A day without new knowledge is a lost day.

Database Technologies - MySQL

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

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

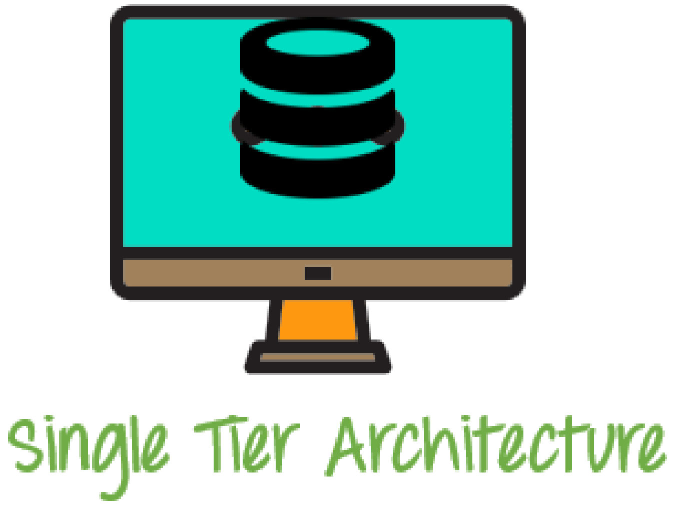
Introduction

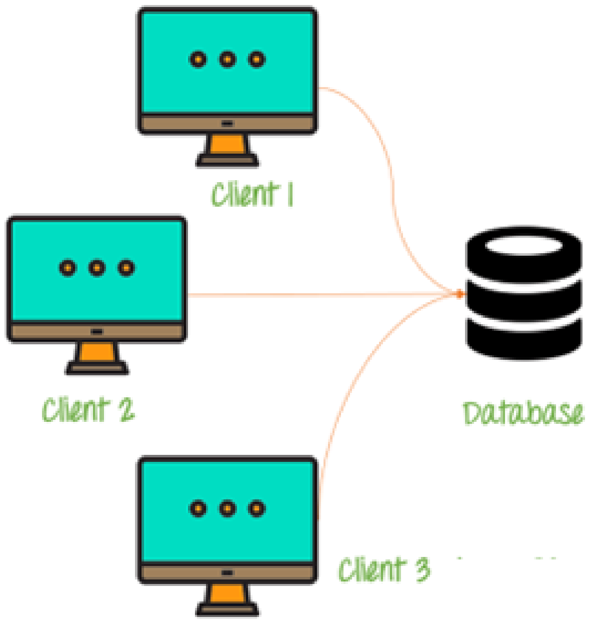
- If anyone who wants to develop a good application
  then he should have the knowledge three major components.

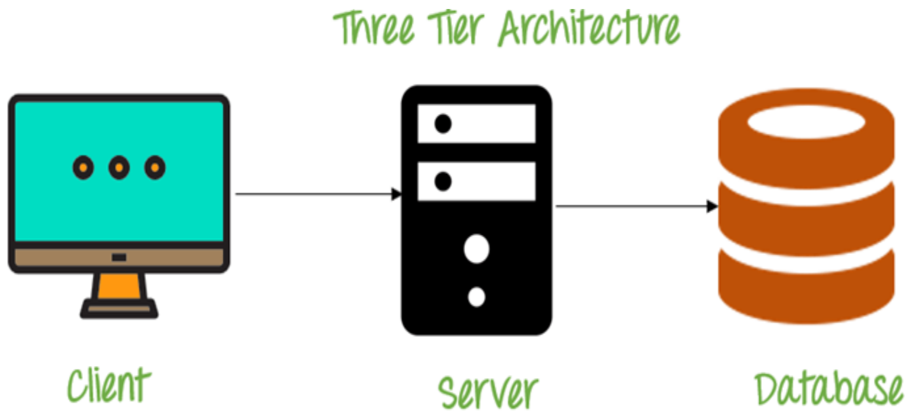
They are . . . . . .

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

  NoSQL }







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.)

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.

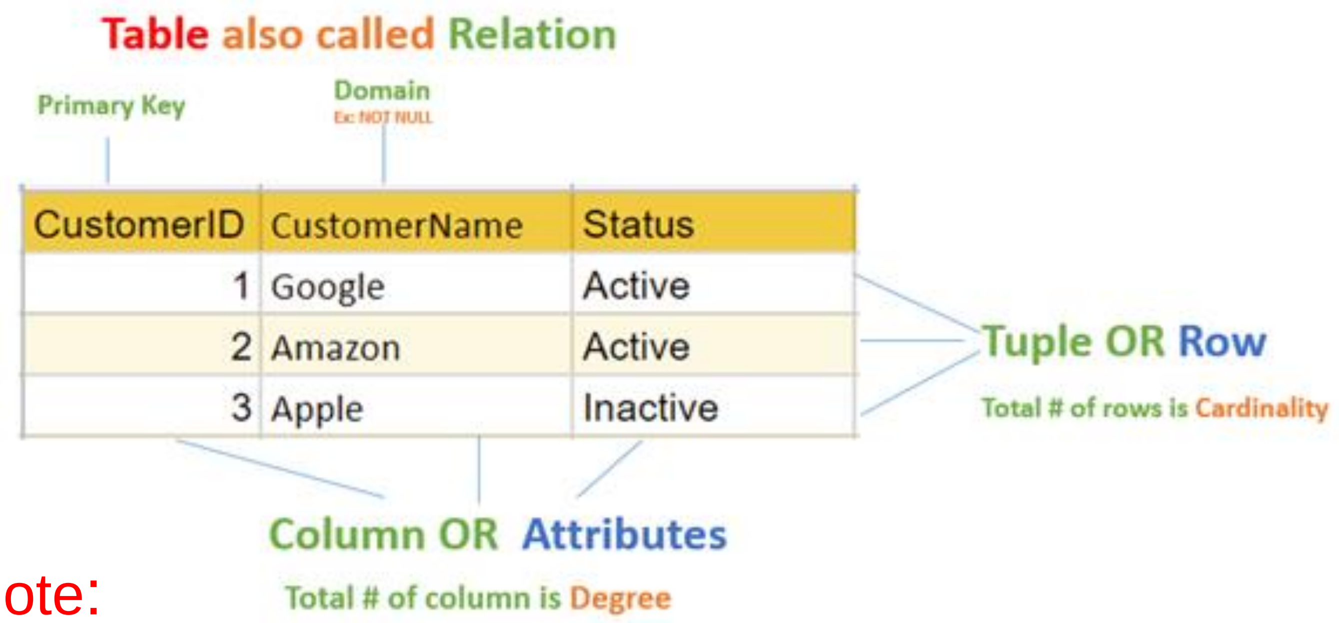


Table also called Relation
Primary Key
Domain
Ex: NOT NULL
Tuple OR Row
Total # of rows is Cardinality
Column OR Attributes
Total # of column is Degree

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive

Remember:

Foreign Key is also know as
- referential constraint
- referential integrity constraint.

