
- Triggers cannot use NEW.COL\_NAME or use OLD.COL\_NAME to refer to generated columns.
- Stored column cannot be converted to virtual column and virtual column cannot be converted to stored column.
- Generated column can be made as invisible column.

### Note:

- The expression can contain literals, built-in functions with no parameters, operators, or references to any column within the same table. If you use a function, it must be scalar and deterministic.

# *virtual column - generated always*

`col_name data_type [GENERATED ALWAYS] AS (expression) [VIRTUAL | STORED]`

- **VIRTUAL**: Column values are not stored, but are evaluated when rows are read, immediately after any BEFORE triggers. A virtual column takes no storage.
- **STORED**: Column values are evaluated and stored when rows are inserted or updated. A stored column does require storage space and can be indexed.

## Note:

- The default is **VIRTUAL** if neither keyword is specified.

- `CREATE TABLE product(`

`productCode INT AUTO_INCREMENT PRIMARY KEY,`

`productName VARCHAR(45),`

`productVendor VARCHAR(45),`

`productDescription TEXT,`

`quantityInStock INT,`

`buyPrice FLOAT,`

`stockValue FLOAT GENERATED ALWAYS AS(quantityInStock * buyPrice) VIRTUAL`

`);`

	Field	Type	Null	Key	Default	Extra
▶	productCode	int	NO	PRI	NULL	auto_increment
	productName	varchar(45)	YES		NULL	
	productVendor	varchar(45)	YES		NULL	
	productDescription	text	YES		NULL	
	quantityInStock	int	YES		NULL	
	buyPrice	float	YES		NULL	
	stockValue	float	YES		NULL	VIRTUAL GENERATED

*col\_name data\_type [GENERATED ALWAYS] AS (expression) [VIRTUAL | STORED]*

**Note:**

- Virtual column cannot be converted to stored column.
  - `CREATE TABLE product1(`  
    `productCode INT,`  
    `quantityInStock INT,`  
    `buyPrice INT,`  
    `stockValue FLOAT GENERATED ALWAYS AS (quantityInStock * buyPrice) VIRTUAL`  
`);`
  - `ALTER TABLE product1 MODIFY stockValue FLOAT (12, 2); // error: We are trying to convert virtual column [`  
    `GENERATED ALWAYS AS ] to stored column by not giving GENERATED ALWAYS AS`
- Then What→ give
- `ALTER TABLE product1 MODIFY stockValue FLOAT (12, 2) GENERATED ALWAYS AS (quantityInStock * buyPrice) VIRTUAL ;`

`col_name data_type [GENERATED ALWAYS] AS (expression) [VIRTUAL | STORED]`

**Note:**

- **Stored column cannot be converted to virtual column.**
- `CREATE TABLE product1(`  
`productCode INT,`  
`quantityInStock INT,`  
`buyPrice INT,`  
`stockValue FLOAT`  
`);`
- `ALTER TABLE product1 MODIFY stockValue FLOAT GENERATED ALWAYS AS (quantityInStock * buyPrice) VIRTUAL; //`  
**error:** We are trying to convert stored column to virtual column by giving GENERATED ALWAYS AS

# visible / invisible columns

Columns are visible by default. To explicitly specify visibility for a new column, use a VISIBLE or INVISIBLE keyword as part of the column definition for CREATE TABLE or ALTER TABLE.

## Note:

- An invisible column is normally hidden to queries, but can be accessed if explicitly referenced. Prior to MySQL 8.0.23, all columns are visible.
- A table must have at least one visible column. Attempting to make all columns invisible produces an error.
- SELECT \* does not include invisible columns.

# *invisible column*

*col\_name data\_type INVISIBLE*

```
CREATE TABLE employee (
    ID INT AUTO_INCREMENT PRIMARY KEY,
    firstName VARCHAR(40),
    salary INT,
    commission INT,
    total INT DEFAULT(salary + commission) INVISIBLE
    tax INT GENERATED ALWAYS AS (total * .25) VIRTUAL INVISIBLE
);
```

- `INSERT INTO employee(firstName, salary, commission) VALUES('ram', 4700, -700);`
- `INSERT INTO employee(firstName, salary, commission) VALUES('pankaj', 3400, NULL);`
- `INSERT INTO employee(firstName, salary, commission) VALUES('rajan', 3200, 250);`
- `INSERT INTO employee(firstName, salary, commission) VALUES('ninad', 2600, 0);`
- `INSERT INTO employee(firstName, salary, commission) VALUES('omkar', 4500, 300);`
- `SELECT * FROM employee;`
  
- `ALTER TABLE employee MODIFY total INT VISIBLE;`
- `ALTER TABLE employee MODIFY total INT INVISIBLE;`

```
CREATE TABLE employee (
    ID INT PRIMARY KEY AUTO_INCREMENT INVISIBLE ,
    firstName VARCHAR(40)
);
```

# varbinary column

TODO

## Note:

- TODO
- TODO
- TODO

# *varbinary column*

*col\_name* VARBINARY

```
CREATE TABLE login (
    ID INT AUTO_INCREMENT PRIMARY KEY,
    userName VARCHAR(40),
    password VARBINARY(40) INVISIBLE
);
```

- `INSERT INTO login(userName, password) VALUES('ram', 'ram@123');`
- `INSERT INTO login(userName, password) VALUES('pankaj', 'pankaj');`
- `INSERT INTO login(userName, password) VALUES('rajan', 'rajan');`
- `INSERT INTO login(userName, password) VALUES('ninad', 'ninad');`
- `INSERT INTO login(userName, password) VALUES('omkar', 'omkar');`
  
- `SELECT * FROM login;`
- `SELECT username, CAST(password as CHAR) FROM login;`

a1 INT(10) zerofill	0	0	0	0	0	0	1	5	0	0	length(ID1) -> 10
a2 INT(10)							3	6	7	5	length(ID2) -> 4

## zerofill value

When you select a column with the type ZEROFILL it pads the displayed value of the field with zeros up to the display width specified in the column definition.

### Note:

- ZEROFILL attribute is use for numeric datatype.
- If you specify ZEROFILL for a column, MySQL automatically adds the **UNSIGNED** attribute to the column.

## *zerofill column*

*col\_name data\_type* **ZEROFILL**

```
CREATE TABLE employee (
    ID INT ZEROFILL AUTO_INCREMENT PRIMARY KEY,
    firstName VARCHAR(40),
    salary INT,
    commission INT,
    total INT DEFAULT(salary + commission)
);
```

- `INSERT INTO employee VALUES(NULL, 'ram', 4700, NULL, default);`
- `INSERT INTO employee VALUES(0, 'pankaj', 3400, 400 , default);`
- `INSERT INTO employee VALUES(100, 'rajan', 3200, NULL , default);`
- `INSERT INTO employee VALUES(NULL, 'ninad', 2600, 0, default);`
- `INSERT INTO employee VALUES(0, 'omkar', 4500, 300, default);`
- `INSERT INTO employee VALUES (-200, 'rahul', 3000, 300 , default);`
  
- `SELECT * FROM employee;`
- `SELECT ID, LENGTH(ID), salary, LENGTH(salary) FROM employee;`

## *zerofill column*

- `CREATE TABLE account (`  
    `accountNumber INT ZEROFILL PRIMARY KEY,`  
    `balance FLOAT,`  
    `openingBalance FLOAT,`  
    `accountName VARCHAR(45),`  
    `customerID INT,`  
    `openingDate DATE`  
`);`

	Field	Type	Null	Key	Default	Extra
▶	accountNumber	int(10) unsigned zerofill	NO	PRI	NULL	
	balance	float	YES		NULL	
	openingBalance	float	YES		NULL	
	accountName	varchar(45)	YES		NULL	
	customerID	int	YES		NULL	
	openingDate	date	int	YES	NULL	

### Note:

- If you specify ZEROFILL for a numeric column, MySQL automatically adds the **UNSIGNED** attribute to the column.

MySQL Constraints define specific rules to the column(s) data in a database table. While inserting, updating, or deleting the data rows, if the rules of the constraint are not followed, the system will display an error message and the action will be terminated. The SQL Constraints are defined while creating a new table. We can also alter the table and add new SQL Constraints. The MySQL Constraints are mainly used to maintain data integrity.

# 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*

The column level constraints can apply only to one column where as table level constraints are applied to the entire table.

## 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:

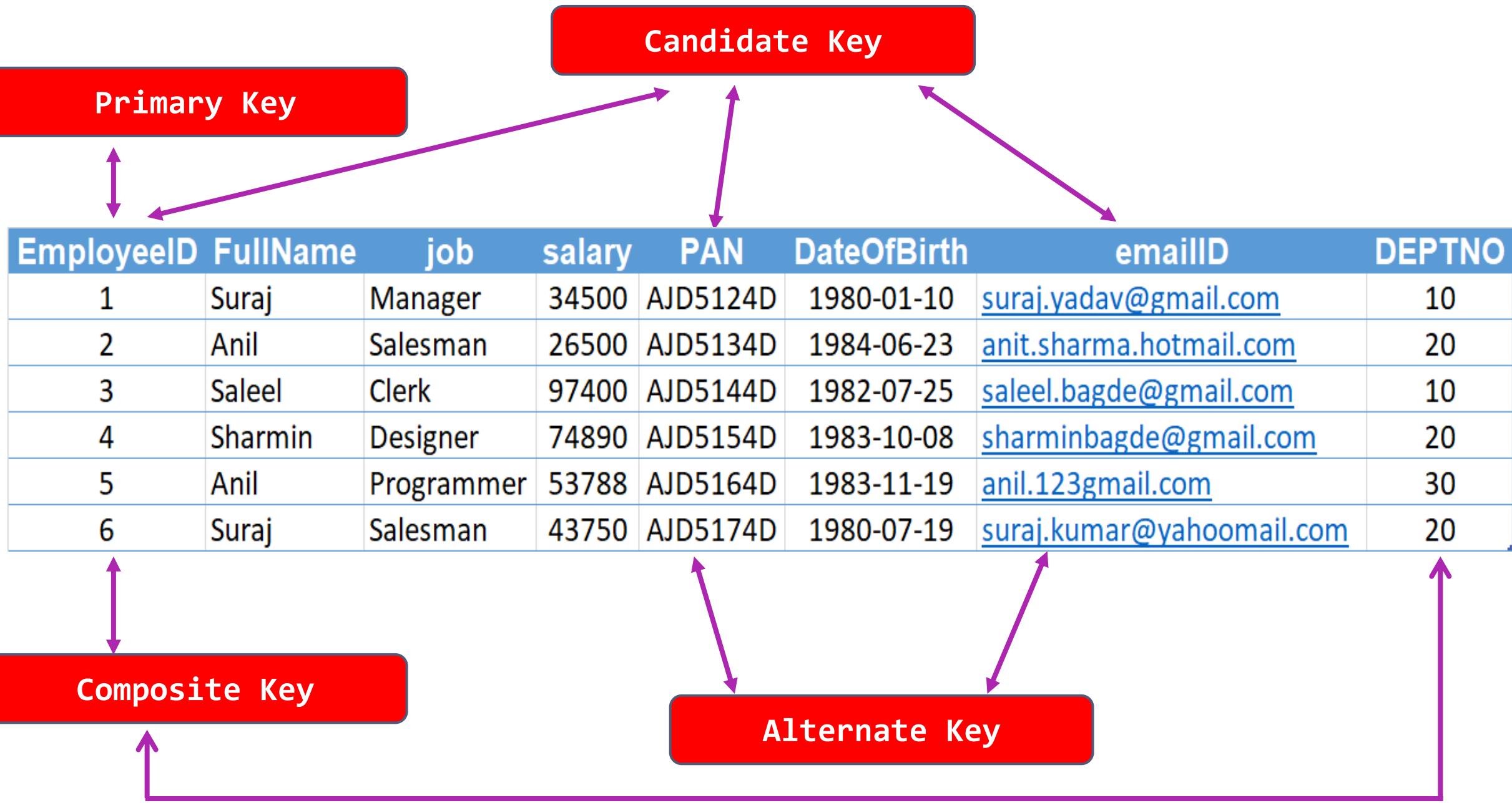
- a table with a foreign key that references another table's primary key is **MUL**.
- If more than one of the Key values applies to a given column of a table, Key displays the one with the highest priority, in the order **PRI**, **UNI**, and **MUL**.
- If a table has a PRIMARY KEY or UNIQUE NOT NULL 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.

Keys are used to establish relationships between tables and also to uniquely identify any record in the table.

## *types of Keys?*

$r = \text{Employee}(\text{EmployeeID}, \text{FullName}, \text{job}, \text{salary}, \text{PAN}, \text{DateOfBirth}, \text{emailID}, \text{deptno})$

- **Candidate Key:** are individual columns in a table that qualifies for uniqueness of all the rows. Here in Employee table EmployeeID, PAN or emailID are Candidate keys.
- **Primary Key:** is the columns you choose to maintain uniqueness in a table. Here in Employee table you can choose either EmployeeID, PAN or emailID columns, EmployeeID is preferable choice.
- **Alternate Key:** Candidate column other the primary key column, like if EmployeeID is primary key then , PAN or emailID columns would be the Alternate key.
- **Super Key:** If you add any other column to a primary key then it become a super key, like EmployeeID + FullName is a Super Key.
- **Composite Key:** If a table do not have any single column that qualifies for a Candidate key, then you have to select 2 or more columns to make a row unique. Like if there is no EmployeeID, PAN or emailID columns, then you can make FullName + DateOfBirth as Composite key. But still there can be a narrow chance of duplicate row.



## 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.
- The primary key must be given a value when a new record is inserted.
- **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.**

# PRIMARY KEY constraint

Choosing a primary key is one of the most important steps in good database design. A primary key is a column that serves a special purpose. 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. Each database table needs a primary key.

## 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.
- Stability: The value of the primary key should be stable over time and not change frequently.

# *clustered and non-clustered index*

Indexing in MySQL is a process that helps us to return the requested data from the table very fast. If the table does not have an index, it scans the whole table for the requested data.

MySQL allows two different types of Indexing:

- Clustered Index
- Non-Clustered Index

**Clustered Index:-** The InnoDB table uses a clustered index for optimizing the speed of most common lookups (SELECT statement) and DML operations like INSERT, UPDATE, and DELETE command. Clustered indexes sort and store the data rows in the table based on their key values that can be sorted in only one way. If the table column contains a primary key or unique key, MySQL creates a clustered index named PRIMARY based on that specific column.

**Non-Clustered Index:-** The indexes other than PRIMARY indexes (i.e. clustered indexes) called a non-clustered index. The non-clustered indexes are also known as secondary indexes. The non-clustered index and table data are both stored in different places. It is not sorted (ordering) the table data.

- `CREATE TABLE test(c1 INT, c2 INT, c3 INT, c4 INT,c5 INT, c6 INT, c7 INT, c8 INT, c9 INT, c10 INT, c11 INT, c12 INT, c13 INT, c14 INT, c15 INT, c16 INT, c17 INT, c18 INT, c19 INT, c20 INT, PRIMARY KEY (c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15, c16, c17, c18 )); // error`

# *constraints – add primary key*

*col\_name data\_type PRIMARY KEY*

The following example creates tables with **PRIMARY KEY** column.

- `CREATE TABLE users (  
 ID INT PRIMARY KEY,  
 userName VARCHAR(25),  
 password VARCHAR(25),  
 email VARCHAR(255)  
);`
- `CREATE TABLE supplier (  
 supplier_id INT,  
 supplier_name VARCHAR(50),  
 contact_name VARCHAR(50),  
 constraint pk_supplier_id PRIMARY KEY(supplier_id)  
);`
- `CREATE TABLE purchase_orders (  
 po_number INT,  
 vendor_id INT NOT NULL,  
 po_status INT NOT NULL,  
 PRIMARY KEY(po_number)  
);`
- `CREATE TABLE person (  
 ID INT NOT NULL UNIQUE,  
 lastName VARCHAR(45),  
 firstName VARCHAR(45),  
 age INT,  
 email VARCHAR(255)  
);`

# *constraints – add composite primary key*

The following example creates tables with **COMPOSITE PRIMARY KEY** column.

- `CREATE TABLE salesDetails (  
 customerID INT,  
 productID INT,  
 timeID INT,  
 qty INT,  
 salesDate DATE,  
 salesAmount INT,  
 PRIMARY KEY(customerID , productID, timeID)  
) ;`

customerID	productID	timeID	quantity	salesDate	salesAmount
Cust-001	PRD-1	D1-T1	100	•••••	25,00,000
Cust-001	PRD-1	D2-T1	100	••••••	25,00,000
Cust-001	PRD-2	D1-T1	200	••••••	50,00,000
Cust-002	PRD-1	D1-T1	100	••••••	25,00,000
Cust-004	PRD-1	D1-T1	100	••••••	25,00,000
Cust-004	PRD-2	D3-T1	200	••••••	50,00,000

# *constraints – add composite primary key*

The following example creates tables with **COMPOSITE PRIMARY KEY** column.

- `CREATE TABLE payments ( paymentID INT, orderID INT, amount INT, bankDetails VARCHAR(255), PRIMARY KEY(paymentID , orderID) );`
- `CREATE TABLE order_product ( orderID INT, productID INT, qty INT, rate INT, constraint pk_orderID_productID PRIMARY KEY(orderID, productID) );`

Try It

```
CREATE TABLE try(c1 INT, c2 INT, c3 INT, c4 INT, c5 INT, c6 INT  
,c7 INT, c8 INT, c9 INT, c10 INT,c11 INT,c12 INT, c13 INT, c14 INT,  
c15 INT, c16 INT,c17 INT, c18 INT, c19 INT, c20 INT, c21 INT, c22  
INT, c23 INT, c24 INT, c25 INT, c26 INT, c27 INT, c28 INT, c29 INT ,  
c30 INT, c31 INT, c32 INT, c33 INT, PRIMARY KEY(c1, c2, c3, c4,  
c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15, c16, c17));
```

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

Add Primary Key using Alter

## *constraints – add primary key using alter*

You can use the **ALTER TABLE** statement to **ADD PRIMARY KEY** on existing column.

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

- `CREATE TABLE vendors (`  
 `vendor_id INT,`  
 `vendor_name VARCHAR(25),`  
 `address VARCHAR(255)`  
`);`
- `ALTER TABLE vendors ADD PRIMARY KEY(vendor_id);`
- `ALTER TABLE vendors ADD constraint pk_vendor_id PRIMARY KEY(vendor_id);`

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

# Drop Primary Key

## *constraints – drop primary key*

You can use the **ALTER TABLE** statement to **DROP PRIMARY KEY**.

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

- `CREATE TABLE vendors (  
 vendor_id INT,  
 vendor_name VARCHAR(25),  
 address VARCHAR(255)  
)`

```
ALTER TABLE vendors 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.

# 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.

## *constraints – add unique key*

*col\_name data\_type UNIQUE KEY*

The following example creates table with **UNIQUE KEY** column.

- `CREATE TABLE clients ( client_id INT, first_name VARCHAR(50), last_name VARCHAR(50), company_name VARCHAR(255), email VARCHAR(255) UNIQUE );`
- `CREATE TABLE contacts ( ID INT, first_name VARCHAR(50), last_name VARCHAR(50), phone VARCHAR(15), UNIQUE(phone) );`
- `CREATE TABLE brands ( ID INT, brandName VARCHAR(30), constraint uni_brandName UNIQUE(brandName) );`
- `SHOW INDEX FROM clients;`

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

Add Unique Key using Alter

## *constraints – add unique key using alter*

You can use the **ALTER TABLE** statement to **ADD UNIQUE KEY** on existing column.

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

- `CREATE TABLE shop (`  
 `ID INT,`  
 `shop_name VARCHAR(30)`  
`);`
- `ALTER TABLE shop ADD UNIQUE(shop_name);`
- `ALTER TABLE shop ADD constraint uni_shop_name UNIQUE(shop_name);`

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

# Drop Unique Key

# *constraints – drop unique key*

You can use the **ALTER TABLE** statement to **DROP UNIQUE KEY**.

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

- `SELECT table_name, constraint_name, constraint_type  
FROM information_schema.table_constraints WHERE  
constraint_schema = 'z' AND table_name IN ('A', 'B');`
- `ALTER TABLE users DROP INDEX <COLUMN_NAME>;`
- `ALTER TABLE users DROP INDEX U_USER_ID; #CONSTRAINT NAME`
- `CREATE TABLE users (  
 ID INT PRIMARY KEY,  
 userName VARCHAR(40),  
 password VARCHAR(255),  
 email VARCHAR(255) UNIQUE KEY  
) ;`
- `CREATE TABLE users (  
 ID INT PRIMARY KEY,  
 userName VARCHAR(40),  
 password VARCHAR(255),  
 email VARCHAR(255),  
 constraint uni_email UNIQUE KEY(email)  
) ;`
- `ALTER TABLE users DROP INDEX email;`
- `ALTER TABLE users DROP INDEX uni_email;`

```
[CONSTRAINT symbol] FOREIGN KEY (col_name, ...) REFERENCES tbl_name (col_name, ...)  
[ON DELETE CASCADE | SET NULL]  
[ON UPDATE 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.

# *insert, update, & delete – (primary key/foreign key)*

A referential constraint could be violated in following cases.

- An **INSERT** attempt to add a row to a child table that has a value in its foreign key columns that does not match a value in the corresponding parent table's column.
- An **UPDATE** attempt to change the value in a child table's foreign key columns to a value that has no matching value in the corresponding parent table's parent key.
- An **UPDATE** attempt to change the value in a parent table's parent key to a value that does not have a matching value in a child table's foreign key columns.
- A **DELETE** attempt to remove a record from a parent table that has a matching value in a child table's foreign key columns.

**Note:**

- PARENT and CHILD tables must use the same storage engine,
- and they cannot be defined as temporary tables.
- If we don't give constraint name. System will automatically generated the constraint name and will assign to foreign key constraint. e.g. **login\_ibfk\_1, login\_ibfk\_2, ....**

Remember:

## *anomaly – (primary key/foreign key)*

Student (parent) Table

RollNo	Name	Mobile	City	State	isActive
1	Ramesh	••••	Pune	MH	1
2	Amit	••••	Baroda	GJ	1
3	Rajan	••••	Surat	GJ	1
4	Bhavin	••••	Baroda	GJ	1
5	Pankaj	••••	Surat	GJ	1

student\_course (child) Table

RollNo	CourseDuration	CourseName
1	1.5 month	RDBMS
2	1.2 month	NoSQL
3	2 month	Networking
1	2 month	Java
2	2 month	.NET

### Insertion anomaly:

- If we try to insert a record in Student\_Course (child) table with RollNo = 7, it will not allow.

### Updation and Deletion anomaly:

- If you try to chance the RollNo from Student (parent) table with RollNo = 6 whose RollNo = 1 , it will not allow.
- If you try to chance the RollNo from Student\_Course (child) table with RollNo = 9 whose RollNo = 3 , it will not allow.
- If we try to delete a record from Student (parent) table with RollNo = 1 , it will not allow.

Remember:

## *alter, drop – (primary key/foreign key)*

### Parent Table

```
student = {  
    rollno INT, * (PK)  
    name VARCHAR(10),  
    mobile VARCHAR(10),  
    city VARCHAR(10),  
    state VARCHAR(10),  
    isActive BOOL  
}
```

### Child Table

```
student_course = {  
    rollno INT, * (FK)  
    courceduration VARCHAR(10),  
    courcename VARCHAR(10)  
}
```

DDL command could be violated in following cases.

### Alter command:

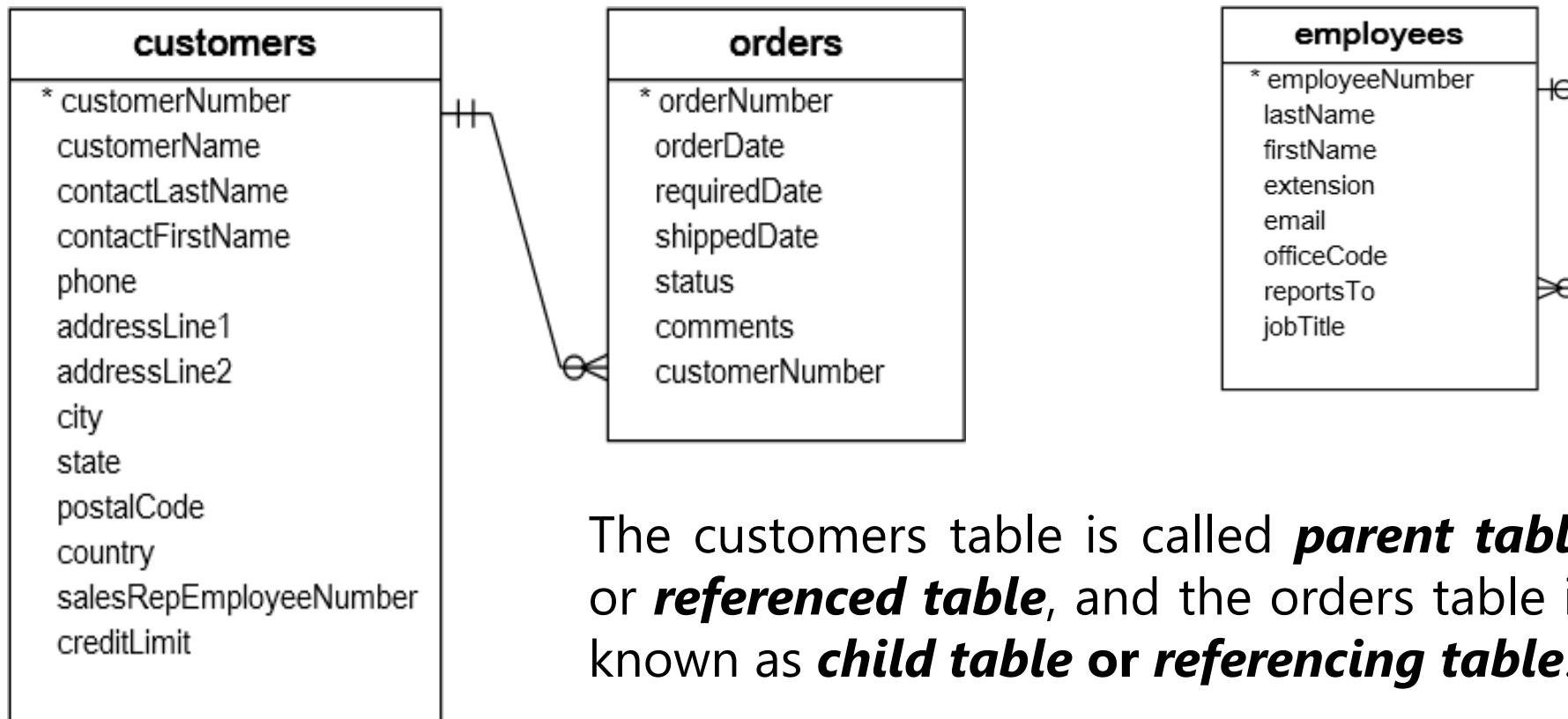
- If we try to modify datatype of RollNo in Student or Student\_Course table with VARCHAR, it will not allow.
- If we try to apply auto\_increment to RollNo in Student table, it will not allow
- If we try to drop RollNo column from Student table , it will not allow.

### Drop command:

- If we try to drop Student (parent) table, it will not allow.

# *constraints – foreign key*

A foreign key is a field in a table that matches another field of another table. A foreign key places constraints on data in the related tables, which enables MySQL to maintain referential integrity.

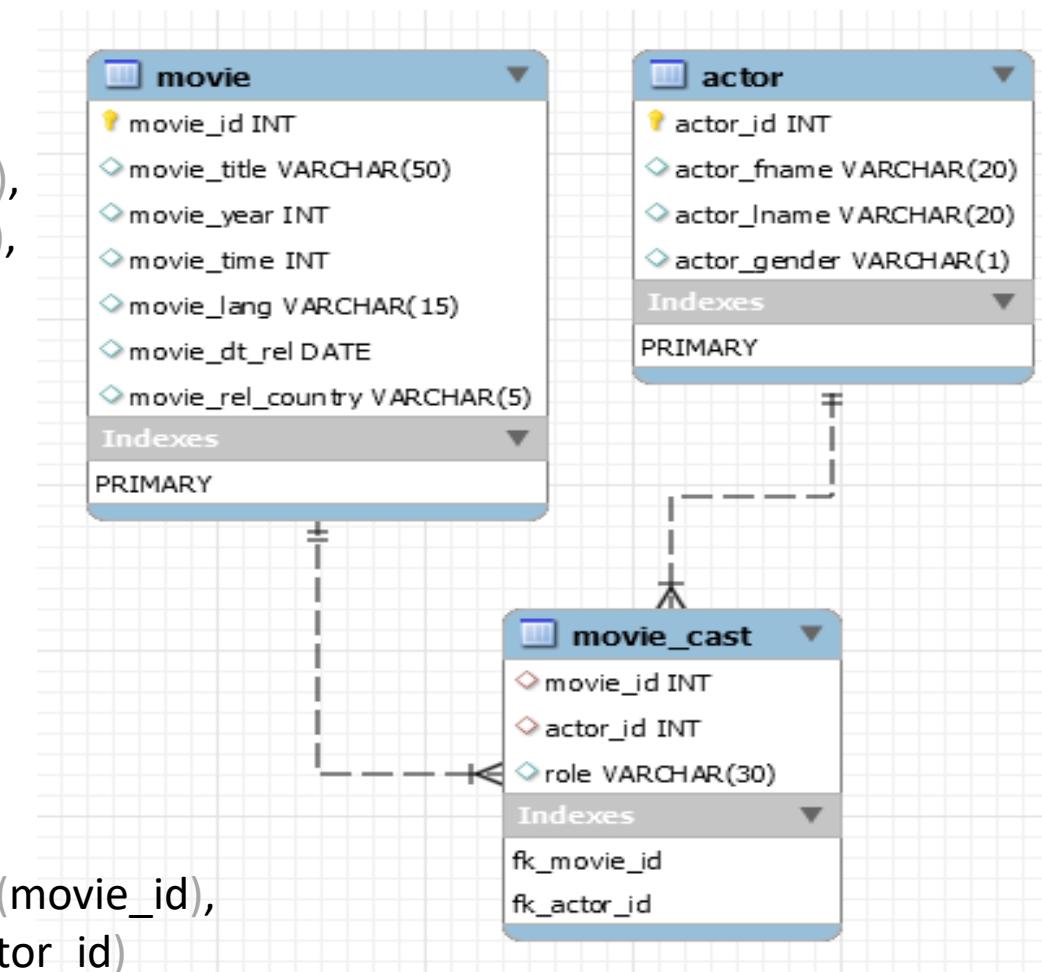


The **customers** table is called ***parent table*** or ***referenced table***, and the **orders** table is known as ***child table* or *referencing table***.

# *constraints – foreign key*

A foreign key is a field in a table that matches another field of another table. A foreign key places constraints on data in the related tables, which enables MySQL to maintain referential integrity.

- `CREATE TABLE movie ( movie_id INT PRIMARY KEY, movie_title VARCHAR(50), movie_year INT, movie_time INT, movie_lang VARCHAR(15), movie_dt_rel DATE, movie_rel_country VARCHAR(5) );`
- `CREATE TABLE actor ( actor_id INT PRIMARY KEY, actor_fname VARCHAR(20), actor_lname VARCHAR(20), actor_gender VARCHAR(1) );`
- `CREATE TABLE movie_cast ( movie_id INT, actor_id INT, role VARCHAR(30), constraint fk_movie_id FOREIGN KEY(movie_id) REFERENCES movie(movie_id), constraint fk_actor_id FOREIGN KEY(actor_id) REFERENCES actor(actor_id) );`



# QUESTION – *find foreign key columns*

The following example find **Foreign Key** columns.

- `CREATE TABLE owner (  
 owner_id INT PRIMARY KEY,  
 first_name VARCHAR(50),  
 last_name VARCHAR(50),  
 email VARCHAR(255)  
);`
- `CREATE TABLE shop (  
 shop_id INT,  
 owner_id INT,  
 shop_name VARCHAR(30)  
);`
- `CREATE TABLE brands (  
 brand_id INT PRIMARY KEY,  
 brand_name VARCHAR(30) UNIQUE  
);`
- `CREATE TABLE contacts (  
 contact_id INT PRIMARY KEY,  
 owner_id INT,  
 contact_number VARCHAR(15)  
);`
- `CREATE TABLE shop_brand (  
 ID INT PRIMARY KEY,  
 shop_id INT,  
 brand_id INT  
);`

```
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 Foreign Key Constraint using  
Alter

# *constraints – add foreign key using alter*

You can use the **ALTER TABLE** statement to **ADD FOREIGN KEY** on existing column.

```
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);
```

```
CREATE TABLE users (  
ID INT PRIMARY KEY,  
userName VARCHAR(40),  
password VARCHAR(255),  
email VARCHAR(255) UNIQUE KEY  
);
```

```
CREATE TABLE login (  
ID INT PRIMARY KEY,  
userID INT,  
loginDate DATE,  
loginTime TIME  
);
```

- **ALTER TABLE login ADD FOREIGN KEY(userID) REFERENCES users(ID);**
- **ALTER TABLE login ADD constraint fk(userID) FOREIGN KEY(userID) REFERENCES users(ID);**

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

Drop Foreign Key Constraint  
using Alter

# *constraints – drop foreign key*

You can use the **ALTER TABLE** statement to **DROP FOREIGN KEY**.

```
CREATE TABLE users (
    ID INT PRIMARY KEY ,
    userName VARCHAR(40),
    password VARCHAR(255),
    email VARCHAR(255)
) ;
```

```
CREATE TABLE login (
    ID INT PRIMARY KEY,
    userID INT,
    loginDate DATE,
    loginTime TIME,
    constraint fk_userID FOREIGN KEY(userID) REFERENCES users(ID)
) ;
```

```
CREATE TABLE login (
    ID INT PRIMARY KEY,
    userID INT,
    loginDate DATE,
    loginTime TIME,
    FOREIGN KEY(userID) REFERENCES users(ID)
) ;
```

- **ALTER TABLE** login **DROP FOREIGN KEY** fk\_userID;
- **ALTER TABLE** login **DROP FOREIGN KEY** login\_ibfk\_1; // **login\_ibfk\_1** is the default constraint name.
- **SELECT** table\_name, constraint\_name, constraint\_type **FROM** information\_schema.table\_constraints **WHERE** table\_schema = 'DB2';

## *constraints – foreign key*

- [ON DELETE *reference\_option*]
- [ON UPDATE *reference\_option*]

Cascaded FOREIGN KEY actions do not activate triggers.

*reference\_option:* RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT

### Remember:

- When an UPDATE or DELETE operation affects a key value in the parent table that has matching rows in the child table, the result depends on the referential action specified using ON UPDATE and ON DELETE sub clauses of the FOREIGN KEY clause.
- ON DELETE or ON UPDATE that is not specified, the default action is always RESTRICT.

# *constraints – foreign key*

Remember:

- CASCADE: DELETE or UPDATE the row from the parent table, and automatically DELETE or UPDATE the matching rows in the child table. Both ON DELETE CASCADE and ON UPDATE CASCADE are supported.
- SET NULL: DELETE or UPDATE the row from the parent table, and set the foreign key column or columns in the child table to NULL. Both ON DELETE SET NULL and ON UPDATE SET NULL clauses are supported.

```
CREATE TABLE users (
    ID INT PRIMARY KEY ,
    userName VARCHAR(40),
    password VARCHAR(255),
    email VARCHAR(255)
);
```

```
CREATE TABLE login (
    ID INT PRIMARY KEY,
    userID INT,
    loginDate DATE,
    loginTime TIME,
    FOREIGN KEY(userID) REFERENCES users(ID) ON DELETE CASCADE ON UPDATE CASCADE
);
```

```
CREATE TABLE login (
    ID INT PRIMARY KEY,
    userID INT,
    loginDate DATE,
    loginTime TIME,
    FOREIGN KEY(userID) REFERENCES users(ID) ON DELETE SET NULL ON UPDATE SET NULL
);
```

## *on delete / on update – foreign key*

**SET NULL**: Delete or update the row from the parent table and set the foreign key column or columns in the child table to NULL. If you specify a SET NULL action, make sure that you have not declared the columns in the child table as NOT NULL.

**RESTRICT**: Rejects the delete or update operation for the parent table. Specifying RESTRICT (or NO ACTION) is the same as omitting the ON DELETE or ON UPDATE clause.

**NO ACTION**: Is equivalent to RESTRICT. The MySQL Server rejects the delete or update operation for the parent table if there is a related foreign key value in the referenced table.

**SET DEFAULT**: This action is recognized by the MySQL parser, but both InnoDB and NDB reject table definitions containing ON DELETE SET DEFAULT or ON UPDATE SET DEFAULT clauses.

1. CREATE TABLE test (c1 INT, c2 INT, c3 INT, check (c3 = SUM(c1)));



// ERROR

SUM(SAL) MIN(SAL) COUNT(\*)  
AVG(SAL) MAX(SAL) COUNT(JOB)

## Check Constraint

## *constraints – check*

### 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

## *constraints – check*

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

The following example creates **USERS** table with **CHECK** column.

```
CREATE TABLE users (
    ID INT PRIMARY KEY,
    userName VARCHAR(40),
    password VARCHAR(255),
    email VARCHAR(255),
    ratings INT CHECK(ratings > 50)
);
```

```
CREATE TABLE users (
    ID INT PRIMARY KEY,
    userName VARCHAR(40),
    password VARCHAR(255),
    email VARCHAR(255),
    ratings INT,
    constraint chk_ratings CHECK(ratings > 50)
);
```

```
CREATE TABLE users (
    ID INT PRIMARY KEY,
    userName VARCHAR(40),
    password VARCHAR(255),
    email VARCHAR(255),
    ratings INT,
    CHECK(ratings > 50)
);
```

```
CREATE TABLE users (
    ID INT PRIMARY KEY,
    userName VARCHAR(40),
    password VARCHAR(255),
    email VARCHAR(255),
    ratings INT,
    constraint chk_ratings CHECK(ratings > 50),
    constraint chk_email CHECK(LENGTH(email) > 12)
);
```

## *constraints – check*

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

The following example creates **USERS** table with **CHECK** column.