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

file-oriented system

File Anomalies

```
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];
```

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
```

file-oriented system

File Anomalies

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

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

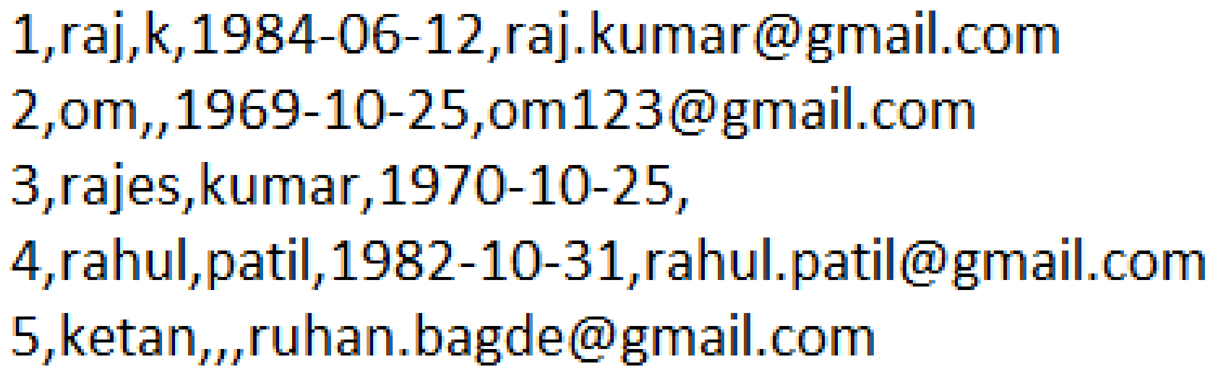
```
1 suraj 4000
2 ramesh 6000
3 rajan 4500
.        |
.        |
.        |
500 sam 3500
.        |
.        |
.        |
1000 amit 2300
```

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.
- **Editing***:* It is easy to edit any information stored in computers in form of files.
- **Remote access***:* It is possible to access data from remote location.
- **Sharing***:* The files stored in systems can be shared among multiple users at a same time.

disadvantage of file-oriented system

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

Disadvantage of File-oriented system

- **Data redundancy***:* It is possible that the same information may be duplicated in different files. This leads to data redundancy results in memory wastage.

(Suppose a customer having both kind of accounts- saving and current account. In such a situation a customers detail are stored in both the file, saving.txt- file and current.txt- file , which leads to Data Redundancy.)

- **Data inconsistency***:* Because of data redundancy, it is possible that data may not be in consistent state.

(Suppose customer changed his/her address. There might be a possibility that address is changed in only one file (saving.txt) and other (current.txt) remain unchanged.)

- **Limited data sharing***:* Data are scattered in various files and also different files may have different formats (for example: .txt, .csv, .tsv and .xml) and these files may be stored in different folders so, due to this it is difficult to share data among different applications.
- **Data Isolation:** Because data are scattered in various files, and files may be in different formats (for example: .txt, .csv, .tsv and .xml), writing new application programs to retrieve the appropriate data is difficult.
- **Data security:** Data should be secured from unauthorized access, for example a account holder in a bank should not be able to see the account details of another account holder, such kind of security constraints are difficult to apply in file processing systems.

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

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

DBMS

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

Difference between File System and DBMS

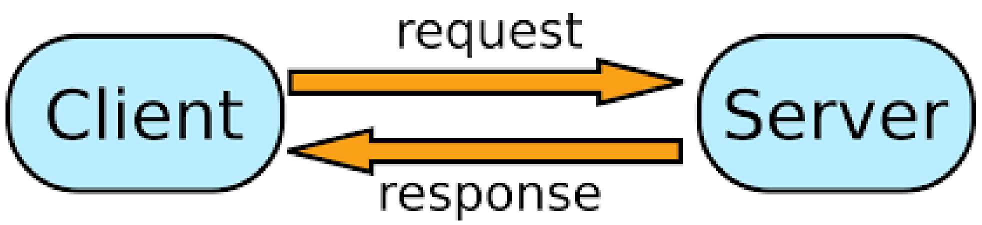
File Management System	Database Management System
• File System is easy-to-use system to store data which require less security and constraints.	• Database Management System is used when security constraints are high.
• Data Redundancy is more in File System.	• Data Redundancy is less in Database Management 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 device that provides a service to another computer program and its user, 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.

difference between dbms and rdbms

DBMS	RDBMS
• Data is stored as file.	• Data is stored as tables.
• There is no relationship between data in DBMS.	• Data is present in multiple tables which can be related to each other.
• DBMS has no support for distributed databases.	• RDBMS supports distributed databases.
• Normalization cannot be achieved.	• Normalization can be achieved.
• DBMS supports single user at a time.	• RDBMS supports multiple users at a time.
• Data Redundancy is common in DBMS.	• Data Redundancy can be reduced in RDBMS.
• DBMS provides low level of security during data manipulation.	• RDBMS has high level of security during data manipulation.

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 (columns) 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, and 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, and 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?

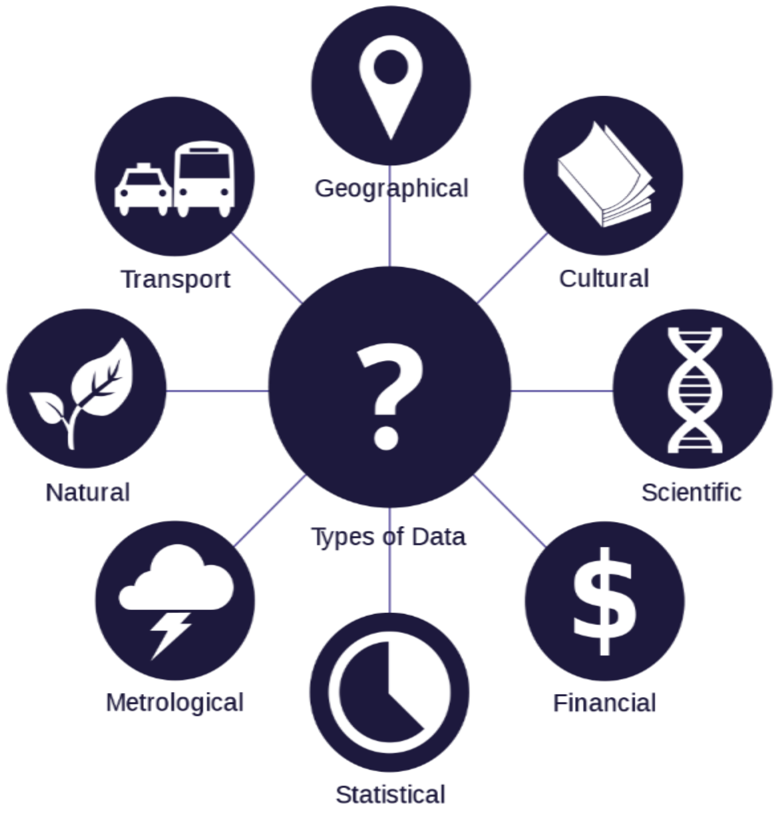
what is data?

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



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?

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.

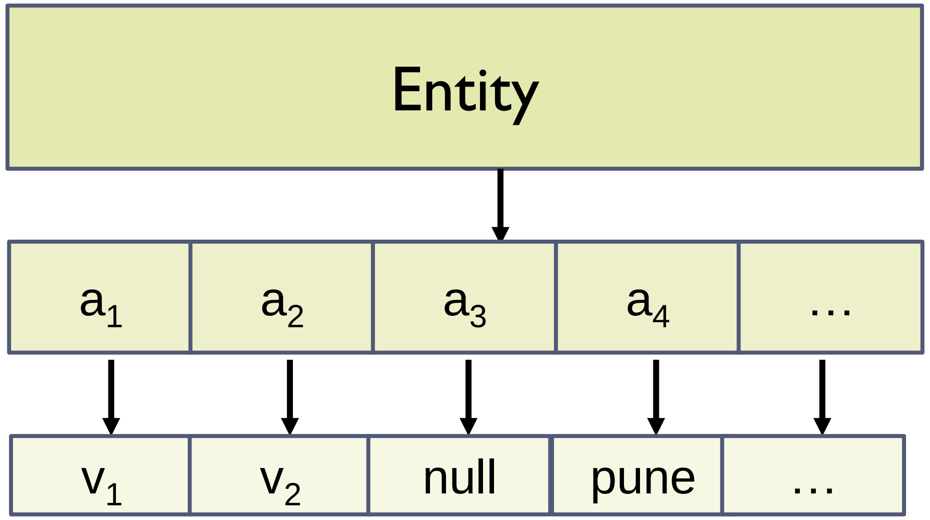
When you are designing attributes for your entities, **you will sometimes find that an attribute does not have a value**. For example, you might want an attribute for a person's middle name, but you can't require a value because some people have no middle name. **For these, you can define the attribute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

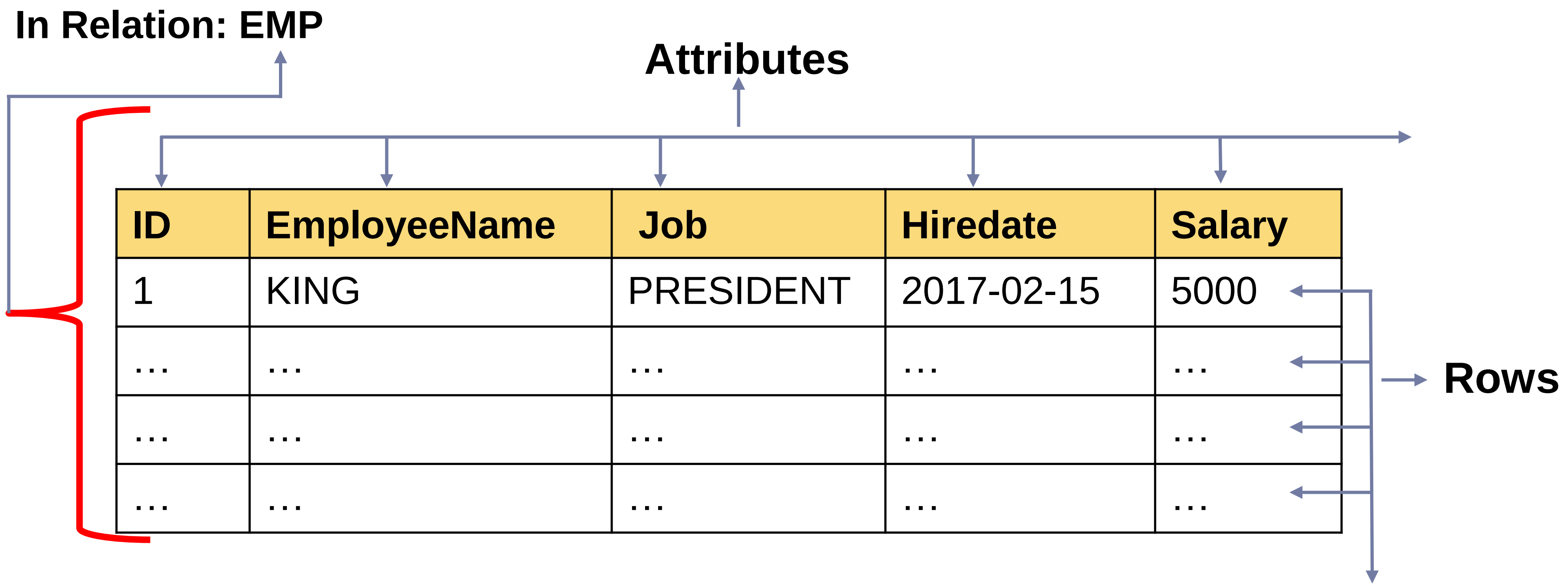
Attributes are the properties that define a relation.
e.g. student(ID, firstName, middleName, lastName, city)

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**.



ID	EmployeeName	Job	Hiredate	Salary
1	KING	PRESIDENT	2017-02-15	5000
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…

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

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

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

- **Atomic Attribute:** An attribute that cannot be divided into smaller independent attribute is known as atomic attribute.
  e.g. ID's, PRN, age, gender, zip, marital status cannot further divide.

- **Single Value Attribute:** An attribute that has only single value is known as single valued attribute.
  e.g. manufactured part can have only one serial number, voter card, blood group, price, quantity, branch can have only one value.

- **Stored Attribute:** The stored attribute are such attributes which are already stored in the database and from which the value of another attribute is derived.