```
CREATE TABLE users (
    ID INT PRIMARY KEY,
    startDate DATE,
    endDate DATE,
    constraint chk_endDate CHECK(endDate > startDate + INTERVAL 7 day)
);
```

- `SELECT * FROM check_constraints WHERE CONSTRAINT_SCHEMA = 'z';`

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

Add Check Constraint using Alter

## *constraints – add check using alter*

You can use the **ALTER TABLE** statement to **ADD CHECK KEY** on existing column.

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

```
CREATE TABLE users (  
    ID INT PRIMARY KEY,  
    userName VARCHAR(40),  
    password VARCHAR(255),  
    email VARCHAR(255),  
    ratings INT  
) ;
```

- **ALTER TABLE** users **ADD CHECK(ratings > 50);**
- **ALTER TABLE** users **ADD constraint chk\_ratings CHECK(ratings > 50);**

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

drop check constraint

# *constraints – drop check key*

You can use the **ALTER TABLE** statement to **DROP CHECK KEY**.

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

```
CREATE TABLE users (  
    ID INT PRIMARY KEY,  
    userName VARCHAR(40),  
    password VARCHAR(255),  
    email VARCHAR(255),  
    ratings INT,  
    constraint chk_ratings CHECK(ratings > 50)  
);
```

- **ALTER TABLE** users **DROP CHECK** chk\_ratings;
  - **ALTER TABLE** users **DROP constraint** chk\_ratings;
  - **ALTER TABLE** users **DROP CHECK** users\_chk\_1;
  - **SELECT** table\_name, constraint\_name, constraint\_type **FROM** information\_schema.table\_constraints **WHERE** table\_schema = 'DB2' **AND** (table\_name LIKE 'U%' **OR** table\_name LIKE 'L%');
- ```
CREATE TABLE users (  
    ID INT PRIMARY KEY,  
    userName VARCHAR(40),  
    password VARCHAR(255),  
    email VARCHAR(255),  
    ratings INT,  
    CHECK(ratings > 50)  
);
```

The check constraint defined on a table must refer to only columns in that table. It can not refer to columns in other tables.

- `CREATE TABLE test (CHECK(c3 > (c1 + c2)), c1 INT, c2 INT, c3 INT);`
- `CREATE TABLE test (c1 INT, c2 INT, c3 INT, CHECK(c3 > (c1 + c2)));`
- `CREATE TABLE test (CHECK(c3 > (c1 + c2)), PRIMARY KEY(c1), c1 INT, c2 INT, c3 INT);`
- `CREATE TABLE test (a INT CHECK (a >= 0),b INT CHECK (b >= 0), CHECK ( a + b <= 10));`
  
- `ALTER TABLE test ADD constraint chk_id CHECK(ID > 10);`
- `ALTER TABLE test DROP CHECK chk_id;`

## *check with (in, like, and between)*

The **CHECK** constraint using **IN**, **LIKE**, and **BETWEEN**.

```
CREATE TABLE users (
    ID INT PRIMARY KEY,
    userName VARCHAR(40),
    password VARCHAR(255) CHECK(LENGTH(password) > 5),
    email VARCHAR(255),
    country VARCHAR(255) CHECK(country LIKE ('I%') OR country LIKE ('U%')),
    ratings INT CHECK(ratings BETWEEN 1 and 5 OR ratings BETWEEN 12 and 25),
    isActive BOOL CHECK(isActive IN (1, 0)),
    startDate DATE,
    endDate DATE,
    constraint chk_endDate CHECK(endDate > startDate + INTERVAL 7 day)
);
```

# 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**.

# *alter table*

## **syntax**

**ALTER TABLE** *tbl\_name*

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

- | **ADD [COLUMN]** *col\_name* *column\_definition* [**FIRST** | **AFTER** *col\_name* ]
- | **ADD [COLUMN]** (*col\_name* *column\_definition*, ...)
- | **ADD {INDEX|KEY}** [*index\_name*] (*index\_col\_name*, ...)
- | **ADD [CONSTRAINT** [ *symbol* ]] **PRIMARY KEY**
- | **ADD [CONSTRAINT** [*symbol*]] **UNIQUE KEY**
- | **ADD [CONSTRAINT** [*symbol*]] **FOREIGN KEY** *reference\_definition*
- | **CHANGE [COLUMN]** *old\_col\_name* *new\_col\_name* *column\_definition* [**FIRST** | **AFTER** *col\_name* ]
- | **MODIFY [COLUMN]** *col\_name* *column\_definition* [**FIRST** | **AFTER** *col\_name* ]
- | **DROP [COLUMN]** *col\_name*
- | **DROP PRIMARY KEY**
- | **DROP {INDEX|KEY}** *index\_name*
- | **DROP FOREIGN KEY** *fk\_symbol*
- | **RENAME [TO|AS]** *new\_tbl\_name*
- | **RENAME COLUMN** *old\_col\_name* **TO** *new\_col\_name*
- | **ALTER [COLUMN]** *col\_name* { **SET DEFAULT** {*literal* | (*expr*)} | **SET {VISIBLE | INVISIBLE}** } | **DROP DEFAULT** }

# *alter table*

## Remember:

- **Change Columns** :- You can rename a column using a CHANGE old\_col\_name new\_col\_name column\_definition clause. To do so, specify the old and new column names and the definition that the column currently has.
- **Modify Columns** :- You can also use MODIFY to change a column's type without renaming it.
- **Dropping Columns** :- If a table contains only one column, the column cannot be dropped. If columns are dropped from a table, the columns are also removed from any index of which they are a part. If all columns that make up an index are dropped, the index is dropped as well.

## Note:

- To convert a table from one storage engine to another, use an ALTER TABLE statement that indicates the new engine:

```
ALTER TABLE tbl_name ENGINE = InnoDB;
```

```
ALTER TABLE tbl_name ADD col1 INT, ADD col2 INT;
```

```
ALTER TABLE tbl_name DROP COLUMN col1, DROP COLUMN col2 , ADD col3 INT;
```

# add column

`ALTER TABLE tbl_name [alter_specification [, alter_specification] ...]`

`alter_specification`

- `ADD [COLUMN] col_name column_definition [FIRST | AFTER col_name ]`
- `ADD [COLUMN] (col_name column_definition, ...)`

# *add column*

- `CREATE TABLE vehicles (`  
    `vehicleID INT PRIMARY KEY ,`  
    `year INT,`  
    `make VARCHAR(100)`  
`);`

|   | Field     | Type         | Null | Key | Default | Extra |
|---|-----------|--------------|------|-----|---------|-------|
| ▶ | vehicleID | int          | NO   | PRI | NULL    |       |
|   | year      | int          | YES  |     | NULL    |       |
|   | make      | varchar(100) | YES  |     | NULL    |       |

- `INSERT INTO vehicles VALUES (111, 2000, 'Honda');`
  - `INSERT INTO vehicles VALUES (112, 2002, 'Hyundai');`
  - `INSERT INTO vehicles VALUES (113, 2000, 'Jeep');`
  - `INSERT INTO vehicles VALUES (114, 2005, 'Toyota');`

- ALTER TABLE vehicles  
    ADD ID INT UNIQUE auto\_increment first,  
    ADD model VARCHAR(100) not null,  
    ADD color VARCHAR(50),  
    ADD note VARCHAR(255);

|   | Field     | Type         | Null | Key | Default | Extra          |
|---|-----------|--------------|------|-----|---------|----------------|
| ▶ | ID        | int          | NO   | UNI | NULL    | auto_increment |
|   | vehicleID | int          | NO   | PRI | NULL    |                |
|   | year      | int          | YES  |     | NULL    |                |
|   | make      | varchar(100) | YES  |     | NULL    |                |
|   | model     | varchar(100) | NO   |     | NULL    |                |
|   | color     | varchar(50)  | YES  |     | NULL    |                |
|   | note      | varchar(255) | YES  |     | NULL    |                |

# modify column

`ALTER TABLE tbl_name [alter_specification [, alter_specification] ...]`

`alter_specification`

- `MODIFY [COLUMN] col_name column_definition [FIRST | AFTER col_name]`

- `CREATE TABLE` vehicles (
   
vehicleID INT PRIMARY KEY ,
   
year INT,
   
make VARCHAR(100),
   
model VARCHAR(100) not null,
   
color VARCHAR(50),
   
note VARCHAR(255)
 );

|   | Field    | Type         | Null | Key | Default | Extra |
|---|----------|--------------|------|-----|---------|-------|
| ▶ | vehideID | int          | NO   | PRI | NULL    |       |
|   | year     | int          | YES  |     | NULL    |       |
|   | make     | varchar(100) | YES  |     | NULL    |       |
|   | model    | varchar(100) | NO   |     | NULL    |       |
|   | color    | varchar(50)  | YES  |     | NULL    |       |
|   | note     | varchar(255) | YES  |     | NULL    |       |

## modify column

- `ALTER TABLE` vehicles
   
`MODIFY year SMALLINT not null,`
  
`MODIFY make VARCHAR(150) not null,`
  
`MODIFY color VARCHAR(20) not null;`

|   | Field    | Type         | Null | Key | Default | Extra |
|---|----------|--------------|------|-----|---------|-------|
| ▶ | vehideID | int          | NO   | PRI | NULL    |       |
|   | year     | smallint     | NO   |     | NULL    |       |
|   | make     | varchar(150) | NO   |     | NULL    |       |
|   | model    | varchar(100) | NO   |     | NULL    |       |
|   | color    | varchar(20)  | NO   |     | NULL    |       |
|   | note     | varchar(255) | YES  |     | NULL    |       |

- `INSERT INTO` vehicles `VALUES (111, 2000, 'Honda', 'A1', 'silver', ' Honda was the first Japanese automobile manufacturer to release a dedicated luxury brand, Acura, in 1986.');`
- `INSERT INTO` vehicles `VALUES (112, 2002, 'Hyundai', 'AC1', 'white', ' Hyundai operates the world's largest integrated automobile manufacturing facility in Ulsan, South Korea which has an annual production capacity of 1.6 million units.');`
- `INSERT INTO` vehicles `VALUES (113, 2000, 'Jeep', 'D2', 'black', ' Fiat Chrysler Automobiles has owned Jeep since 2014. Previous owners include the Kaiser Jeep Corporation and American Motors Corporation. Most Jeeps are American-made, except for a select few models. The Toledo Assembly Complex in Ohio manufactures the Jeep Wrangler.');`

# rename column

`ALTER TABLE tbl_name [alter_specification [, alter_specification] ...]`

`alter_specification`

- `RENAME COLUMN old_col_name TO new_col_name`

# rename column

- `CREATE TABLE vehicles ( vehicleID INT, year SMALLINT, make VARCHAR(150), model VARCHAR(100), color VARCHAR(20), note VARCHAR(255) );`
- `ALTER TABLE vehicles RENAME COLUMN year TO model_year`

|   | Field     | Type         | Null | Key | Default | Extra |
|---|-----------|--------------|------|-----|---------|-------|
| ▶ | vehicleID | int          | YES  |     | NULL    |       |
|   | year      | smallint     | YES  |     | NULL    |       |
|   | make      | varchar(150) | YES  |     | NULL    |       |
|   | model     | varchar(100) | YES  |     | NULL    |       |
|   | color     | varchar(20)  | YES  |     | NULL    |       |
|   | note      | varchar(255) | YES  |     | NULL    |       |

|   | Field            | Type         | Null | Key | Default | Extra |
|---|------------------|--------------|------|-----|---------|-------|
| ▶ | vehicleID        | int          | YES  |     | NULL    |       |
|   | model_year       | int          | NO   |     | NULL    |       |
|   | make             | varchar(150) | YES  |     | NULL    |       |
|   | model            | varchar(100) | YES  |     | NULL    |       |
|   | model_color      | varchar(20)  | YES  |     | NULL    |       |
|   | vehicleCondition | varchar(150) | YES  |     | NULL    |       |

# change column

`ALTER TABLE tbl_name [alter_specification [, alter_specification] ...]`

`alter_specification`

- `CHANGE [COLUMN] old_col_name new_col_name column_definition [ FIRST | AFTER col_name ]`

# change column

- `CREATE TABLE vehicles ( vehicleID INT,  
year SMALLINT,  
make VARCHAR(150),  
model VARCHAR(100),  
color VARCHAR(20),  
note VARCHAR(255)  
);`
- `ALTER TABLE vehicles  
CHANGE year model_year INT,  
CHANGE color model_color VARCHAR(20),  
CHANGE note vehicleCondition VARCHAR(150);`

|   | Field     | Type         | Null | Key | Default | Extra |
|---|-----------|--------------|------|-----|---------|-------|
| ▶ | vehicleID | int          | YES  |     | NULL    |       |
|   | year      | smallint     | YES  |     | NULL    |       |
|   | make      | varchar(150) | YES  |     | NULL    |       |
|   | model     | varchar(100) | YES  |     | NULL    |       |
|   | color     | varchar(20)  | YES  |     | NULL    |       |
|   | note      | varchar(255) | YES  |     | NULL    |       |

|   | Field            | Type         | Null | Key | Default | Extra |
|---|------------------|--------------|------|-----|---------|-------|
| ▶ | vehicleID        | int          | YES  |     | NULL    |       |
|   | model_year       | int          | NO   |     | NULL    |       |
|   | make             | varchar(150) | YES  |     | NULL    |       |
|   | model            | varchar(100) | YES  |     | NULL    |       |
|   | model_color      | varchar(20)  | YES  |     | NULL    |       |
|   | vehicleCondition | varchar(150) | YES  |     | NULL    |       |

# change column

- `CREATE TABLE users (`  
    `ID INT PRIMARY KEY,`  
    `userName VARCHAR(40),`  
    `password VARCHAR(25),`  
    `email VARCHAR(255)`  
`) ;`
- `CREATE TABLE login (`  
    `ID INT PRIMARY KEY,`  
    `userID INT,`  
    `loginDate DATE,`  
    `loginTime TIME,`  
    `constraint fk(userID) FOREIGN KEY(userID) REFERENCES users(ID)`  
`) ;`
- `INSERT INTO users VALUES (1, 'rajan', 'ranaj123', 'rajan447@gmail.com');`
- `INSERT INTO users VALUES (2, 'raj', 'raj', 'raj@gmail.com');`
- `INSERT INTO login VALUES (1, 1, curdate(), curtime());`
- `INSERT INTO login VALUES (2, 1, curdate(), curtime());`
- `INSERT INTO login VALUES (3, 2, curdate(), curtime());`
- `INSERT INTO login VALUES (4, NULL, curdate(), curtime());`
- `ALTER TABLE users CHANGE ID userID INT;`
- `ALTER TABLE login CHANGE userID UID INT;`
- `INSERT INTO login VALUES (5, NULL, curdate(), curtime());`

# drop column

`ALTER TABLE tbl_name [alter_specification [, alter_specification] ...]`

`alter_specification`

- `DROP [COLUMN] col_name`

# *drop column*

- `CREATE TABLE vehicles ( vehicleID INT,  
model_year SMALLINT,  
make VARCHAR(150),  
model VARCHAR(100),  
model_color VARCHAR(20),  
vehicleCondition VARCHAR(150) );`
- `ALTER TABLE vehicles  
CHANGE model_year year INT not null,  
DROP model,  
DROP model_color,  
DROP vehicleCondition;`

|   | Field            | Type         | Null | Key | Default | Extra |
|---|------------------|--------------|------|-----|---------|-------|
| ▶ | vehicleID        | int          | YES  |     | NULL    |       |
|   | model_year       | smallint     | YES  |     | NULL    |       |
|   | make             | varchar(150) | YES  |     | NULL    |       |
|   | model            | varchar(100) | YES  |     | NULL    |       |
|   | model_color      | varchar(20)  | YES  |     | NULL    |       |
|   | vehicleCondition | varchar(150) | YES  |     | NULL    |       |

|   | Field     | Type         | Null | Key | Default | Extra |
|---|-----------|--------------|------|-----|---------|-------|
| ▶ | vehicleID | int          | YES  |     | NULL    |       |
|   | year      | int          | NO   |     | NULL    |       |
|   | make      | varchar(150) | YES  |     | NULL    |       |

# *alter table*

## Sample table

```
CREATE TABLE vehicles (
    vehicleID INT PRIMARY KEY ,
    year INT,
    make VARCHAR(100)
);
```

## Add new columns to a table

```
ALTER TABLE vehicles
ADD model VARCHAR(100) NOT NULL,
ADD color VARCHAR(50),
ADD note VARCHAR(255);
```

## Modify columns

```
ALTER TABLE vehicles
MODIFY year SMALLINT NOT NULL,
MODIFY color VARCHAR(20) NOT NULL,
MODIFY make VARCHAR(150) NOT NULL;
```

## Rename columns

```
ALTER TABLE vehicles
CHANGE year model_year SMALLINT NOT NULL,
CHANGE color model_color VARCHAR(20),
CHANGE note vehicleCondition VARCHAR(150);
```

## DROP columns

```
ALTER TABLE vehicles
CHANGE model_year year INT NOT NULL,
DROP model,
DROP model_color,
DROP vehicleCondition;
```

# 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 drops any VIEW for the table.
- If you try to drop a PARENT/MASTER TABLE, it will not get dropped.

# *drop table*

DROP [TEMPORARY] TABLE [IF EXISTS] tbl\_name [,tbl\_name] ...

## Note:

- All table data and the table definition are removed/dropped.
- If it is desired to delete only the records but to leave the table definition for future use, then the ***DELETE*** command should be used instead of ***DROP TABLE***.
  
- **DROP** login;
- **DROP TABLE** users;
- **DROP TABLE** login, users;

create table using different engines

```
show engines;  
set default_storage_engine = memory;
```

## *create table with memory engine*

- **MEMORY** storage engine tables are visible to another client/user.
- Structure is stored and rows will be removed, after re-starting mysql server (MySQL80) from Services.
- Provides in-memory tables, formerly known as HEAP.
- It stores all data in RAM for faster access than storing data on disks.
- Operations involving non-critical data such as session management or caching.

e.g. `CREATE TABLE temp(c1 INT, c2 INT) ENGINE = MEMORY;`

- `INSERT INTO temp VALUES(10, 10);`
- `SELECT * FROM temp;`

re-start mysql server.

- `SELECT * FROM temp;`

```
show engines;  
set default_storage_engine = csv;
```

## *create table with csv engine*

- **CSV** storage engine tables are visible to another client.
- The CSV storage engine stores data in text/csv files using comma-separated values format.
- The storage engine for the table doesn't support nullable (NULL) columns.
- Doesn't support AUTO\_INCREMENT columns.
- Doesn't support PRIMARY KEY and UNIQUE KEY constraints.
- CHECK constraint with NOT NULL is allowed.

e.g. CREATE TABLE csv(

```
    ID INT not null,  
    ename VARCHAR(10) not null,  
    job VARCHAR(10) not null,  
    sal INT not null) ENGINE = CSV;
```

- INSERT INTO csv VALUES(1, 'saleel', 'manager', 3400);
- SELECT \* FROM csv;

```
show engines;  
set default_storage_engine = blackhole;
```

## *create table with blackhole engine*

- **BLACKHOLE** tables are visible to another client.
- storage engine acts as a “black hole” that accepts data but throws it away and does not store it.
- Triggers can be written on this type of tables

e.g. `CREATE TABLE temp(c1 INT PRIMARY KEY AUTO_INCREMENT, c2 INT UNIQUE, c3 INT NOT NULL, c4 INT CHECK(c4 >= 100)) ENGINE = BLACKHOLE;`

- `INSERT INTO temp(c2, c3, c4) VALUES(100, 200, 300);`
- `SELECT * FROM temp;`
- `DROP TRIGGER IF EXISTS triggername;`  
`delimiter $$`  
`CREATE TRIGGER triggername BEFORE INSERT ON temp FOR EACH ROW`  
`begin`  
 `INSERT INTO temp1 VALUES (NEW.c1, NEW.c2);`  
`end $$`  
`delimiter ;`

## *create temporary table*

- **TEMPORARY** tables are not visible to another client.
- Structure and rows is removed, after exit.

e.g. `CREATE TEMPORARY TABLE temp(c1 INT, c2 INT);`

- `INSERT INTO temp VALUES(10, 10);`
- `SELECT * FROM temp;`
- `EXIT`

# table partitioning

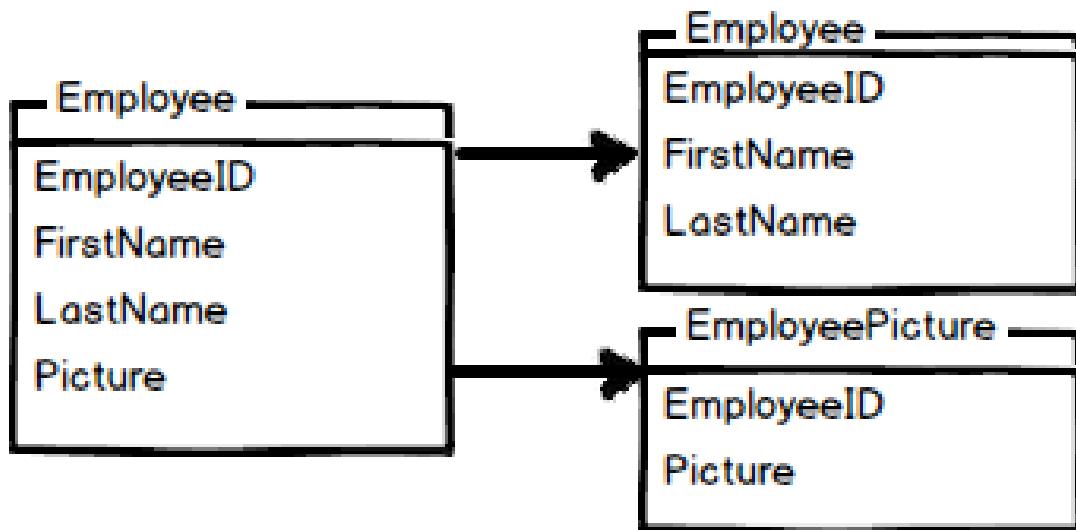
Partitioning separates data into logical units.

# table partitioning

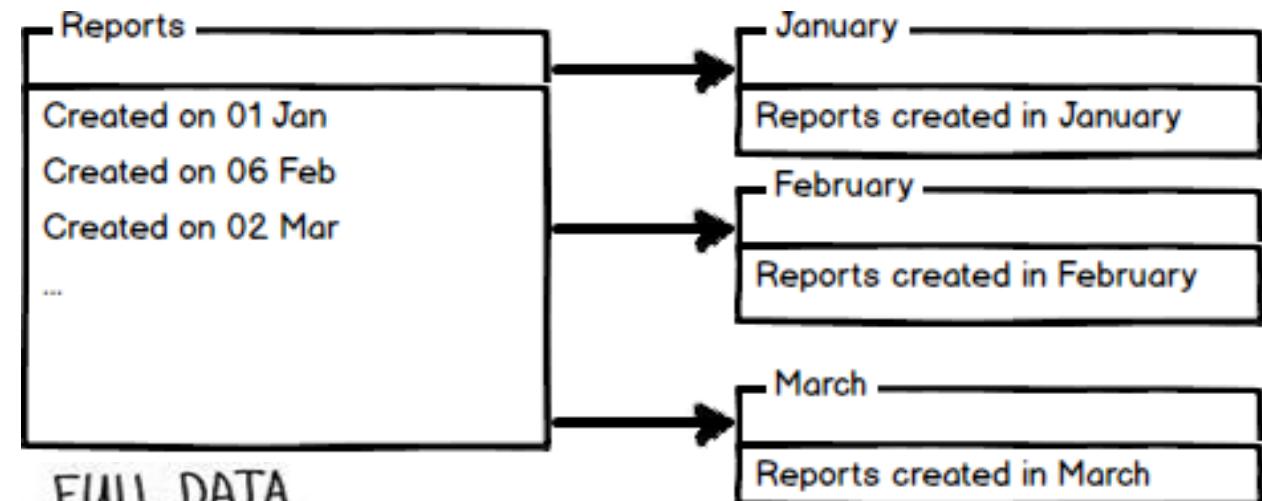
## What is a database table partitioning?

Partitioning is the database process where very large tables are divided into multiple smaller parts. By splitting a large table into smaller, individual tables. The main goal of partitioning is to aid in maintenance of large tables and to reduce the overall response time to read and load data for particular SQL operations.

### Vertical Partitioning

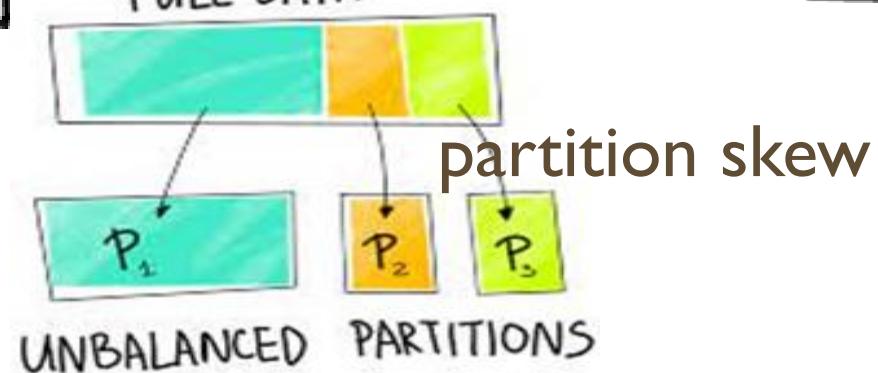


### Horizontal Partitioning



MySQL has mainly six types of partitioning, which are given below:

- RANGE Partitioning
- LIST Partitioning
- COLUMNS Partitioning
- HASH Partitioning
- KEY Partitioning
- Subpartitioning



# *table partitioning*

## Original Table

| CUSTOMER ID | FIRST NAME | LAST NAME | FAVORITE COLOR |
|-------------|------------|-----------|----------------|
| 1           | TAEKO      | OHNUKI    | BLUE           |
| 2           | O.V.       | WRIGHT    | GREEN          |
| 3           | SELDAA     | BAĞCAN    | PURPLE         |
| 4           | JIM        | PEPPER    | AUBERGINE      |

## Vertical Partitions

VP1

| CUSTOMER ID | FIRST NAME | LAST NAME |
|-------------|------------|-----------|
| 1           | TAEKO      | OHNUKI    |
| 2           | O.V.       | WRIGHT    |
| 3           | SELDAA     | BAĞCAN    |
| 4           | JIM        | PEPPER    |

VP2

| CUSTOMER ID | FAVORITE COLOR |
|-------------|----------------|
| 1           | BLUE           |
| 2           | GREEN          |
| 3           | PURPLE         |
| 4           | AUBERGINE      |

## Horizontal Partitions

HP1

| CUSTOMER ID | FIRST NAME | LAST NAME | FAVORITE COLOR |
|-------------|------------|-----------|----------------|
| 1           | TAEKO      | OHNUKI    | BLUE           |
| 2           | O.V.       | WRIGHT    | GREEN          |

HP2

| CUSTOMER ID | FIRST NAME | LAST NAME | FAVORITE COLOR |
|-------------|------------|-----------|----------------|
| 3           | SELDAA     | BAĞCAN    | PURPLE         |
| 4           | JIM        | PEPPER    | AUBERGINE      |

# *partitioning by range / list*

## RANGE Partitioning

PARTITION BY RANGE (COLUMNS)

```
(  
    PARTITION part_name1 VALUES LESS THAN (int_value),  
    PARTITION part_name2 VALUES LESS THAN (int_value),  
    PARTITION part_name3 VALUES LESS THAN MAXVALUE  
)
```

## LIST Partitioning

PARTITION BY LIST (COLUMNS)

```
(  
    PARTITION part_name1 VALUES IN (int_value_list),  
    PARTITION part_name2 VALUES IN (int_value_list),  
    PARTITION part_name3 VALUES IN (int_value_list)  
)
```

## RANGE Partitioning

```
e.g. CREATE TABLE employee(  
    empno INT,  
    ename VARCHAR(10),  
    salary INT  
)  
PARTITION BY RANGE (salary) (  
    PARTITION p0 VALUES LESS THAN (2000),  
    PARTITION p1 VALUES LESS THAN (4000),  
    PARTITION p2 VALUES LESS THAN (6000),  
    PARTITION p3 VALUES LESS THAN MAXVALUE  
);
```

- `INSERT INTO employee PARTITION(p0) VALUES(1, 'saleel', 1500);`
- `SELECT * FROM employee PARTITION(p0);`
- `UPDATE employee PARTITION(p0) set salary = 1500;`
- `UPDATE employee PARTITION(p0) set salary = 3000 WHERE empno = 1; // Invalid statement`
- `DELETE FROM employee PARTITION(p0);`

| Warehouse      | storeID        |
|----------------|----------------|
| AC Warehouse   | 1, 3, 5, 7     |
| National       | 2, 4, 6, 8     |
| Global         | 10, 12, 14, 16 |
| Migrant System | 11, 13, 15, 17 |

## LIST Partitioning

```
e.g. CREATE TABLE item(  
    itemID INT,  
    itemDesc VARCHAR(10),  
    storeID INT  
)  
PARTITION BY LIST(storeID) (  
    PARTITION p0 VALUES IN(1, 3, 5, 7),  
    PARTITION p1 VALUES IN(2, 4, 6, 8),  
    PARTITION p2 VALUES IN(10, 12, 14, 16),  
    PARTITION p3 VALUES IN(11, 13, 15, 17)  
);
```

# *alter / drop partitioning by range / list*

## Alter Partitioning

### RANGE Partitioning

```
ALTER TABLE a ADD PARTITION (PARTITION p3 VALUES LESS THAN(130));
```

- MAXVALUE can only be used in last partition definition

### LIST Partitioning

```
ALTER TABLE a ADD PARTITION (PARTITION p3 VALUES IN (10, 11));
```

### DROP Partitioning

```
ALTER TABLE a ADD PARTITION p3;
```

# 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*

# *temporary table*

e.g.

```
CREATE TEMPORARY TABLE student (
    ID INT PRIMARY KEY,
    namefirst VARCHAR(45),
    namelast VARCHAR(45),
    DOB DATE,
    emailID VARCHAR(128)
);
```

```
CREATE TEMPORARY TABLE temp (
    ID INT PRIMARY KEY,
    firstName VARCHAR(45),
    phone INT,
    city VARCHAR(10) DEFAULT 'PUNE',
    salary INT,
    comm INT,
    total INT GENERATED ALWAYS AS(salary + comm) VIRTUAL
);
```

---

## *create temporary table ... like*

Use CREATE TABLE ... LIKE to create an empty table based on the definition of another table.

```
CREATE TEMPORARY TABLE [IF NOT EXISTS] new_tbl LIKE orig_tbl;
```

- `CREATE TEMPORARY TABLE tempEmployee LIKE employee;`

### Remember:

- LIKE works only for base tables, not for VIEWS.
- 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.
- `CREATE TEMPORARY TABLE new_tbl SELECT * FROM orig_tbl LIMIT 0;`

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.

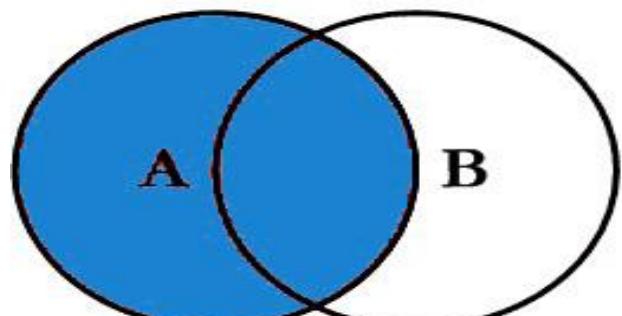
*R*

| 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     |

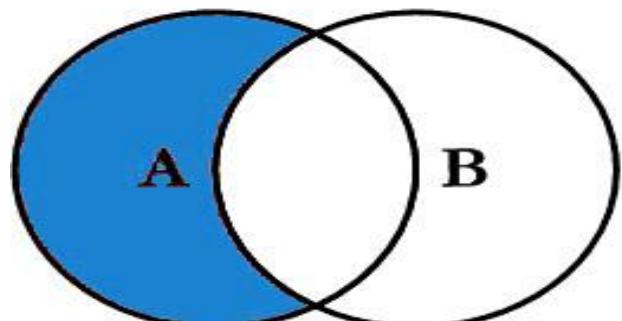
*S*

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

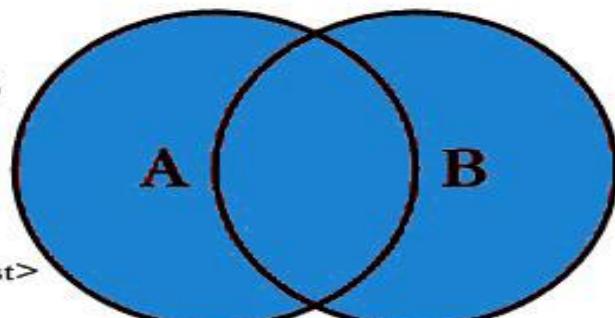
# SQL JOINS



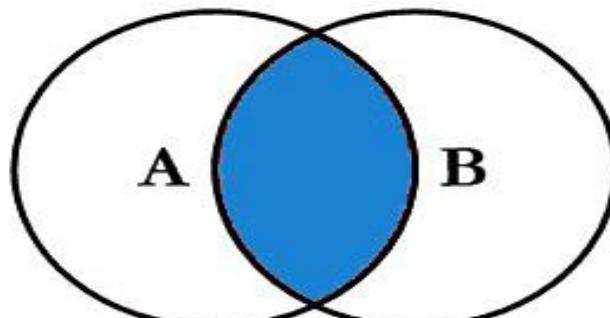
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



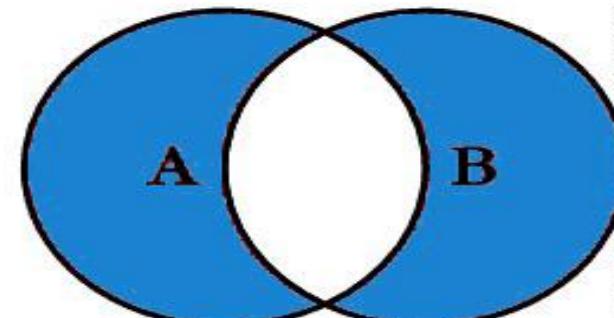
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



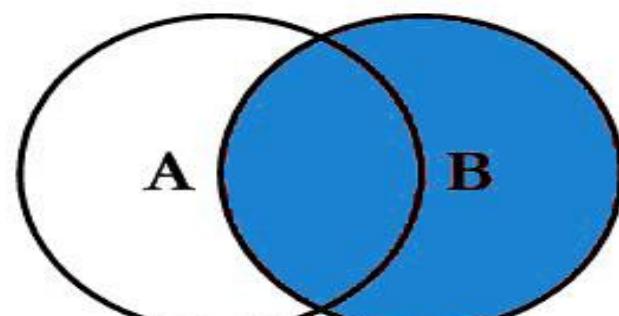
```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```



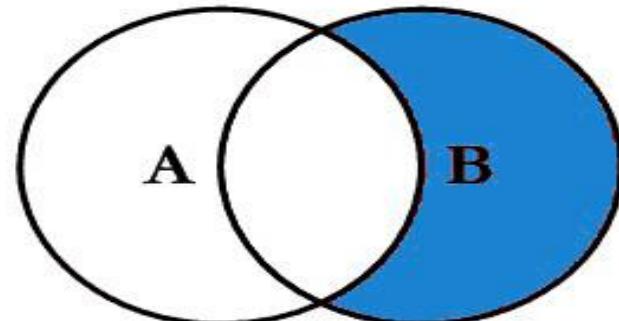
```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



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



```
SELECT <select_list>
FROM TableA A
RIGHT 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
```

# *select statement*

## 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>

column-list  
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)

- `SELECT 'HELLO' ' WORLD';`
- `SELECT 'HELLO' AS 'WORLD';`
- `SELECT ename `EmployeeName` FROM emp;`
- `SELECT ename AS `EmployeeName` FROM emp;`

## 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 A1 [ AS ] alias_name, A2 [ AS ] alias_name, . . . , AN 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**.
- 
- `SELECT empno AS EmployeeID, ename EmployeeName FROM emp;`
  - `SELECT ID AS 'Employee ID', ename "Employee Name" FROM emp;`
  - `SELECT * FROM emp employee;`

# comparison functions and operator

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

## *assignment\_operator*

= (assignment), :=

- The value on the right hand side may be a literal value, another variable storing a value, or any legal expression that yields a scalar value, including the result of a query (provided that this value is a scalar value). You can perform multiple assignments in the same SET statement. You can perform multiple assignments in the same statement.
- Unlike =, the := operator is never interpreted as a comparison operator. This means you can use := in any valid SQL statement (not just in SET statements) to assign a value to a variable.

# *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.

e.g.

1. operand1 \* operand2
2. operand1 = operand2
3. operand IS [NOT] NULL
4. operand [NOT] LIKE 'pattern'
5. expr AND expr
6. Operand := 1001

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

**Note:**

- AND has higher precedence than OR.

- WHERE `col * 4 < 16`
- WHERE `col < 16 / 4`
- `SELECT CONCAT(1, "saleel");`

## column - expressions

"Strings are automatically converted to numbers and numbers to strings as necessary." This means that in order to compare a string to a number, it tries to parse a number from the start of the string. In this case there is no number there, so it converts to 0, and  $0 = 0$  is true.

# *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 ;

# coalesce()

Return the first non-NULL argument

# *coalesce()*

## TODO

- `SELECT primaryphone, bphone, cphone, hphone, IFNULL(COALESCE(bphone, cphone, hphone), 'Contact customer care') 'Active Phone' FROM coalesce;`

Note:

*between ... and ...*

If any expression is NULL, the BETWEEN operator returns a NULL value.

Check whether a value is within a range of values

### Comparison BETWEEN ... AND ...

All three expressions: expr, begin\_expr, and end\_expr must have the same data type.