  e.g. (HRA, DA…) can be derive from salary, age can be derived from DoB, total marks or average marks of a student can be derived from marks.

Composite **VS** Multi Valued Attribute

composite / multi valued attributes

Composite Attribute

Person Entity

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

$$\begin{tikzcd}
\{Address\} \arrow[d] \\
\{street,\ city,\ state,\ postal\text{-}code\} \arrow[d] \\
\{street\text{-}number,\ street\text{-}name,\ apartment\text{-}number\}
\end{tikzcd}$$

Multi Valued Attribute

Person Entity

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

What is an Prime, Non-Prime Attribute?

attributes

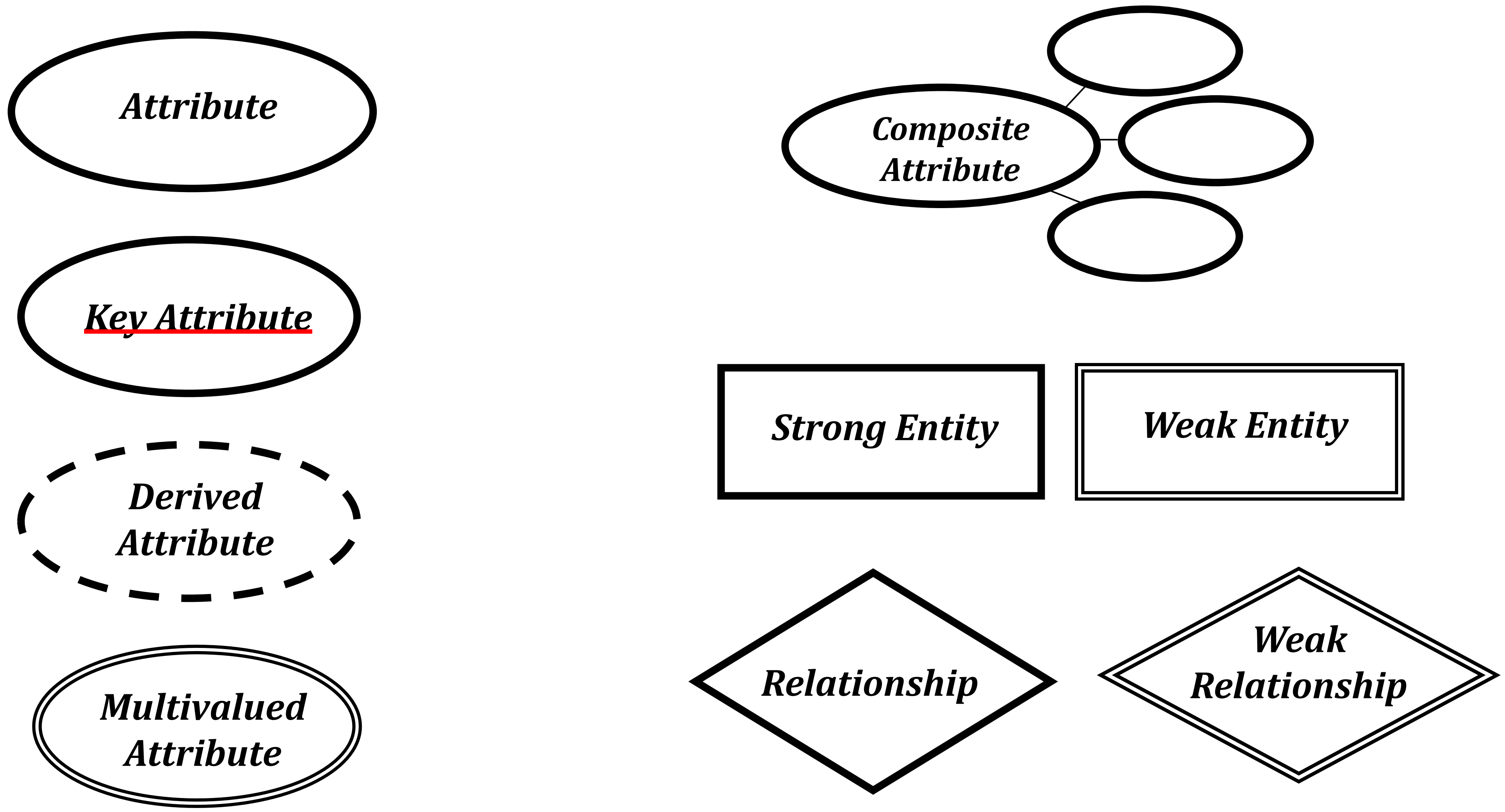
Prime attribute (*Entity integrity*)

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



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.

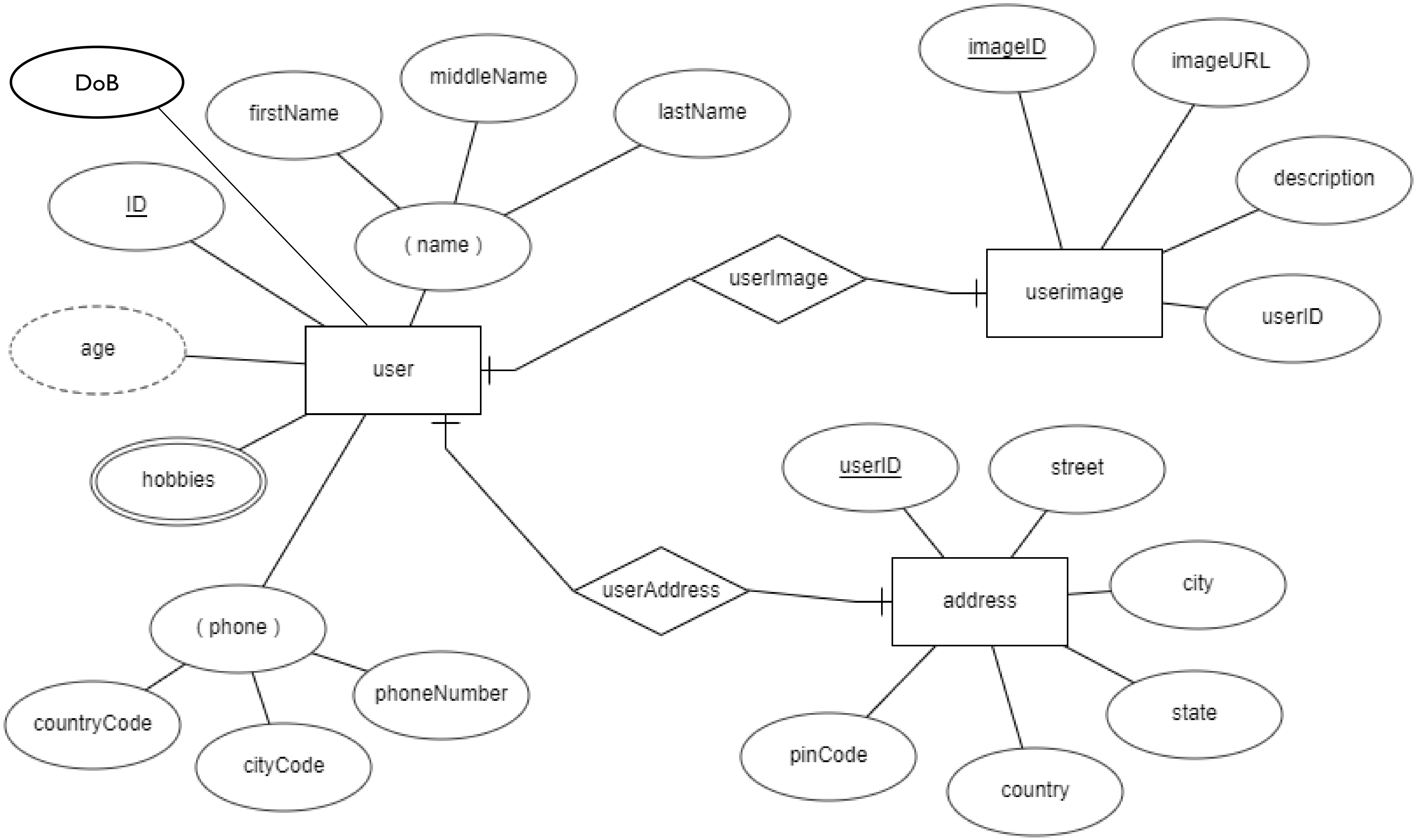
Example 1 – A loan entity can not be created for a customer if the customer doesn't exist

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

Example 3 – A dependents list entity can not be created if the employee doesn't exist

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

entity relationship diagram



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

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

- **Degree d(R) / Arity**: Total number of **attributes/columns** present in a relation/table is called **degree of the relation** and is denoted by $\mathbf{d}(R)$.

- **Cardinality |R|**: Total number of **tuples/rows** present in a relation/table, **is called cardinality of a relation** and is denoted by $|R|$.

- **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** $\mathbf{d}(\mathbf{R})$.
  2. Domains of the respective attributes should also be same.

What is domain constraint and types of data integrity constraints?

Data integrity refers to the correctness and completeness of data.

A domain constraint and types of data integrity constraints

❖ **Domain Constraint** = data type + Constraints (**not null/unique/primary key/foreign key/check/default**)

e.g. custID INT, constraint pk_custid PRIMARY KEY(custID)

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

- **Entity integrity:** Entity Integrity Constraint is used to ensure the uniqueness of each record the table. There are primarily two types of integrity constraints that help us in ensuring the uniqueness of each row, namely, UNIQUE constraint and PRIMARY KEY constraint.
-
- **Referential integrity:** Referential Integrity Constraint ensures that there always exists a valid relationship between two tables. This makes sure that if a foreign key exists in a table relationship then it should always reference a corresponding value in the second table $t_1[FK] = t_2[PK]$ or it should be null.

- **Domain integrity:** A domain is a set of values of the same type. For example, we can specify if a particular column can hold null values or not, if the values have to be unique or not, the data type or size of values that can be entered in the column, the default values for the column, etc..

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 = Employee(EmployeeID, FullName, job, salary, PAN, DateOfBirth, emailID, deptno)

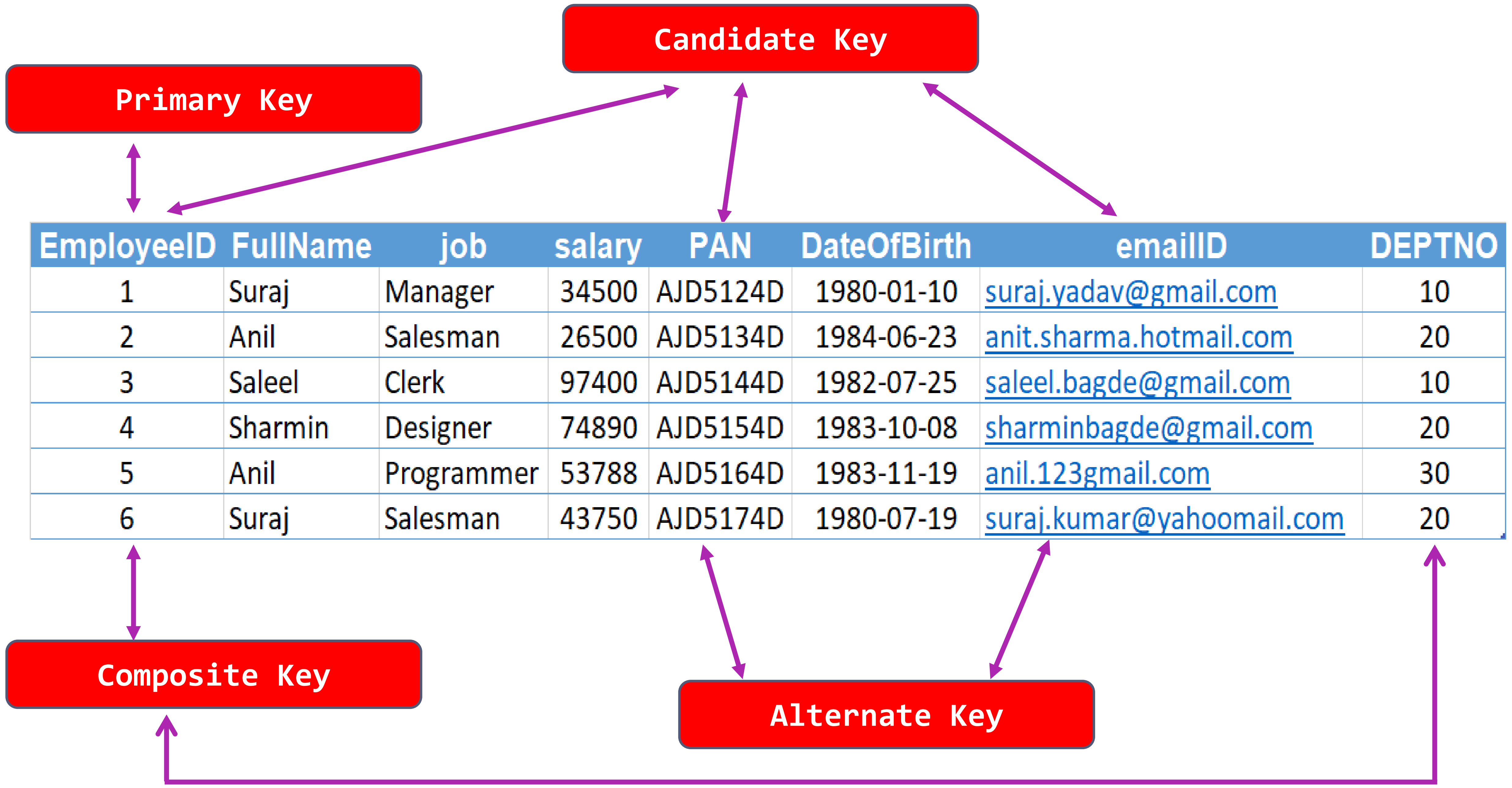
- **Candidate Key:** are individual columns in a table that qualifies for uniqueness of all the rows. Here in Employee table EmployeeID, PAN or emailID are Candidate keys.

- **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.



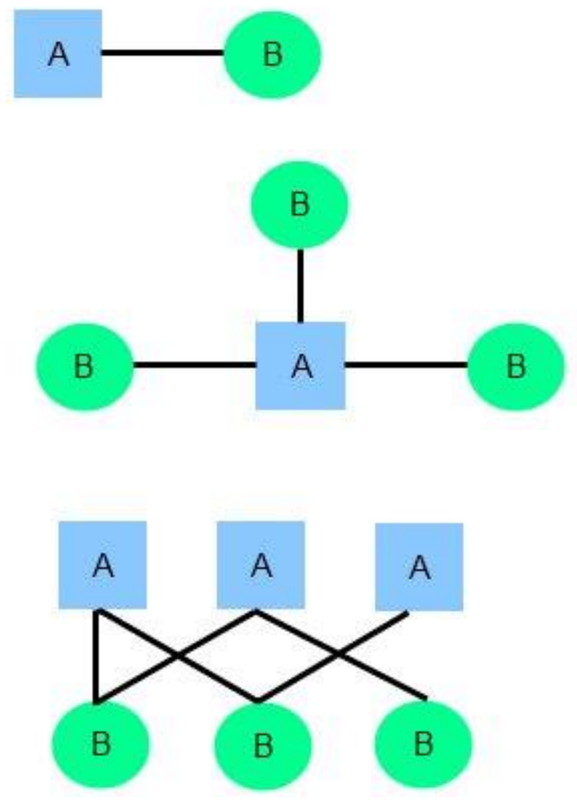
EmployeeID	FullName	job	salary	PAN	DateOfBirth	emailID	DEPTNO
1	Suraj	Manager	34500	AJD5124D	1980-01-10	suraj.yadav@gmail.com	10
2	Anil	Salesman	26500	AJD5134D	1984-06-23	anit.sharma.hotmail.com	20
3	Saleel	Clerk	97400	AJD5144D	1982-07-25	saleel.bagde@gmail.com	10
4	Sharmin	Designer	74890	AJD5154D	1983-10-08	sharminbagde@gmail.com	20
5	Anil	Programmer	53788	AJD5164D	1983-11-19	anil.123gmail.com	30
6	Suraj	Salesman	43750	AJD5174D	1980-07-19	suraj.kumar@yahoomail.com	20

Common relationships

relationships

Common relationship

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

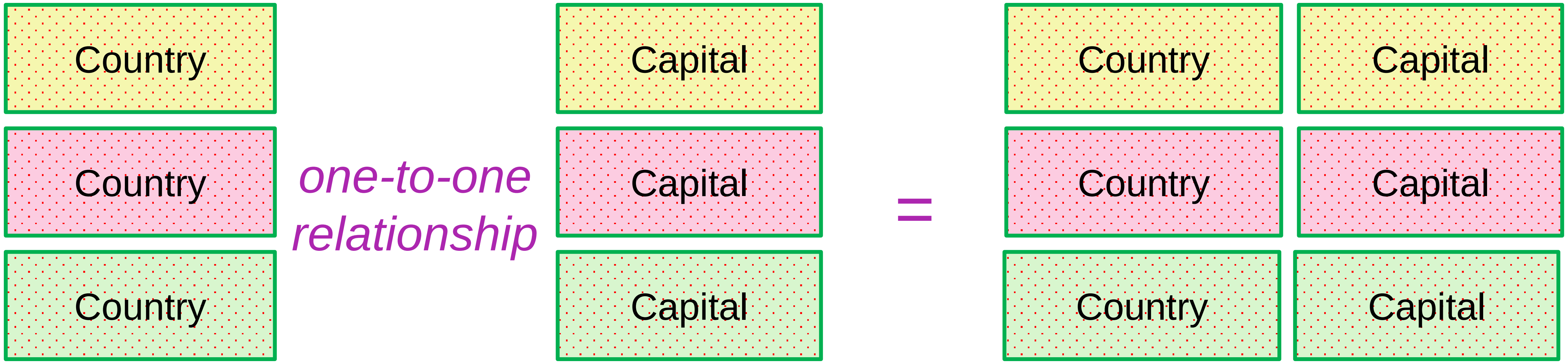


one-to-one relationship

one-to-one relationship

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

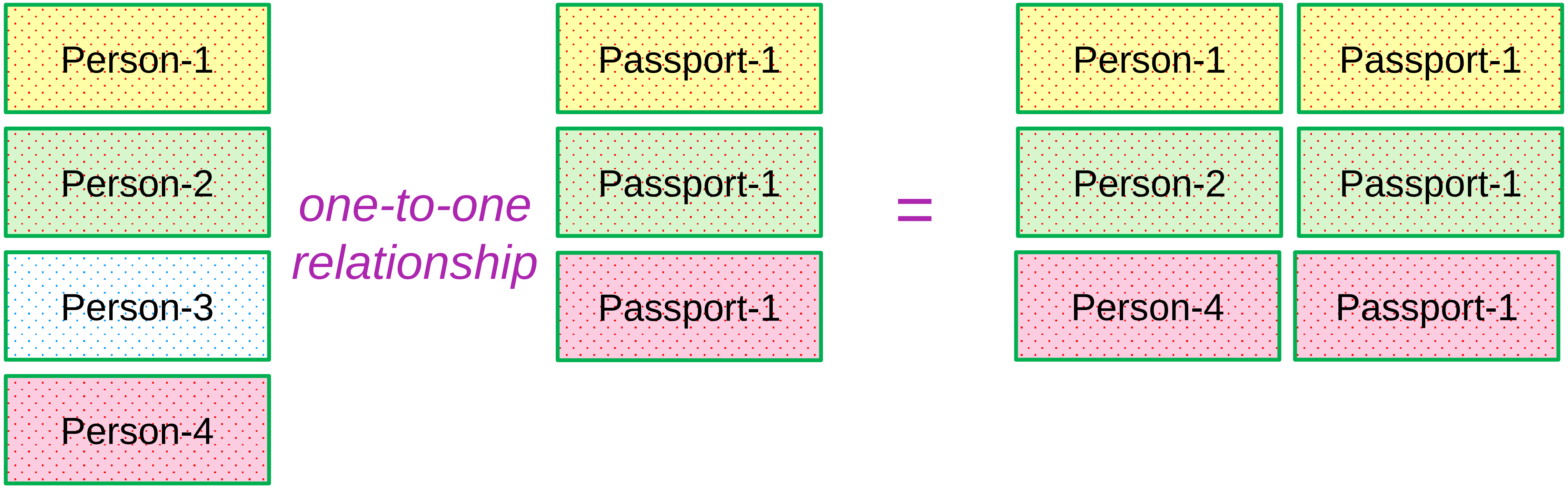
A *one-to-one* relationship is a type of cardinality that refers to the relationship between two entities ***R*** and ***S*** in which one element of entity ***R*** may only be linked to zero/one element of entity ***S***, and vice versa.



one-to-one relationship

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

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

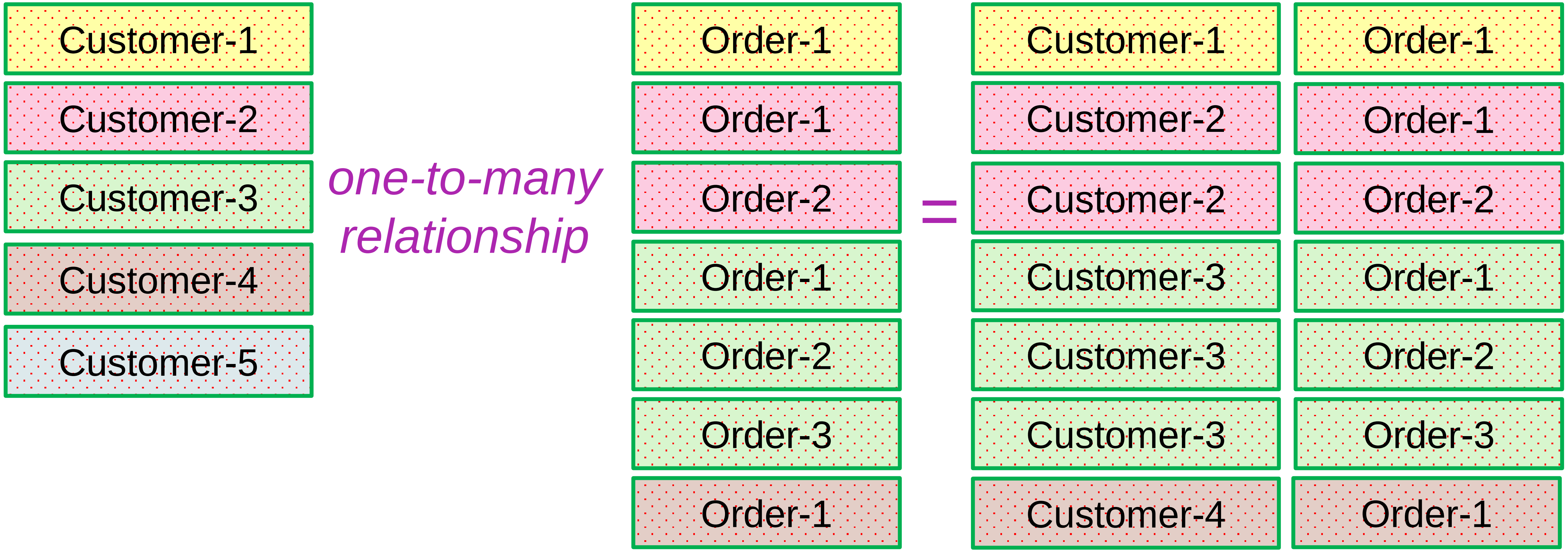


one-to-many relationship

one-to-many relationship

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

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



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

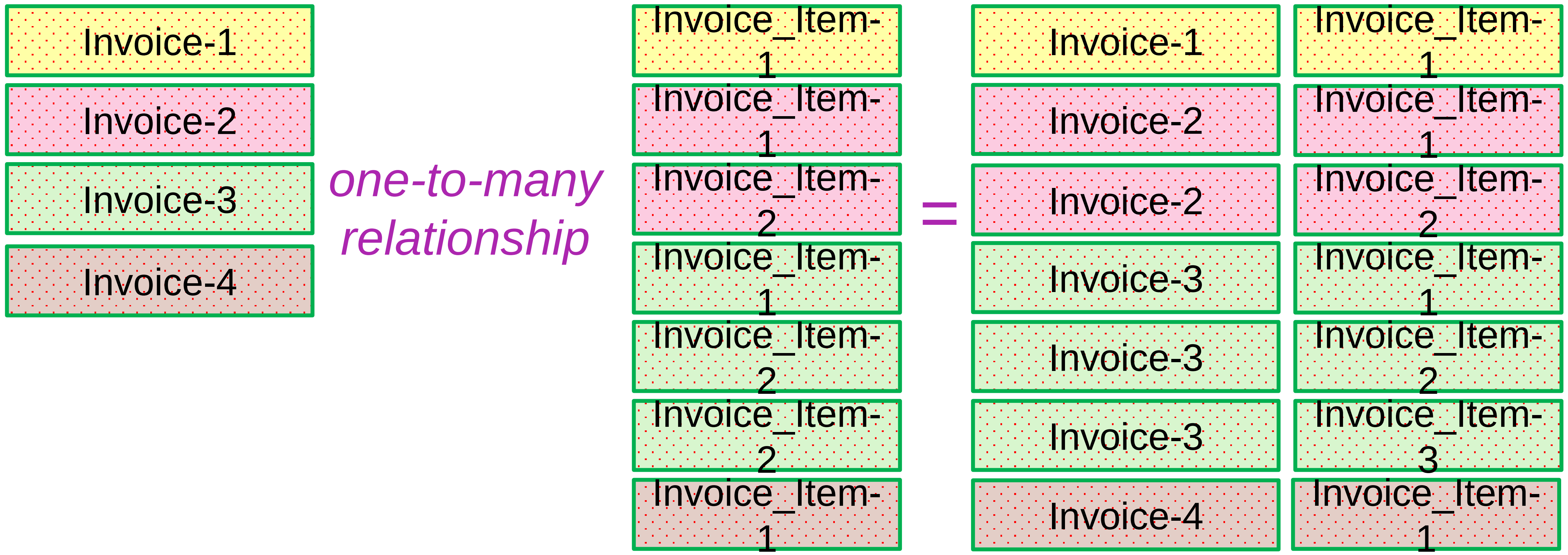
Order-1
Order-1
Order-2
Order-1
Order-2
Order-3
Order-1

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 row in the table on the other side of their relationship.

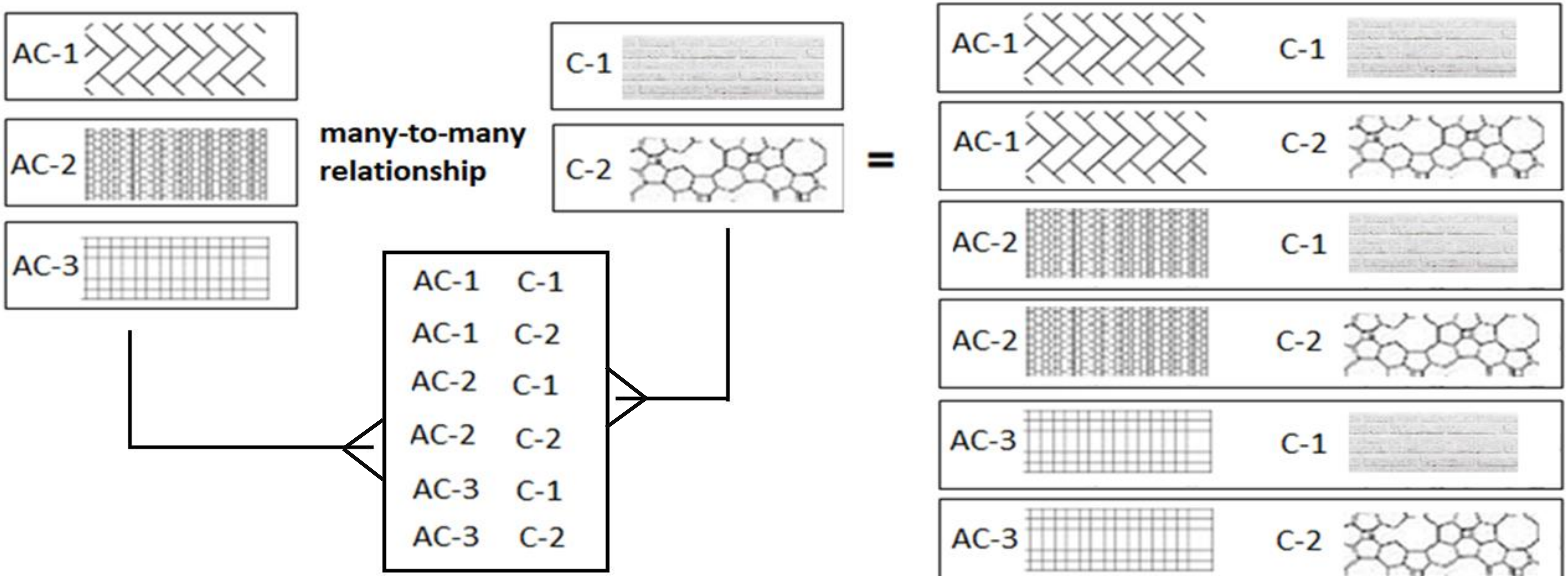
a *one-to-many* relationship is a type of cardinality that refers to the relationship between two entities ***R*** and ***S*** in which an element of ***R*** may be linked to many elements of ***S***, but a member of ***S*** is linked to only one element of ***R***.



many-to-many relationship

many-to-many relationship

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

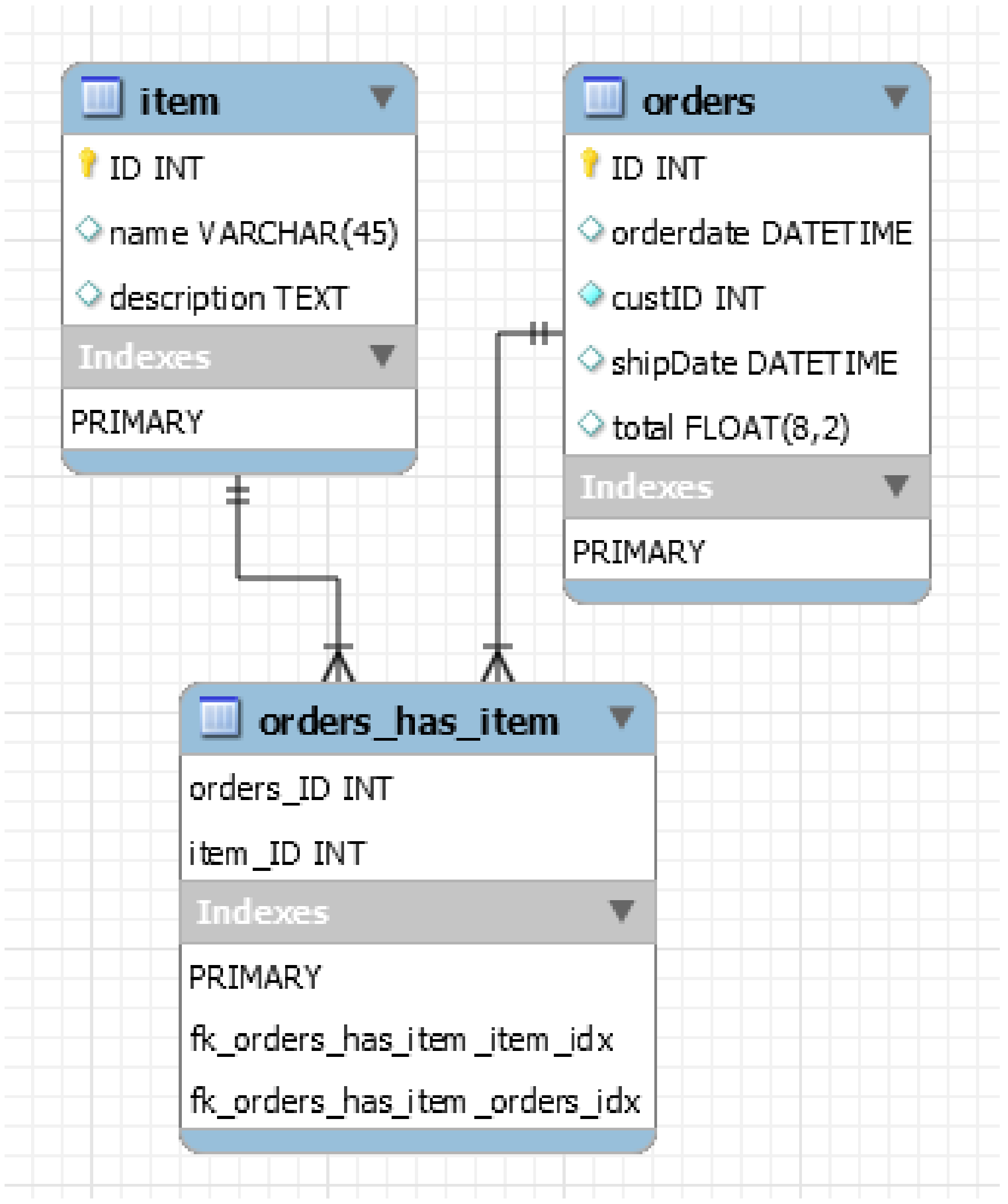


how to create many-to-many relationship

```
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)
);
```

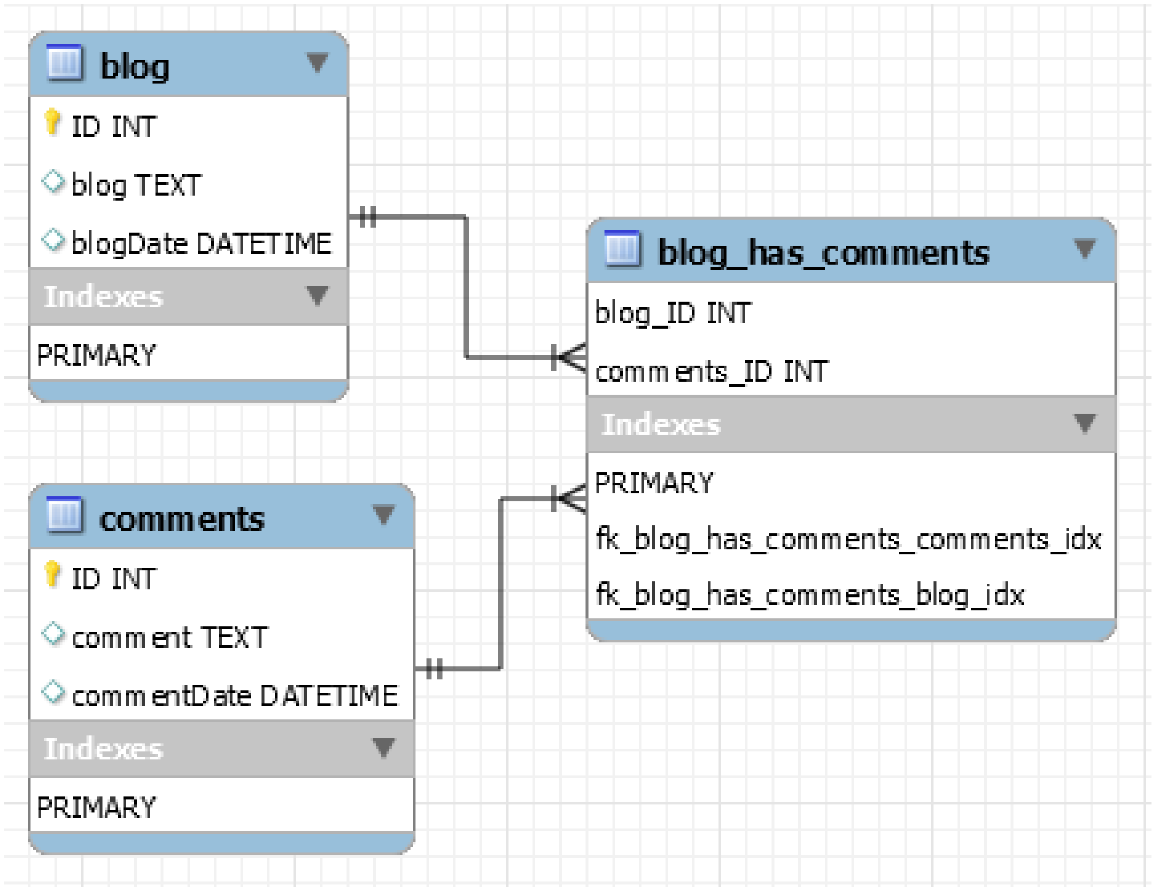


how to create many-to-many relationship