```
SELECT 5 BETWEEN 0 AND 10 // Returns 1
```

```
SELECT 5 BETWEEN 6 AND 10 // Returns 0
```

Note:

- If any expression is NULL, the BETWEEN operator returns a NULL value.

*is*

Tests a value against a boolean value, where boolean\_value can be TRUE, FALSE, or UNKNOWN.

### Comparison IS

```
SELECT 0 is false //return 1
```

```
SELECT 0 is true //return 0
```

```
SELECT 1 is true //return 1
```

```
SELECT 1 is false //return 0
```

```
SELECT 0 is unknown //return 0
```

```
SELECT 1 is unknown //return 0
```

```
SELECT null is unknown //return 1
```

## Note:

In SQL, the same name can be used for two (or more) attributes as long as the attributes are in different relations. If this is the case, and a multi-table query refers to two or more attributes with the same name, we must ***qualify*** the attribute name with the relation name to prevent ambiguity. This is done by prefixing the relation name to the attribute name and separating the two by a period (.).

# 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`.

## **Note:**

- You need not specify a **`tbl_name`** or **`db_name.tbl_name`** prefix for a column reference unless the reference would be ambiguous.
- The identifier quote character is the backtick (`)

- NULL value does not occupy space in memory.
- NULL value is independent of data type.
- NULL can be written in any lettercase.

## null-to-null comparisons

- SELECT NULL = NULL, NULL IS NULL; // Output will be NULL and 1
- SELECT NULL = NULL, NULL IS NOT NULL; // Output will be NULL and 0

## *the null value*

The **NULL** value is special. It means "**NO VALUE**" or "**UNKNOWN VALUE**".

### Remember:

- You cannot compare two "**UNKNOWN VALUE**" the way you compare two "**KNOWN VALUE**" to each other.
  - If you attempt to use **NULL** with the usual arithmetic comparison operators, the result is **NULL** (*undefined*).
  - Instead of using `=`, `<>`, or `!=` to test for equality or inequality with **NULL** values, use **IS NULL** or **IS NOT NULL**
  - MySQL specific `<=>` comparison operator is used for **NULL**-to-**NULL** comparisons.
- 

```
SELECT * FROM EMP WHERE COMM <=> NULL
```

### Note:

The result of (FALSE AND UNKNOWN) is FALSE, whereas the result of (FALSE OR UNKNOWN) is UNKNOWN.

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)

- `SELECT IFNULL (1, 2) AS R1;`
- `SELECT IFNULL (NULL, 2) AS R1;`
- `SELECT IFNULL (1/0, 2) AS R1;`
- `SELECT IFNULL (1/0, 'Yes') AS R1;`
- `SELECT comm, IFNULL(comm + comm*.25, 1000) FROM emp;`

## 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)**

- `SELECT IF(1 > 2, 2, 3) AS R1;`
- `SELECT sal, IF(sal = 3000, 'Ok', 'Not Bad') R1 FROM emp;`
- `SELECT ename, sal, IF(sal = 3000 AND ename = 'FORD', 'Y', 'N') R1 FROM emp;`
- `SELECT ename, sal, comm, IF(comm IS NULL && ename = 'FORD', 'Y', 'N') R1 FROM emp;`
- `SELECT deptno, IF(deptno = 10, 'Sales', IF(deptno = 20 , 'Purchase','N/A')) R1 FROM emp;`
- `SELECT productid, productname, unitprice, unitsinstock, reorderlevel, IF(unitsinstock < reorderlevel, 'Stock is less', 'Good Stock') as 'Stock Report' FROM products;`
- `SELECT hiredate, IF(( YEAR(hiredate) % 4 = 0 AND YEAR(hiredate) % 100 <> 0 ) OR YEAR(hiredate) % 400 = 0 , 'Leap Year', 'Not A Leap Year') FROM emp;`

## NULLIF function

Returns **NULL** if expr1 = expr2 is true, otherwise returns expr1.

**NULLIF(expr1, expr2)**

- `SELECT NULLIF(1, 1) AS R1;`
- `SELECT NULLIF(1, 2) AS R1;`

# *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**

- `SELECT deptno, CASE deptno WHEN 10 THEN 'Accounts' WHEN 20 THEN 'Sales' ELSE 'N/A' END R1 FROM emp;`
- `SELECT deptno, CASE deptno WHEN 10 THEN 'Accounts' ELSE 'N/A' END CASE FROM emp; # error`
- `SELECT custId, type, amount, CASE type WHEN 'd' THEN amount WHEN 'c' THEN amount * -1 END amount FROM transactions;`
  
- `SELECT job, SUM(CASE job WHEN 'manager1' THEN 1 ELSE 0 END) R1 FROM emp; # returns 0`
- `SELECT job, SUM(CASE job WHEN 'manager1' THEN 1 END) R1 FROM emp; # returns NULL`

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

- `SELECT deptno, CASE WHEN deptno = 10 THEN 'Sales' WHEN deptno = 20 THEN 'Purchase' ELSE 'N/A' END R1 FROM emp;`
- `SELECT companyname,  
CASE WHEN country IN ('USA', 'Canada') THEN 'North America'  
WHEN country = 'Brazil' THEN 'South America'  
WHEN country IN ('Japan', 'Singapore') THEN 'Asia'  
WHEN country = 'Australia' THEN 'Australia'  
ELSE 'Europe' END as Continent  
FROM suppliers  
ORDER BY companyname;`
- `SELECT hiredate, CASE WHEN (YEAR(hiredate) % 4 = 0 AND YEAR(hiredate) % 100 <> 0) OR YEAR(hiredate) % 400 = 0  
THEN 'LEAP YEAR' END R1 FROM emp;`

# *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 WHEN [condition] THEN result [WHEN [condition] THEN result . . .] [ELSE result] END
```

### \* Count(custID)

```
ORDER BY CASE orderCount
WHEN 1 THEN 'One-time Customer'
WHEN 2 THEN 'Repeated Customer'
WHEN 3 THEN 'Frequent Customer'
ELSE 'Loyal Customer' END customerType
```

### \* ORDER BY CASE

```
WHEN filter = 'Debit' THEN 1
WHEN filter = 'Credit' THEN 2
WHEN filter = 'Total' THEN 3
END transactionType;
```

```
* ORDER BY FIELD(status, 'In Process',
'On Hold', 'Cancelled', 'Resolved',
'Disputed', 'Shipped');
```

### \* ORDER BY CASE status

```
WHEN 'active' THEN 1
WHEN 'approved' THEN 2
WHEN 'rejected' THEN 3
WHEN 'submitted' THEN 4
ELSE 5 END statusType
```

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

- ```
SELECT cnum, COUNT(*), CASE
    WHEN COUNT(*) = 1 THEN 'one-time-customer'
    WHEN COUNT(*) = 2 THEN 'repeated-customer'
    WHEN COUNT(*) = 3 THEN 'frequent-customer'
    WHEN COUNT(*) >= 4 THEN 'loyal-customer'
END "Customer Report"
FROM orders GROUP BY cnum ORDER BY 2;
```

- `DATEDIFF(CURDATE(), hiredate) / 365.25`

## 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()**.

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

NOW() returns a constant time that indicates the time at which the statement began to execute. (Within a stored function or trigger, NOW() returns the time at which the function or triggering statement began to execute.) This differs from the behavior for SYSDATE(), which returns the exact time at which it executes.

- [SELECT SYSDATE\(\)](#)
- [SELECT NOW\(\)](#)
- [SELECT CURDATE\(\)](#)
- [SELECT CURTIME\(\)](#)

***Result in something like this:***

**SYSDATE()**  
2017-02-11 10:22:31

**NOW()**  
2017-02-11 10:22:31

**CURDATE()**  
2017-02-11

**CURTIME()**  
10:22:31

```
mysql> SELECT NOW(), SLEEP(7), NOW();
mysql> SELECT SYSDATE(), SLEEP(7), SYSDATE();
```

The DATE() function extracts the date value from a date or datetime expression.

### DATE(expr)

- `SELECT DATE("2017-06-15 09:34:21");`
- `SELECT NOW(), DATE(NOW());`

#### Note:

- Returns NULL if expression is not a date or a datetime value.
- `SELECT hiredate, DAYNAME(hiredate), IF(DAYNAME(hiredate)='Saturday', hiredate + INTERVAL 2 DAY,  
IF(DAYNAME(hiredate)='Sunday', hiredate + INTERVAL 1 DAY, IF(DAYNAME(hiredate)='Monday', hiredate + INTERVAL 7  
DAY, IF(DAYNAME(hiredate)='Tuesday', hiredate + INTERVAL 6 DAY, IF(DAYNAME(hiredate)='Wednesday', hiredate +  
INTERVAL 5 DAY, IF(DAYNAME(hiredate)='Thursday', hiredate + INTERVAL 4 DAY, IF(DAYNAME(hiredate)='Friday', hiredate +  
INTERVAL 3 DAY, 'TODO')))))) R1 FROM emp;`

## + 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		

## + 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 hiredate, DAYNAME(hiredate), IF(DAYNAME(hiredate) = 'Saturday', hiredate + INTERVAL 2 DAY,  
IF(DAYNAME(hiredate) = 'Sunday', hiredate + INTERVAL 1 DAY, IF(DAYNAME(hiredate) = 'Monday', hiredate + INTERVAL 7  
DAY, IF(DAYNAME(hiredate) = 'Tuesday', hiredate + INTERVAL 6 DAY, IF(DAYNAME(hiredate) = 'Wednesday', hiredate +  
INTERVAL 5 DAY, IF(DAYNAME(hiredate) = 'Thursday', hiredate + INTERVAL 4 DAY, IF(DAYNAME(hiredate) = 'Friday',  
hiredate + INTERVAL 3 DAY, 'TODO')))))) R1 FROM emp;`

*adddate()*

# **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

# *addtime()*

ADDTIME() adds expr2 to expr1 and returns the result. expr1 is a time or datetime expression, and expr2 is a time expression.

## ADDTIME(expr1, expr2)

- `SELECT NOW(), ADDTIME(NOW(), '1:1:1') # 'HH:MM:SS'`
- `SELECT NOW(), ADDTIME(NOW(), '2 2:10:5') # 'DAY HH:MM:SS'`
- Adding 15 seconds.  
`SELECT NOW(), ADDTIME(NOW(), "15") AS Updated_time;`
- Adding 10 minutes.  
`SELECT NOW(), ADDTIME(NOW(), "00:10:00") AS Updated_time;`
- Adding 2 hours.  
`SELECT NOW(), ADDTIME(NOW(), "2:00:00") AS Updated_time;`
- Adding 10 hours 30 minute 25 second and 100000 Microseconds.  
`SELECT NOW(), ADDTIME(NOW(), "10:30:25.100000") AS Updated_time;`
- Adding 1 day 2 hours 30 minute 25 second.  
`SELECT NOW(), ADDTIME(NOW(), "1 2:30:25") AS Updated_time;`

=

## subtime()

SUBTIME() returns expr1 – expr2 expressed as a value in the same format as expr1. expr1 is a time or datetime expression, and expr2 is a time expression.

### SUBTIME(expr1, expr2)

- `SELECT NOW(), SUBTIME(NOW(), '1:1:1'); # 'HH:MM:SS'`
- `SELECT NOW(), SUBTIME(NOW(), '2 1:1:1'); # 'DAY HH:MM:SS'`
- Subtract 15 seconds.  
`SELECT NOW(), SUBTIME(NOW(), "15") AS Updated_time;`
- Subtract 10 minutes.  
`SELECT NOW(), SUBTIME(NOW(), "00:10:00") AS Updated_time;`
- Subtract 2 hours.  
`SELECT NOW(), SUBTIME(NOW(), "2:00:00") AS Updated_time;`
- Subtract 10 hours 30 minute 25 second and 100000 Microseconds.  
`SELECT NOW(), SUBTIME(NOW(), "10:30:25.100000") AS Updated_time;`
- Subtract 1 day 2 hours 30 minute 25 second.  
`SELECT NOW(), SUBTIME(NOW(), "1 2:30:25") AS Updated_time;`

# 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		

- `SELECT EXTRACT(MONTH FROM NOW());`
- `SELECT EXTRACT(YEAR_MONTH FROM NOW());`

Note:

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

e.g.

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

## *period\_diff()*

MySQL PERIOD\_DIFF() returns the difference between two periods. Periods should be in the same format i.e. YYYYMM or YYMM.

`PERIOD_DIFF(Period1, Period2);`

- `SELECT PERIOD_DIFF(201701, 201601);`
- `SELECT PERIOD_DIFF(EXTRACT (YEAR_MONTH FROM NOW()), EXTRACT(YEAR_MONTH FROM hiredate)) AS R1 FROM emp;`

# *datetime functions*

Syntax	Result
DAY(date)	DAY() is a <b>synonym for DAYOFMONTH()</b> .
DAYNAME(date)	Returns the name of the weekday for date.
DAYOFMONTH(date)	Returns the day of the month for date, in the range 1 to 31
DAYOFWEEK(date)	Returns the weekday index for date (1 = Sunday, 2 = Monday, ..., 7 = Saturday).
DAYOFYEAR(date)	Returns the day of the year for date, in the range 1 to 366
LAST_DAY(date)	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.
MONTH(date)	Returns the month for date, in the range 1 to 12 for January to December
MONTHNAME(date)	Returns the full name of the month for date.
YEAR(date)	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 NOW(), NOW() + INTERVAL 1 DAY, WEEKDAY(NOW() + INTERVAL 1 DAY);`
- `SELECT * FROM emp WHERE DAY(hiredate) = 17;`
- `SELECT YEAR(hiredate), ( YEAR(hiredate) % 4 = 0 AND YEAR(hiredate) % 100 != 0 ) OR YEAR(hiredate) % 400 = 0 R1`  
`FROM emp ;`
- `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
%l	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)

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

# *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

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

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. <b>e.g.</b> <ul style="list-style-type: none"><li>• <code>SELECT ASCII(ename) FROM emp;</code></li></ul>
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> <b>e.g.</b> <ul style="list-style-type: none"><li>• <code>SELECT CHAR(65, 66, 67); / SELECT CAST(CHAR(65 66, 67) AS CHAR);</code></li></ul>
CONCAT(str1, str2, ...)	Returns the string that results from concatenating the arguments. CONCAT() <b>returns NULL if any argument is NULL.</b> <b>e.g.</b> <ul style="list-style-type: none"><li>• <code>SELECT CONCAT('Mr. ', ename) FROM emp;</code></li><li>• <code>SELECT CONCAT('My', NULL, 'SQL');</code> #op will be NULL</li></ul>
ELT(N, str1, str2, str3, ...)	ELT() returns the Nth element of the list of strings: str1 if N = 1, str2 if N = 2, and so on. Returns NULL if N is less than 1 or greater than the number of arguments. <b>e.g.</b> <ul style="list-style-type: none"><li>• <code>SELECT ELT(1, 'Bank', 'Of', 'India', 'Kothrud', 'Pune');</code></li><li>• <code>SELECT ELT(1, ename, job, sal) FROM emp;</code></li><li>• <code>SELECT hiredate, ELT(MONTH(hiredate),'Winter', 'Winter', 'Spring', 'Spring', 'Spring', 'Summer', 'Summer', 'Summer', 'Autumn', 'Autumn', 'Autumn', 'Winter') R1 FROM emp;</code></li></ul>

# *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 synonym for <code>LOWER()</code> .
<code>UCASE(str)</code>	Returns upper case string. <code>UCASE()</code> is a synonym for <code>UPPER()</code> .
<code>LENGTH(str)</code>	Returns the length of the string.
<code>LPAD(str, len, padstr)</code>	Returns the string str, left-padded with the string padstr to a length of len characters.
<code>RPAD(str, len, padstr)</code>	Returns the string str, right-padded with the string padstr to a length of len characters.
<code>REPEAT(str, count)</code>	Returns a string consisting of the string str repeated count times. If count is less than 1, returns an empty string. Returns NULL if str or count are NULL.

- `SELECT UCASE(ename) FROM emp;`
- `SELECT sal, LPAD(sal, 20, '*') FROM emp;`

# *string functions*

Syntax	Result
<code>LEFT(str, len)</code>	Returns the leftmost len characters from the string str, or NULL if any argument is NULL.
<code>RIGHT(str, len)</code>	Returns the rightmost len characters from the string str, or NULL if any argument is NULL.
<code>LTRIM(str)</code>	Returns the string str with leading space characters removed.
<code>RTRIM(str)</code>	Returns the string str with trailing space characters removed.
<code>TRIM(str)</code>	Returns the string str with leading and trailing space characters removed.
<code>BINARY</code> value	Convert a value to a binary string.

- `SELECT ename, BINARY ename FROM emp;`

# *string functions*

Syntax	Result
INSTR(str, substr)	Returns the position of the first occurrence of substring substr in string str.
REPLACE(str, from_str, to_str)	Returns the string str with all occurrences of the string from_str replaced by the string to_str. REPLACE() performs a case-sensitive match when searching for from_str. <i>e.g.</i> <ul style="list-style-type: none"><li>• <code>SELECT REPLACE('Hello', 'l', 'x');</code></li></ul>
REVERSE(str)	Returns the string str with the order of the characters reversed.
SUBSTR(str, pos, len)	<b>SUBSTR() is a synonym for SUBSTRING().</b> <i>e.g.</i> <ul style="list-style-type: none"><li>• <code>SELECT SUBSTR ('This is the test by IWAY', 6);</code></li><li>• <code>SELECT SUBSTR ('This is the test by IWAY', -4, 4);</code></li></ul>
MID(str, pos, len)	MID function is a synonym for SUBSTRING.

- `SELECT ename, job, IF(ISNULL(phone), '*****', RPAD(LEFT(phone, 4), 10, '*')) FROM emp;`
- `SELECT ename, job, phone, IF(ISNULL(phone), REPEAT('*', 10), RPAD(LEFT(phone, 4), 10, '*')) FROM emp;`
- `SELECT `user name`, IF(LENGTH(SUBSTR(`user name`, INSTR(`user name`, " "))) = 0, "Weak User", `user name`) R1 FROM emp;`
- `UPDATE emp SET job = REPLACE(job, job, LOWER(job));`

# *string functions - examples*

- `SELECT sal, REPEAT('$', sal/100) FROM emp;`
- `SELECT emailid, REPEAT('*', LENGTH(emailid)) FROM emp;`
- `SELECT pwd, REPEAT('*', LENGTH(pwd)) password FROM emp;`
- `SELECT c1, CONCAT(REPEAT('0', 10 - LENGTH(c1)), c1) FROM leading_zeroes;`
- `SELECT ename, job, IF(ISNULL(phone), '*****', RPAD(LEFT(phone, 4), 10, '*')) FROM emp;`
- `SELECT `user name`, IF(LENGTH(SUBSTR(`user name`, INSTR(`user name`, " "))) = 0, "Weak User", `user name`) R1 FROM emp;`
- `SELECT LENGTH('saleel') - LENGTH(REPLACE('saleel', 'e', ""));`
- `SELECT empno, datePresent, LENGTH(datePresent) - LENGTH(REPLACE(datePresent, ",", "")) + 1 "Days Present" FROM emp_attendance;`
- `SELECT CandidateID, REPLACE(REPLACE(response, ',', ''), 'n', '') R1, LENGTH(REPLACE(REPLACE(response, ',', ''), 'n', '')) R2 FROM vote_response;`
- `SELECT c1, c1 / 1, SUBSTR(c1, LENGTH(c1 / 1) + 1) FROM numberString;`
- `SELECT c1, REVERSE(c1) / 1, LENGTH(REVERSE(c1) / 1), REVERSE(SUBSTR(REVERSE(c1), LENGTH(REVERSE(c1) / 1) + 1)) FROM Stringnumber;`
- `UPDATE emp SET job := REPLACE(job, 'officers', 'Officers');`

## *string functions - examples*

- `SELECT * FROM emp1 WHERE ename = BINARY "sherlock";`
- `SELECT * FROM emp1 WHERE ename = BINARY "Sherlock";`
- `SELECT * FROM emp1 WHERE ename = BINARY UPPER(ename);`
- `SELECT * FROM emp1 WHERE ename = BINARY LOWER(ename);`
- `SELECT CONCAT(UCASE(LEFT(ename, 1)), LCASE(SUBSTRING(ename, 2))) "Title Case" FROM emp;`
- `SELECT * FROM emp1 WHERE ename = BINARY CONCAT(UCASE(LEFT(ename, 1)), LCASE(SUBSTRING(ename, 2)));`

# *string functions*

- `SELECT * FROM emp WHERE LEFT(ename, 1) IN ('a', 'e', 'o', 'i', 'u');`
- `SELECT * FROM emp WHERE REGEXP_LIKE(ename, '(a|e|i|o|u)');`
- `SELECT * FROM emp WHERE REGEXP_LIKE(ename, '^ (a|m)'); //starts with`
- `SELECT * FROM emp WHERE REGEXP_LIKE(ename, '(n|r)$'); //ends with`
- `SELECT "abc,,abc,,,,bc,,,,,abc" ; / SELECT REPLACE(REPLACE(REPLACE("abc,,abc,,,,bc,,,,,abc", ",","."),".",""),".","")) R1 ;`

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.

```
SELECT FLOOR(RAND() * (b - a + 1) + a );
```

## *mathematical functions*

e.g.

- `SELECT CEIL(1.23);`
- `SELECT CEIL(-1.23);`
- `SELECT FLOOR(1.23);`
- `SELECT FLOOR(-1.23);`
- `SELECT ROUND(-1.23);`
- `SELECT ROUND(-1.58);`
- `SELECT ROUND(RAND() * 100);`
- `SELECT FLOOR(RAND() * 899999 + 100000) OTP;`
- `SELECT weight, TRUNCATE(weight, 0) AS kg, MID(weight, INSTR(weight, ".") + 1) AS gms FROM mass_table;`
- `SELECT weight, TRUNCATE(weight, 0) AS kg, RIGHT(MOD(weight , 1), 2) AS gms FROM mass_table;`

## 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 statement...

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

## *table statement*

The TABLE statement in some ways acts like SELECT. You can order and limit the number of rows produced by TABLE using ORDER BY and LIMIT clauses, respectively.

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

- `TABLE emp;`
- `TABLE emp ORDER BY 2;`
- `TABLE emp ORDER BY 2 LIMIT 1, 2;`
- `TABLE t1 UNION TABLE t2;`

## 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 [ALL / DISTINCT / DISTINCTROW] identifier.* / identifier.A1 [ [as] alias_name], identifier.A2 [ [as] alias_name], identifier.A3 [ [as] alias_name], expression1 [ [as] alias_name], expression2 [ [as] alias_name] ...`

- [ `FROM <identifier.r1> [as] alias_name], <identifier.r2> [as] alias_name], ... ]`
- [ `WHERE < where_condition1 > { and | or } < where_condition2 > ... ]`
- [ `GROUP BY < { col_name | expr | position }, ... [ WITH ROLLUP ] > ]`
- [ `HAVING < having_condition1 > { and | or } < having_condition2 > ... ]`
- [ `ORDER BY < { col_name | expr | position } [ ASC | DESC ], ... > ]`
- [ `LIMIT < { [offset,] row_count | row_count OFFSET offset } > ]`
- [ `FOR { UPDATE } ]`
- [ { `INTO OUTFILE 'file_name' | INTO DUMPFILE 'file_name' | INTO var_name [, var_name], ... }` } ]

## **select statement**

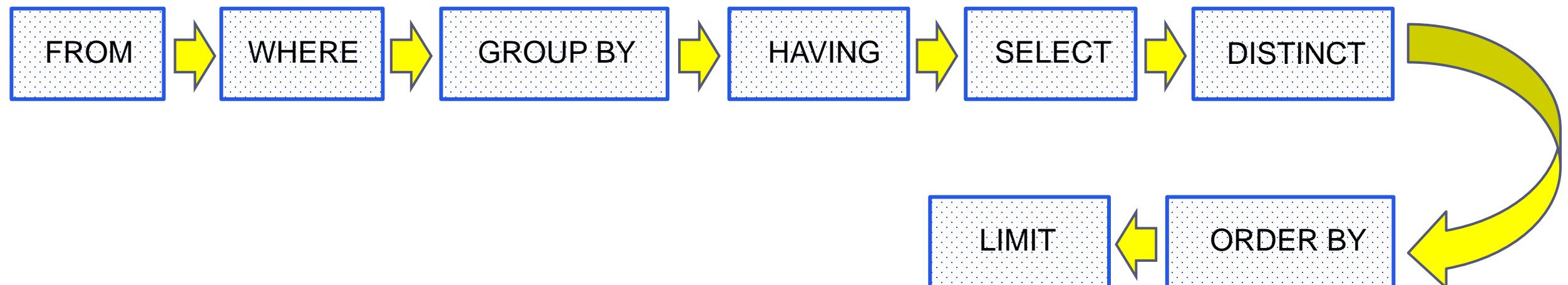
### **Remember:**

- **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



# select statement... (is checks for)

## Syntax Check

MySQL Database must check each SQL statement for syntactic validity.

```
mysql> SELECT * FORM emp;
```

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'FORM emp' at line 1

## Semantic Check

A semantic check determines whether a statement is meaningful, for example, whether the objects and columns in the statement exist.

```
mysql> SELECT * FROM nonexistent_table;
```

ERROR 1146 (42S02): Table 'db1.nonexistent\_table' doesn't exist

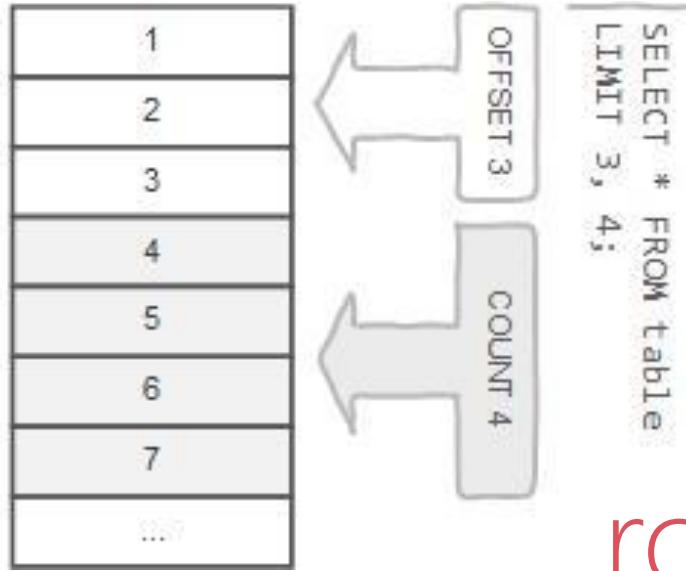
# UUID() / UUID\_SHORT()

A UUID is a Universal Unique Identifier and 128-bit long value.

## Remember:

- UUID values in MySQL are **unique** across tables, databases, and servers..
- `SELECT UUID() AS R1, UUID_SHORT() AS R2 FROM tbl_name;`

```
SELECT * FROM table;
```



## 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.

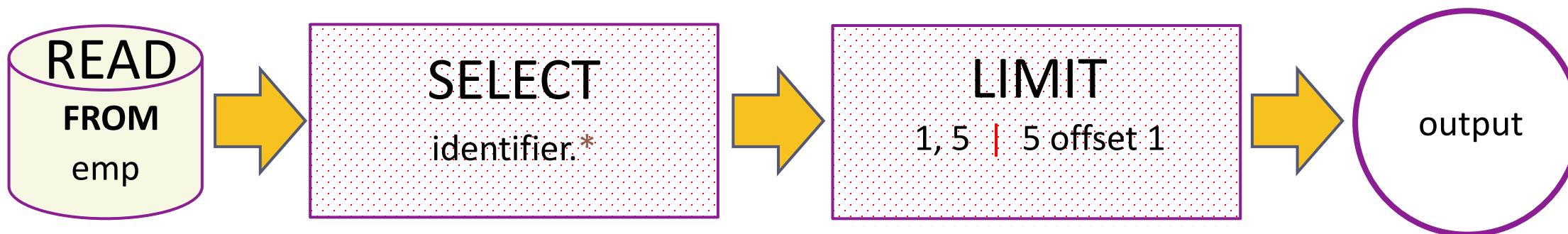
## *select - limit*

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;

## *select - limit*

This variable is used to return the maximum number of rows from SELECT statements. Its default value is unlimited. But if you changed the limit then SELECT statement returns the rows equals to the value.

`SQL_SELECT_LIMIT = { value | DEFAULT }`

This variable does not apply to SELECT statement that executed in the stored procedures or functions.

- `SET SQL_SELECT_LIMIT = 2;`
- `SET SQL_SELECT_LIMIT = DEFAULT;`
- `SELECT * FROM emp;`

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 A1, A2, A3, An FROM r`

`[ORDER BY {A1, A2, A3, ... | expr | position} [ASC | DESC], ...]`

"Ordered by attributes  $A_1, A_2, A_3 \dots$ "

- 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  $A_3$

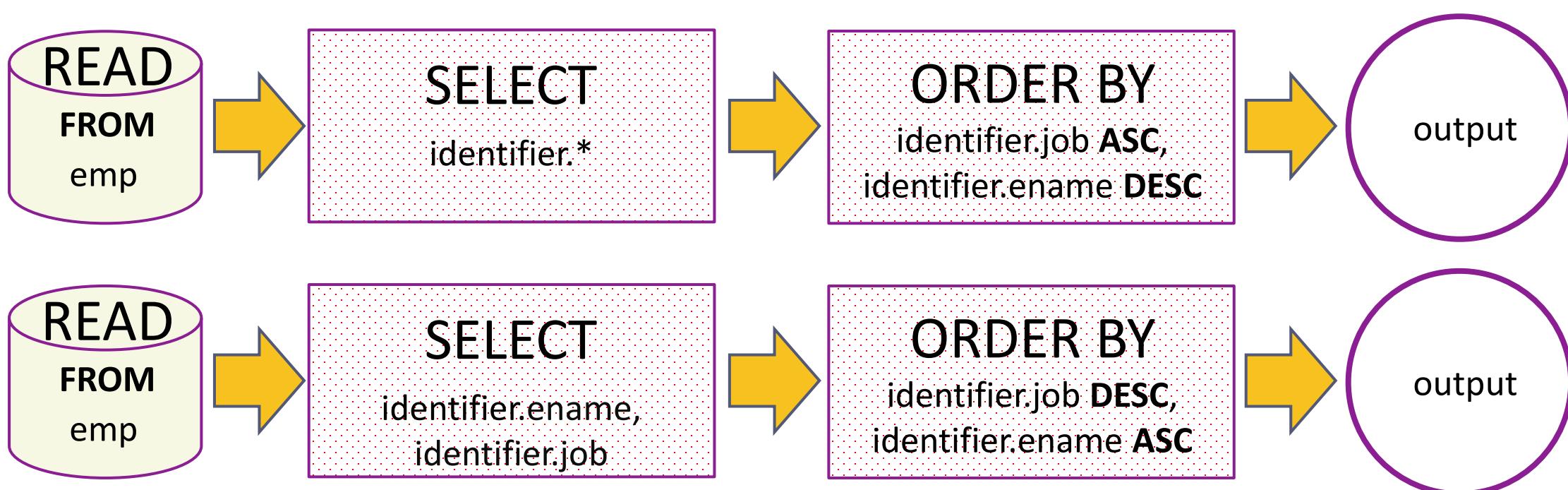
- `SELECT * FROM r ORDER BY key_part1, key_part2;` // optimizer does not use the index.
- `SELECT key_part1, key_part2 FROM r ORDER BY key_part1, key_part2;` // optimizer uses the index.

# *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 { $A_1, A_2, A_3, \dots$  | expr | 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;`

## *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;`

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?

- 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  $P$  ]

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

WHERE salary > 20000 AND < 30000 --Illegal

WHERE state NOT = 'CA' --Illegal

## 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.

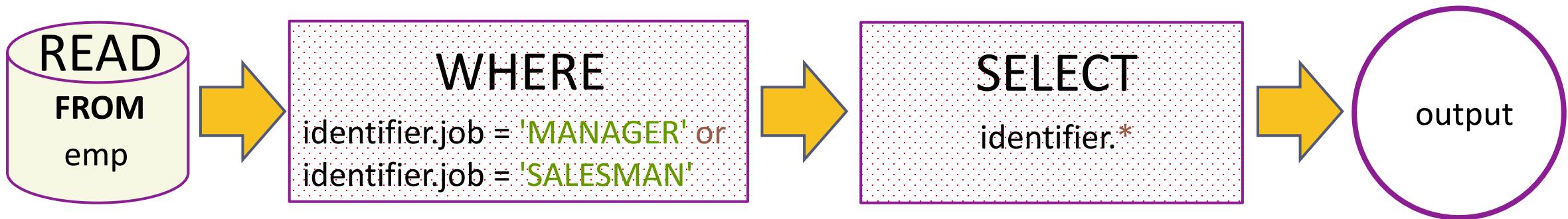
- SELECT GREATEST(10, 20, 30), # 30  
LEAST(10, 20, 30); # 10
- SELECT GREATEST(10, null, 30), # null  
LEAST(10, NULL, 30); # null
- SELECT \* FROM emp WHERE (deptno, pwd) = (SELECT deptno, pwd FROM dept WHERE deptno = 30);

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	a12recppm	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 andor WHERE ename = 'saleel' AND city = 'pune' OR city = 'baroda';`
- `SELECT * FROM andor 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:

*is null / is not null*

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;

	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;`

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

	<code>id</code>	<code>title</code>	<code>completed</code>
▶	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;`

	<code>id</code>	<code>title</code>	<code>completed</code>
▶	1	Task1	0
	3	Task3	0
	7	Task7	0
	12	Task12	0
	NULL	NULL	NULL

## *select – boolean*

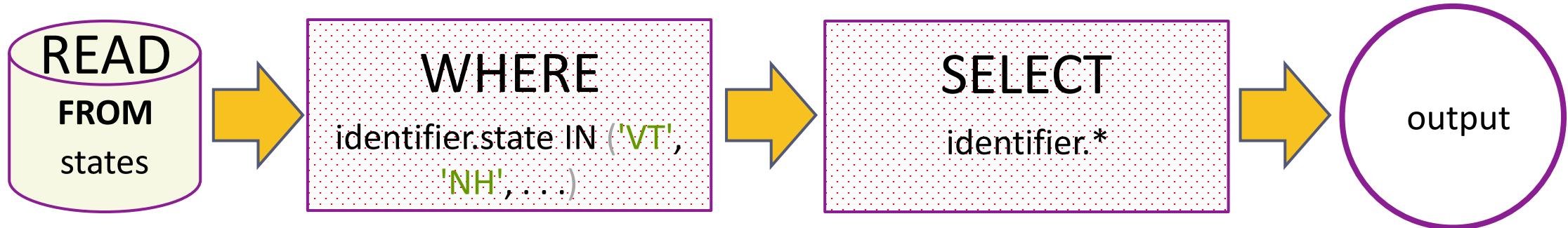
- BOOL and BOOLEAN are synonym of TINYINT(1)

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

in

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 r1 WHERE (`

`state = 'VT' OR  
 state = 'NH' OR  
 state = 'ME' OR  
 state = 'MA' OR  
 state = 'CT' OR  
 state = 'RI'`



- `SELECT ... FROM r1 WHERE state IN ('VT', 'NH', 'ME', 'MA', 'CT', 'RI');`
- `SELECT ... FROM r1 WHERE state IN (SELECT ...);`

`);`

**A IN (B<sub>1</sub>, B<sub>2</sub>, B<sub>3</sub>, etc.)**    A is found in the list (B<sub>1</sub>, B<sub>2</sub>, 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)"

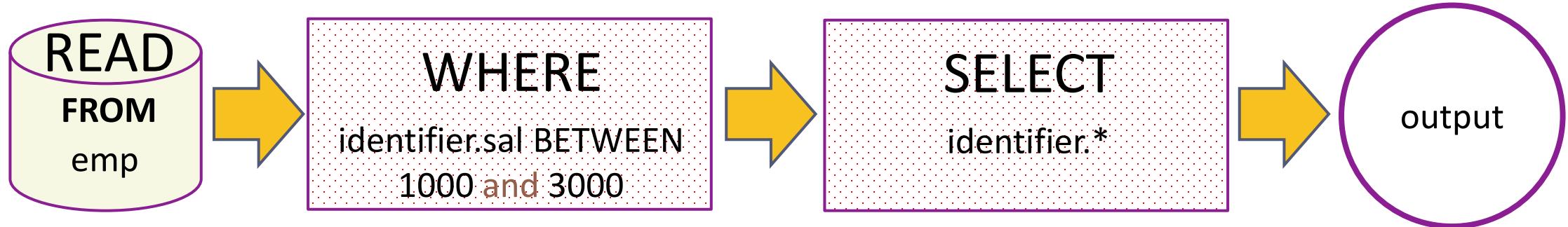
## Remember:

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)`



#### 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.

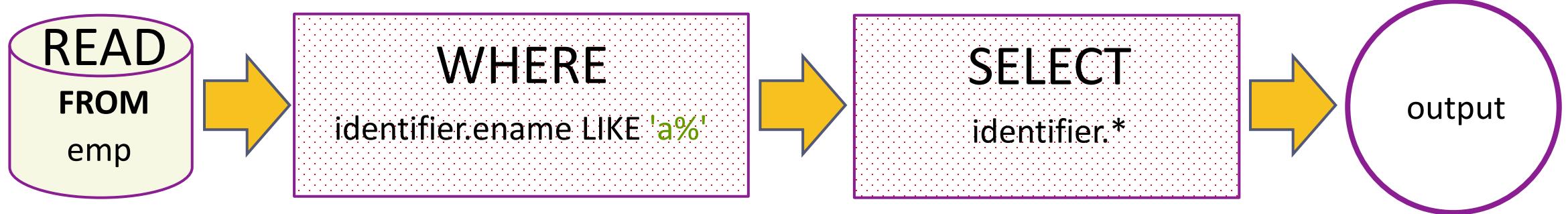
SELET empno, ename, job, hiredate, sal, comm, deptno, isactive FROM emp WHERE sal BETWEEN 1000 AND NULL;

- 
- 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.
-

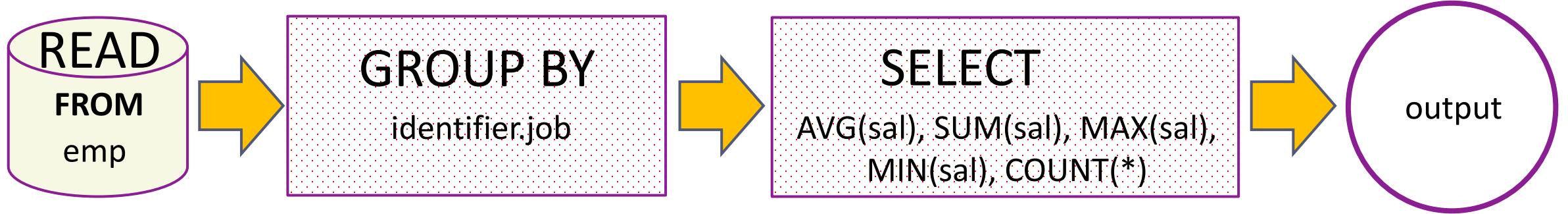
like

- 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.