```
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)
);
```



MySQL

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?

what is sql?

Remember:

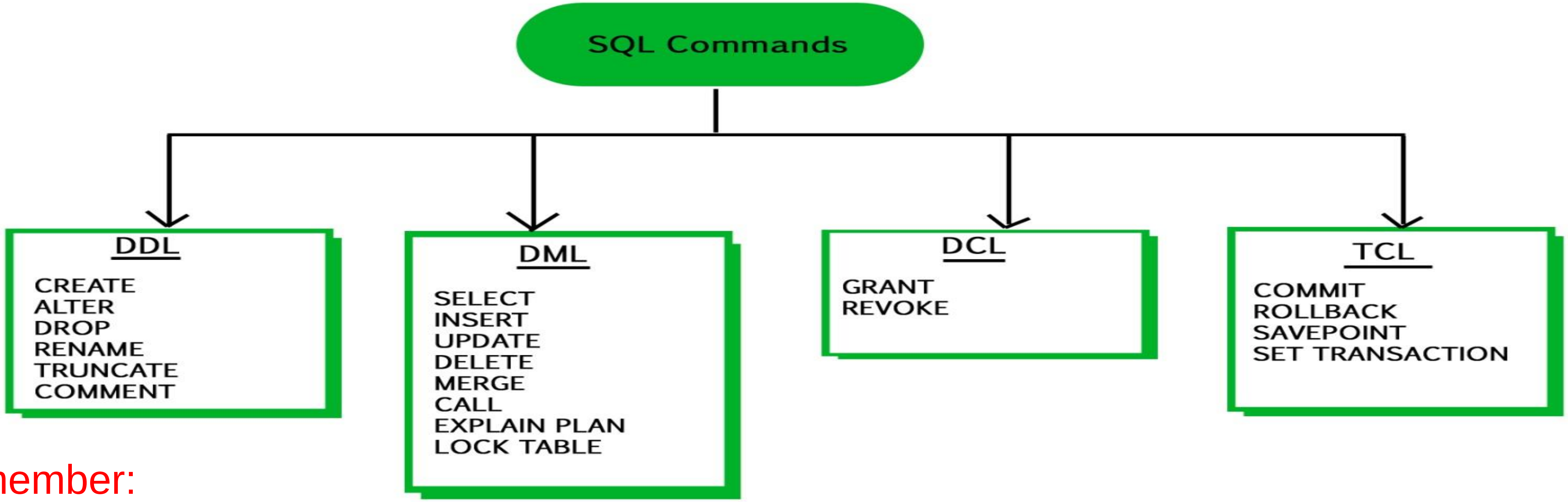
- **EXPLICIT or IMPLICIT commit will commit the data.**

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.

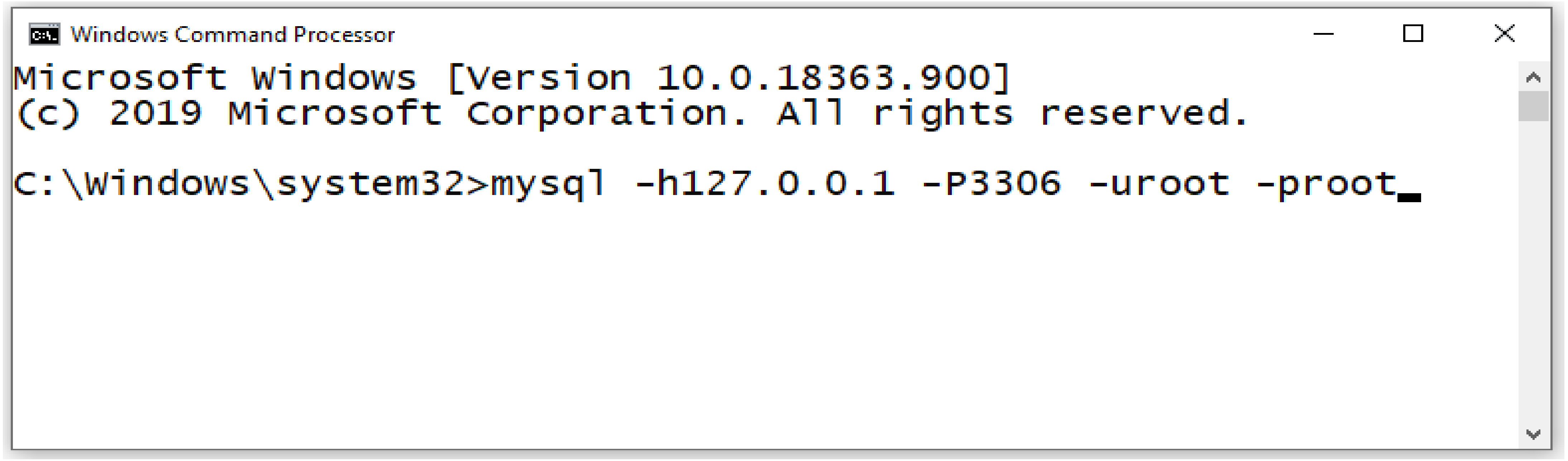
Reconnect to the server	\r
Execute a system shell command	\!
Exit mysql	\q
Change your mysql prompt.	prompt str or \R str

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*]