## 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;`

## 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;`

## 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;`

## 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;

Things to... Remember:

*aggregate functions*

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.**

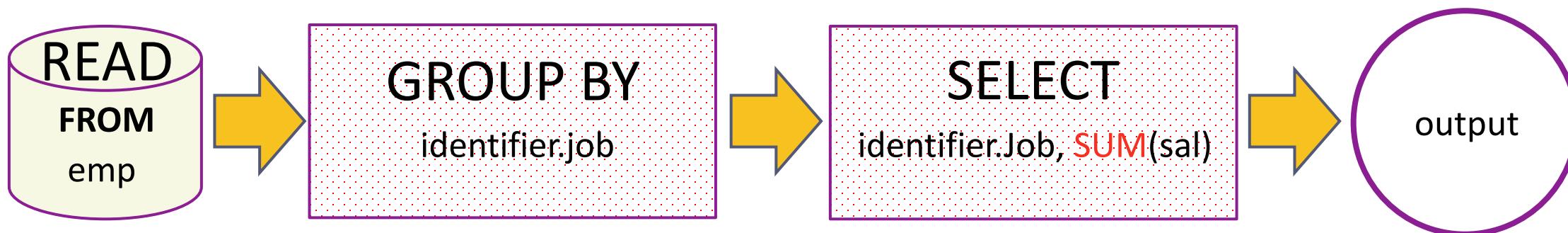
**select - group by**

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

- SET *SQL\_MODE* = '';
- SET *SQL\_MODE* = 'ONLY\_FULL\_GROUP\_BY';

*select - 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';` \*
  - **HAVING** is merged with **WHERE** if you do not use GROUP BY or Aggregate Functions (COUNT(), . . .)
- \* **ERROR: Unknown column '...'**  
**in 'having clause'**

## *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, whereas **HAVING** clause can only be used with the **SELECT** statement.
  - **WHERE** clause filters rows before aggregation (GROUPING), whereas, **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 whereas **HAVING** clause acts as a post-filter.

# *where vs having*

**WHERE**

**Vs**

**HAVING**

<b>WHERE</b>	<b>HAVING</b>
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;
```

# 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	r50mom
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	r50mom
4	7566	JONES	MANAGER	7839	1981-04-02 00:00:00	2975.00	NULL	20	4	JONES	a12recmom
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	a12recmom

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

## 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:

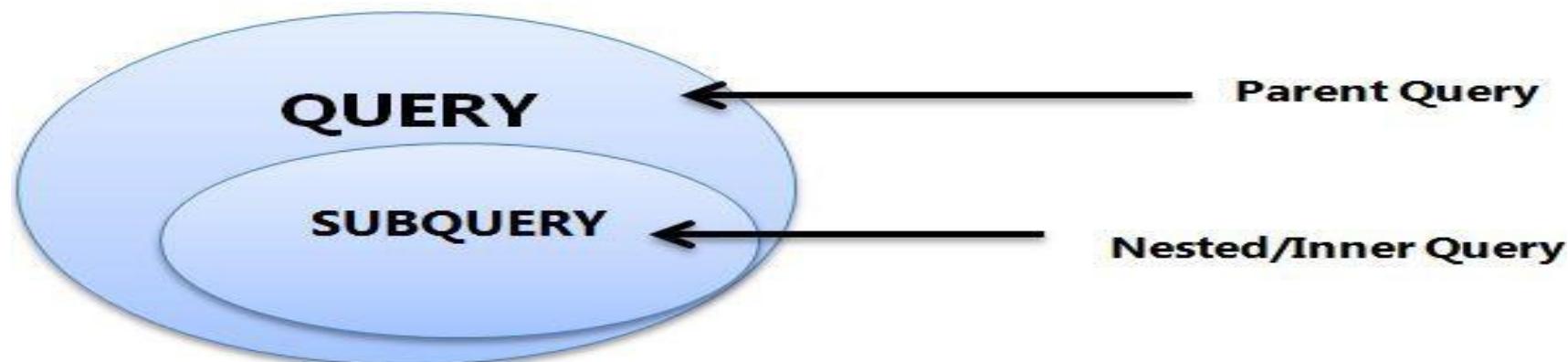
- a **SELECT** clause
- a **FROM** clause
- a **WHERE** clause
- a **HAVING** clause

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

**Note:**

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

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



## *single row subqueries*

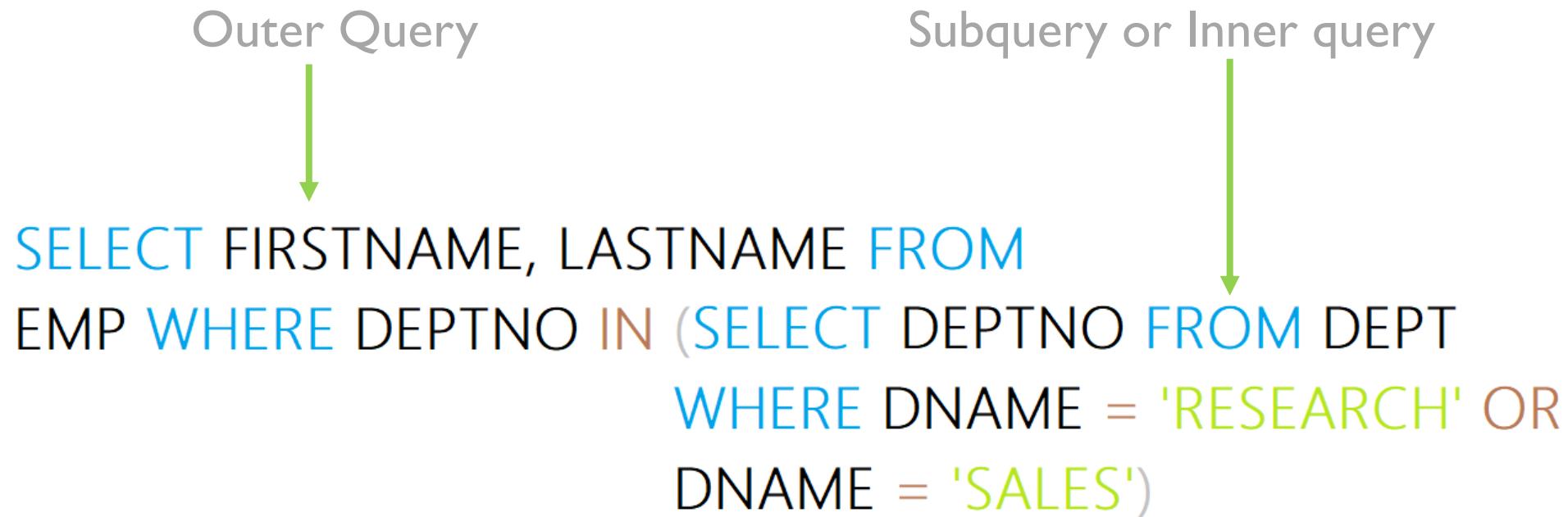
A single row subquery returns **zero or one row** to the outer SQL statement. You can place a subquery in a **WHERE** clause, a **HAVING** clause, or a **FROM** clause of a SELECT statement.

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

Subquery or Inner query  
↓

## *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.



## *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 A1, A2, A3, (subquery) as A4, ... 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 A1, A2, A3, (subquery) as A4, ... FROM (subquery) [AS] name, ...`

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`

	<b>MAX(R1)</b>
	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 A1, A2, A3, (subquery) as A4, ... FROM r WHERE p = (subquery)`

**Remember:**

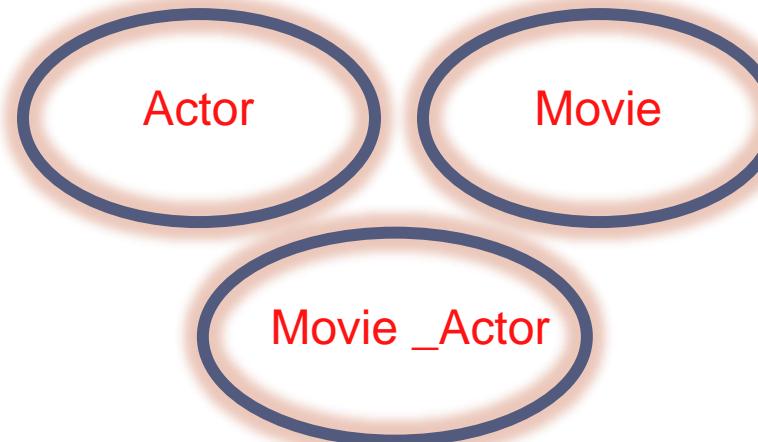
- A subquery can be used before or after any of the comparison operators.
  - The subquery can return **at most one value**.
  - The value can be the result of an arithmetic expression or a function.
- 
- `SELECT * FROM emp WHERE deptno = (SELECT 5 + 5);`
  - `SELECT * FROM emp WHERE sal = (SELECT MAX(sal) 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	Akshay 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*

- *operand comparison\_operator ANY (subquery)* The word SOME is an alias for ANY.
- *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*

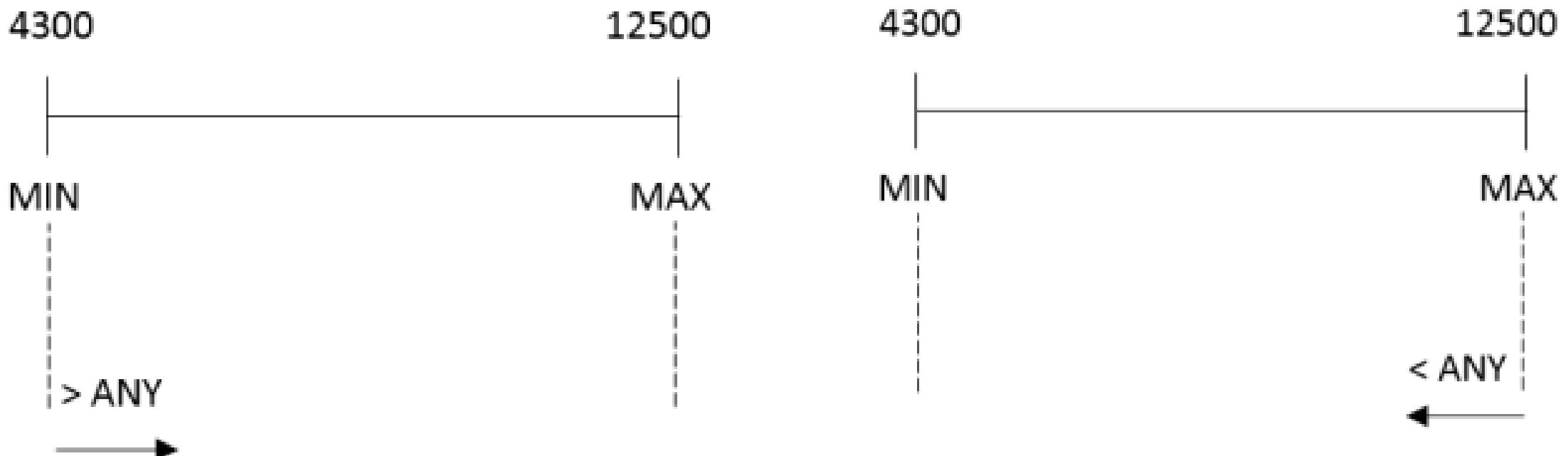
TODO

SELECT  $A_1, A_2, A_3, (\text{subquery})$  as  $A_4, \dots$  FROM  $(\text{subquery})$  [AS] Alias WHERE  $p$  IN  $(\text{subquery})$   
SELECT  $A_1, A_2, A_3, (\text{subquery})$  as  $A_4, \dots$  FROM  $(\text{subquery})$  [AS] Alias WHERE  $p$  ANY  $(\text{subquery})$   
SELECT  $A_1, A_2, A_3, (\text{subquery})$  as  $A_4, \dots$  FROM  $(\text{subquery})$  [AS] Alias WHERE  $p$  ALL  $(\text{subquery})$

- 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`

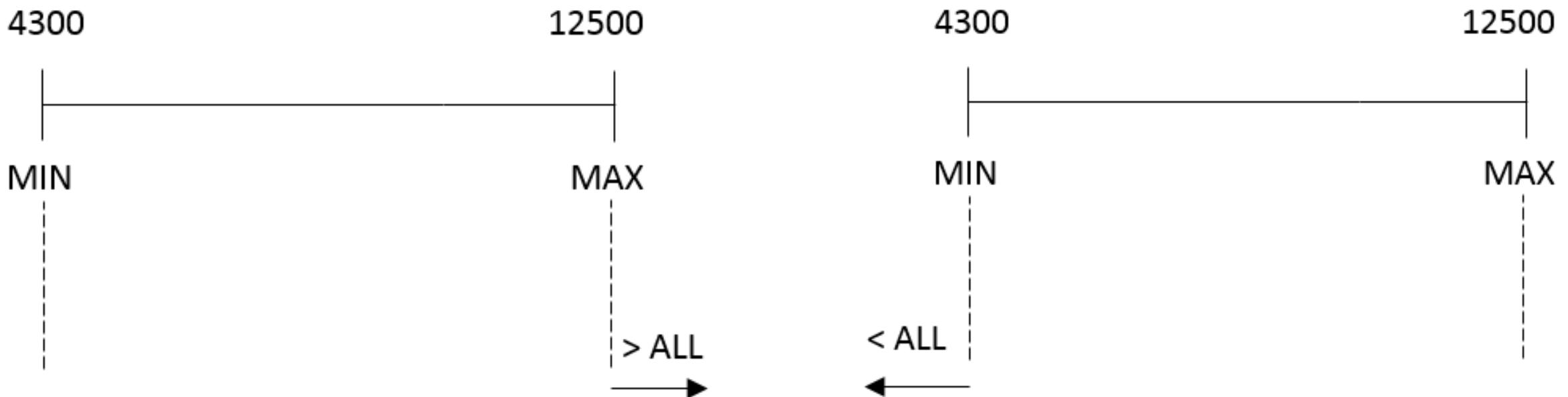
## *any / some*

- "`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.



*all*

- " $x = \text{ALL}(\dots)$ ": The value must match all the values in the list to evaluate to TRUE.
- " $x \neq \text{ALL}(\dots)$ ": The value must not match any values in the list to evaluate to TRUE.
- " $x > \text{ALL}(\dots)$ ": The value must be greater than the biggest value in the list to evaluate to TRUE.
- " $x < \text{ALL}(\dots)$ ": The value must be smaller than the smallest value in the list to evaluate to TRUE.
- " $x \geq \text{ALL}(\dots)$ ": The value must be greater than or equal to the biggest value in the list to evaluate to TRUE.
- " $x \leq \text{ALL}(\dots)$ ": The value must be smaller than or equal to the smallest value in the list to evaluate to TRUE.



*all*

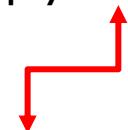
The expression is TRUE, if table T2 is empty.

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

This statement will return all rows from EMP table.



empty 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 d1 WHERE c1 not in (SELECT min(c1) FROM d1 GROUP BY deptno, dname, loc, walletid) ORDER BY deptno;
```

row subquery

## *row subqueries*

Scalar or column Subqueries return a single value or a column of values. A row subquery is a subquery variant that returns a single row and can thus return more than one column value. You can use = , >, <, >=, <=, <>, !=, <=>

1. `SELECT * from EMP where deptno, 1 = (SELECT deptno, 1 from DEPT where deptno=10);`
2. `SELECT * FROM emp WHERE ROW (deptno, 1) = (SELECT deptno, 1 FROM dept WHERE deptno = 10);`

- `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 A1, A2, A3, A4, ... 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);`

IS NULL  
NULL

joins

*joins*

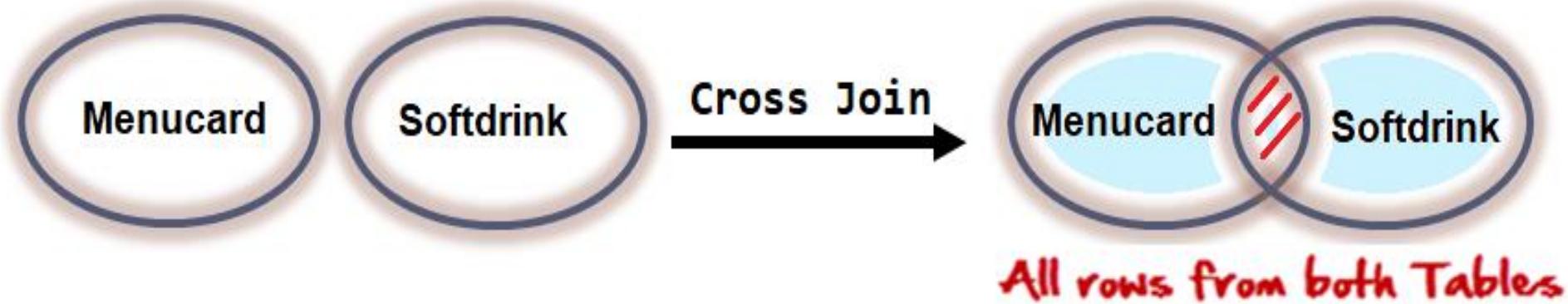
**JOINS** are used to **retrieve data from multiple tables.**

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

*joins*

## 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

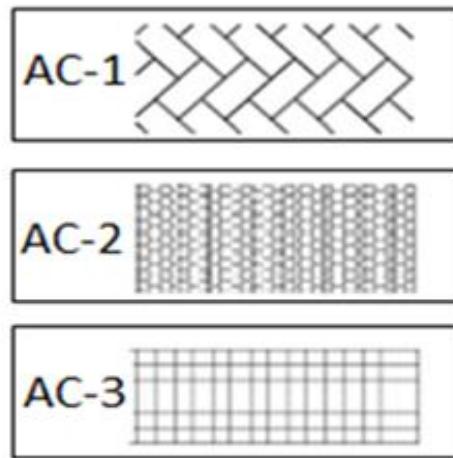
- **Cartesian/Product means** Number of Rows present in Table 1 Multiplied by Number of Rows present in Table 2.
- **Cross Join in MySQL** does not require any common column to **join** two table.

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.

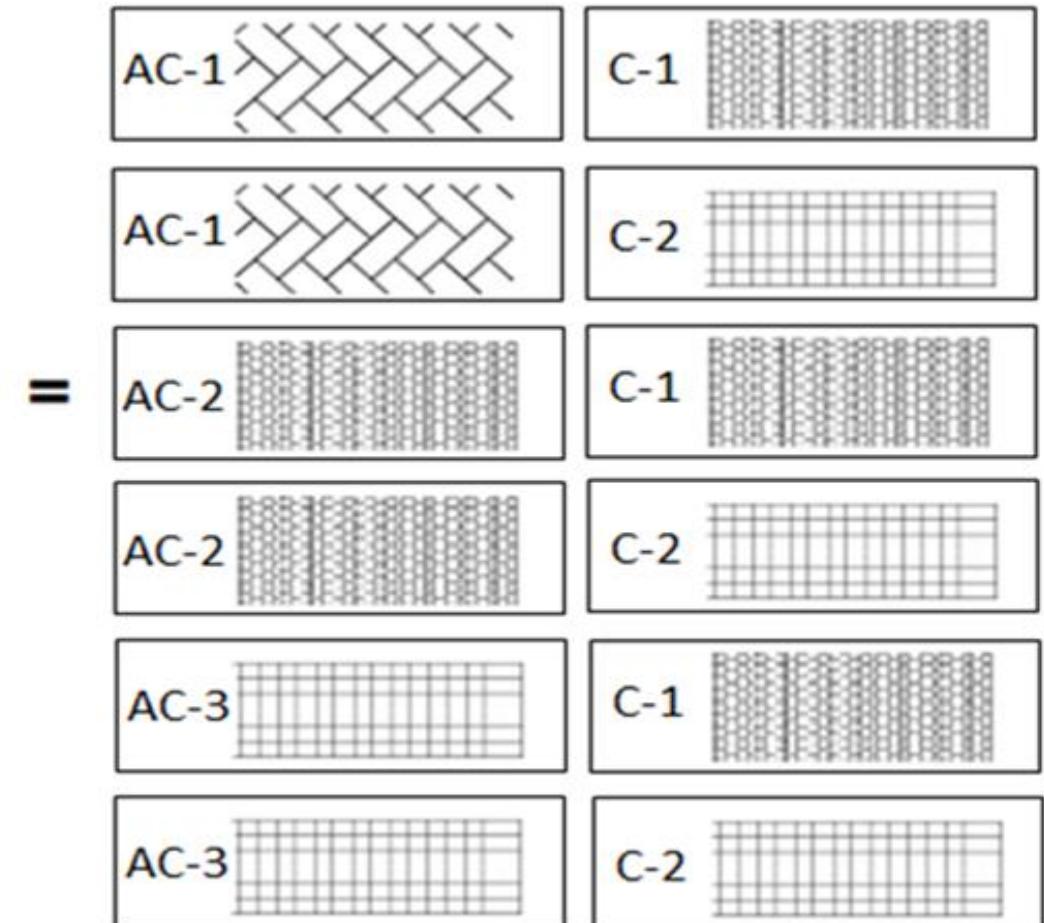
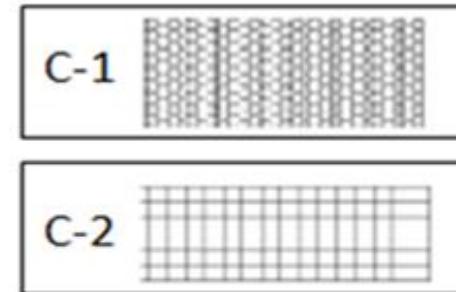
$$\begin{aligned} \text{Degree } d(R \times S) &= d(R) + d(S). \\ \text{Cardinality } |R \times S| &= |R| * |S| \end{aligned}$$

## *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.



CROSS JOIN



## *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  $r_1, r_2, \dots$

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

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

$$r_1 \times r_2$$

$$R = \{ (1, a),$$

$$(2, a),$$

$$(1, b),$$

$$(2, b),$$

$$(1, c),$$

$$(2, c) \}$$

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

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

$$r_1 \times r_2$$

$$R = \{ (1, a),$$

$$(1, b),$$

$$(2, a),$$

$$(2, b),$$

$$(3, a),$$

$$(4, b) \}$$

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

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

$$r_1 \times r_2$$

$$R = \{ (1, a),$$

$$(2, a),$$

$$(1, b),$$

$$(2, b),$$

$$(1, \text{null}),$$

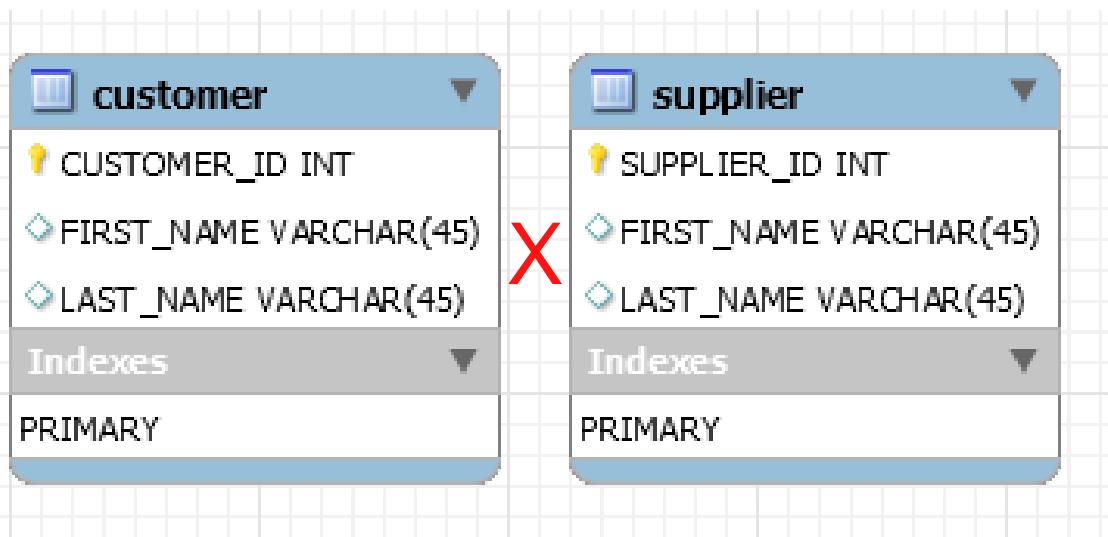
$$(2, \text{null}) \}$$

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

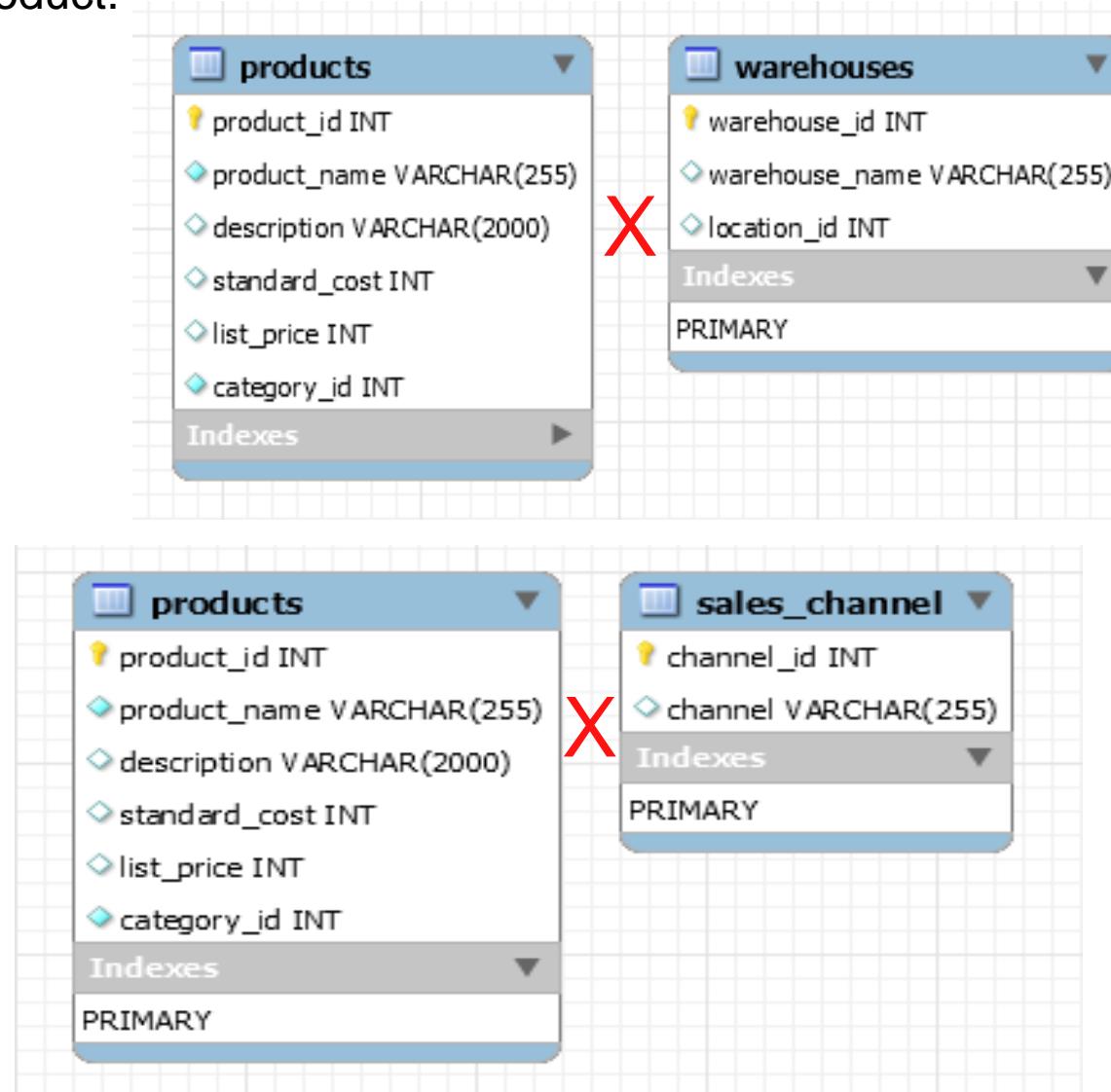
# *joins - cartesian or product*

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



# 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 menocard 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 Cherry	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 A1, A2, A3, ... FROM r1 CROSS JOIN r2, ...`

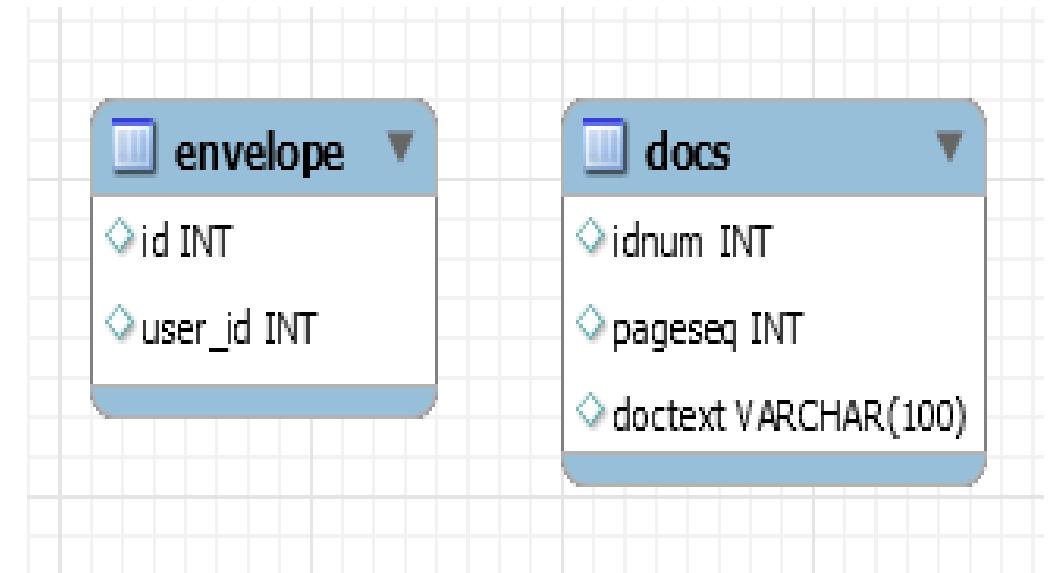
**envelope Table**

	<b>id</b>	<b>user_id</b>
▶	1	1
	2	2
	3	3

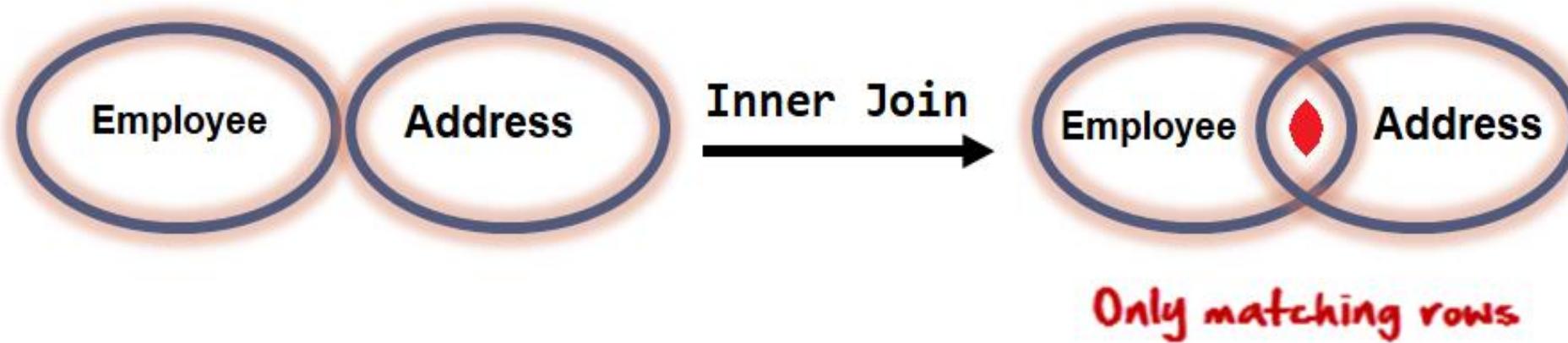
**docs Table**

	<b>idnum</b>	<b>pageseq</b>	<b>doctext</b>
▶	1	5	NULL
	2	6	NULL
	HULL	0	NULL

- `SELECT * FROM envelope CROSS JOIN docs;`



	<b>id</b>	<b>user_id</b>	<b>idnum</b>	<b>pageseq</b>	<b>doctext</b>
▶	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	HULL	0	NULL
	2	2	HULL	0	NULL
	3	3	HULL	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  $R(A_1, A_2, \dots, A_n) \bowtie_{\text{join condition}} S(B_1, B_2, \dots, B_m)$  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.

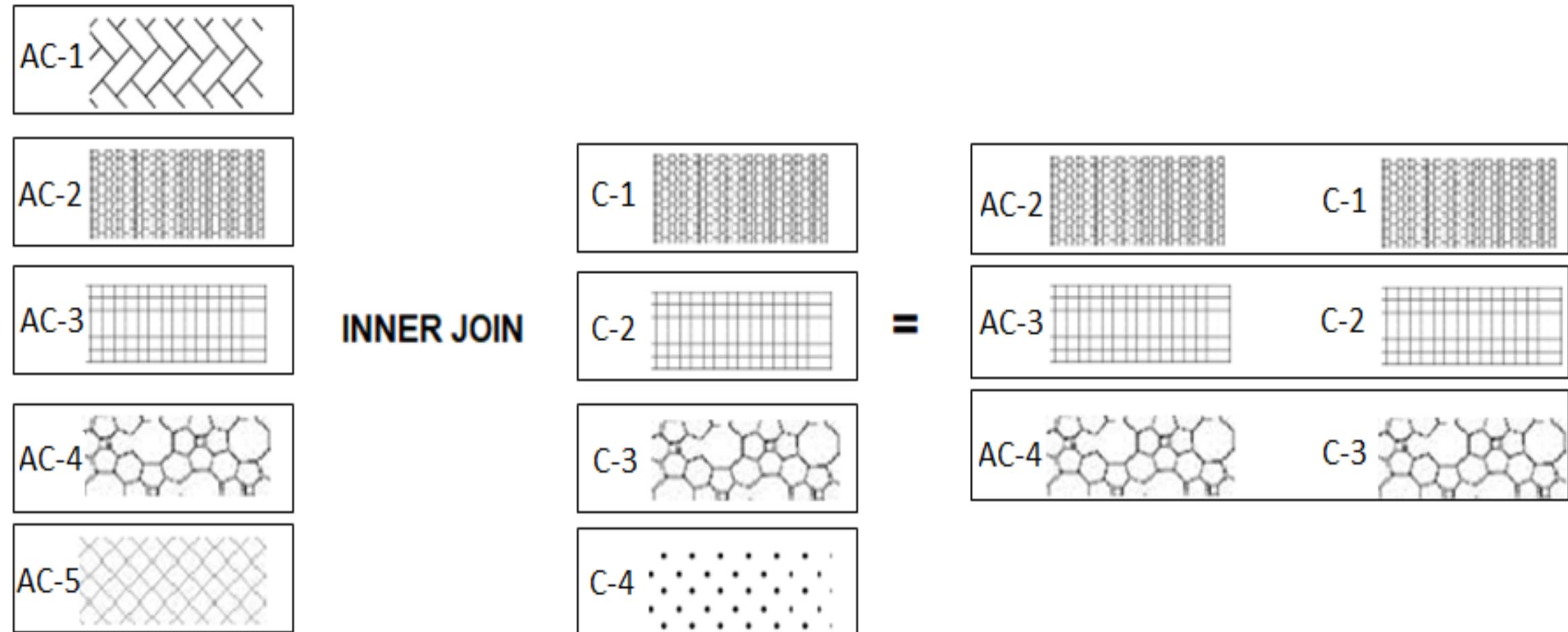
employeeID	Name	deptNo
1	Saleel	10 
2	Sharmin	20 
3	Vrushali	10 

deptNo	Name
10	Sales
20	Accounting
30	Manager

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  $r_1(\text{AC-1}, \text{AC-2}, \text{AC-3}, \text{AC-4}, \text{AC-5})$  and  $r_2(\text{C-1}, \text{C-2}, \text{C-3}, \text{C-4})$ . The result includes rows: (2,A), (3,B), and (4,C) as they have the same patterns.