```
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 stores on the disk
InnoDB	.idb (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.

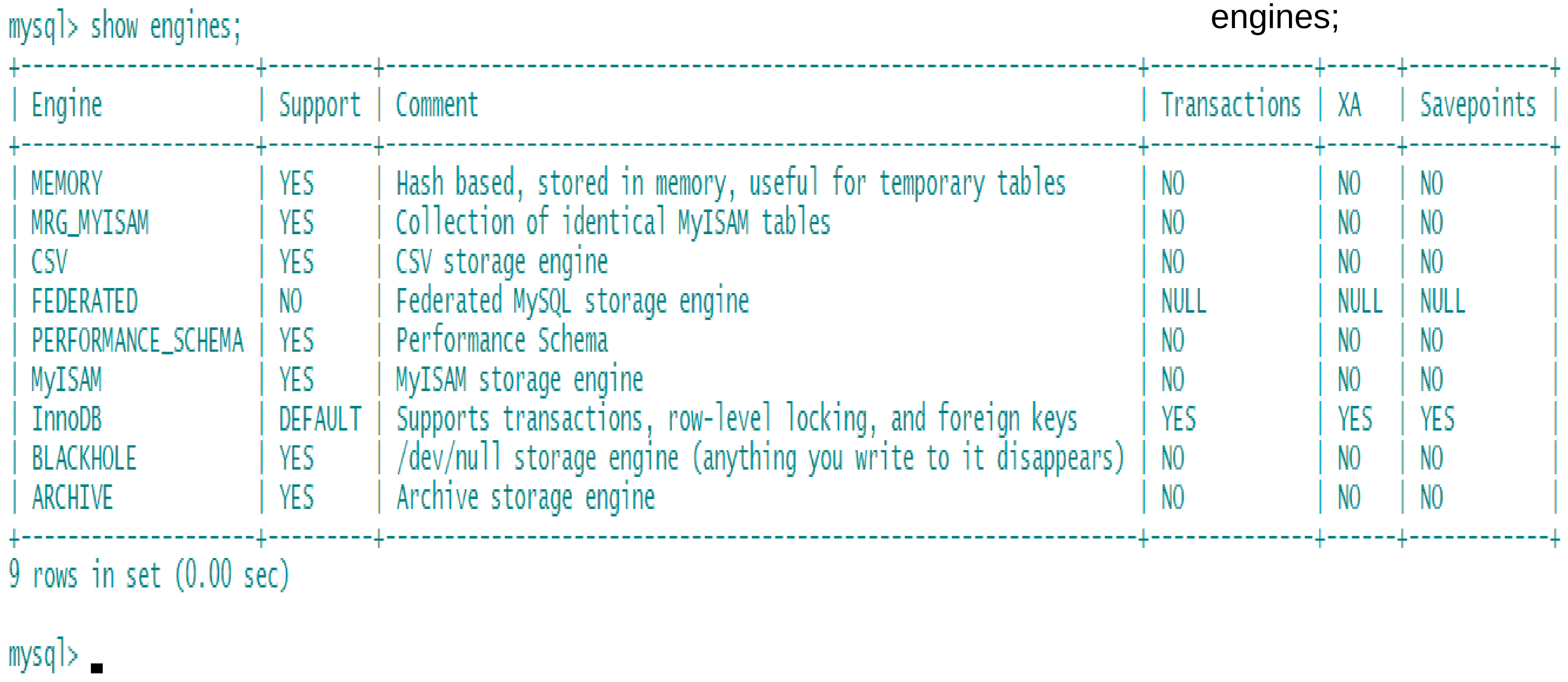
INFORMATION_SCHEMA.ENGINES

- 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.

SHOW DATABASES

SHOW DATABASES Syntax

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

SHOW SCHEMAS is a synonym for SHOW DATABASES.