## *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 <condition> AND <condition> AND ... AND <condition>, where each <condition> 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	7	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');`

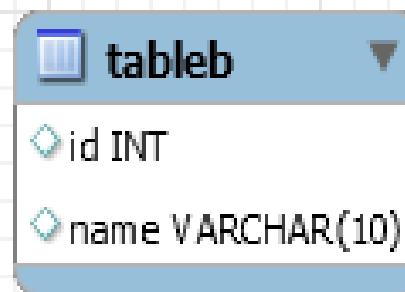
tableA Table

	id	name
▶	5	aa
	1	a
	2	b
	3	y
	NONE	d
	5	NONE
	1	NONE
	1	b
	8	a

tableB Table

	id	name
▶	1	a
	2	x
	4	b
	NONE	c
	6	NONE
	NONE	NONE
	7	z
	2	NONE
	5	z
	9	u

## joins – equi join



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

	id	name	id	name
▶	1	a	1	a
	1	NONE	1	a
	1	b	1	a
	2	b	2	x
	2	b	2	NONE
	5	aa	5	z
	5	NONE	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*

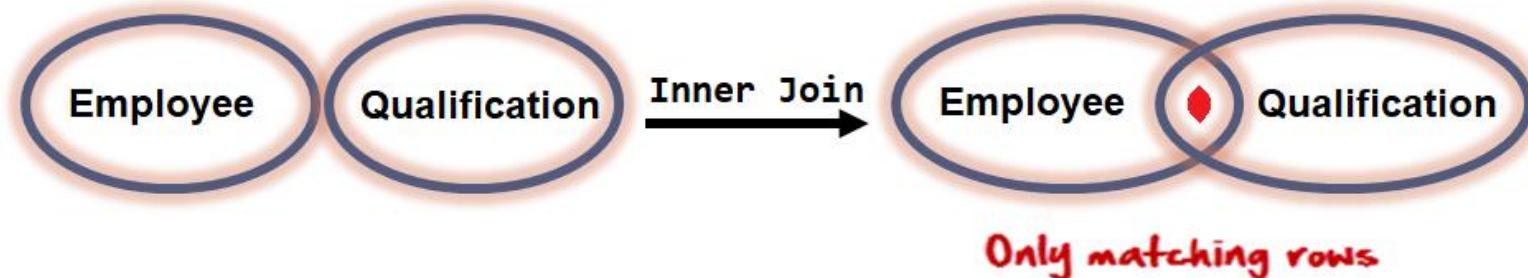
The INNER JOIN selects all rows from both participating tables as long as there is a match between the columns. An SQL INNER JOIN is same as JOIN clause, combining rows from two or more tables.

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

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

ID	FIRSTNAME	LASTNAME	GENDER	HIREDATE	ID	EMPLOYEEID	NAME	Stream	ADMISSIONYEAR	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	1965-01-10	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.

**SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1$  [INNER] JOIN  $r_2$  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	COMMPLAN	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 Pro
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 Hold
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	Canceled
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	Resolved
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	Disputed
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	Shipped
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 Progress
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	Shipped
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 Hold
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 Progress
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	Shipped
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	Canceled
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	Shipped
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 Progress
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 Progress
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	Resolved
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	Shipped
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 Hold
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 Progress
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	Canceled
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	Shipped

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

## *joins – natural join*

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

**SELECT  $A_1, A_2, A_3, \dots$  FROM  $r_1$  NATURAL [INNER] JOIN  $r_2$  NATURAL [INNER] JOIN  $r_3 \dots$**

- **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	NATURAL JOIN
Inner Join joins two table on the basis of the column which is explicitly specified in the ON clause.	Natural Join joins two tables based on same attribute name.
In Inner Join, The resulting table will contain all the attribute of both the tables including duplicate columns also	In Natural Join, The resulting table will contain all the attributes of both the tables but keep only one copy of each common column
In Inner Join, only those records will return which exists in both the tables	Same as Inner Join
<b>SYNTAX:</b> <ul style="list-style-type: none"><li>• <code>SELECT * FROM r<sub>1</sub> INNER JOIN r<sub>2</sub> ON r<sub>1</sub>.A<sub>1</sub> = r<sub>2</sub>.A<sub>1</sub>;</code></li><li>• <code>SELECT * FROM r<sub>1</sub> INNER JOIN r<sub>2</sub> USING(A<sub>1</sub>, [A<sub>2</sub>]);</code></li></ul>	<b>SYNTAX:</b> <ul style="list-style-type: none"><li>• <code>SELECT * FROM r<sub>1</sub> NATURAL JOIN r<sub>2</sub>;</code></li></ul>

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  $r_2$

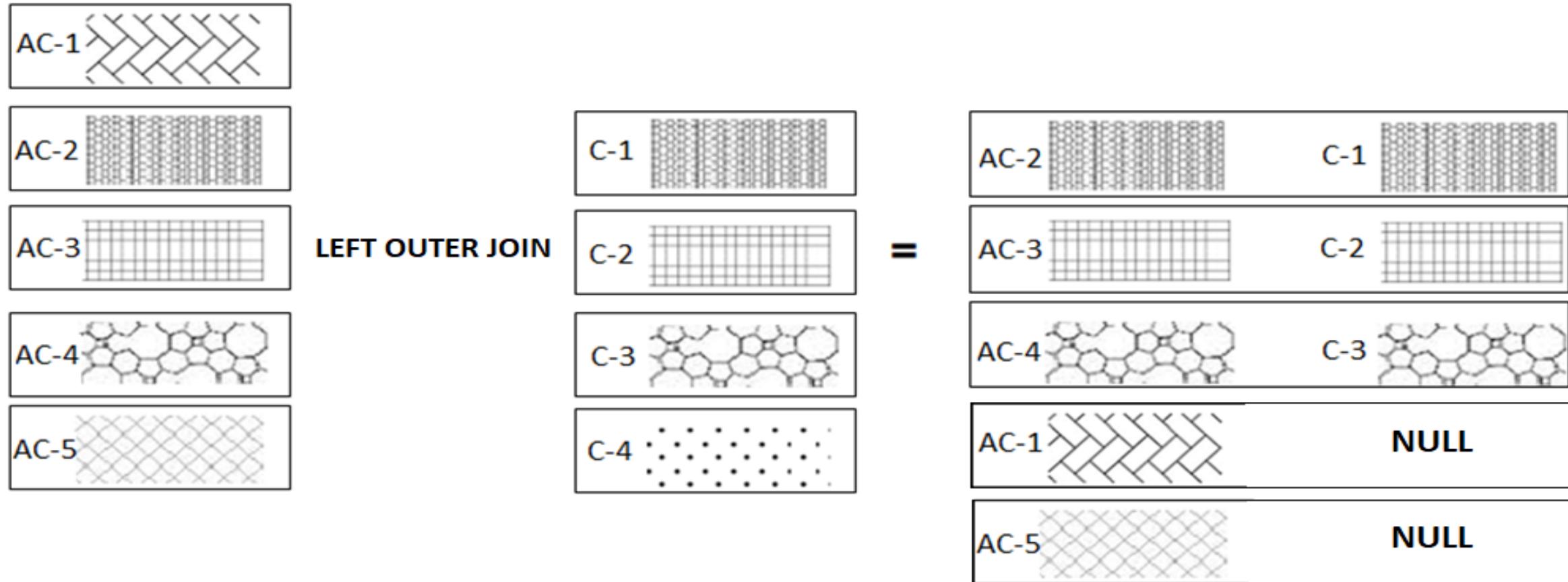
$$R = \{(1, 1, a), (2, 2, b), (1, 1, c), (3, 3, d), (2, 2, e), (1, 1, f), (4, \text{NULL}, \text{NULL})\}$$

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

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

## *joins – left outer join*

The following example shows the LEFT JOIN of two tables  $r_1(\text{AC-1}, \text{AC-2}, \text{AC-3}, \text{AC-4}, \text{AC-5})$  and  $r_2(\text{C-1}, \text{C-2}, \text{C-3}, \text{C-4})$ . 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 A1, A2, A3, ... FROM r1 LEFT [OUTER] JOIN r2 ON r1. A1 = r2. A1
```

```
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 A1, A2, A3, ... FROM r1 LEFT [OUTER] JOIN r2 ON r1. A1 = r2. A1 WHERE r2. A1 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  $r_2$

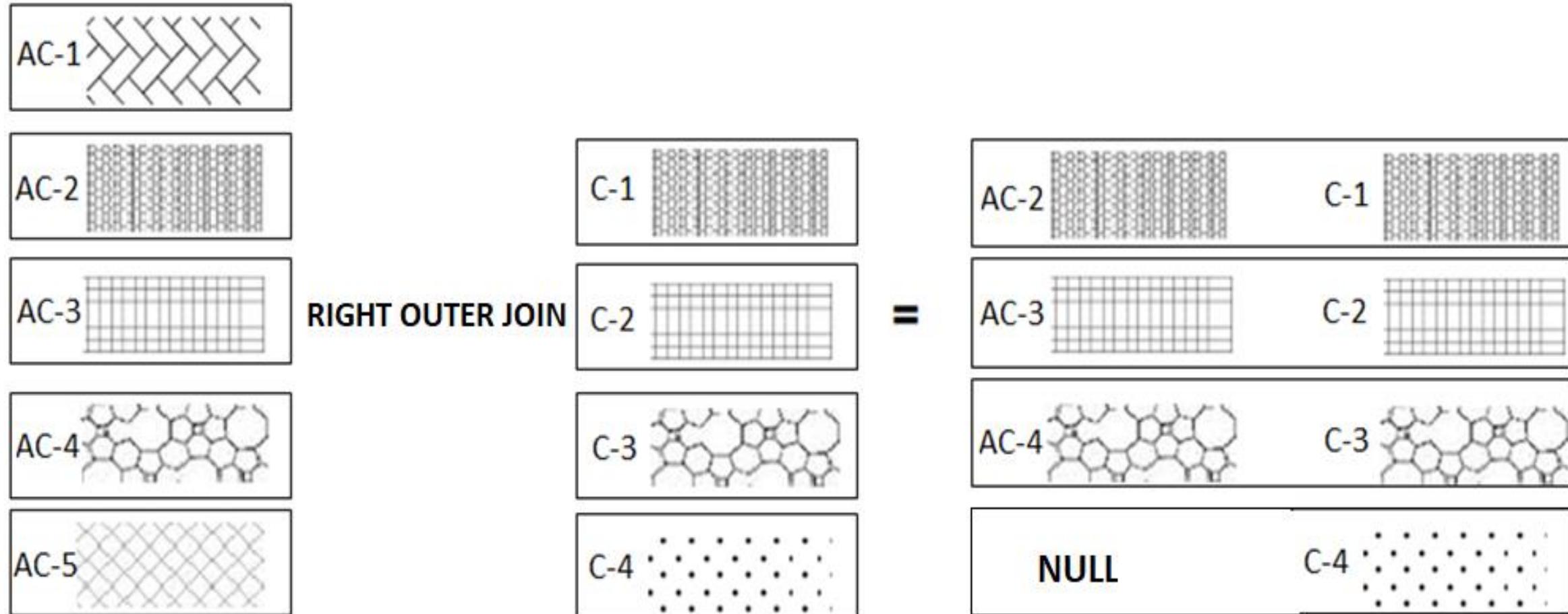
$$R = \{(1, 1, a), (1, 1, c), (1, 1, f), (2, 2, b), (2, 2, e), (3, 3, d), (\text{NULL}, 5, z)\}$$

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

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

## *joins – right outer join*

The following example shows the RIGHT OUTER JOIN of two tables  $r_1(\text{AC-1}, \text{AC-2}, \text{AC-3}, \text{AC-4}, \text{AC-5})$  and  $r_2(\text{C-1}, \text{C-2}, \text{C-3}, \text{C-4})$ . 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 A1, A2, A3, ... FROM r1 RIGHT [OUTER] JOIN r2 ON r1.A1 = r2.A1
```

```
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;`

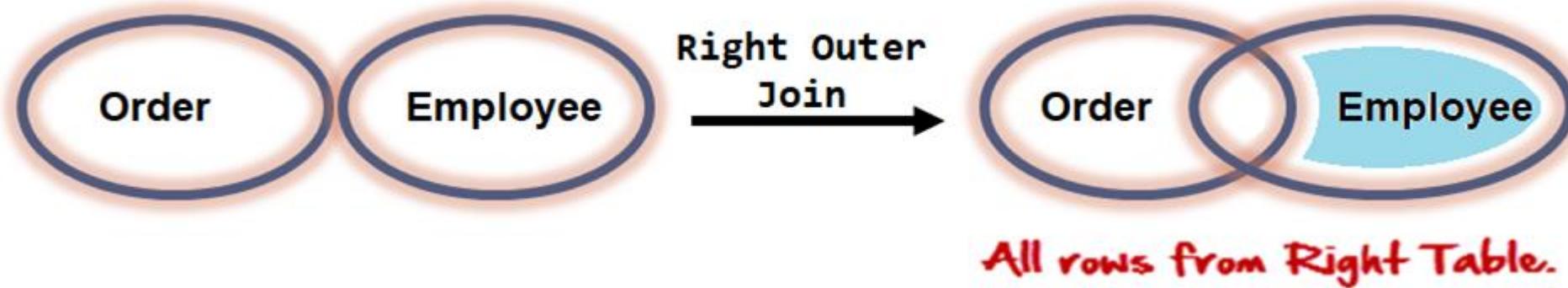
	ID	namefirst	namelast	DOB	emailID	ID	studentID	orderdate	amount
	6	lala	prasad	1980-12-01	lala.prasad@gmail.com	7	6	2019-11-11	4000
	1	saleel	bagde	1986-12-14	saleel.bagde@gmail.com	8	1	2019-07-19	1270
	5	ruhan	bagde	1984-01-12	ruhan.bagde@gmail.com	9	5	2019-04-07	2000
	7	sharmin	bagde	1986-12-14	sharmin.bagde@gmail.com	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
						15	NULL	2019-04-07	4000
						16	NULL	2019-10-10	1270
						17	NULL	2019-11-11	4588
						18	NULL	2019-07-21	1200
						19	NULL	2019-11-02	125
						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 A1, A2, A3, ... FROM r1 RIGHT [OUTER] JOIN r2 ON r1.A1 = r2.A1 WHERE r1.A1 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  $A_1, A_2, A_3, \dots$  FROM  $r_1$  RIGHT [OUTER] JOIN  $r_2$  USING ( $A_1, \dots$ )**

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

**SELECT  $A_1, A_2, A_3, \dots$  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.  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.	All the tuples of the right table remain in the result.  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.	All the tuples from left as well as right table remain in the result.  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 rx.A1, rx.A2, ry.A1, ry.A2, ... FROM r1 rx, r1 ry WHERE rx.A1 = ry.A1
```

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

```
SELECT C1 FROM T1 UNION SELECT C1 FROM T2  
ORDER BY C1
```

(UNION ALL, EXCEPT ALL, INTERSECT ALL)

```
(SELECT C1 FROM T1 ORDER BY C1) UNION  
(SELECT C1 FROM T2 ORDER BY C1)
```

```
SELECT C1 FROM T1 ORDER BY C1 UNION  
SELECT C1 FROM T2 ORDER BY C1 //ERROR
```

# set operation in sql

**Set operators** are used to join the results of two (or more) SELECT statements.

There are set union (**UNION**), set difference (**EXCEPT**), and set intersection (**INTERSECT**) operations. The relations resulting from these set operations are sets of tuples; that is, duplicate tuples are eliminated from the result.

## Remember:

- The result set column names are taken from the column names of the first SELECT statement.
- SELECT statement should have the same data type. (Not in MySQL)
- UNION: To apply ORDER BY or LIMIT to an individual SELECT, place the clause inside the parentheses that enclose the SELECT.

e.g. (SELECT ...) UNION (SELECT ...)

- `CREATE TABLE Facebook (`  
    `ID INT PRIMARY KEY,`  
    `name VARCHAR(45),`  
    `location VARBINARY(45)`  
`);`
- `CREATE TABLE LinkedIn (`  
    `ID INT PRIMARY KEY,`  
    `name VARCHAR(45),`  
    `location VARBINARY(45)`  
`);`

```
SELECT name FROM students  
UNION  
SELECT name FROM contacts;
```

*/\* Fetch the union of queries \*/*

```
SELECT name FROM students  
UNION ALL  
SELECT name FROM contacts;
```

*/\* Fetch the union of queries with duplicates \*/*

```
SELECT name FROM students  
EXCEPT  
SELECT name FROM contacts;
```

*/\* Fetch names from students \*/  
/\* that aren't present in contacts \*/*

```
SELECT name FROM students  
INTERSECT  
SELECT name FROM contacts;
```

*/\* Fetch names from students \*/  
/\* that are present in contacts as well \*/*

# union

## syntax

```
SELECT ... UNION [ALL | DISTINCT]
SELECT ... [UNION [ALL | DISTINCT]
SELECT ...]
```

- `SELECT DISTINCT * FROM duplicate;`
- `SELECT * FROM duplicate UNION SELECT * FROM duplicate;`
- `SELECT deptno, dname, loc, walletid FROM (SELECT ROW_NUMBER()
OVER(PARTITION BY deptno) R1, duplicate.* FROM duplicate) T1 WHERE R1=1;`

## Note:

- It is used to combine two or more result sets (SELECT statements) into a single set
  - it removes duplicate rows between the various SELECT statements.
  - each SELECT statement within the UNION operator must have the same number of fields in the result sets.
  - default behaviour for UNION is that duplicate rows are removed from the result.
- 
- `(SELECT deptno FROM emp LIMIT 1) UNION (SELECT deptno FROM dept LIMIT 1);`
  - `SELECT 'EMP' as 'Table Name', COUNT(*) FROM emp UNION SELECT 'DEPT', COUNT(*) FROM dept UNION SELECT 'BONUS', COUNT(*) FROM bonus;`
  - `SELECT COUNT(*) FROM customer UNION SELECT COUNT(*) FROM ord;`
  - `SELECT * FROM emp WHERE deptno NOT IN (SELECT deptno FROM emp m WHERE m.deptno NOT IN (SELECT deptno FROM emp f WHERE gender = 'F')) UNION SELECT deptno FROM emp m WHERE m.deptno NOT IN (SELECT deptno FROM emp f WHERE gender = 'M'));`

*union*

books

	bookID	bookName	Type	Cost
▶	1	DS	Hardcover	950
	1	DS	Hardcover	950
	2	JavaScript	Paperback	700
	2	JavaScript	Paperback	700

newbooks

	bookID	bookName	Type	Cost
▶	1	Redis	Paperback	850
	1	Redis	Paperback	850
	2	JavaScript	Paperback	700
	2	JavaScript	Paperback	700

Output

	bookID	bookName	Type	Cost
▶	1	DS	Hardcover	950
	2	JavaScript	Paperback	700
	1	Redis	Paperback	850

*Duplicate rows  
not repeated  
in result set*

- `SELECT * FROM books UNION SELECT * FROM newbooks;`

Note:

The following statement will give an error

- `SELECT bookName, type FROM books ORDER BY bookname  
UNION  
SELECT bookName, type FROM newbooks;`

*union all*

books

	bookID	bookName	Type	Cost
▶	1	DS	Hardcover	950
	1	DS	Hardcover	950
	2	JavaScript	Paperback	700
	2	JavaScript	Paperback	700

newbooks

	bookID	bookName	Type	Cost
▶	1	Redis	Paperback	850
	1	Redis	Paperback	850
	2	JavaScript	Paperback	700
	2	JavaScript	Paperback	700

Output

	bookID	bookName	Type	Cost
▶	1	DS	Hardcover	950
	1	DS	Hardcover	950
	2	JavaScript	Paperback	700
	2	JavaScript	Paperback	700
	1	Redis	Paperback	850
	1	Redis	Paperback	850
	2	JavaScript	Paperback	700
	2	JavaScript	Paperback	700

### Note:

- It is used to combine two or more result sets into a single set, including duplicates.
- `SELECT * FROM books UNION ALL SELECT * FROM newbooks;`
- `SELECT * FROM emp UNION ALL SELECT * FROM emp;`
- `SELECT bookname, COUNT(*) FROM (SELECT bookname FROM books UNION ALL SELECT bookname FROM newbooks) b GROUP BY bookname;`

Duplicate rows  
are repeated  
in result set

It is used to combine two result sets and returns the data which are common in both the result set.

## *intersect / intersect all*

books

	bookID	bookName	Type	Cost
▶	1	DS	Hardcover	950
	1	DS	Hardcover	950
	2	JavaScript	Paperback	700
	2	JavaScript	Paperback	700

newbooks

	bookID	bookName	Type	Cost
▶	1	Redis	Paperback	850
	1	Redis	Paperback	850
	2	JavaScript	Paperback	700
	2	JavaScript	Paperback	700

Output

	bookID	bookName	Type	Cost
▶	2	JavaScript	Paperback	700

Output

	bookID	bookName	Type	Cost
▶	2	JavaScript	Paperback	700
	2	JavaScript	Paperback	700

An INTERSECT query returns the intersection of 2 or more data sets. If a record exists in both data sets, it will be included in the INTERSECT results.

- `SELECT * FROM books INTERSECT SELECT * FROM newbooks;`
- `SELECT * FROM books INTERSECT ALL SELECT * FROM newbooks;`
- `SELECT bookName, type FROM books WHERE EXISTS ( SELECT bookname, type FROM newbooks WHERE books.bookName = newbooks.bookName);`
- `SELECT bookName, type FROM books WHERE bookName IN (SELECT bookName FROM newbooks);`

It is used to combine two result sets and returns the data from the first result set which is not present in the second result set.

*except / except all*

books

	bookID	bookName	Type	Cost
▶	1	DS	Hardcover	950
	1	DS	Hardcover	950
	2	JavaScript	Paperback	700
	2	JavaScript	Paperback	700

newbooks

	bookID	bookName	Type	Cost
▶	1	Redis	Paperback	850
	1	Redis	Paperback	850
	2	JavaScript	Paperback	700
	2	JavaScript	Paperback	700

Output

	bookID	bookName	Type	Cost
▶	1	DS	Hardcover	950

Output

	bookID	bookName	Type	Cost
▶	1	DS	Hardcover	950
	1	DS	Hardcover	950

An EXCEPT returns rows from first dataset, which are not available in the second dataset.

- `SELECT * FROM books EXCEPT SELECT * FROM newbooks;`
- `SELECT * FROM books EXCEPT ALL SELECT * FROM newbooks;`
  
- `SELECT bookName, type FROM books WHERE NOT EXISTS (SELECT bookName, type FROM newbooks WHERE books.bookName = newbooks.bookName);`
- `SELECT bookName, type FROM newbooks WHERE NOT EXISTS (SELECT bookName, type FROM books WHERE books.bookName = newbooks.bookName);`

Note:

*except*

- There is no **MINUS** operator in MySQL, you can easily simulate this type of query using either the **EXCEPT** , **NOT IN** clause or the **NOT EXISTS** clause.

1. `SELECT * FROM books /* Fetch everything from books */  
 EXCEPT /* that are not present in newbooks */  
 SELECT * FROM newbooks;`

2. `SELECT * FROM newbooks /* Fetch everything from newbooks */  
 EXCEPT /* that are not present in books */  
 SELECT * FROM books;`

create table ... like statement

## *create table ... like*

Use CREATE TABLE ... LIKE to create an empty table based on the definition of another table, including any column attributes and indexes defined in the original table.

```
CREATE TABLE [IF NOT EXISTS] new_tbl LIKE orig_tbl;
```

- LIKE works only for base tables, not for VIEWS.
- You cannot execute CREATE TABLE or CREATE TABLE ... LIKE while a LOCK TABLES statement is in effect.
- `CREATE TABLE e LIKE emp;`

MEMORY tables are visible to another client, but  
TEMPORARY tables are not visible to another client.

*create temporary table ... like  
statement*

## *create temporary table ... like*

Use CREATE TABLE ... LIKE to create an empty table based on the definition of another table.

```
CREATE TEMPORARY TABLE [IF NOT EXISTS] new_tbl LIKE orig_tbl;
```

- LIKE works only for base tables, not for VIEWS.
- You cannot execute CREATE TABLE or CREATE TABLE ... LIKE while a LOCK TABLES statement is in effect.

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.

```
CREATE TEMPORARY TABLE e like emp;
```

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.

create table ... select statement

## *create table ... select*

You can create one table from another by adding a SELECT statement at the end of the CREATE TABLE statement.

```
CREATE TABLE new_tbl [AS] SELECT * FROM orig_tbl;
```

You cannot use FOR UPDATE as part of the SELECT in a statement such as CREATE TABLE new\_table  
SELECT ... FROM old\_table .... FOR UPDATE.

If you attempt to do so, the statement fails.

- CREATE TABLE e **SELECT \* FROM emp;**
- CREATE TABLE e as **SELECT \* FROM emp;**
- CREATE TABLE e (ID INT) **SELECT \* FROM emp;**
- CREATE TABLE e as **SELECT 1+1, ename FROM emp;**
- CREATE TABLE e as **SELECT 1+1 "R1", ename FROM emp;**
- CREATE TABLE e as **SELECT \* FROM emp WHERE 1=2;**

### Note:

- By default, this statement does not copy all column attributes such as AUTO\_INCREMENT

- `SET autocommit = { 0 | 1 }` 0 - disable; 1 - enable
  - show variables like 'auto%';
  - `SELECT @@AUTOCOMMIT;`

Turning the autocommit off globally. Open **my.ini** file and add the following statement.

[mysqld]

`autocommit = 0`

After adding the line re-start the MySQL server.

## commit and rollback

Remember:

- When you **UPDATE** or **DELETE** a row, that row is automatically locked for other users.
- ROW LOCKING IS AUTOMATIC IN MYSQL AND ORACLE.
- When you **UPDATE** or **DELETE** a row, that row becomes READ ONLY for other users.
- Other users can **SELECT** from that table; they will view the old data before your changes.
- Other users can **INSERT** rows into that table.
- Other users can **UPDATE** or **DELETE** "other" rows of that table.
- No other user can **UPDATE** or **DELETE** your locked row, till you have issued a Rollback or Commit.
- To lock the rows manually you require **SELECT** statement with a **FOR UPDATE** clause.
- **LOCKS ARE AUTOMATICALLY RELEASED WHEN YOU ROLLBACK OR COMMIT.**

# *commit and rollback*

To undo MySQL statements you use the ROLLBACK statement. To write the changes into the database within a transaction, you use the COMMIT statement.

START TRANSACTION

[transaction\_characteristic]

transaction\_characteristic:

| READ WRITE  
| READ ONLY

## Note:

- It is important to note that MySQL automatically commits the changes to the database by default.
- START TRANSACTION READ ONLY;
- INSERT INTO dept VALUES (60, 'HRD', 'Pune' );
- COMMIT

## *commit and rollback*

To undo MySQL statements you use the ROLLBACK statement. To write the changes into the database within a transaction, you use the COMMIT statement.

BEGIN [WORK]  
COMMIT [WORK]  
SAVEPOINT *identifier*  
ROLLBACK [WORK] TO [SAVEPOINT] *identifier*  
SET autocommit = {0 | 1}

### Note:

- It is important to note that MySQL automatically commits the changes to the database by default.
  - BEGIN WORK
  - INSERT INTO dept VALUES (60, 'HRD', 'Pune' );
  - ROLLBACK / COMMIT WORK
- 
- SAVEPOINT pt0;
  - INSERT INTO dept VALUES (60, 'HRD', 'Pune' );
  - SAVEPOINT pt1;
  - INSERT INTO dept VALUES (61, 'HRD', 'baroda' );
  - ROLLBACK TO SAVEPOINT pt1;

select .... for update

## *select - for update*

Sets a shared mode lock on any rows that are read. Other sessions can read the rows, but cannot modify them until your transaction commits. If any of these rows were changed by another transaction that has not yet committed, your query waits until that transaction ends and then uses the latest values.

- `SELECT * FROM emp FOR UPDATE;`

# 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 retains 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*

<b>DELETE</b>	<b>TRUNCATE</b>
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 deleting 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*

<b>DROP</b>	<b>TRUNCATE</b>
The DROP command is used to remove table definition and its contents.	Whereas the TRUNCATE command is used to delete all the rows from the table.
In the DROP command, VIEW of table does not exist.	In the TRUNCATE command, VIEW of table exist.
In the DROP command, integrity constraints will be removed.	In the TRUNCATE command, integrity constraints will not be removed.
In the DROP command, INDEX associated to the table will be removed.	In the TRUNCATE command, INDEX associated to the table will not be removed.
In the DROP command, TRIGGER associated to the table will execute.	In the TRUNCATE command, TRIGGER associated to the table will not execute.
In the DROP command, TRIGGER associated to the table will be removed.	In the TRUNCATE command, TRIGGER associated to the table will be not removed.

# rename table

Change the table name.

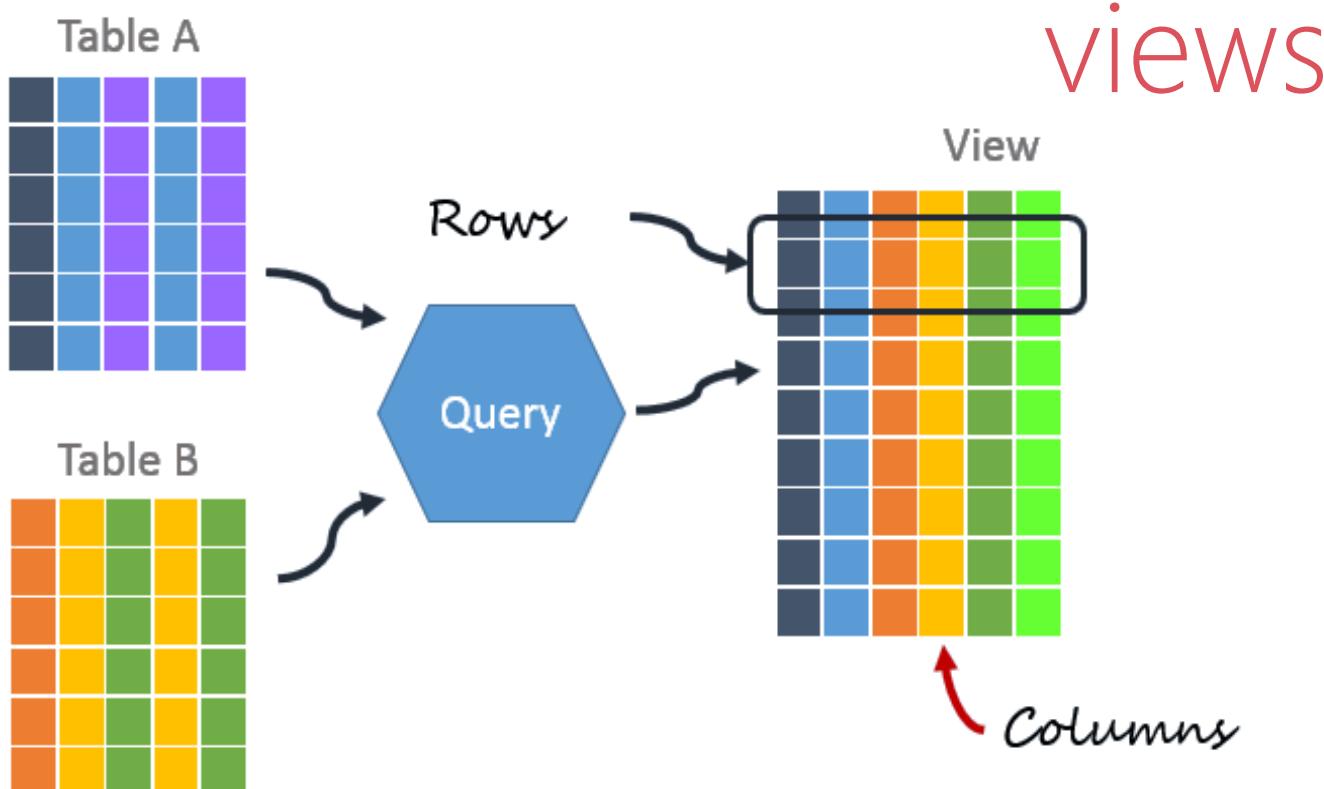
`RENAME TABLE old_tbl_name TO new_tbl_name`

- `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.



## Theater

ID	Name	Address
1	A	B
2	C	D
3	E	F
4	G	H

## Opera

ID	Name	Address
1	I	J
2	K	L
3	M	N
4	O	P

## Cinema

ID	Name	Address
1	R	S
2	T	U
3	V	W
4	X	Y

view

## Executed View

ID	Name	Address
1	A	B
2	C	D
3	E	F
4	G	H
5	I	J
6	K	L
7	M	N
8	O	P
9	R	S
10	T	U
11	V	W
12	X	Y

SELECT ID, Name, Address **FROM** theater

**UNION**

SELECT ID, Name, Address **FROM** opera

**UNION**

SELECT ID, Name, Address **FROM** cinema;

## Remember:

- 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).

view

## 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.
  - `DESC dept;`
  - `CREATE VIEW v1 AS SELECT * FROM dept;`
  - `DROP TABLE dept;`
  - `DESC v1;`

## 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.

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(A<sub>1</sub>, A<sub>2</sub>) 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	RQFY8GZNDF9E	*****	*****	1
	2	2	LMV	Light motor vehicles including motorcars, jeeps, taxis, delivery vans	ZFVHYH40AHGS	*****	*****	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	2EFFSXY040AW	*****	*****	3
	8	5	HMV	Heavy Motor Vehicles	GC1BMW KCTVQ4	*****	*****	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
*								

# 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;

create user and grant/revoke  
privileges

## *create user*

TODO

`CREATE USER [IF NOT EXISTS] 'user_name' @'localhost' IDENTIFIED BY 'password'`

`SET PASSWORD FOR 'user_name'@'localhost' = 'auth_string'`

`DROP USER [IF EXISTS] 'user_name' @'localhost'`

- `CREATE USER 'saleel'@'localhost' IDENTIFIED BY 'saleel';`
- `SET PASSWORD FOR 'saleel'@'localhost' = 'sharmin';`
- `DROP USER 'saleel'@'localhost';`

# *grant/revoke privileges*

TODO

`GRANT priv_type ON object_type TO 'user_name'@'localhost'`

`REVOKE priv_type ON object_type FROM 'user_name'@'localhost'`

Privilege	Privilege
ALL [PRIVILEGES]	SELECT
CREATE	INSERT
ALTER	UPDATE
DROP	DELETE
EXECUTE	

- `GRANT ALL PRIVILEGES ON db1.* TO 'saleel'@'localhost';`
- `GRANT ALL PRIVILEGES ON *.* TO 'saleel'@'localhost';`
- `GRANT INSERT, UPDATE ON emp TO 'saleel';`
- `REVOKE INSERT, UPDATE ON emp FROM 'saleel';`

Try Out

*create view*

```
CREATE TABLE customer (
    ID INT PRIMARY KEY ,
    name VARCHAR(50),
    description TEXT
);
```

```
CREATE TABLE orders (
    orderID INT ,
    customerID INT,
    orderDate DATE,
    amount INT,
    constraint fk_customerID FOREIGN KEY(customerID) REFERENCES customer(ID));
```

```
INSERT INTO customer VALUES (1, 'saleel', 'description Text 1'), (2, 'vrushali', 'description Text 2'), (3, 'sharmin', 'description Text 3'), (4, 'ruhan', 'description Text 4');
```

```
INSERT INTO orders VALUES(1, 1, NOW(), 3000), (2, 1, NOW(), 1000), (3, 2, NOW(), 100), (4, 2, NOW(), 300), (5, 2, NOW(), 200);
```

- CREATE or REPLACE VIEW v1 AS SELECT \* FROM customer, orders WHERE ID = customerID;
  - CREATE or REPLACE VIEW v2 AS SELECT customer.\* FROM customer, orders WHERE ID = customerID;
  - CREATE or REPLACE VIEW v3 AS SELECT orders.\* FROM customer, orders WHERE ID = customerID;
- // We may get Error: Can not insert into join view 'db5.v1'
- ```
• INSERT INTO v1(ID, name, description, orderID, customerID, orderdate, amount) VALUES (10,'S10','D1', 10, 1, NOW(), 111); // Error: Can not modify more than one base table through a join view 'db5.v1'
```
- // We may get Error: Can not insert into join view 'db5.v2' without fields list.
- INSERT INTO v1(ID, name, description) VALUES(1, 'neel', 'description Text 5');
  - INSERT INTO v1(orderID, customerID, orderDate, amount) VALUES(1, 1, now(), 7000);
  - INSERT INTO v2(ID, name, description) VALUES(5, 'neel', 'description Text 5');
  - INSERT INTO v3(orderID, customerID, orderdate, amount) VALUES(6, 1, NOW(), 200);

# SHOW CREATE VIEW Syntax

`SHOW CREATE VIEW view_name`

`show create VIEW v1;`

`SHOW [FULL] TABLES [ { FROM | IN } db_name ]  
[ LIKE 'pattern' | WHERE expr ]`

`show full tables where table_type like 'VIEW';`

## *alter / drop view*

This statement changes the definition of a view, which must exist.

```
ALTER VIEW view_name [(column_list)]
    AS select_statement
    [WITH CHECK OPTION]
```

e.g.

- `ALTER VIEW studentview AS SELECT namefirst, namelast, emailid FROM student;`

`DROP VIEW` removes one or more views.

```
DROP VIEW [IF EXISTS]
    view_name [, view_name, ...]
```

e.g.

- `DROP VIEW studentview;`
- `DROP VIEW studentid10view, studentviewwithcheck;`
- `DROP VIEW studentTotalMarksView, studentAddressView;`

MySQL indexes (PRIMARY KEY, UNIQUE, INDEX, and FULLTEXT) are stored in B-trees.

# index

- it is a schema object.
- it is used by the server to speed up the retrieval of rows.
- it reduces disk I/O(input/output) process.
- it helps to speed up select queries where clauses, but it slows down data input, with the update and the insert statements.
- it can be created or dropped with no effect on the data.
- multiple columns index may consist of up to 16 columns.

Indexes are used to find rows with specific column values quickly. Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows. If the table has an index for the columns in question, MySQL can quickly determine the position (**from the index file**) to seek to in the middle of the data file without having to look at all the data. This is much faster than reading every row sequentially.

- To find the rows matching a WHERE clause quickly.
- If the table has a multiple-column index, any leftmost prefix of the index can be used by the optimizer to look up rows. For example, if you have a three-column index on (col1, col2, col3), you have indexed search capabilities on (col1), (col1, col2), and (col1, col2, col3).

## Note:

- Index name can have max 64 char.
- It is not possible to create an INDEX on a VIEW.
- If we drop the BASE TABLE, the INDEX will be dropped automatically.

## *create index*

Indexes are used to find rows with specific column values quickly.

```
CREATE [UNIQUE] INDEX index_name  
ON tbl_name (index_col_name, ...)
```

e.g.

- `CREATE INDEX indexOnName ON emp(ename);`
- `CREATE INDEX indexOnUniversity ON student_qualifications(university);`
- `CREATE UNIQUE INDEX uniqueIndexOnName ON emp(ename);`

# *clustered and non-clustered index*

Indexing in MySQL is a process that helps us to return the requested data from the table very fast. If the table does not have an index, it scans the whole table for the requested data.

MySQL allows two different types of Indexing:

- [Clustered Index](#)
- [Non-Clustered Index](#)

**Clustered Index:-**

- Clustered index is used to optimize the speed of most common lookups and DML operations like INSERT, UPDATE, and DELETE command.
- Clustered indexes sort and store the data rows in the table based on their key (primary key) values that can be sorted in only one direction.
- If the table column contains a **primary key** or **unique key**, MySQL creates a clustered index.
- Data retrieval is faster than non-cluster index

**Non-Clustered Index:-**

- The indexes other than PRIMARY indexes (clustered indexes) called a non-clustered index.
- The non-clustered indexes are also known as secondary indexes.
- The non-clustered index and table data are both stored in different places.
- It is not sorted (ordering) the table data.
- Extra space is required to store logical structure
- Data update is faster than clustered index

# SHOW INDEX Syntax

To get the index of a table, you specify the table name after the FROM keyword. The statement will return the index information associated with the table in the current database.

```
SHOW { INDEX | INDEXES | KEYS }
```

```
  { FROM | IN } tbl_name
```

```
  [{ FROM | IN } db_name]
```

```
  [WHERE expr]
```

e.g.

- SHOW INDEX FROM emp;
- SHOW INDEX FROM student\_qualifications;

SHOW INDEX returns table index information.

# *drop index*

DROP INDEX drops the index named index\_name from the table tbl\_name.

**DROP INDEX index\_name ON tbl\_name**

e.g.

- **DROP INDEX indexOnName ON emp;**
- **DROP INDEX indexOnUniversity ON student\_qualifications;**
- **DROP INDEX uniqueIndexOnName ON emp;**

- UPDATE j SET doc = JSON\_SET(doc, '\$.ename', 'sharmin');

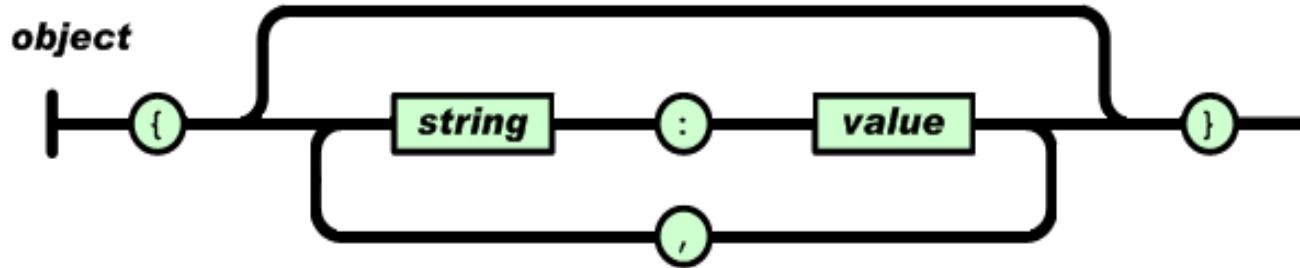
json

# *datatype – json*

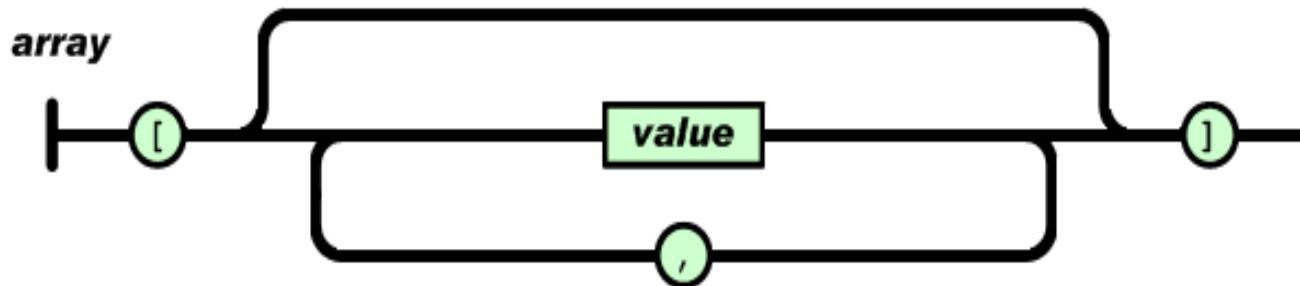
JSON (JavaScript Object Notation) documents.

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

An object is an unordered set of name/value pairs.



An array is an ordered collection of values.



# *dml- insert json object... values*

INSERT inserts new rows into an existing table. The INSERT ... VALUES

```
JSON_OBJECT("key1", 1, "key2", "abc", "key3", JSON_ARRAY( value1, "value2", ... ), ... );  
'{ "key1" : 1, "key2" : "abc" , "key3" : [ value1, "value2" , ... ] , ... }'
```

```
CREATE TABLE customer (  
    ID INT PRIMARY KEY AUTO_INCREMENT,  
    customer_details VARCHAR(1000),  
    order_details JSON );
```

e.g.

- `INSERT INTO customer VALUES(default,'{"CID": 1001,"name":"saleel", "city":"pune"}', JSON_OBJECT("orderID", 1, "item", "computer", "qty", 1, "rate", 75000));`
- `INSERT INTO customer VALUES(default,'{"CID": 1002,"name":"saleel", "city": "pune", "phone": [11111, 22222] }', JSON_OBJECT("orderID", 1, "item", "computer", "qty", 1, "rate", 75000));`
- `INSERT INTO customer VALUES(default,'{"CID": 1003,"name":"sharmin", "city": "pune"}', JSON_OBJECT("orderID", 1, "item", JSON_ARRAY("maggi", "burger", "coffee"), "qty", JSON_ARRAY(4, 2, 2), "rate", JSON_ARRAY(45, 375, 125)));`
- `INSERT INTO customer VALUES(default, JSON_OBJECT("CID", 1004, "name", "ruhan", "city", "Pune", "phone", JSON_ARRAY(1111, 2222)), JSON_ARRAY(JSON_OBJECT("orderID", 1, "item", "mouse", "qty", 2, "rate", 1200 ), JSON_OBJECT("orderID", 2, "item", "pd", "qty", 1, "rate", 1600)));`

# json\_extract

Returns data from a JSON document, selected from the parts of the document matched by the path arguments.

## *json\_extract (select)*

MySQL supports a native JSON data type that enables efficient access to data in JSON (JavaScript Object Notation) documents.

`JSON_EXTRACT(json_doc, path[, path] . . .`

- `SELECT JSON_EXTRACT(document, '$.*') FROM empj;`
- `SELECT JSON_EXTRACT(document, '$.empID') FROM empj;`
- `SELECT JSON_EXTRACT(document, '$.phone[1]') FROM empj;`
- `SELECT JSON_EXTRACT(document, "$.address.street") FROM empj;`
- `SELECT JSON_EXTRACT(document, "$.empID", "$.ename") FROM empj;`

-> / ->> ... (*select*)

MySQL supports a native JSON data type that enables efficient access to data in JSON (JavaScript Object Notation) documents.

- `SELECT id, document->'$.ename', document->'$.sal' FROM empJ;`
- `SELECT id, document->'$.ename', document->'$.sal', document->'$.comm' FROM empJ;`
- `SELECT id, document->'$.ename', document->'$.sal', document->'$.comm', document->'$.sal' + document->'$.comm' FROM empJ;`
- `SELECT id, document->'$.address.coord[0]' FROM empJ;`
- `SELECT id, document->'$.phone[0]' FROM empJ;`
- `SELECT author_details->>'$. authorName', book_details->'$[1].*' FROM author_book;`
- `SELECT id, document->"$.phone", JSON_LENGTH(document,("$.phone")) FROM empJ;`

## *json\_pretty ... (select)*

Print a JSON document in human-readable format

- `SELECT oid, JSON_PRETTY(orderheader) FROM orderj;`
- `SELECT oid, JSON_PRETTY(order_items) FROM orderitemsj;`
- `SELECT orderj.oid, JSON_PRETTY(orderheader), JSON_PRETTY(order_items) FROM orderj, orderitemsj WHERE orderj.oid = orderitemsj.oid;`

Note:

- TODO

# json functions

TODO

# *json function*

| Syntax                                                         | Result                                                                                                    |
|----------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| <code>-&gt;</code>                                             | Return value from JSON column after evaluating path; equivalent to <code>JSON_EXTRACT()</code> .          |
| <code>-&gt;&gt; / JSON_UNQUOTE(json_val)</code>                | Return value from JSON column after evaluating path and unquoting the result.                             |
| <code>JSON_EXTRACT(json_doc, path[, path] ...)</code>          | Returns data from a JSON document, selected from the parts of the document matched by the path arguments. |
| <code>JSON_ARRAY([val[, val] ...])</code>                      | Evaluates a list of values and returns a JSON array containing those values.                              |
| <code>JSON_OBJECT([key, val[, key, val] ...])</code>           | Create JSON object.                                                                                       |
| <code>JSON_INSERT(json_doc, path, val[, path, val] ...)</code> | INSERT a key, if not present [ with UPDATE ]                                                              |
| <code>JSON_PRETTY(json_val)</code>                             | Provides pretty-printing of JSON values                                                                   |

# *json function*

| Syntax                                                               | Result                                          |
|----------------------------------------------------------------------|-------------------------------------------------|
| <code>JSON_REPLACE(json_doc, path, val[, path, val] ...)</code>      | Replace values in JSON document [ with UPDATE ] |
| <code>JSON_SET(json_doc, path, val[, path, val] ...)</code>          | Insert data into JSON document [ with UPDATE ]  |
| <code>JSON_REMOVE(json_doc, path[, path] ...)</code>                 | Remove data from JSON document [ with UPDATE ]  |
| <code>JSON_KEYS(json_doc[, path])</code>                             | Array of keys from JSON document                |
| <code>JSON_ARRAY_APPEND(json_doc, path, val[, path, val] ...)</code> | Append data to JSON document [ with UPDATE ]    |
| <code>JSON_LENGTH(json_doc[, path])</code>                           | Number of elements in JSON document             |

## *dml- update/remove json object... values*

Inserts or Updates data in a JSON document and returns the result.

`JSON_SET(json_doc, path, value1[, path, value2], ...)`

`JSON_REMOVE(json_doc, path[, path], ...)`

```
CREATE TABLE customer (
    ID INT PRIMARY KEY AUTO_INCREMENT,
    customer_details VARCHAR(1000),
    order_details JSON
);
```

e.g.

- `UPDATE customer SET order_details = JSON_SET(order_details, '$.state', 'MH');`
- `UPDATE customer SET order_details = JSON_SET(order_details, '$.city', 'baroda') WHERE id = 1;`
- `UPDATE customer SET order_details = JSON_REMOVE(order_details, "$. state") WHERE id = 1;`

e.g.

## pk/fk in json

- `CREATE TABLE orderJ(oid INT PRIMARY KEY, orderheader JSON);`
- `INSERT INTO orderJ VALUES(1, JSON_OBJECT('customerID', 1001, 'customerName', 'saleel', 'mobile', 9850884228, 'city', 'baroda'));`
- `INSERT INTO orderJ VALUES(2, JSON_OBJECT('customerID', 1002, 'customerName', 'sharmin', 'mobile', 9850884229, 'city', 'pune'));`
- `INSERT INTO orderJ VALUES(3, JSON_OBJECT('customerID', 1003, 'customerName', 'vrushali', 'mobile', 9850884230, 'city', 'pune'));`
- `INSERT INTO orderj VALUES(4, JSON_OBJECT('customerID', 1004, 'customerName', 'ruhan', 'mobile', JSON_ARRAY(11111111, 22222222), 'city', 'pune'));`

- 
- `CREATE TABLE orderItemsJ(oid INT, order_items JSON,  
FOREIGN KEY(oid) REFERENCES orderJ(oid));`
  - `INSERT INTO orderItemsJ VALUES(1, JSON_OBJECT('itemID', 1001, 'itemName', 'butter', 'rate', 175, 'qty', 5));`
  - `INSERT INTO orderItemsJ VALUES(1, JSON_OBJECT('itemID', 1002, 'itemName', 'cheese', 'rate', 225, 'qty', 2));`
  - `INSERT INTO orderItemsJ VALUES(2, JSON_ARRAY(JSON_OBJECT('itemID', 1001, 'itemName', 'butter', 'rate', 175, 'qty', 2),  
JSON_OBJECT('itemID', 1003, 'itemName', 'bread', 'rate', 45, 'qty', 2), JSON_OBJECT('itemID', 1002, 'itemName', 'cheese', 'rate', 225, 'qty', 4)));`

lock / unlock table

# *lock / unlock*

MySQL enables client sessions to acquire table locks to prevent from modifying tables

Sessions holding a READ lock, DROP TABLE and TRUNCATE TABLE operations are not permitted.

```
LOCK TABLES tbl_name lock_type [,tbl_name]  
lock_type] ...
```

`lock_type:`

READ | WRITE

```
UNLOCK TABLES
```

- `LOCK TABLE DEPT READ;`
- `LOCK TABLE DEPT WRITE;`
- `UNLOCK TABLE`

select ... into

An INTO clause should not be used in a nested SELECT because such a SELECT must return its result to the outer context.

## *select ... into*

The SELECT ... INTO form of SELECT enables a query result to be stored in variables or written to a file.

`SELECT ... INTO var_list`

`SELECT ... INTO OUTFILE`

`SELECT ... INTO DUMPFILE`

- Selects column values and stores them into variables.
- Writes the selected rows to a file. Column and line terminators can be specified to produce a specific output format.
- Writes a single row to a file without any formatting.

select ... into var\_list

## *select ... into var\_list*

The SELECT ... INTO form of SELECT enables a query result to be stored in variables or written to a file.

### `SELECT ... INTO var_list`

- The selected values are assigned to the variables.
- The number of variables must match the number of columns.
- The query should return a single row.
- If the query returns no rows, a warning with error code 1329 occurs (No data), and the variable values remain unchanged.
- If the query returns multiple rows, error 1172 occurs (Result consisted of more than one row).
- The statement may retrieve multiple rows, you can use LIMIT 1 to limit the result set to a single row.

## *select ... into var\_list*

The SELECT ... INTO form of SELECT enables a query result to be stored in variables or written to a file.

### `SELECT ... INTO var_list`

- `SET @x = 0;`  
`SET @y = null;`  
`SELECT empno, ename INTO @x, @y FROM emp WHERE empno = 7788;`
- `SET @x = 0;`  
`SET @y = null;`  
`SELECT MAX(sal), MIN(sal) INTO @x, @y FROM emp WHERE empno = 7788;`

| variable_name    | value                                         |
|------------------|-----------------------------------------------|
| secure_file_priv | C:/ProgramData/MySQL/MySQL Server 8.0/Uploads |

In this example, I can only read files from the / Uploads / directory.

## select ... into outfile

If not working then do changes in *my.ini* file.

```
secure_file_priv = ""  
SHOW VARIABLES LIKE "secure_file_priv";
```

# *select ... into outfile*

Writes the selected rows to a file. Column and line terminators can be specified to produce a specific output format.

## SELECT ... INTO OUTFILE

- `SELECT identifier.* FROM emp INTO OUTFILE "/tmp/emp.csv";`
  - `SELECT identifier.* FROM emp INTO OUTFILE "/tmp/emp.txt";`
  - `SELECT identifier.* FROM emp INTO OUTFILE "/tmp/emp.csv" FIELDS TERMINATED BY ',';`
  - `SELECT identifier.* FROM emp INTO OUTFILE "/tmp/emp.csv" FIELDS TERMINATED BY ',';`
  - `SELECT identifier.* FROM emp INTO OUTFILE "/tmp/emp.csv" FIELDS TERMINATED BY ',' LINES TERMINATED by '\n';`
  - `SELECT identifier.* FROM emp INTO OUTFILE "/tmp/emp.csv" FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '\"'  
LINES TERMINATED BY '\n';`
  - `SELECT "DEPT", "DNAME", "LOC", "PWD", "STARTEDON" UNION SELECT * FROM dept;`
- 
- `TABLE emp INTO OUTFILE "/tmp/emp.csv";`
  - `TABLE emp INTO OUTFILE "/tmp/emp.txt";`

select ... into dumpfile

## *select ... into dumpfile*

If you use INTO DUMPFILE instead of INTO OUTFILE, MySQL writes only one row into the file, without any column or line termination and without performing any escape processing. This is useful if you want to store a BLOB value in a file.

### **SELECT ... INTO DUMPFILE**

- `SELECT identifier.* FROM emp WHERE empno = 7788 INTO DUMPFILE "/tmp/emp.txt";`

- mysql dump -uroot -p db1 >> d:\bk.sql
  - mysql dump -u[username] -p[password] --all-databases > d:\db\_backup.sql
  - mysql dump -P 3306 -h 192.168.100.26 -usaleel -psaleel db\_name > db\_backup.sql
  - mysql dump -P 3306 -h 192.168.100.26 -usaleel -psaleel db\_name table\_name > db\_backup.sql
  - mysql dump -P 3306 -h 192.168.100.26 -usaleel -psaleel db\_name table\_name --WHERE ="deptno=10" > db\_backup.sql
  - mysql dump -P 3306 -h 192.168.100.26 -usaleel -psaleel db\_name table\_name --WHERE ="job='manager'" > db\_backup.sql
  - mysql dump --column-statistics=0 -P 3306 -h 192.168.100.26 -usaleel -psaleel db\_name > db\_backup.sql
  - mysql -P 3306 -h 192.168.100.74 -uroot -p root db2 < d:\ backup\_fileName.sql
  - mysql -P 3306 -h 192.168.100.74 -uroot -p root < d:\ backup\_fileName.sql db2
- 

## snapshots

To create a raw data snapshot of MyISAM tables, you can use standard copy tools such as cp or copy, a remote copy tool such as scp or rsync, an archiving tool such as zip or tar, or a file system snapshot tool such as dump, providing that your MySQL data files exist on a single file system. If you are replicating only certain databases, copy only those files that relate to those tables.

CountryName,CapitalName,CapitalLatitude,CapitalLongitude,CountryCode,ContinentName,remark  
Somaliland,Hargeisa,9.55,44.05,NULL,Africa,  
South Georgia and South Sandwich Islands,King Edward Point,-54.283333,-36.5,GS,Antarctica,  
French Southern and Antarctic Lands,Port-aux-Français,-49.35,70.216667,TF,Antarctica,  
Palestine,Jerusalem,31.7666667,35.233333,PS,Asia,  
Aland Islands,Mariehamn,60.116667,19.9,AX,Europe,  
Nauru,Yaren,-0.5477,166.920867,NR,Australia,  
Saint Martin,Marigot,18.0731,-63.0822,MF,North America,  
Tokelau,Atafu,-9.166667,-171.833333,TK,Australia,  
Western Sahara,El-Aaiún,27.153611,-13.203333,EH,Africa,

load data infile "D:\abc1.csv" into table a fields terminated by ","  
ignore 1 rows (id,name,@dt) SET dt =  
STR\_TO\_DATE(@dt,"%d-%m-%Y");

## import .csv / .tsv file

```
CREATE TABLE countries (
    CountryName VARCHAR(45),
    CapitalName VARCHAR(45),
    CapitalLatitude VARCHAR(45),
    CapitalLongitude VARCHAR(45),
    CountryCode VARCHAR(45),
    ContinentName VARCHAR(45),
    remark VARCHAR(45)
);
```

```
LOAD DATA INFILE 'path/file-name.csv' INTO TABLE countries
FIELDS TERMINATED BY ',' or FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

- ```
SELECT "DEPT", "DNAME", "LOC", "PWD", "STARTEDON" UNION
SELECT * FROM dept;
```

## *Import .CSV file with dateFormat*

moduleId,moduleName,duration,startedON

```
1,OracleCR,6 months,23-10-2022  
2,PIG,2 months,12-10-2022  
3,Neo4j,4 months,20-10-2022  
4,Cassandra,7 months,03-10-2022  
5,HIVE,3 months,09-10-2022  
6,MongoDB,2 months,25-10-2022  
7,Redis,1 months,06-10-2022
```

```
CREATE TABLE module (  
moduleId INT,  
moduleName VARCHAR(45),  
duration VARCHAR(45),  
startedON DATE  
);
```

```
LOAD DATA INFILE 'path/file-name.csv' INTO TABLE module  
FIELDS TERMINATED BY ','  
LINES TERMINATED BY '\n'  
IGNORE 1 ROWS  
(moduleId,modulename,moduleDuration,@startedOn)  
SET startedOn = STR_TO_DATE(@startedOn,'%d-%m-%Y')  
;
```

# cluster

A cluster comprises multiple interconnected computers or servers that appear as if they are one server to end users and applications.

# data dictionary

A data dictionary contains metadata i.e data about the database. The data dictionary is very important as it contains information such as what is in the database, who is allowed to access it, where is the database physically stored etc. The users of the database normally don't interact with the data dictionary, it is only handled by the database administrators.

The data dictionary in general contains information about the following –

- Names of all the database tables and their schemas.
- Details about all the tables in the database, such as their owners, their security constraints, when they were created etc.
- Physical information about the tables such as where they are stored and how.
- Table constraints such as primary key attributes, foreign key information etc.
- Information about the database views that are visible.

# *data dictionary - information\_schema*

## Data Dictionary

INFORMATION\_SCHEMA.COLUMNS

INFORMATION\_SCHEMA.TABLES

INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS

INFORMATION\_SCHEMA.STATISTICS

INFORMATION\_SCHEMA.KEY\_COLUMN\_USAGE

INFORMATION\_SCHEMA.ROUTINES

INFORMATION\_SCHEMA.PARAMETERS

INFORMATION\_SCHEMA.TRIGGERS

**Class Room**

# **Session 1**

# pl/sql

PL/SQL is a procedural language extension to Structured Query Language (SQL). The purpose of PL/SQL is to combine database language and procedural programming language.

## Note:

- Statements that return a **result set** can be used within a STORED PROCEDURE but **not** within a STORED FUNCTION. This prohibition includes **SELECT** statements that do not have an **INTO var\_list** clause.
- DDL statements, such as **CREATE**, **ALTER**, and **DROP** etc. are permits STORED PROCEDURES, (**but not in** STORED FUNCTIONS).
- ERROR 1314 (0A000): USE command is not allowed in stored procedures
- Stored PROCEDURES and FUNCTIONS name can have max 64 char.

## create procedure and create function

- GRANT EXECUTE ON [ PROCEDURE | FUNCTION ] object\_name **TO** user;
- REVOKE EXECUTE ON [ PROCEDURE | FUNCTION ] object\_name **FROM** user;
- SHOW **CREATE PROCEDURE** <routine\_name>
- SHOW **CREATE FUNCTION** <routine\_name>

# *stored procedure and function*

These statements create stored routines. By default, a routine is associated with the default database. To associate the routine explicitly with a given database, specify the name as db\_name.sp\_name when you create it.

`CREATE PROCEDURE sp_name ([proc_parameter[,...]])`

`proc_parameter:`

`[ IN | OUT | INOUT ] param_name type`

`CREATE FUNCTION sp_name ([func_parameter[,...]])`

`RETURNS type`

`DETERMINISTIC`

`DROP PROCEDURE IF EXISTS procedurename;`  
delimiter \$\$

`CREATE PROCEDURE procedurename()`  
`begin`

`statements.....`

`end $$`

`delimiter ;`

`SELECT * FROM information_schema.parameters  
WHERE SPECIFIC_NAME = 'pro1';`

`desc mysql.proc`

`DROP FUNCTION IF EXISTS functionname;  
delimiter $$  
CREATE FUNCTION functionname() RETURNS INT  
DETERMINISTIC  
begin  
statements.....  
return (something);  
end $$  
delimiter ;`

**Parameter:** Is an unknown thing. Means it exists but don't know it.

e.g. Blood sugar it exists but we don't know it.

*proc\_parameter - out | inout param-name type*

TODO.

`CREATE PROCEDURE sp_name ([proc_parameter[,...]])`

`proc_parameter:`

`[ IN | OUT | INOUT ] param_name type`

### Note:

- For each **OUT** or **INOUT** parameter, pass a user-defined variable in the **CALL** statement that invokes the **PROCEDURE**. So that you can obtain its value when the **PROCEDURE** returns.
- If you are calling the **PROCEDURE** from within another stored **PROCEDURE** or **FUNCTION**, you can also pass a routine parameter or local routine variable as an **OUT** or **INOUT** parameter.

A parameter is a variable in a method definition. When a method is called, the arguments are the data you pass into the method's parameters. Parameter is variable in the declaration of function. Argument is the actual value of this variable that gets passed to function.

## difference between arguments and parameters?

- **Parameter** is variable in the declaration of *procedure* or a *function*.
- **Argument** is the actual value of this variable that gets passed to a *procedure* or a *function*.

# *stored procedure and function*

Parameter List

```
CREATE PROCEDURE EMPLOYEE(ID INT)
BEGIN
    SELECT * FROM EMP WHERE EMPNO = ID;
END;
```

Argument List

```
CALL EMPLOYEE(7788);
```

difference between stored  
procedure and function?

## *stored procedure and function*

- A FUNCTION always returns a value using the return statement. PROCEDURE may return one or more values through OUT parameter(s) or may not return any at all.
- FUNCTIONS are normally used for computations whereas PROCEDURES are normally used for executing business logic.
- A FUNCTION returns 1 value only. PROCEDURE can return multiple values (max 1024).
- A FUNCTION can be called directly by SQL statement like select func\_name from dual while PROCEDURES cannot.
- A FUNCTION does not allow the use of DDL statements (like: CREATE, ALTER, DROP etc.) while PROCEDURE can.

The major difference is that UDFs can be used like any other expression within SQL statements, whereas stored procedures must be invoked using the CALL statement.

### Note:

- MySQL permits routines to contain DDL statements, such as CREATE, ALTER and DROP in **stored procedure**,
- but DDL statements, such as CREATE, ALTER and DROP are not permitted in **stored function**.

# *calling procedure from function and contrariwise*

- DROP FUNCTION IF EXISTS F1;

delimiter \$\$

CREATE FUNCTION F1() returns VARCHAR(100)

DETERMINISTIC

begin

declare x VARCHAR(100) default 'Hello World';

# SELECT "Hello World";

call p1(x);

return x;

end \$\$

delimiter ;



# ERROR 1415 (0A000): Not allowed to  
return a result set from a function

- DROP PROCEDURE IF EXISTS P1;

delimiter \$\$

CREATE PROCEDURE P1(OUT para1 VARCHAR(100))

begin

# SELECT "Hello World123";



# ERROR 1415 (0A000): Not allowed to  
return a result set from a function

SELECT "Hello World123" INTO para1;

end \$\$

delimiter ;



This will work

source and call stored procedure

## *source and call stored procedure*

If you are already running mysql, you can execute an SQL script file using the **source** command or **\.** command

`source file_name.sql`

`\. file_name.sql`

```
MySQL> source pl1.sql;  
MySQL> \. file_name.sql;
```

The CALL statement invokes a stored procedure that was defined previously with CREATE PROCEDURE. Stored procedures that take no arguments can be invoked without parentheses. That is, CALL sp\_name and CALL sp\_name() are equivalent.

`CALL sp_name([parameter[,...]])`

`CALL sp_name[()]`

```
MySQL> call sp_name;  
MySQL> call sp_name();
```

## *delimiter - statement*

By default, mysql itself recognizes the semicolon as a statement delimiter, so you must redefine the delimiter temporarily to cause mysql to pass the entire stored program definition to the server. To redefine the mysql delimiter, use the delimiter command.

```
DROP PROCEDURE IF EXISTS procedurename;  
delimiter $$  
CREATE PROCEDURE procedurename()  
begin  
    statements.....  
end $$  
delimiter ;
```

```
DROP FUNCTION IF EXISTS functionname;  
delimiter $$  
CREATE FUNCTION functionname() RETURNS INT  
DETERMINISTIC  
begin  
    statements.....  
    return (something);  
end $$  
delimiter ;
```

# begin ... end compound-statement

A compound statement is a block that can contain:

- other blocks
- declarations of variables
- declarations of cursors
- declarations of exception handlers
- compound statements
- flow control constructs such as loops and conditional tests.

# *begin ... end*

BEGIN ... END syntax is used for writing compound statements, which can appear within stored programs (stored procedures and functions, triggers, and events). A compound statement can contain multiple statements, enclosed by the BEGIN and END keywords. statement\_list represents a list of one or more statements, each terminated by a semicolon (;) statement delimiter. The statement\_list itself is optional, so the empty compound statement (BEGIN END) is legal.

[begin\_label:] BEGIN

[statement\_list]

....

....

....

END [end\_label]

- DROP PROCEDURE IF EXISTS procedurename;  
delimiter \$\$  
CREATE PROCEDURE procedurename()  
begin  
    declare x BOOL default false;  
    begin  
        declare y BOOL default true;  
        SELECT x;  
    end ;  
    SELECT y;  
end \$\$  
delimiter ;

**DECLARE** variable\_name datatype(size) [ **DEFAULT** default\_value ];

1. declare x INT DEFAULT 0;  
SET x := 10;
2. declare y INT;  
SET y := 10;
3. **SET x := (SELECT 1001)**

**Declaring  
variables**

**DECLARE** variable\_name datatype(size)

**Assigning variables**

- 1 **DECLARE z INT;**
- 2 **SET z = 10;**

# declare variables

**Note:**

- **DECLARE** is permitted only inside a **BEGIN ... END** compound statement and must be at its start, before any other statements.
- Only scalar values can be assigned. For example, a statement such as **SET x = (SELECT 1, 2)** **is invalid**.

**Remember:**

- First, you specify the variable name after the **DECLARE** keyword. The variable name must follow the naming rules of MySQL table column names.
- Second, you specify the data type of the variable and its size. A variable can have any MySQL data types such as **INT**, **VARCHAR**, and **DATETIME**.
- Third, when you declare a variable, its initial value is **NULL**. You can assign the variable a default value using the **DEFAULT** keyword.

## *local variable declare*

### Remember:

- A local variable takes precedence over a routine parameter or table column.
- A routine parameter takes precedence over a table column.

C1	C2
7	7

- ```
DROP PROCEDURE IF EXISTS pro1;
delimiter $
CREATE PROCEDURE pro1 (c1 INT)
begin
    declare c1 INT default 0;
    SELECT c1 FROM x;
end $
delimiter ;
```

```
mysql> CALL pro1(12);
```

# *local variable declare and user-defined variables*

Local variables are declared within stored procedures and are only valid within the BEGIN...END block where they are declared. Local variables can have any SQL data type.

`DECLARE var_name [, var_name] ... type [DEFAULT value]`

---

User variables in MySQL stored procedures, user variables are referenced with an ampersand (@) prefixed to the user variable name.

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

- `DROP PROCEDURE IF EXISTS procedurename;  
delimiter $$  
CREATE PROCEDURE procedurename()  
begin  
declare x VARCHAR(12) default 'Infoway';  
SELECT x;  
end $$  
delimiter ;`
- `DROP PROCEDURE IF EXISTS procedurename;  
delimiter $$  
CREATE PROCEDURE procedurename()  
begin  
set @x = 'Infoway';  
SELECT @x;  
end $$  
delimiter ;`

# *local variable declare and user-defined variables*

Local variables are declared within stored procedures and are only valid within the BEGIN...END block where they are declared. Local variables can have any SQL data type.

`DECLARE var_name [, var_name] ... type [DEFAULT value]`

---

User variables in MySQL stored procedures, user variables are referenced with an ampersand (@) prefixed to the user variable name.

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

- `DROP PROCEDURE IF EXISTS procedurename;`  
`delimiter $$`  
`CREATE PROCEDURE procedurename(x INT)`  
`begin`  
`SET x:=(SELECT 1001);`  
`SET @y := (SELECT 2001);`  
`end $$`  
`delimiter ;`
- `DROP FUNCTION IF EXISTS functionname;`  
`delimiter $$`  
`CREATE PROCEDURE functionname(x INT) RETURNS INT`  
`begin`  
`SET x := (SELECT 1002);`  
`SET @y := (SELECT 2002);`  
`return(x);`  
`end $$`  
`delimiter ;`

## *local variable declare*

Local variables are declared within stored procedures and are only valid within the BEGIN...END block where they are declared. Local variables can have any SQL data type.

DECLARE var\_name [, var\_name] ... type [DEFAULT value]

---

- DROP PROCEDURE IF EXISTS procedurename;

delimiter \$\$

CREATE PROCEDURE procedurename()

begin

declare x VARCHAR(20);

set x = 'Hello World';

SELECT x;

end \$\$

delimiter ;

## *local variable declare*

- ```
DROP PROCEDURE IF EXISTS procedurename;
delimiter $$

CREATE PROCEDURE procedurename()
begin
    declare a INT default 10;
    declare b, c INT;
    set a = a + 100;
    set b = 2;
    set c = a + b;
    begin
        declare c INT;
        set c = 5;
        SELECT a, b, c;
    end;
    SELECT a, b, c;
end $$

delimiter ;
```

# delimiter problem

- ```
DROP PROCEDURE IF EXISTS procedurename;
delimiter$$
CREATE PROCEDURE procedurename(OUT para1 VARCHAR(100))
begin
    SELECT "Hello World123" INTO para1;
end $$
```

 delimiter;

if and loop label

# *if and loop*

```
IF search_condition THEN statement_list  
    [ELSEIF search_condition THEN statement_list] ...  
    [ELSE statement_list]  
END IF
```

---

**ITERATE** means "start the loop again."

**LEAVE** statement is used to exit the flow control construct that has the given label.

```
[begin_label:] LOOP  
    statement_list  
END LOOP [end_label]
```

ITERATE label

LEAVE label

*if*

```
IF search_condition THEN statement_list  
[ELSEIF search_condition THEN statement_list] ...  
[ELSE statement_list]  
END IF
```

e.g.

```
DROP PROCEDURE IF EXISTS procedurename;  
delimiter $$  
CREATE PROCEDURE procedurename(v_value INT)  
begin  
if v_value = 0 then  
    SELECT 'The enter value is 0' as "Message Box";  
elseif v_value = 1 THEN  
    SELECT 'The enter value is 1' as "Message Box";  
else  
    SELECT 'The enter value is neither 0 or 1' as "Message Box";  
end if;  
end $$  
delimiter ;
```

# *loop*

```
[begin_label:] LOOP  
    statement_list  
END LOOP [end_label]
```

- ITERATE label
- LEAVE label

e.g.

```
DROP PROCEDURE IF EXISTS procedurename;  
delimiter $$  
CREATE PROCEDURE procedurename(v_value INT)  
begin  
    declare x INT default 0;  
    lbl:loop  
        set x = x + 1;  
        SELECT x;  
        if x = v_value then  
            leave lbl;  
        end if;  
    end loop lbl;  
end $$  
delimiter ;
```

**ITERATE** means "start the loop again."

**LEAVE** statement is used to exit the flow control construct that has the given label.

A procedure is a group of PL/SQL statements that you can call by name.

# stored procedure

A procedure is a group of PL/SQL statements that you can call by name.