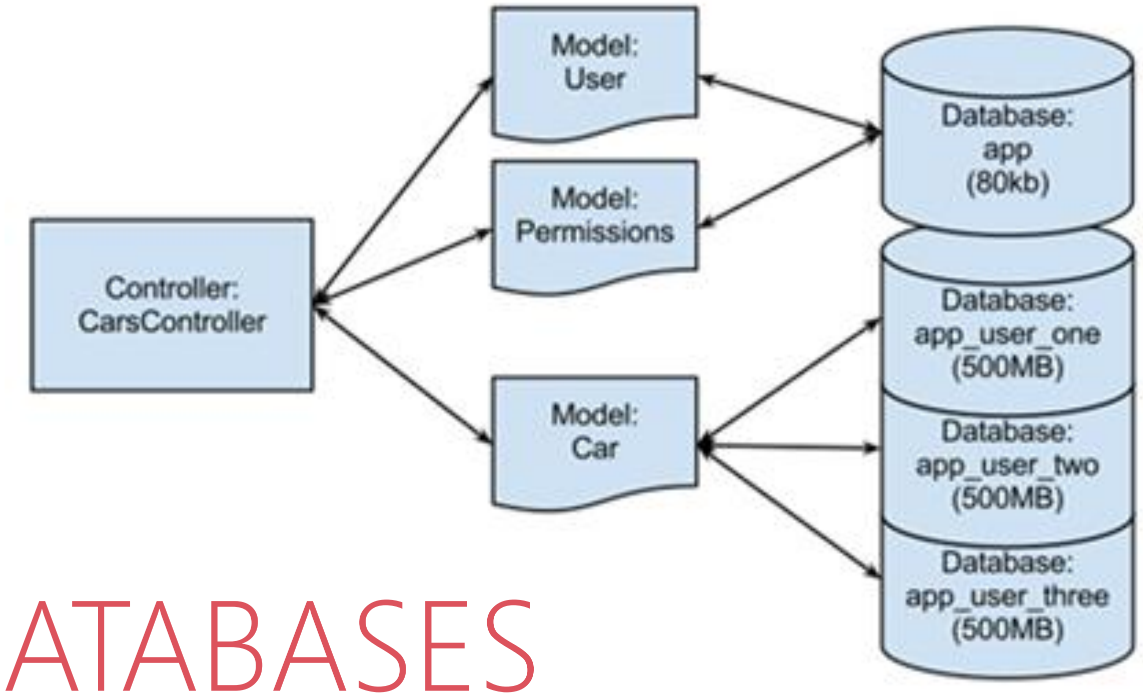
```
SHOW DATABASES;

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;
```

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 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');

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 ');

datatype - numeric

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

For: float(M,D), double(M,D) or decimal(M,D), M must be >= 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

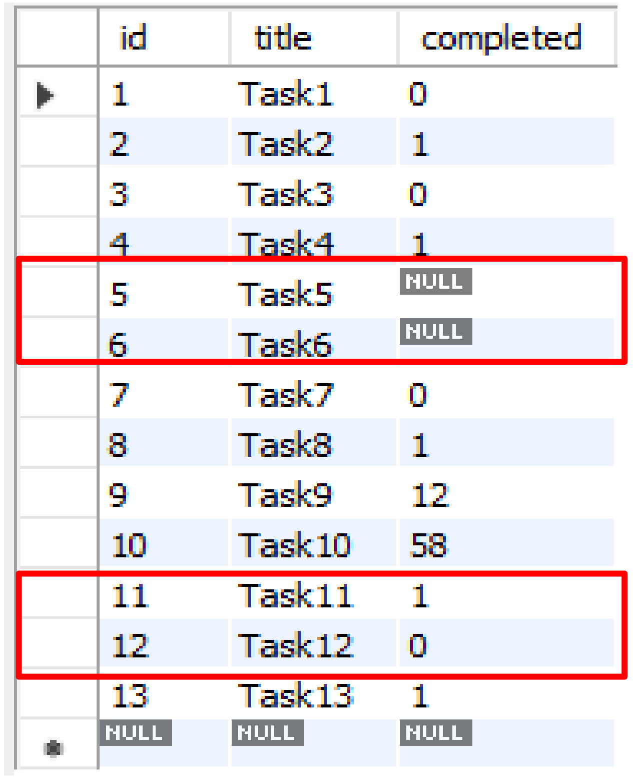
Datatypes	**Size**	**Description**
YEAR	1 byte	YYYY
DATE	3 bytes	YYYY-MM-DD
TIME	3 bytes	HH:MM:SS
DATETIME	8 bytes	YYYY-MM-DD hh:mm:ss

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)**

datatype – enum

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

datatype – enum

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

```
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', '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_defineation, . . .)
    [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

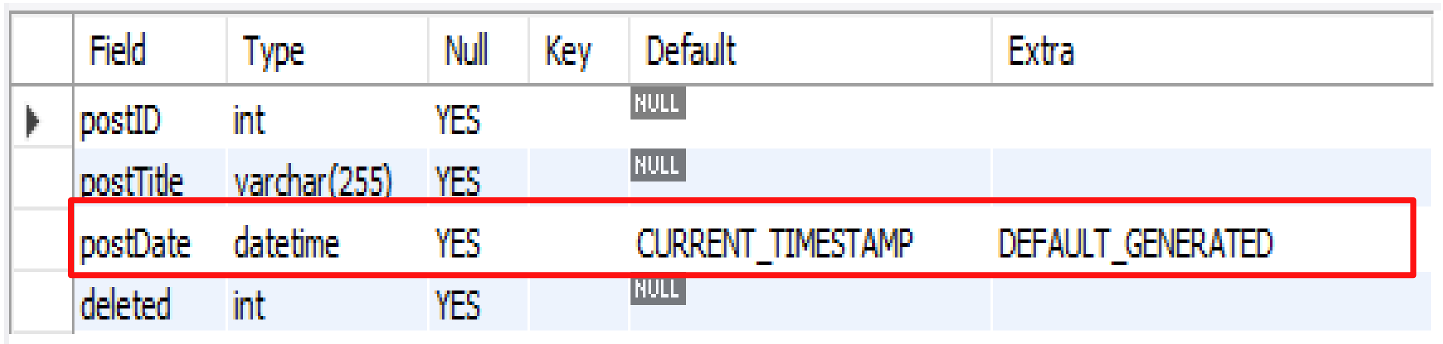
The DEFAULT specifies a default value for the column.

default value

col_name data_type DEFAULT value

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

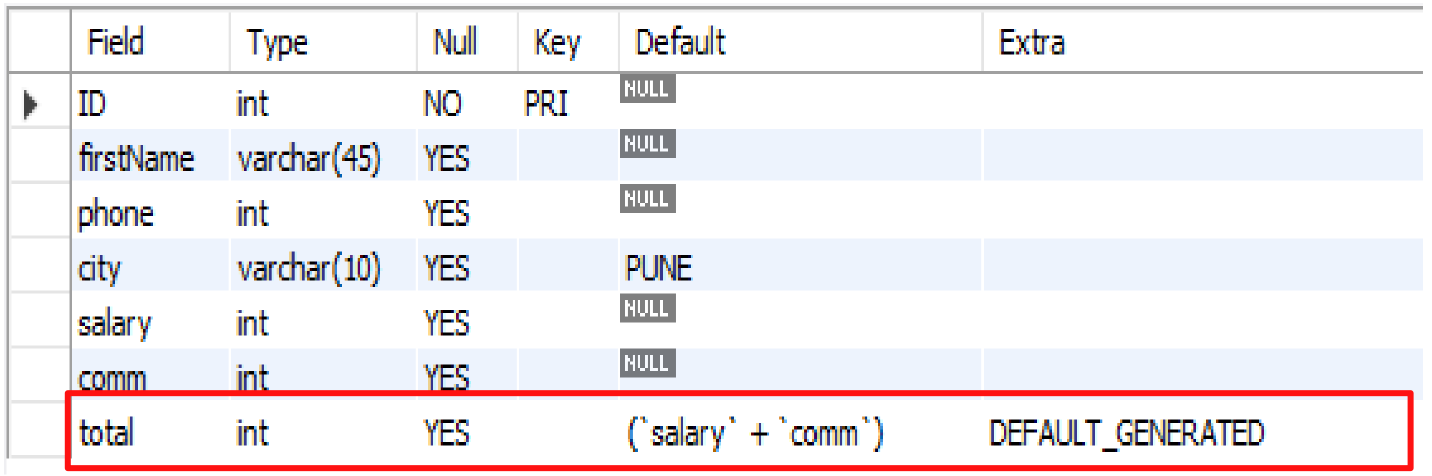
-
  ```
  CREATE TABLE posts (
      postID INT,
      postTitle VARCHAR(255),
      postDate DATETIME DEFAULT NOW(),
      deleted INT
  );
  ```



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	

# 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)
  );
  ```

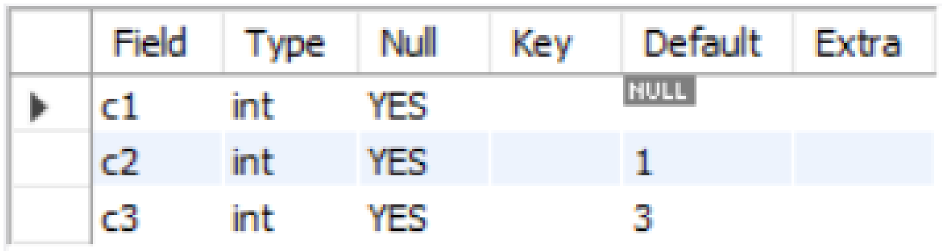


Field	Type	Null	Key	Default	Extra
ID	int	NO	PRI	NULL	
firstName	varchar(45)	YES		NULL	
phone	int	YES		NULL	
city	varchar(10)	YES		PUNE	
salary	int	YES		NULL	
comm	int	YES		NULL	
total	int	YES		(`salary` + `comm`)	DEFAULT_GENERATED

default value - insert

The **DEFAULT** example.

```
• CREATE TABLE t (
    c1 INT,
    c2 INT DEFAULT 1,
    c3 INT DEFAULT 3,
  );
```



	Field	Type	Null	Key	Default	Extra
▶	c1	int	YES		NULL	
	c2	int	YES		1	
	c3	int	YES		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));
```

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 giving you the error.
- Inserting a string into a string column (CHAR, VARCHAR, TEXT, or BLOB) that exceeds the column maximum length. The value is truncated to the column maximum length.

- **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');`

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

SELECT statement…

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

SELECT CLAUSE

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

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

Capabilities of SELECT Statement

1. SELECTION
2. PROJECTION
3. JOINING

Capabilities of SELECT Statement

➢SELECTION

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

R

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

Capabilities of SELECT Statement

➢PROJECTION

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

R

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

Capabilities of SELECT Statement

➢JOINING

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

R

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

S

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

select statement

SELECTION Process

SELECT $\underbrace{\text{ * }}_{\text{selection-list | field-list | column-list}}$ FROM <table_references>

Remember:

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

PROJECTION Process

SELECT $\underbrace{\text{column-list}}_{\text{selection-list | field-list | column-list}}$ FROM <table_references>

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.

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

Note:

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.

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.

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.

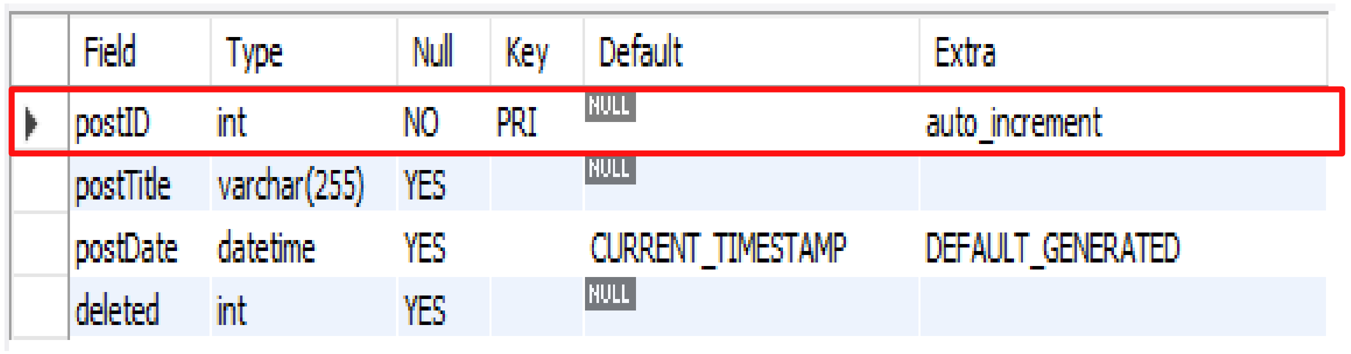
- `SELECT @@IDENTITY`
- `SELECT LAST_INSERT_ID()`
- `SET INSERT_ID = 7`