## Remember:

- It is **not permitted** to assign the value DEFAULT to stored PROCEDURE or FUNCTION parameters  
e.g.  
`PROCEDURE procedurename(IN p_ID INT DEFAULT 10)`
- Or stored program local variables  
e.g.  
`SET var_name = DEFAULT some_value.`

## *stored procedure - examples*

- `DROP PROCEDURE IF EXISTS procedurename;`

`delimiter $$`

`CREATE PROCEDURE procedurename()`

`begin`

`SELECT * FROM emp;`

`end $$`

`delimiter ;`

- `DROP PROCEDURE IF EXISTS procedurename;`

`delimiter $$`

`CREATE PROCEDURE procedurename()`

`begin`

`declare x VARCHAR(12);`

`SELECT ename INTO x FROM emp WHERE empno = 7788;`

`SELECT x;`

`end $$`

`delimiter ;`

## *stored procedure - examples*

- `DROP PROCEDURE IF EXISTS procedurename;`

`delimiter $$`

`CREATE PROCEDURE procedurename(IN para1 INT)`

`begin`

`declare x VARCHAR(12);`

`SELECT ename INTO x FROM emp WHERE empno = para1;`

`SELECT x;`

`end $$`

`delimiter ;`

- `DROP PROCEDURE IF EXISTS procedurename;`

`delimiter $$`

`CREATE PROCEDURE procedurename(IN para1 VARCHAR(15))`

`begin`

`declare x VARCHAR(12);`

`SELECT job INTO x FROM emp WHERE ename = para1;`

`SELECT x;`

`end $$`

`delimiter ;`

## *stored procedure - examples*

- DROP PROCEDURE IF EXISTS procedurename;

delimiter \$\$

CREATE PROCEDURE procedurename(**IN** para1 VARCHAR(12))

begin

    SELECT \* FROM emp WHERE job = para1;

end \$\$

delimiter ;

- DROP PROCEDURE IF EXISTS procedurename;

delimiter \$\$

CREATE PROCEDURE procedurename(**IN** para1 VARCHAR(12), **OUT** s INT, **OUT** j VARCHAR(12))

begin

    SELECT para1, SUM(Sal) INTO varJob, varSumSal FROM emp WHERE job = para1;

end \$\$

delimiter ;

## *stored procedure - examples*

- DROP PROCEDURE IF EXISTS procedurename;

delimiter \$\$

CREATE PROCEDURE procedurename(**IN** para1 VARCHAR(12))

begin

if (SELECT true FROM dept WHERE deptno = x) then

    SELECT \* FROM emp WHERE deptno = x;

else

    SELECT "Data Not Found";

end if;

end \$\$

delimiter ;

## *stored procedure - examples*

- `DROP PROCEDURE IF EXISTS procedurename;`

`delimiter $$`

`CREATE PROCEDURE procedurename(IN p1 INT(2), IN p2 VARCHAR(12), IN p3 VARCHAR(10), IN p4 VARCHAR(20))`

`begin`

`INSERT into DEPT values (p1, p2, p3, p4);`

`end $$`

`delimiter ;`

- `DROP PROCEDURE IF EXISTS procedurename;`

`delimiter $$`

`CREATE PROCEDURE myprocedure (INOUT p1 INT(2))`

`begin`

`set p1 = p1+2;`

`end $$`

`delimiter ;`

`mysql> SET @x = 10`

`mysql> CALL procudeureName(@x);`

`mysql> SELECT @x`

## *stored procedure - examples*

- `DROP PROCEDURE IF EXISTS procedurename;`

`delimiter $$`

`CREATE PROCEDURE procedurename()`

`begin`

`CREATE TABLE temp (IN col1 INT);`

`end $$`

`delimiter ;`

- `DROP PROCEDURE IF EXISTS procedurename;`

`delimiter $$`

`CREATE PROCEDURE procedurename(IN v1 INT, IN v2 VARCHAR(20), IN v3 VARCHAR(20), IN v4 VARCHAR(20))`

`begin`

`INSERT INTO dept VALUES (v1, v2, v3, v4);`

`end $$`

`delimiter ;`

- `DROP PROCEDURE IF EXISTS procedurename;`

`delimiter $$`

`CREATE PROCEDURE procedurename(IN var1 INT)`

`begin`

`SELECT * FROM emp LIMIT var1;`

`end $$`

`delimiter ;`

## *stored procedure - examples*

- `DROP PROCEDURE IF EXISTS procedurename;`

`delimiter $$`

`CREATE PROCEDURE procedurename()`

`begin`

`CREATE TABLE temp (col1 INT);`

`end $$`

`delimiter ;`

- `DROP PROCEDURE IF EXISTS procedurename;`

`delimiter $$`

`CREATE PROCEDURE procedurename()`

`begin`

`#CREATE TABLE temp (col1 INT);`

`#ALTER TABLE temp ADD col2 INT;`

`#DROP TABLE temp;`

`end $$`

`delimiter ;`

## *stored procedure - examples*

- ```
DROP PROCEDURE IF EXISTS procedurename;
delimiter $$

CREATE PROCEDURE procedurename()
begin
    declare exit handler for 1146 CREATE TABLE abc(id INT, ename VARCHAR(20));
    SELECT * FROM abc;
end $$

delimiter ;
```

## *stored procedure - examples*

- `DROP PROCEDURE IF EXISTS procedurename;`  
`delimiter $$`  
`CREATE PROCEDURE procedurename(IN x INT, IN y INT, OUT z INT, OUT z1 INT )`  
`begin`  
    `set z = x + y;`  
    `set z1 = x * y;`  
`end $$`  
`delimiter ;`

---

```
mysql> SET @a := 0;
mysql> SET @b := 0;
mysql> CALL pro1(5, 4, @a, @b);
```

*dynamic stored procedure*

prepared sql statement

## *prepared sql statement*

The PREPARE statement prepares a SQL statement and assigns it a name, stmt\_name. The prepared statement is executed with EXECUTE and released with DEALLOCATE PREPARE.

PREPARE stmt\_name FROM preparable\_stmt

EXECUTE stmt\_name  
[ USING @var\_name [, @var\_name ] . . . ]

{ DEALLOCATE | DROP } PREPARE stmt\_name

```
SET @skip=1; SET @numrows=5;  
PREPARE STMT FROM 'SELECT * FROM table_name LIMIT ?, ?';  
EXECUTE STMT USING @skip, @numrows;
```

# *prepared sql statement*

## Example of prepared statement

- PREPARE STMT FROM 'SELECT \* FROM emp';  
EXECUTE STMT;
- SET @x := 'SELECT \* FROM emp WHERE deptno = 10';  
PREPARE STMT FROM @x;  
EXECUTE STMT;
- SET @x:=10;  
PREPARE STMT FROM 'SELECT \* FROM emp WHERE deptno = ?';  
EXECUTE STMT USING @x;
- SET @x:=10, @y := 3000;  
PREPARE STMT FROM 'SELECT \* FROM emp WHERE deptno = ? and sal >= ?';  
EXECUTE STMT USING @x, @y;
- DEALLOCATE PREPARE stat1;

# *dynamic stored procedure - examples*

- `DROP PROCEDURE IF EXISTS procedurename;`

`delimiter $$`

`CREATE PROCEDURE procedurename(table_name VARCHAR(10))`

`begin`

`set @t1 = CONCAT('SELECT * FROM ', table_name );`

`PREPARE STMT FROM @t1;`

`EXECUTE STMT;`

`end $$`

`delimiter ;`

- `DROP PROCEDURE IF EXISTS procedurename;`

`delimiter $$`

`CREATE PROCEDURE procedurename(c1 VARCHAR(10), x VARCHAR(10))`

`begin`

`set @t1 = CONCAT('SELECT ', c1 , ' FROM ',x );`

`PREPARE STMT FROM @t1;`

`EXECUTE STMT;`

`end $$`

`delimiter ;`

## *dynamic stored procedure - examples*

- `DROP PROCEDURE IF EXISTS procedurename;`

`delimiter $`

```
CREATE PROCEDURE procedurename(in _DEPTNO INT, _DNAME VARCHAR(20), _LOC VARCHAR(20), _PWD  
VARCHAR(20))
```

`begin`

```
set @a := _DEPTNO;
```

```
set @b := _DNAME;
```

```
set @c := _LOC;
```

```
set @d := _PWD;
```

```
set @x:= concat("INSERT INTO dept VALUES(?, ?, ?, ?)");
```

```
PREPARE STMT FROM @x;
```

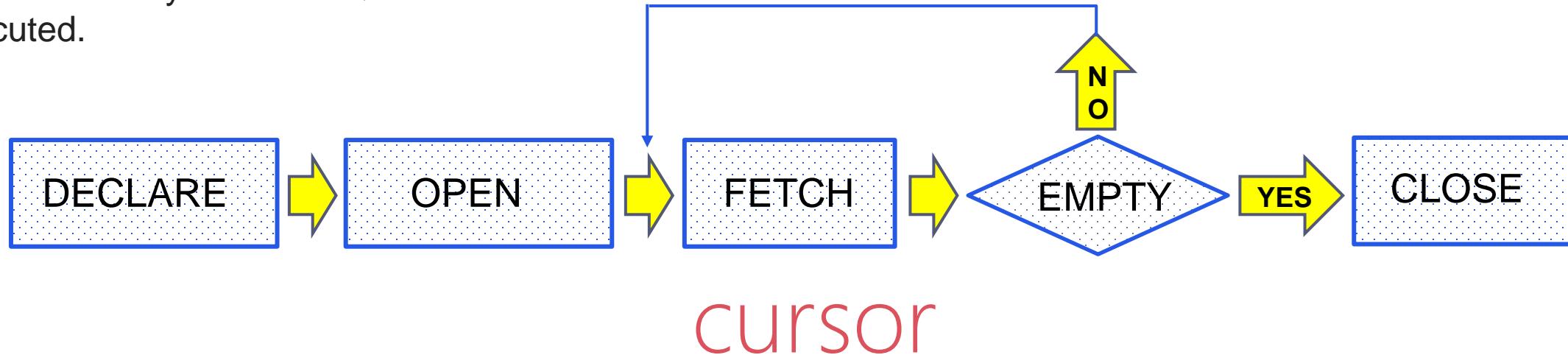
```
EXECUTE STMT USING @a, @b, @c ,@d;
```

`end $`

`delimiter ;`

- A **cursor** is a temporary work area created in the system memory when a SQL statement is executed.

`SELECT found_rows();`



A cursor allows you to iterate a set of rows returned by a query and process each row individually.

### Remember:

- **Read only:** Not updatable.
- **Nonscrollable:** Can be traversed only in one direction and cannot skip rows.
- **Asensitive:** there are two kinds of cursors: asensitive cursor and insensitive cursor. An asensitive cursor points to the actual data, whereas an insensitive cursor uses a temporary copy of the data. MySQL cursor is asensitive.

### Note:

- Cursor declarations must appear before handler declarations. Variable and condition declarations must appear before cursor or handler declarations.

## STEPS:

- **DECLARE cursor\_name CURSOR FOR select\_statement**

This statement declares a cursor and associates it with a SELECT statement that retrieves the rows to be traversed by the cursor.

- **OPEN cursor\_name**

This statement opens a previously declared cursor.

- **FETCH [[NEXT] FROM] cursor\_name INTO var\_name [, var\_name] ...**

This statement fetches the next row for the SELECT statement associated with the specified cursor (which must be open), and advances the cursor pointer. If a row exists, the fetched columns are stored in the named variables. The number of columns retrieved by the SELECT statement must match the number of output variables specified in the FETCH statement.

If no more rows are available, a **No Data condition**.

- **CLOSE cursor\_name**

This statement closes a previously opened cursor.

Q1. Using cursor get all student details.

```
DROP PROCEDURE IF EXISTS getAllStudent;
delimiter $$

CREATE PROCEDURE getAllStudent()
begin
    declare v_ID INT;
    declare v_namefirst, v_namelast VARCHAR(45);
    declare v_dob DATE;
    declare v_emailID VARCHAR(128);
    declare c1 CURSOR FOR SELECT * FROM student;
    declare EXIT HANDLER FOR NOT FOUND SELECT "No more student found";
    OPEN c1;
    lbl: loop
        FETCH c1 INTO v_ID, v_namefirst, v_namelast, v_dob, v_emailID;
        SELECT v_ID, v_namefirst, v_namelast, v_dob, v_emailID;
    end loop lbl;
    CLOSE c1;
end $$

delimiter ;
```

## CURSOR

- `DROP PROCEDURE IF EXISTS procedurename;`  
delimiter \$\$  
`CREATE PROCEDURE procedurename(_custID INT)`  
`b1:begin`  
    `if (SELECT true FROM customer WHERE custid = _custID) then`  
        `b2: begin`  
            `declare _ordID INT;`  
            `declare _orderdate, _shipdate DATETIME;`  
            `declare _status, _commplan VARCHAR(45);`  
            `declare _total FLOAT(8,2);`  
            `declare c1 CURSOR for SELECT ordID, orderdate, commplan, shipdate, status, total FROM ord`  
            `WHERE custid = _custID;`  
            `OPEN c1;`  
            `lbl:loop`  
                `FETCH c1 INTO _ordID, _orderdate, _commplan, _shipdate, _status, _total;`  
                `select _ordid, _orderdate, _commplan, _shipdate, _status, _total;`  
                `end loop lbl;`  
                `CLOSE c1;`  
            `end b2;`  
        `end if;`  
    `end b1 $$`  
delimiter ;

## CURSOR

- `DROP PROCEDURE IF EXISTS procedurename;`  
delimiter \$\$  
`CREATE PROCEDURE procedurename(_custID INT, _status VARCHAR(45))`  
b1: `begin`  
    `if (SELECT true FROM customer WHERE custid= _custID) then`  
        `SELECT * FROM customer WHERE custid= _custID;`  
    b2: `begin`  
        `declare _ordID INT;`  
        `declare _orderdate, _shipdate DATETIME;`  
        `declare _commplan VARCHAR(45);`  
        `declare _total FLOAT(8,2);`  
        `declare c1 CURSOR for SELECT ordID, orderdate, commplan, shipdate, total FROM ord WHERE custid = _custID and status= _status;`  
        `OPEN c1;`  
        `lbl:loop`  
            `FETCH c1 INTO _ordID, _orderdate, _commplan, _shipdate, _total;`  
            `SELECT _ordID, _orderdate, _commplan, _shipdate, _status _total;`  
        `end loop lbl;`  
        `CLOSE c1;`  
    `end b2;`  
    `end if;`  
  `end b1 $$`  
delimiter ;

Q2 Write a program to enter studentID and using cursor get his student qualification details.

```
DROP PROCEDURE if EXISTS getStudentQualification;
```

```
CREATE PROCEDURE getStudentQualification(v_ID INT)
```

```
b1:begin
```

```
    declare x INT;
```

```
        SELECT ID into x FROM student WHERE ID = v_ID;
```

```
    if x is null then
```

```
        SELECT "Student not found....";
```

```
    else
```

```
        b2: begin
```

```
            declare v_name, v_college, v_university VARCHAR(128);
```

```
            declare v_marks VARCHAR(45);
```

```
            declare v_year INT;
```

```
            declare c1 CURSOR for SELECT name, college, university, marks, year FROM student_qualifications where studentID = x;
```

```
            declare EXIT HANDLER FOR NOT FOUND SELECT "Done";
```

```
            OPEN c1;
```

```
            lbl:loop
```

```
                FETCH c1 INTO v_name, v_college, v_university, v_marks, v_year;
```

```
                SELECT v_name, v_college, v_university, v_marks, v_year;
```

```
            end loop lbl;
```

```
            CLOSE c1;
```

```
        end b2;
```

```
    end if;
```

```
end b1 $$
```

```
delimiter ;
```

- `DROP PROCEDURE IF EXISTS procedurename;`  
`delimiter $$`  
`CREATE PROCEDURE procedurename(para1 VARCHAR(25))`  
`begin`  
    `declare done TINYINT DEFAULT FALSE;`  
    `declare varempno INT;`  
    `declare varename VARCHAR(20);`  
    `declare c1 CURSOR FOR SELECT empno, ename FROM emp WHERE job = para1;`  
    `declare EXIT HANDLER FOR NOT FOUND set done=true;`  
`OPEN c1;`  
    `lbl: loop`  
        `FETCH c1 INTO varempno, varename;`  
        `if done then`  
            `LEAVE lbl;`  
        `else`  
            `SELECT varempno as "Employee No.", varename as "Employee Name";`  
        `end if;`  
    `end loop lbl;`  
`CLOSE c1;`  
`end I$$`  
`delimiter ;`

# cursor

- `DROP PROCEDURE IF EXISTS procedurename;`  
`delimiter $$`  
`CREATE PROCEDURE procedurename()`  
`begin`  
    `declare x INT;`  
    `declare p, q, r VARCHAR(20);`  
    `declare c1 CURSOR FOR SELECT * FROM d;`  
    `declare EXIT HANDLER FOR 1329 SELECT "No More data to extract" as R1;`  
`OPEN c1;`  
        `SELECT sleep(20);`  
    `lbl:loop`  
        `FETCH c1 INTO x, p, q, r;`  
        `SELECT x, p, q, r;`  
    `end loop;`  
`CLOSE c1;`  
`end $$`  
`delimiter ;`

USER root session 1  
mysql> call pro1();

USER root session 2  
mysql> `INSERT INTO dept VALUES(1,1,1,1);`  
mysql> commit;

exception / signal

## *exception / signal*

- ```
DROP PROCEDURE IF EXISTS procedurename;
delimiter $$

CREATE PROCEDURE procedurename()
begin
    declare a INT DEFAULT 101;
    declare myException CONDITION FOR SQLSTATE '45000';

/* declare EXIT HANDLER FOR 1146 SELECT 'table not found';
   declare EXIT HANDLER FOR 1064
*/
    if a = 10 then
        SELECT 'ok';
    else
        SIGNAL myException
        SET MESSAGE_TEXT = 'bad';
    end if;
end $$

delimiter ;
```

A DETERMINISTIC function always returns the same result for the same input parameters whereas a non-deterministic function returns different results for the same input parameters. If you don't use DETERMINISTIC or NOT DETERMINISTIC , MySQL uses the NOT DETERMINISTIC option by default.

```
SET GLOBAL log_bin_trust_function_creators = 1;
```

# functions

The RETURNS clause may be specified *only for a FUNCTION*, for which it is mandatory. It indicates the return type of the function, and the function body must contain a RETURN value statement.

## Note:

- By default, all parameters are IN parameters.
- You cannot specify IN, OUT or INOUT modifiers to the parameters.
- It is possible to create a stored function with the same name as a built-in function, but to invoke the stored function it is necessary to qualify it with a schema name.

# *user defined function*

## Remember:

- It is possible to create a stored function with the same name as a built-in function, but to invoke the stored function it is necessary to qualify it with a schema name.  
e.g. `SELECT db1.length();`
- ERROR 1415 (0A000): Not allowed to return a result set from a function  
e.g.  
`SELECT "Hello World"; // will not work in FUNCTION`  
`SELECT "Hello World" INTO x; // will work in FUNCTION`
- ERROR 1336 (0A000) : **Dynamic SQL** is not allowed in **stored function or trigger**.
- ERROR 1422 (HY000) : **EXPLICIT** or **IMPLICIT** commit is not allowed in **stored function or trigger**.

## *user defined function - examples*

- DROP FUNCTION IF EXISTS functionname;  
delimiter \$\$  
CREATE FUNCTION functionname() RETURNS INT  
DETERMINISTIC  
begin  
declare x INT;  
set x = 1001;  
return (x);  
end \$\$  
delimiter ;
- DROP FUNCTION IF EXISTS functionname;  
delimiter \$\$  
CREATE FUNCTION functionname(para1 VARCHAR(12)) RETURNS INT  
DETERMINISTIC  
begin  
declare total INT default 0;  
SELECT SUM(sal) INTO total FROM emp WHERE job=para1;  
return (total);  
end \$\$  
delimiter ;

## *user defined function - examples*

- `DROP FUNCTION IF EXISTS functionname;`  
`delimiter $$`  
`CREATE FUNCTION functionname() RETURNS INT`  
`DETERMINISTIC`  
`begin`  
    `return (SELECT MAX(deptno) + 1 FROM dept);`  
`end $$`  
`delimiter ;`
- `DROP FUNCTION IF EXISTS functionname;`  
`delimiter $$`  
`CREATE FUNCTION functionname() RETURNS INT`  
`DETERMINISTIC`  
`begin`  
    `return CONCAT(UPPER(LEFT(x,1)), SUBSTR(LOWER(x), 2 ));`  
`end $$`  
`delimiter ;`

# *user defined function - examples*

- `DROP FUNCTION IF EXISTS functionname;`

`delimiter $`

`CREATE FUNCTION functionname() RETURNS VARCHAR(40)`

`deterministic`

`begin`

`declare x, y, z VARCHAR(40) default "";`

`declare cnt INT default 1;`

`lbl:loop`

`SELECT CAST(CHAR(FLOOR(65 + RAND() * 27)) as CHAR) INTO y;`

`SELECT CAST(CHAR(FLOOR( 97 + RAND() * 27)) as CHAR) INTO z;`

`SET x := CONCAT(x, y, z);`

`if cnt > 2 then`

`leave lbl;`

`end if;`

`SET cnt := cnt + 1;`

`end loop lbl;`

`return(x);`

`end $`

`delimiter ;`

# *user defined function - examples*

- `DROP FUNCTION IF EXISTS functionname;`

`delimiter $$`

`CREATE FUNCTION functionname(xx INT) RETURNS VARCHAR(50)`

`DETERMINISTIC`

`begin`

`declare x BOOL;`

`SELECT True INTO x FROM emp WHERE empno = xx;`

`if x then`

`return (SELECT CONCAT(ename, " ", job, " ", sal) FROM emp WHERE empno = xx);`

`else`

`return "deptno is not valid";`

`end if;`

`end $$`

`delimiter ;`

## *user defined function - examples*

- `DROP FUNCTION IF EXISTS functionname;`  
`delimiter $$`  
`CREATE FUNCTION functionname(no INT) RETURNS VARCHAR(20)`  
`DETERMINISTIC`  
`begin`  
`declare x VARCHAR(20);`  
`SELECT distinct 'Number present' INTO x FROM t1 WHERE c1 = no;`  
`if x is not null then`  
`return(x);`  
`else`  
`return(no);`  
`end if;`  
`end $$`  
`delimiter ;`

- **DROP FUNCTION IF EXISTS** functionname;

delimiter \$

**CREATE FUNCTION** functionname(x **VARCHAR(1000)**) **RETURNS** **VARCHAR(1000)**

**DETERMINISTIC**

**begin**

declare y **INT** default 0;

declare z **VARCHAR(1000)** default " ";

lbl:loop

set y := y+1;

set z := TRIM(CONCAT( z, SUBSTR( x, y, 1 ), ' ' ));

if y >= LENGTH(x) then

  leave lbl;

end if;

end loop lbl;

**return** (SUBSTR( z, 1, LENGTH( z ) - 1 ));

**end** \$

delimiter ;

- **DROP FUNCTION IF EXISTS** functionname;

delimiter \$

**CREATE FUNCTION** functionname( x **VARCHAR**(1000)) **RETURNS** **VARCHAR**(1000)

**DETERMINISTIC**

**begin**

declare y **INT** default 0;

declare z **VARCHAR**(1000) default " ";

lbl:loop

set y := y+1;

if ASCII(SUBSTR( x, y, 1 )) NOT BETWEEN 48 AND 57 then

    set z := CONCAT( z, SUBSTR( x, y, 1 ));

end if;

if y > LENGTH(x) then

    leave lbl;

end if;

end loop lbl;

**return** (SUBSTR(TRIM(z),1, LENGTH(z)));

**end** \$

delimiter ;

- **DROP FUNCTION IF EXISTS** functionname;

delimiter \$

**CREATE FUNCTION** functionname( x **VARCHAR**(1000) ) **RETURNS** **VARCHAR**(1000)

**DETERMINISTIC**

**begin**

declare y **INT** default 0;

declare z **VARCHAR**(1000) default " ";

lbl:loop

set y := y+1;

if ASCII(SUBSTR( x, y, 1 )) NOT BETWEEN 65 AND 90 and ASCII(SUBSTR( x, y, 1 )) not between 97 AND 122 then

    set z := CONCAT( z, SUBSTR( x, y, 1 ));

end if;

if y > LENGTH(x) then

    leave lbl;

end if;

end loop lbl;

**return** (SUBSTR(TRIM(z),1, LENGTH(z)));

**end** \$

delimiter ;

# *user defined function with cursor - examples*

- `DROP FUNCTION IF EXISTS functionname;`

`delimiter $`

`CREATE FUNCTION functionname() RETURNS VARCHAR(10000)`

`DETERMINISTIC`

`begin`

`declare x, y VARCHAR(100);`

`declare z VARCHAR(10000) default "";`

`declare c1 CURSOR for SELECT deptno, dname FROM dept ORDER BY deptno DESC;`

`declare exit handler for 1329 return z;`

`open c1;`

`lbl:loop`

`fetch c1 into x, y;`

`set z := concat(x, " ", y, " ", "\n ", z);`

`end loop lbl;`

`end $`

`delimiter ;`

## *user defined function - examples*

- `DROP FUNCTION IF EXISTS functionname;`

`delimiter $$`

`CREATE FUNCTION functionname() RETURNS VARCHAR(20)`

`DETERMINISTIC`

`begin`

`#CREATE TABLE temp (col1 INT);`

`#ALTER TABLE temp ADD col2 INT;`

`#DROP TABLE temp;`

`return "done";`

`end $$`

`delimiter ;`

**ERROR 1422 (HY000): Explicit or implicit commit is not allowed in stored function or trigger.**

## Triggers are mainly required for the following purposes:

- To maintain complex integrity constraints
- Auditing table information by recording the changes
- Signaling other program actions when changes are made to the table
- Enforcing complex business rules
- Preventing invalid transactions

Triggers activate only for changes made to tables by SQL statements.

# triggers

A trigger is a special type of stored procedure that automatically runs when an event occurs in the database server. DML triggers run when a user tries to modify data through a data manipulation language (DML) event. DML events are INSERT, UPDATE, or DELETE statements on a TABLE or VIEW.

### Remember:

- The trigger is always associated with the table named `tbl_name`, which must refer to a permanent table.
- You cannot associate a trigger with a TEMPORARY table or a VIEW.
- If you drop a table, any triggers for the table are also dropped.
- ERROR 1415 (0A000): Not allowed to return a result set from a trigger.
- It is possible to define multiple triggers for a given table that have the same trigger event and action time. For example, you can have two BEFORE UPDATE triggers for a table. **By default, triggers that have the same trigger event and action time activate in the order they were created.**

# *trigger*

The trigger is always associated with the table named `tbl_name`, which must refer to a permanent table.

```
CREATE TRIGGER trigger_name  
  trigger_time trigger_event  
  ON tbl_name FOR EACH ROW  
  trigger_body
```

`trigger_time`: { BEFORE | AFTER }

`trigger_event`: { INSERT | UPDATE | DELETE }

- `SHOW CREATE TRIGGER trigger_name`

```
DESC INFORMATION_SCHEMA.TRIGGERS
```

# *trigger*

You put the trigger name after the CREATE TRIGGER statement. The trigger name should follow the naming convention [trigger time]\_[table name]\_[trigger event], for example before\_employees\_update.

Trigger activation time can be BEFORE or AFTER. You must specify the activation time when you define a trigger. You use the BEFORE keyword if you want to process action prior to the change is made on the table and AFTER if you need to process action after the change is made.

The trigger event can be INSERT, UPDATE or DELETE. This event causes the trigger to be invoked. A trigger only can be invoked by one event. To define a trigger that is invoked by multiple events, you have to define multiple triggers, one for each event.

A trigger must be associated with a specific table. Without a table trigger would not exist therefore you have to specify the table name after the ON keyword.

You place the SQL statements between BEGIN and END block. This is where you define the logic for the trigger.

# *what is trigger\_time and trigger\_event ?*

**trigger\_time** : trigger\_time is the trigger action time. It can be BEFORE or AFTER to indicate that the trigger activates before or after each row to be modified.

**trigger\_event** : trigger\_event indicates the kind of operation that activates the trigger.

**INSERT**: The trigger activates whenever a new row is inserted into the table; for example, **through INSERT statements**.

**UPDATE**: The trigger activates whenever a row is modified; for example, **through UPDATE statements**.

**DELETE**: The trigger activates whenever a row is deleted from the table; for example, **through DELETE and REPLACE statements**.

## Remember:

- DROP TABLE and TRUNCATE TABLE statements on the table do not activate this trigger, because they do not use DELETE.
- Dropping a partition does not activate DELETE triggers, either.

## *before and after*

- If a BEFORE trigger fails, the operation on the corresponding row is not performed.
- A BEFORE trigger is activated by the attempt to insert or modify the row or delete the row, regardless of whether the attempt subsequently succeeds.
- An AFTER trigger is executed only if any BEFORE triggers and the row operation execute successfully.
- An error during either a BEFORE or AFTER trigger results in failure of the entire statement that caused trigger invocation.

## *new and old*

The **OLD** and **NEW** keywords enable you to access columns in the rows affected by a trigger. **OLD** and **NEW** are MySQL extensions to triggers; they are not case sensitive.

A column named with **OLD** is read only. In a BEFORE trigger, you can also change its value with `SET NEW.col_name = value`.

- In an INSERT trigger, only **NEW.col\_name** can be used; there is no **OLD** row.
- In a DELETE trigger, only **OLD.col\_name** can be used; there is no **NEW** row.
- In an UPDATE trigger, you can use **OLD.col\_name** to refer to the columns of a row before it is updated and **NEW.col\_name** to refer to the columns of the row after it is updated.

The trigger occurs. This can either be BEFORE or AFTER an INSERT, UPDATE or DELETE. A BEFORE trigger must be used if you need to modify incoming data. An AFTER trigger must be used if you want to reference the new/changed record as a foreign key for a record in another table.

**A column named with **OLD** is read only.**

## **NOTE**

- The trigger cannot use the **CALL** statement to invoke stored procedures that return data to the client or that use dynamic SQL. (Stored procedures are permitted to return data to the trigger through OUT or INOUT parameters.)

# *drop trigger*

This statement drops a trigger. The schema (database) name is optional. If the schema is omitted, the trigger is dropped from the default schema.

`DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name`

If you drop a table, any triggers for the table are also dropped.

As of MySQL 5.7.2, it is possible to define multiple triggers for a given table that have the same trigger event and action time. For example, you can have two BEFORE UPDATE triggers for a table. By default, triggers that have the same trigger event and action time activate in the order they were created.

## *example of trigger*

- DROP TRIGGER IF EXISTS triggername;  
delimiter \$\$  
CREATE TRIGGER triggername BEFORE INSERT ON dept FOR EACH ROW  
begin  
    SELECT 'Hello World'; #error  
end \$\$  
delimiter ;

ERROR 1415 (0A000): Not allowed to return a result set from a trigger

## *example of trigger*

- `DROP TRIGGER IF EXISTS triggername;`  
delimiter \$\$  
`CREATE TRIGGER triggername AFTER INSERT ON dept FOR EACH ROW`  
`begin`  
    `INSERT INTO d1 VALUES (NEW.deptno, NEW.dname, NEW.loc, NEW.pwd);`  
`end $$`  
delimiter ;
- `DROP TRIGGER IF EXISTS triggername;`  
delimiter \$\$  
`CREATE TRIGGER triggername AFTER UPDATE ON dept FOR EACH ROW`  
`begin`  
    `INSERT INTO d1 VALUES(OLD.dname, NEW.dname);`  
`end $$`  
delimiter ;
- `DROP TRIGGER IF EXISTS triggername;`  
delimiter \$\$  
`CREATE TRIGGER triggername AFTER DELETE ON dept FOR EACH ROW`  
`begin`  
    `INSERT INTO d VALUES(OLD.deptno, OLD.dname, OLD.loc, OLD.pwd, now(), user());`  
`end $$`  
delimiter ;

## *example of trigger*

### Difference between BEFORE and AFTER trigger.

- ```
DROP TRIGGER IF EXISTS triggername;
delimiter $$

CREATE TRIGGER triggername BEFORE INSERT ON
dept FOR EACH ROW
begin
declare x INT;
SELECT COUNT(*) INTO x FROM dept;
if x > 4 then
    signal sqlstate '42000' set message_text = 'error';
end if;
end $$

delimiter ;
```
- ```
DROP TRIGGER IF EXISTS triggername;
delimiter $$

CREATE TRIGGER triggername AFTER INSERT ON dept
FOR EACH ROW
begin
declare x INT;
SELECT COUNT(*) INTO x FROM dept;
if x > 4 then
    signal sqlstate '42000' set message_text = 'error';
end if;
end $$

delimiter ;
```

## *example of trigger*

### Difference between BEFORE and AFTER trigger.

- `CREATE TABLE temp(c1 INT PRIMARY KEY AUTO_INCREMENT, c2 INT);`
- `CREATE TABLE temp1(c1 INT);`
- `DROP TRIGGER IF EXISTS triggername;`  
`delimiter $$`  
`CREATE TRIGGER triggername BEFORE INSERT ON temp`  
`FOR EACH ROW`  
`begin`  
`INSERT INTO temp1 VALUES(NEW.c1);`  
`end $$`  
`delimiter ;`
- `DROP TRIGGER IF EXISTS triggername;`  
`delimiter $$`  
`CREATE TRIGGER triggername AFTER INSERT ON temp`  
`FOR EACH ROW`  
`begin`  
`INSERT INTO temp1 VALUES(NEW.c1);`  
`end $$`  
`delimiter ;`
- `INSERT INTO temp1 VALUES(DEFAULT, 100);`
- `SELECT * FROM temp1;`
- `INSERT INTO temp1 VALUES(DEFAULT, 100);`
- `SELECT * FROM temp1;`

## *example of trigger*

- `DROP TABLE IF EXISTS` dump, disk1, disk2, disk3;
- `CREATE TABLE` dump(`id INT` primary key auto\_increment, name `VARCHAR(20)`, salary `INT`, isActive `BOOL`);
- `CREATE TABLE` disk1(`id INT`, name `VARCHAR(20)`, salary `INT`, isActive `BOOL`);
- `CREATE TABLE` disk2(`id INT`, name `VARCHAR(20)`, salary `INT`, isActive `BOOL`);
- `CREATE TABLE` disk3(`id INT`, name `VARCHAR(20)`, salary `INT`, isActive `BOOL`);

• `DROP TRIGGER IF EXISTS` triggername;

delimiter \$\$

`CREATE TRIGGER` triggername `AFTER INSERT` ON dump `FOR EACH ROW`

`begin`

`declare x INT;`

`set x = NEW.id mod 3;`

`if x = 0 then`

`INSERT INTO disk3 VALUES(NEW.id, NEW.name, NEW.salary, NEW.isActive);`

`elseif x = 2 then`

`INSERT INTO disk2 VALUES(NEW.id, NEW.name, NEW.salary, NEW.isActive);`

`elseif x = 1 then`

`INSERT INTO disk1 VALUES(NEW.id, NEW.name, NEW.salary, NEW.isActive);`

`end if;`

`end $$`

delimiter ;

## *example of trigger*

- ```
DROP TRIGGER IF EXISTS triggername;
delimiter $$

CREATE TRIGGER triggername AFTER INSERT ON dept FOR EACH ROW
begin
    INSERT INTO d VALUES (NEW.deptno, NEW.dname, NEW.loc, NEW.pwd, now(), user());
end $$

delimiter ;
```
- ```
DROP TRIGGER IF EXISTS triggername;
delimiter $$

CREATE TRIGGER triggername AFTER INSERT ON dept FOR EACH ROW
begin
    if DATE_FORMAT (now(), '%W') = 'Wednesday' then
        signal sqlstate '42000' set message_text = 'Invalid';
    end if;
end $$

delimiter ;
```

## *example of trigger*

- `DROP TRIGGER IF EXISTS triggername;`  
delimiter \$\$  
`CREATE TRIGGER triggername BEFORE INSERT ON dept FOR EACH ROW`  
`begin`  
    `declare msg VARCHAR(100);`  
    `if NEW.dname = 'A' then`  
        `set NEW.dname = 'Apple';`  
    `else`  
        `set msg = 'My error message';`  
        `signal sqlstate '42000' set message_text = msg;`  
    `end if;`  
`end $$`  
delimiter ;
- `DROP TRIGGER IF EXISTS triggername;`  
delimiter \$\$  
`CREATE TRIGGER triggername BEFORE INSERT ON dept FOR EACH ROW`  
`begin`  
    `if NEW.deptno < 50 then`  
        `signal sqlstate '42000' set message_text = 'Invalid department number';`  
    `end if;`  
`end $$`  
delimiter ;

## *example of trigger*

- ```
DROP TRIGGER IF EXISTS triggername;
delimiter $$

CREATE TRIGGER triggername BEFORE INSERT ON dept FOR EACH ROW
begin
    if NEW.deptno < 50 then
        signal sqlstate '42000' set message_text = 'Invalid department number';
    end if;
end $$

delimiter ;
```
- ```
DROP TRIGGER IF EXISTS triggername;
delimiter $$

CREATE TRIGGER triggername BEFORE INSERT ON emp FOR EACH ROW
begin
    if NEW.sal <= 0 then
        set NEW.sal = 25000;
    end if;
end $$

delimiter ;
```

ERROR 1362 (HY000): Updating of NEW row is not allowed in after trigger

```
mysql> INSERT INTO emp (empno, ename, sal, mgr, deptno) VALUES(1, 'abc', -10000, 7788, 10);
```

## *example of trigger*

- `DROP TRIGGER IF EXISTS triggername;`  
delimiter \$\$  
`CREATE TRIGGER triggername BEFORE INSERT ON city FOR EACH ROW`  
`begin`  
    `declare msg VARCHAR(100);`  
    `if NEW.c1 <> 'Pune' then`  
        `set msg = 'Invalid city name';`  
        `signal sqlstate '42000' set message_text = msg;`  
    `end if;`  
`end $$`  
delimiter ;
- `DROP TRIGGER IF EXISTS triggername;`  
delimiter \$\$  
`CREATE TRIGGER triggername BEFORE INSERT ON dept FOR EACH ROW`  
`begin`  
    `set NEW.dname = upper(NEW.dname);`  
`end $$`  
delimiter ;  
  
`mysql> INSERT INTO dept VALUES(2, 'abc', 2, 2);`

## *example of trigger*

- `DROP TRIGGER IF EXISTS triggername;`  
delimiter \$\$  
`CREATE TRIGGER triggername BEFORE INSERT ON dept FOR EACH ROW`  
`begin`  
    `declare x INT default 0;`  
    `SELECT MAX(id) + 1 INTO x FROM log;`  
    `INSERT INTO log VALUES(x, 'Data inserted', curdate(),curtime(), user());`  
`end $$`  
delimiter ;
- `DROP TRIGGER IF EXISTS triggername;`  
delimiter \$\$  
`CREATE TRIGGER triggername BEFORE INSERT on dept FOR EACH ROW`  
`begin`  
    `declare x INT;`  
    `set x = (SELECT MAX(deptno) + 1 FROM dept);`  
    `set NEW.deptno = x;`  
`end $$`  
delimiter ;

## *example of trigger*

```
CREATE TABLE tempCustomer(  
    customerID VARCHAR(45),  
    customerName VARCHAR(45)  
);
```

```
INSERT INTO tempCustomer values('Cust-1001', 'saleel');  
INSERT INTO tempCustomer values('Cust-2001', 'sharmin');  
INSERT INTO tempCustomer values('Cust-003', 'ruhan');  
INSERT INTO tempCustomer values('Cust-0075', 'sangita');  
INSERT INTO tempCustomer values('Cust-75', 'bandish');  
INSERT INTO tempCustomer values('Cust-5', 'vrushali');
```

- DROP TRIGGER IF EXISTS triggerName;  
delimiter \$  
CREATE TRIGGER triggerName BEFORE INSERT ON tempCustomer FOR EACH ROW  
begin  
 declare x INT default 0;  
 SELECT MAX(CAST(SUBSTR(customerID, 6) AS signed)) + 1 INTO x FROM tempCustomer;  
 set NEW.customerID := concat('Cust-', x);  
end \$  
delimiter ;

## *example of trigger*

- `DROP TRIGGER IF EXISTS triggername;`  
`delimiter $$`  
`CREATE TRIGGER triggername BEFORE INSERT ON emp FOR EACH ROW`  
`begin`  
    `declare x INT;`  
    `set x = (SELECT deptno FROM dept WHERE deptno = NEW.deptno);`  
    `if x is null then`  
        `signal sqlstate '42000' set message_text = 'error';`  
    `end if;`  
`end $$`  
`delimiter ;`
- `DROP TRIGGER IF EXISTS triggername;`  
`delimiter $$`  
`CREATE TRIGGER triggername BEFORE INSERT ON dept FOR EACH ROW`  
`begin`  
    `declare x INT;`  
    `SELECT MAX(deptno) + 1 INTO x FROM dept;`  
    `set NEW.deptno := x;`  
`end $$`  
`delimiter ;`

## *example of trigger*

- `DROP TRIGGER IF EXISTS triggername;`  
delimiter \$\$  
`CREATE TRIGGER triggername BEFORE INSERT ON dept FOR EACH ROW`  
`begin`  
    `set NEW.dname = TRIM(NEW.dname);`  
    `set NEW.loc = TRIM(NEW.loc);`  
    `set NEW.pwd = TRIM(NEW.pwd);`  
`end $$`  
delimiter ;

# *item, warehouse , item\_ordered, ...      example of trigger*

- DROP TABLE IF EXISTS item;
  - DROP TABLE IF EXISTS warehouse;
  - DROP TABLE IF EXISTS item\_in\_warehouse;
  - DROP TABLE IF EXISTS item\_ordered;
- 
- CREATE TABLE item(itemid INT, itemname VARCHAR(20));
  - CREATE TABLE warehouse(warehouse\_id INT, warehouse\_name VARCHAR(255), channel\_id INT);
  - CREATE TABLE item\_in\_warehouse(id INT PRIMARY KEY AUTO\_INCREMENT, item\_id INT, warehouse\_id INT, minimum\_stock INT, rol INT, stock INT);
  - CREATE TABLE item\_ordered(orderID INT, itemID INT, orderDate DATE, orderTime TIME, qty INT);

*Item, warehouse , item\_ordered, ...*

*example of trigger*

- INSERT INTO item VALUES(1, 'SPORTS SHOES');
  - INSERT INTO item VALUES(2, 'CASUAL SHOES');
  - INSERT INTO item VALUES(3, 'T-SHIRTS');
  - INSERT INTO item VALUES(4, 'JEANS');
  - INSERT INTO item VALUES(5, 'JACKETS');
  - INSERT INTO item VALUES(6, 'SWEATERS');
  - INSERT INTO item VALUES(7, 'WATCHES');
- 
- INSERT INTO warehouse VALUES( 143, 'AC Warehouse', 1);
  - INSERT INTO warehouse VALUES( 156, 'National', 2);
  - INSERT INTO warehouse VALUES( 231, 'Global', 3);
  - INSERT INTO warehouse VALUES( 254, 'NON-STOP', 2);
  - INSERT INTO warehouse VALUES( 321, 'Migrant System', 2);
  - INSERT INTO warehouse VALUES( 464, 'Blaze', 1);

# *Item, warehouse , item\_ordered, ...      example of trigger*

- `INSERT INTO item_in_warehouse (item_id, warehouse_id) VALUES (1, 143);`
- `INSERT INTO item_in_warehouse (item_id, warehouse_id) VALUES (2, 156);`
- `INSERT INTO item_in_warehouse (item_id, warehouse_id) VALUES (3, 231);`
- `INSERT INTO item_in_warehouse (item_id, warehouse_id) VALUES (5, 254);`
- `INSERT INTO item_in_warehouse (item_id, warehouse_id) VALUES (3, 321);`
- `INSERT INTO item_in_warehouse (item_id, warehouse_id) VALUES (1, 464);`
- `INSERT INTO item_in_warehouse (item_id, warehouse_id) VALUES (6, 156);`
- `INSERT INTO item_in_warehouse (item_id, warehouse_id) VALUES (1, 156);`
- `INSERT INTO item_in_warehouse (item_id, warehouse_id) VALUES (7, 464);`
- `INSERT INTO item_in_warehouse (item_id, warehouse_id) VALUES (6, 321);`
- `INSERT INTO item_in_warehouse (item_id, warehouse_id) VALUES (5, 464);`
- `INSERT INTO item_in_warehouse (item_id, warehouse_id) VALUES (4, 321);`
- `INSERT INTO item_in_warehouse (item_id, warehouse_id) VALUES (3, 464);`
- `INSERT INTO item_in_warehouse (item_id, warehouse_id) VALUES (6, 231);`

# *Item, warehouse , item\_ordered, ...      example of trigger*

- DROP TRIGGER IF EXISTS insert\_in\_item\_ordered;  
delimiter \$\$  
CREATE TRIGGER insert\_in\_item\_ordered BEFORE INSERT ON item\_ordered FOR EACH ROW  
begin  
declare x INT;  
set NEW.orderDate := current\_date();  
set NEW.orderTime := current\_time();  
SELECT itemID INTO x FROM item WHERE itemID = NEW.itemID;  
if x is null then  
    signal sqlstate '42000' set message\_text = "Item not found in item table!";  
end if;  
end \$\$  
delimiter ;

# *Item, warehouse , item\_ordered, ...      example of trigger*

- DROP TRIGGER IF EXISTS itemid\_primarykey;

delimiter \$\$

```
CREATE TRIGGER itemid_primarykey AFTER INSERT ON item FOR EACH ROW
begin
```

```
declare x INT;
```

```
SELECT itemid INTO x FROM item WHERE itemid = NEW.itemid;
```

```
if x is null then
```

```
    signal sqlstate '42000' set message_text = 'itemid already present, cannot insert duplicate itemid.';
```

```
end if;
```

```
end $$
```

```
delimiter ;
```

- DROP TRIGGER IF EXISTS readOnlyItemName;

delimiter \$\$

```
CREATE TRIGGER readOnlyItemName AFTER UPDATE ON item FOR EACH ROW
begin
```

```
    signal sqlstate '42000' set message_text = 'Product name is read-only and it can not be changed.';
```

```
end $$
```

```
delimiter ;
```

# *Item, warehouse , item\_ordered, ...*

## *example of trigger*

- DROP TRIGGER IF EXISTS insert\_item\_in\_warehouse;  
delimiter \$\$  
CREATE TRIGGER insert\_item\_in\_warehouse BEFORE INSERT ON item\_in\_warehouse FOR EACH ROW  
begin  
declare itemNotFound BOOL default True;  
declare warehouseNotFound BOOL default True;  
SELECT False INTO warehouseNotFound FROM item WHERE itemid = NEW.item\_id;  
SELECT False INTO warehouseFound FROM warehouse WHERE warehouse\_id = NEW.warehouse\_id;  
if itemNotFound then  
    signal sqlstate '42000' set message\_text = 'Item not found!';  
end if;  
if warehouseNotFound then  
    signal sqlstate '42000' set message\_text = 'Warehouse not found!';  
end if;  
end \$\$  
delimiter ;

*Item, warehouse , item\_ordered, ...*

*example of trigger*

```
DROP TRIGGER IF EXISTS calculate_minimum_stock_ROL;
```

```
delimiter $$
```

```
CREATE TRIGGER calculate_minimum_stock_ROL BEFORE INSERT ON item_in_warehouse FOR EACH ROW
begin
```

```
declare _minimum_stock INT;
```

```
declare _ROL INT;
```

```
set _minimum_stock := NEW.stock / 2;
```

```
set _ROL := _minimum_stock * .25;
```

```
set NEW.minimum_stock := _minimum_stock;
```

```
set NEW.ROL := _ROL;
```

```
end $$
```

```
delimiter ;
```

- `DROP TABLE IF EXISTS customer_Account;`
- `CREATE TABLE customer_Account (`  
    `accountID VARCHAR(20),`  
    `customerName VARCHAR(20),`  
    `account_Type VARCHAR(20),`  
    `openDate DATE,`  
    `phone_number VARCHAR(12)`  
`);`

# Bank

## example of trigger

- ```
DROP TRIGGER IF EXISTS customer_AccountID;
delimiter $$

CREATE TRIGGER customer_AccountID BEFORE INSERT ON customer_account FOR EACH ROW
begin
    declare x int;
    SELECT ifnull(max(cast(substr(accountID, 5) AS UNSIGNED)),0 ) + 1 INTO x FROM customer_account
    WHERE account_Type = NEW.account_Type;

    if NEW.account_Type = 'saving' then
        set NEW.accountID = concat('S/A-', x);
    elseif NEW.account_Type = 'current' then
        set NEW.accountID = concat('C/A-', x);
    else
        signal sqlstate '42000' set message_text = "Invalid Account Type!";
    end if;
end $$

delimiter ;
```

# Bank

## example of trigger

- ```
DROP TRIGGER IF EXISTS customer_phone;
delimiter $$

CREATE TRIGGER customer_phone BEFORE INSERT ON customer_account FOR EACH ROW
begin
    set NEW.phone_number := rpad(left(NEW.phone_number, 4), 10, 'x');
end $$

delimiter ;
```
- ```
DROP TRIGGER IF EXISTS open_date;
delimiter $$

CREATE TRIGGER open_date BEFORE INSERT ON customer_account FOR EACH ROW
begin
    if NEW.opendate > curDate() then
        signal sqlstate '42000' set message_text = 'Invalid Opening Date!';
    end if;
end $$

delimiter ;
```

# Bank

## example of trigger

- DROP PROCEDURE IF EXISTS showCustomerAccount;  
delimiter \$\$  
CREATE PROCEDURE showCustomerAccount()  
begin  
    SELECT \* FROM customer\_Account;  
end \$\$  
delimiter ;
- DROP TRIGGER IF EXISTS lock\_issue\_card;  
delimiter \$\$  
CREATE TRIGGER lock\_issue\_card BEFORE INSERT ON credit\_card FOR EACH ROW  
begin  
    if @TRIGGER\_DISABLED is False or cast(@TRIGGER\_DISABLED as char) is null then  
        set @TRIGGER\_DISABLED := True;  
        signal sqlstate '42000' set message\_text = 'You cannot insert data in issue\_card table';  
    end if;  
end \$\$  
delimiter ;

# Bank

## example of trigger

- ```
DROP TRIGGER IF EXISTS issue_card;
delimiter $$

CREATE TRIGGER issue_card BEFORE INSERT ON credit_card FOR EACH ROW
begin
    declare x VARCHAR(20);
    if @TRIGGER_DISABLED is True then
        set @TRIGGER_DISABLED := False;
        SELECT accountID INTO x FROM customer_Account WHERE accountID = NEW.accountID;
        if x is null then
            signal sqlstate '42000' set message_text = 'Invalid Account Number, card cannot be issued!
                "Contact customer care!"';
        end if;
    end if;
end $$

delimiter ;
```

# Bank

## *example of trigger*

- DROP PROCEDURE IF EXISTS issueCard;

delimiter \$\$

```
CREATE PROCEDURE issueCard(in _accountID VARCHAR(20), _issueDate DATE, _pin INT, _isactive BOOL)
begin
```

```
    if @TRIGGER_DISABLED is True then
```

```
        INSERT INTO credit_card VALUES(_accountID, _issueDate, _pin, _isactive);
```

```
    end if;
```

```
end $$
```

```
delimiter ;
```

TODO

# SQL Injection

TODO

# Injection in Insert/Update

TODO

Load- XML

# XML file

## XML File

```
<?xml version="1.0" encoding="utf-16"?>
<employees>
    <employee>
        <id> be129 </id>
        <firstname> Jane </firstname>
        <lastname> Doe </lastname>
        <title> Engineer </title>
    </employee>

    <employee>
        <id> be130 </id>
        <firstname> William </firstname>
        <lastname> Defoe </lastname>
        <title> Accountant </title>
    </employee>
</employees>
```

# *Table structure for XML file and load XML file.*

Example:

```
CREATE TABLE `employee` (
    `id` varchar (45) DEFAULT NULL,
    `firstname` varchar (45) DEFAULT NULL,
    `lastname` varchar (45) DEFAULT NULL,
    `title` varchar (45) DEFAULT NULL
);
```

```
LOAD XML INFILE 'C:\\EMPLOYEE.XML' INTO TABLE employee ROWS
IDENTIFIED BY '<employee>';
```

```
SELECT * FROM employee;
```

# Select data from XML using MySQL ExtractValue function

Example:

```
SELECT ExtractValue( '<?xml version="1.0" encoding="UTF-8"?>
<student>
    <p1>
        <id>1001</id>
        <name> Saleel Bagde </name>
        <email> saleelbagde@gmail.com </email>
        <email> saleel.bagde@yahoo.com </email>
    </p1>
    <p1>
        <id>1002</id>
        <name> Sharmin Bagde </name>
        <email> sharminbagde@gmail.com </email>
    </p1>
</student>', '//student//p1[$1]//email[$2]');
```

# *Inserting XML script in table.*

Example:

```
CREATE TABLE `xmlTable` ( `xmlData` text);

INSERT INTO XMLTABLE VALUES ("  
<Students>  
    <student1>  
        <RollNumber>1</RollNumber>  
        <Name> Saleel </Name>  
        <email> saleelbagde@gmail.com </email>  
        <email> saleelbagde@yahoo.com </email>            </student1>  
    <student2>  
        <RollNumber> 2 </RollNumber>  
        <Name> Vrushali </Name>  
        <email> vrushalibagde@gmail.com </email>  
    </student2>  
    <student3>  
        <RollNumber> 3 </RollNumber>  
        <Name> Sharmin </Name>  
        <email> sharminbagde@gmail.com </email>  
    </student3>  
</Students>");
```

## *Select from XML script from table.*

Example:

```
SELECT EXTRACTVALUE (xmlData,"//Students//student1//email[$2]") FROM  
XMLTABLE;
```

# *Inserting XML script in table.*

Example:

```
CREATE TABLE `xmlTable` ( `xmlData` text);

INSERT INTO XMLTABLE VALUES ("  
<Students>  
    <student>  
        <RollNumber>1 </RollNumber>  
        <Name> Saleel </Name>  
        <email> saleelbagde@gmail.com </email>  
        <email> saleelbagde@yahoo.com </email>  
    </student>  
    <student>  
        <RollNumber> 2 </RollNumber>  
        <Name> Vrushali </Name>  
        <email> vrushalibagde@gmail.com </email>  
    </student>  
    <student>  
        <RollNumber> 3 </RollNumber>  
        <Name> Sharmin </Name>  
        <email> sharminbagde@gmail.com </email>  
    </student>  
</Students>");
```

## *Select from XML script from table.*

Example:

```
SELECT EXTRACTVALUE(xmlData,"//Students//student[$1]//email[$2]") FROM  
XMLTABLE;
```

# XML file

## Employee.xml File

```
<?xml version="1.0" encoding="utf-16"?>
<employees>
    <employee>
        <id> be129 </id>
        <firstname> Jane </firstname>
        <lastname> Doe </lastname>
        <title> Engineer </title>
    </employee>

    <employee>
        <id> be130 </id>
        <firstname> William </firstname>
        <lastname> Defoe </lastname>
        <title> Accountant </title>
    </employee>
</employees>
```

## *Load and select data from external XML file*

Example:

```
SET @xml = LOAD_FILE("E:/employee.xml");
SELECT CONVERT(EXTRACTVALUE(@xml,"//employee[$1]//firstname"),
CHAR);
```

# *IMP SQL statements*

**1. Find 2<sup>nd</sup> highest salary.**

```
SELECT MAX(sal) FROM emp WHERE sal < (SELECT MAX(sal) FROM emp);
```

**2. Find 2<sup>nd</sup> lowest salary.**

```
SELECT MIN(sal) FROM emp WHERE sal > (SELECT MIN(sal) FROM emp);
```

**3. Find 2<sup>nd</sup> highest salary of each department.**

```
SELECT MAX(sal) FROM emp WHERE sal NOT IN (SELECT MAX(sal) FROM emp GROUP BY deptno) GROUP BY deptno;
```

**4. Find 2<sup>nd</sup> lowest salary of each department.**

```
SELECT MIN(sal) FROM emp WHERE sal NOT IN (SELECT MIN(sal) FROM emp GROUP BY deptno) GROUP BY deptno;
```

**5. Find top 3 highest paid salary**

```
SELECT * FROM emp WHERE sal > (SELECT sal FROM emp GROUP BY sal ORDER BY sal DESC limit 3, 1) ORDER BY sal DESC;
```

**6. Serial number jobwise.**

```
SELECT @CNT := CASE WHEN job = @JB THEN @CNT + 1 ELSE 1 END R1, @JB := JOB FROM emp, (SELECT @CNT :=0, @JB := '') E ORDER BY job;
```

# *IMP SQL statements*

## **6. Find only the duplicate records.**

```
SELECT * FROM (SELECT row_number() over(partition by deptno, dname, loc, walletid) R1, duplicate.* FROM duplicate)
t1 WHERE r1>1;
```

## **7. Delete only the duplicate records.**

```
DELETE FROM duplicate WHERE id NOT IN (SELECT * FROM (SELECT MIN(id) FROM duplicate GROUP BY deptno, dname,
loc, walletid) d );
```

```
DELETE FROM duplicate WHERE id NOT IN (SELECT * FROM (SELECT id FROM duplicate GROUP BY deptno, dname, loc,
walletid) t1 );
```

## **8. Display employee name with highest salary in each job.**

```
SELECT ename, job, sal FROM emp WHERE (job, sal) IN (SELECT job, MAX(sal) FROM emp GROUP BY job) ORDER BY job;
```

## **9. Display employee having the same salary**

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

# *Interview questions*

- What is the difference between CHAR and VARCHAR?
- What is the difference between DELETE, DROP, AND TRUNCATE.
- What is the difference between DELETE TABLE and TRUNCATE TABLE commands in MySQL?
- What is the difference between Inner Join and Natural Join.
- What are types of joins in MySQL?
- What is the difference between primary key and unique key?
- What do you mean by Joins and explain different types of MySQL Joins?

select count(\*), e.\* from e group by empno, ename, job, mgr, hiredate, sal, comm, deptno, bonusid, `user name`, pwd;

select \* from plumber, service, plumber\_service\_map where pid = plumberId and serviceID = sid and pid in (select plumberid from plumber\_service\_map where serviceid in (1, 2, 7) group by plumberid having count(\*) =(select max(RI) from (select plumberid, count(plumberid) RI from plumber\_service\_map where serviceid in (1, 2, 7) group by plumberid) t1)) order by plumberid, serviceid;

# IMP SQL statements

TABLET  
Table

Id	Name	Ingredient	Unit
1	CROCIN	PARACETAMOL	100mg
2	COMBIFLAM	PARACETAMOL, IBUPROFEN	25mg
3	DIVON PLUS	PARACETAMOL, DICLOFENAC	30mg

SYRUP  
Table

Id	Name	Ingredient	Unit
11	BENADRYL	DIPHENHYDRAMINE	.7mg
12	COMBIFLAM	PARACETAMOL, IBUPROFEN	0.12mg
13	COREX	CHLORPHENAMINE	2.3mg
14	DICLO	DICLOFENAC	0.06mg

INJECTION  
Table

Id	Name	Ingredient	Unit
21	BRUFEN	IBUPROFEN	0.10mg
22	CPM	CHLORPHENAMINE	0.25mg
23	VOVERAN	DICLOFENAC	0.09mg

1. Think about how multiplication can be done without actually multiplying

$$7 * 4 = 28$$

$$7 + 7 + 7 + 7 = 28$$

$$5 * 6 = 30$$

$$5 + 5 + 5 + 5 + 5 + 5 = 30$$

2. Square

$$1^2 = (1) = 1$$

$$2^2 = (1 + 3) = 4$$

$$3^2 = (1 + 3 + 5) = 9$$

$$4^2 = (1 + 3 + 5 + 7) = 16$$

"Live as if you were to die tomorrow.  
Learn as if you were to live forever"



## *shell commands*

```
MySQL JS > function fn(x ,y) {  
    -> print(x + y);  
    -> }  
    ->  
MySQL JS > fn(10,20);
```

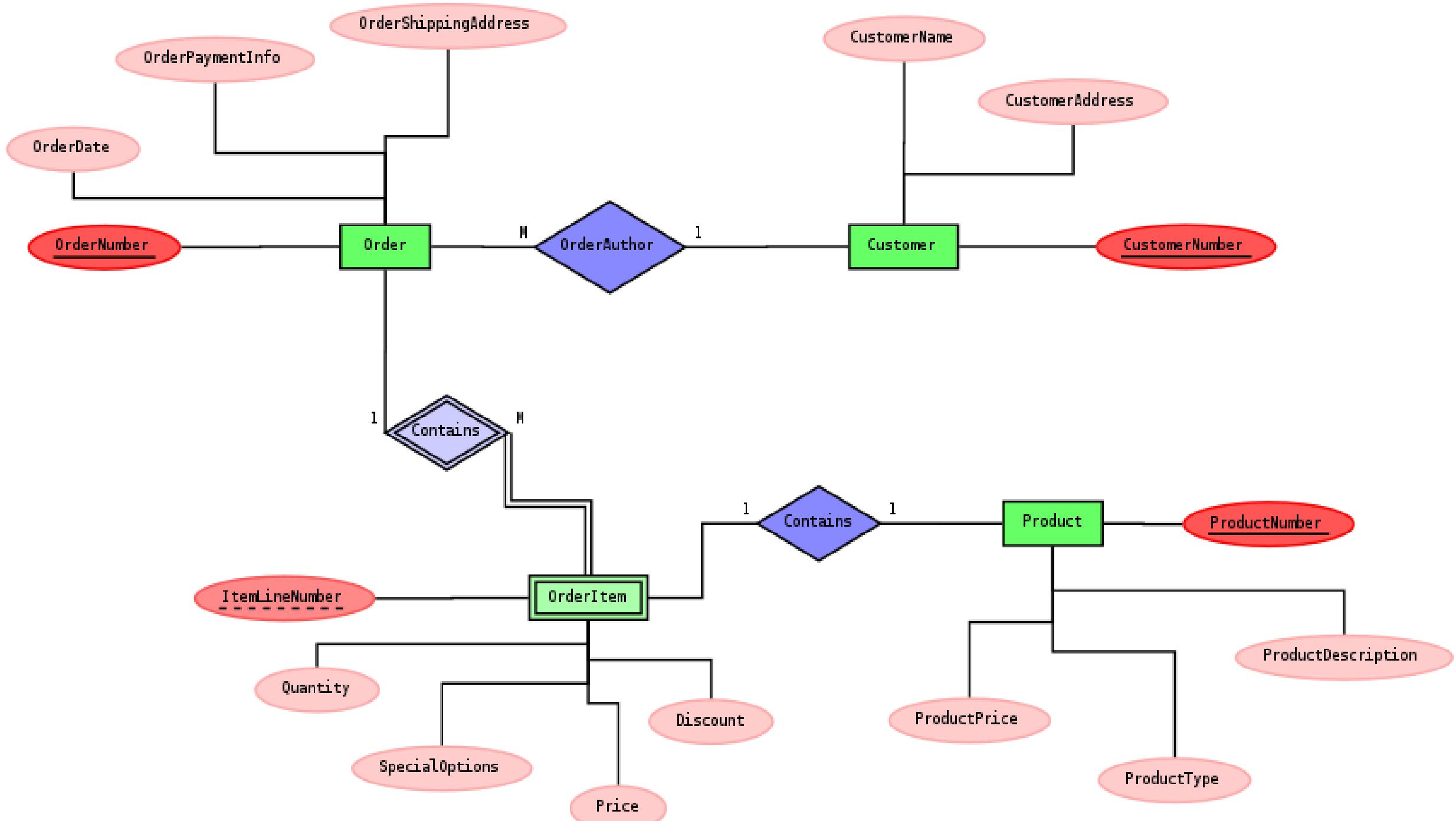
```
MySQL JS > \SQL  
Switching to SQL mode... Commands end with ;
```

```
MySQL SQL > \connect root@localhost  
Creating a session to 'root@localhost'  
Please provide the password for 'root@localhost': ****  
Save password for 'root@localhost'? [Y]es/[N]o/Ne[v]er (default No): n  
Fetching global names for auto-completion... Press ^C to stop.  
Your MySQL connection id is 18 (X protocol)  
Server version: 8.0.31 MySQL Community Server - GPL  
No default schema selected; type \use <schema> to set one.  
MySQL localhost:33060+ ssl SQL > USE DB1;
```

```
select * from plumber,service,plumber_service_map where pid = plumberId and serviceID = sid and pid in (select  
plumberid from plumber_service_map where serviceid in (1,2,7) group by plumberid having count(*) =(select max(R1)  
from (select plumberid,count(plumberid) R1 from plumber_service_map where serviceid in (1,2,7) group by plumberid)  
t1)) order by plumberid,serviceid;
```

## Binary Numbers

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, and so on.



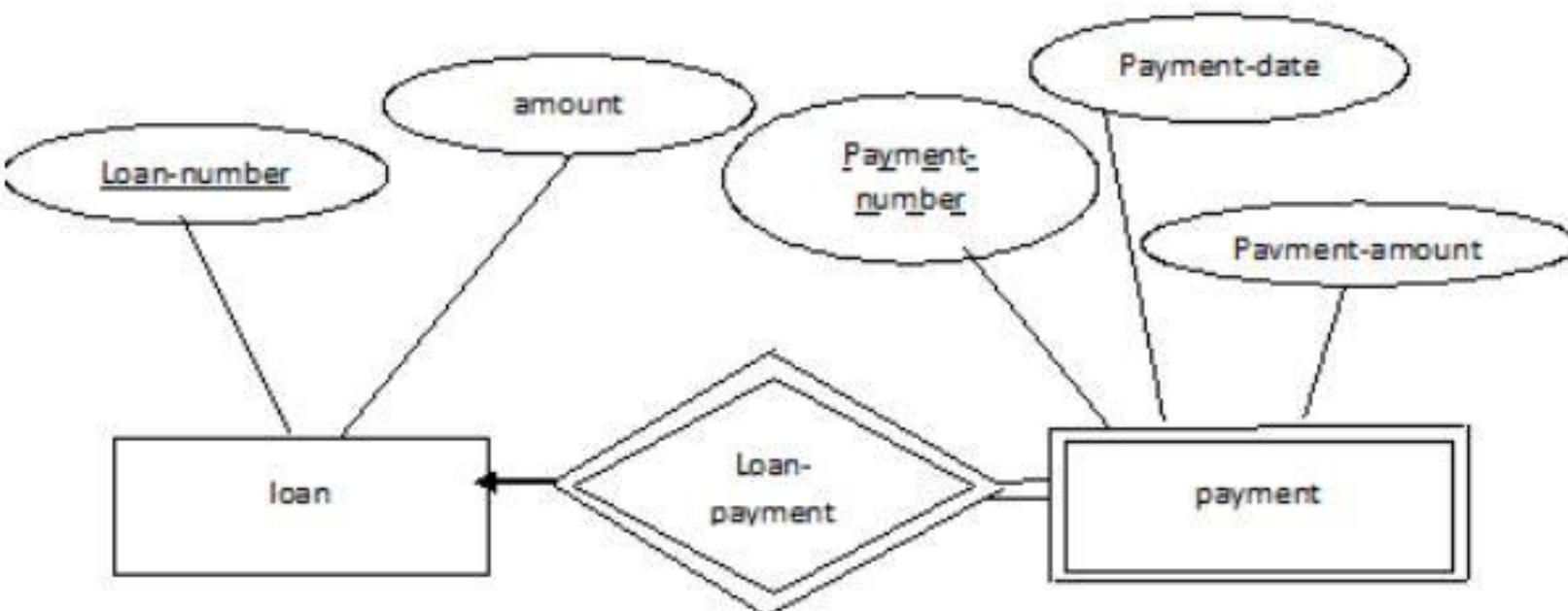
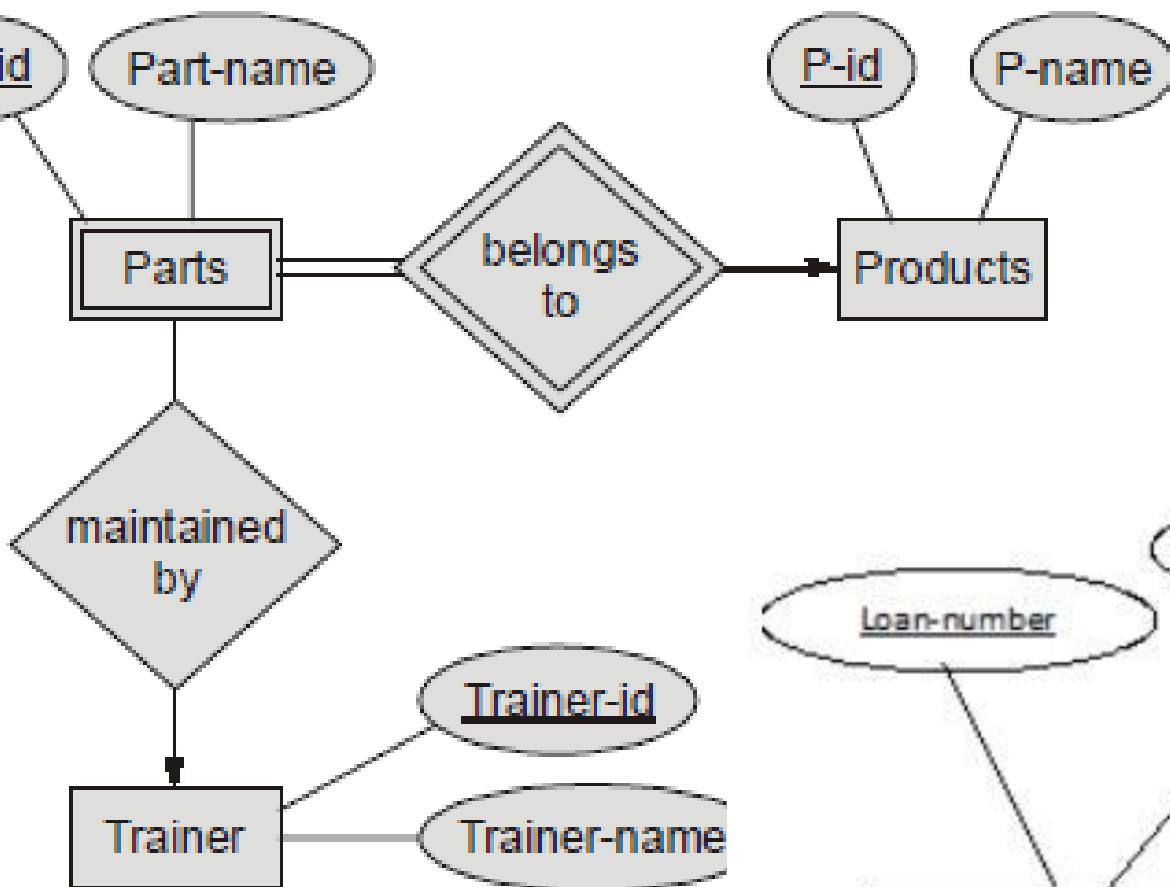


Fig1: Example of weak entity set.

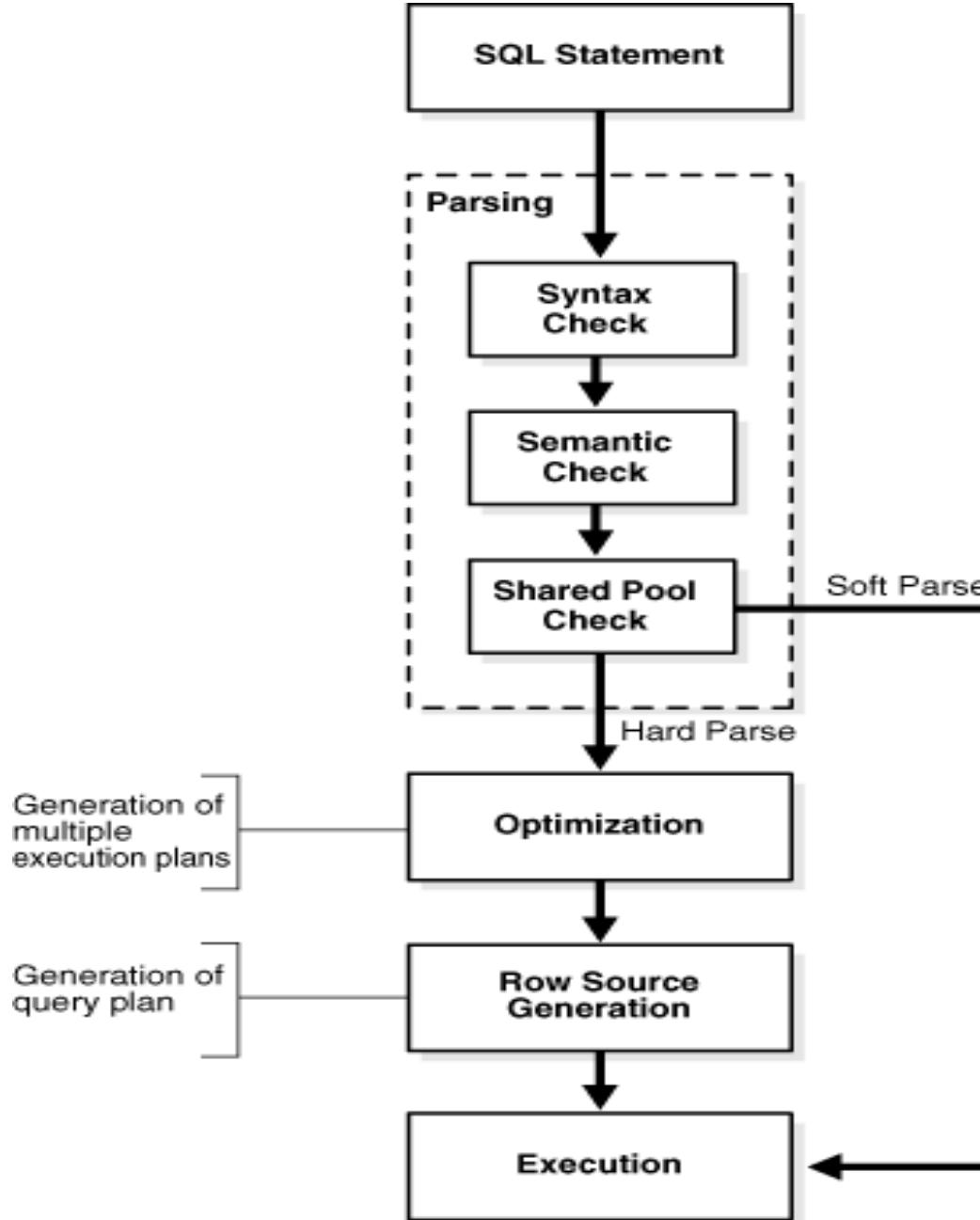
**Table DEPARTMENTS**

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
60	IT	103	1400
90	Executive	100	1700

Projection  
Selection

**Table EMPLOYEES**

EMPLOYEE_ID	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	MANAGER_ID	DEPARTMENT_ID
100	King	SKING		AD_PRES	90	
101	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	100	90
102	De Hann	LDEHANN	13-JAN-93	AD_VP	100	90
103	Hunold	AHUNOLD		IT_PROG	102	60



		item(types)		
		Mumbai	New Delhi	Gurgaon
Time (Quarter)	Q1	986	567	875
	Q2	786	85	987
	Q3			908
	Q4	788	987	765
		678	654	987
		899	875	190
		787	969	908
		436	108	237
		836	987	

## Create Tables:

**Product:** This table will store information about each product item.

Columns: product\_id (primary key), product\_name, description, price, etc.

**Quarter:** This table will store information about quarters or time periods.

Columns: quarter\_id (primary key), year, quarter\_name, start\_date, end\_date, etc.

**Location:** This table will store information about different locations.

Columns: location\_id (primary key), location\_name, address, city, country, etc.

**Sales:** This table will store the sales data, including references to the Product, Quarter, and Location tables.

Columns: sale\_id (primary key), product\_id (foreign key referencing Product table), quarter\_id (foreign key referencing Quarter table), location\_id (foreign key referencing Location table), quantity\_sold, revenue, etc.

```
drop procedure if exists pl1;
delimiter $
CREATE PROCEDURE pl1()
BEGIN
    declare d varchar(20);
    declare cnt int default 1;
    declare x varchar(2000) default "";
    declare c1 cursor for select dname from dept;
    declare exit handler for 1329 select 'end';
    open c1;
    lbl:LOOP
        fetch c1 into d;
        set cnt :=1;
    lbl2:LOOP
        if cnt <= length(d) THEN
            set x := concat(x, substr(d, cnt, 1), ", ");
        else
            leave lbl2;
        end if;
        set cnt := cnt + 1;
    end loop lbl2;
    SELECT x;
    set x="";
end loop lbl;
close c1;
end $
delimiter ;
```