-
  ```
  CREATE TABLE posts (
     c1 INT UNIQUE KEY AUTO_INCREMENT,
     c2 VARCHAR(20)
  ) AUTO_INCREMENT = 2;      // auto_number will start with
  value 2.
  ```

auto_increment

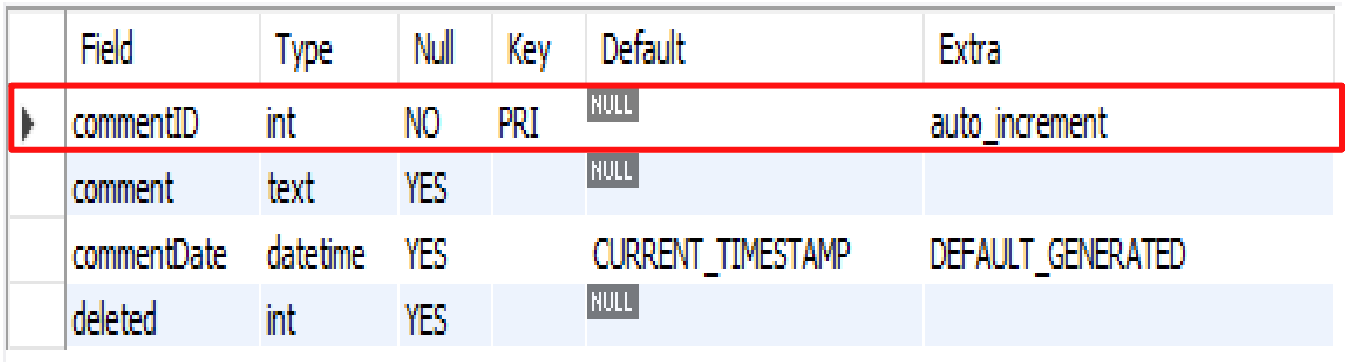
The **auto_increment** specifies a **auto_increment** value for the column.

-
  ```
  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 . . . 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

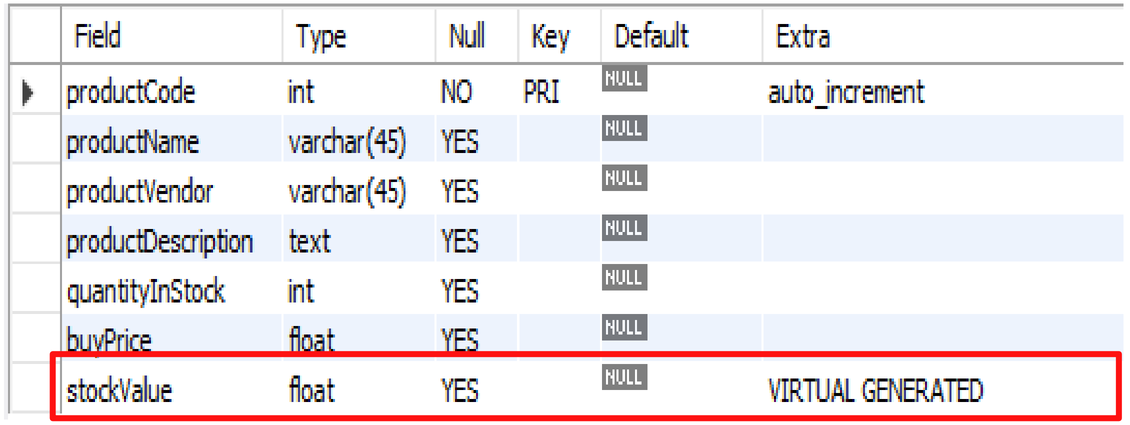
col_name data_type [GENERATED ALWAYS] AS (*expression*) [VIRTUAL | STORED]

- **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

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
) ;
```

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

- `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`

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;

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**.

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

types of Keys?

r = Employee(EmployeeID, FullName, job, salary, PAN, DateOfBirth, emailID, deptno)

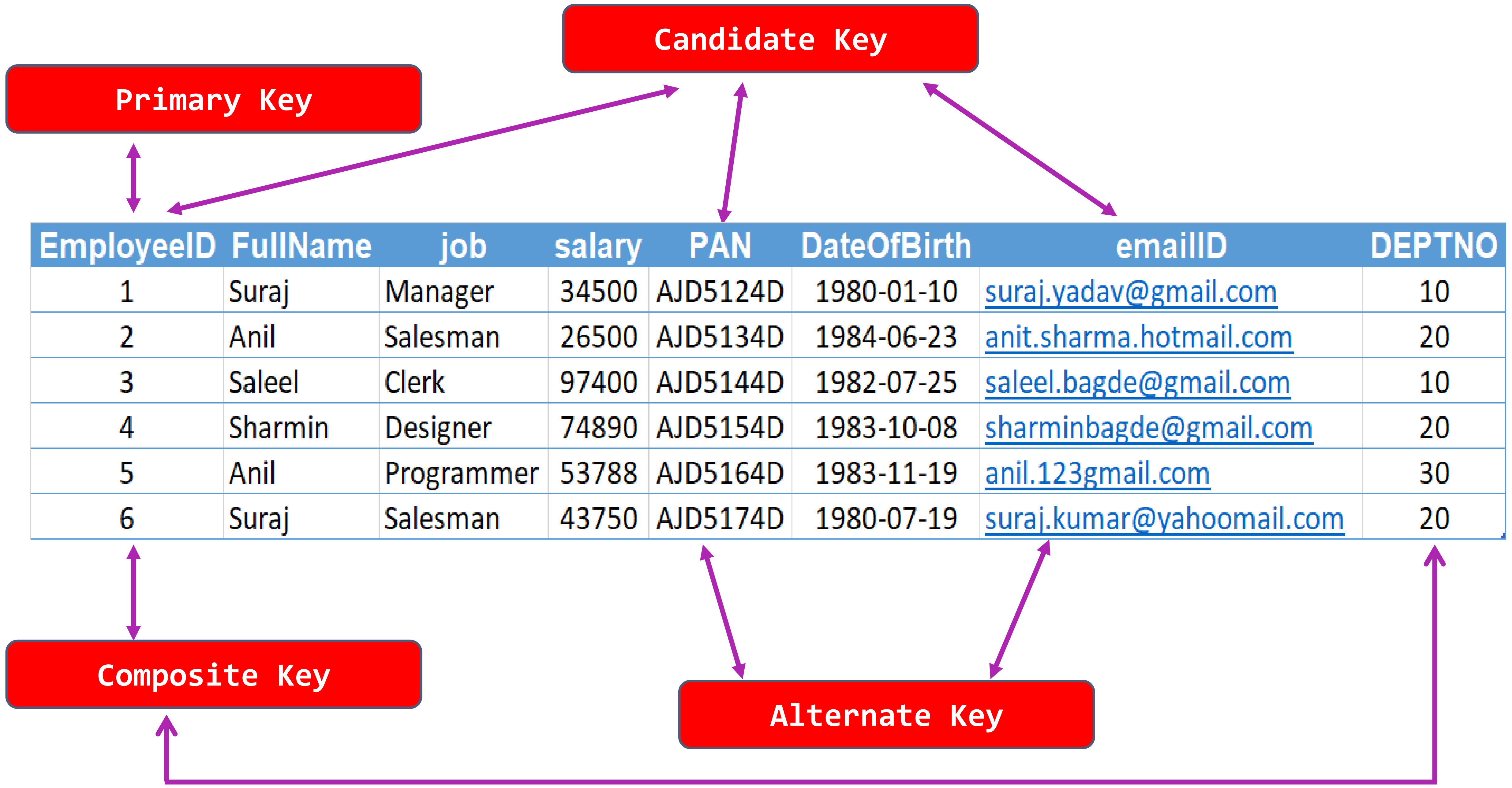
- **Candidate Key:** are individual columns in a table that qualifies for uniqueness of all the rows. Here in Employee table EmployeeID, PAN or emailID are Candidate keys.

- **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.



EmployeeID	FullName	job	salary	PAN	DateOfBirth	emailID	DEPTNO
1	Suraj	Manager	34500	AJD5124D	1980-01-10	suraj.yadav@gmail.com	10
2	Anil	Salesman	26500	AJD5134D	1984-06-23	anit.sharma.hotmail.com	20
3	Saleel	Clerk	97400	AJD5144D	1982-07-25	saleel.bagde@gmail.com	10
4	Sharmin	Designer	74890	AJD5154D	1983-10-08	sharminbagde@gmail.com	20
5	Anil	Programmer	53788	AJD5164D	1983-11-19	anil.123gmail.com	30
6	Suraj	Salesman	43750	AJD5174D	1980-07-19	suraj.kumar@yahoomail.com	20

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
  purchase_orders (
    po_number INT,
    vendor_id INT NOT NULL,
    po_status INT NOT NULL,
    PRIMARY KEY(po_number)
  );
  ```

-
  ```
  CREATE TABLE supplier (
   supplier_id INT,
   supplier_name VARCHAR(50),
   contact_name VARCHAR(50),
   constraint pk_supplier_id PRIMARY KEY(supplier_id)
  );
  ```

-
  ```
  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 brands (
    ID INT,
    brandName VARCHAR(30),
    constraint uni_brandName
  UNIQUE(brandName)
   );
```

```
• SHOW  INDEX FROM clients;
```

```
• CREATE TABLE contacts (
     ID INT,
     first_name VARCHAR(50),
     last_name VARCHAR(50),
     phone VARCHAR(15),
     UNIQUE(phone)
  );
```

```
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;

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	CourceDuration	CourceName
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.

alter, drop – (primary key/foreign key)

Remember:

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.

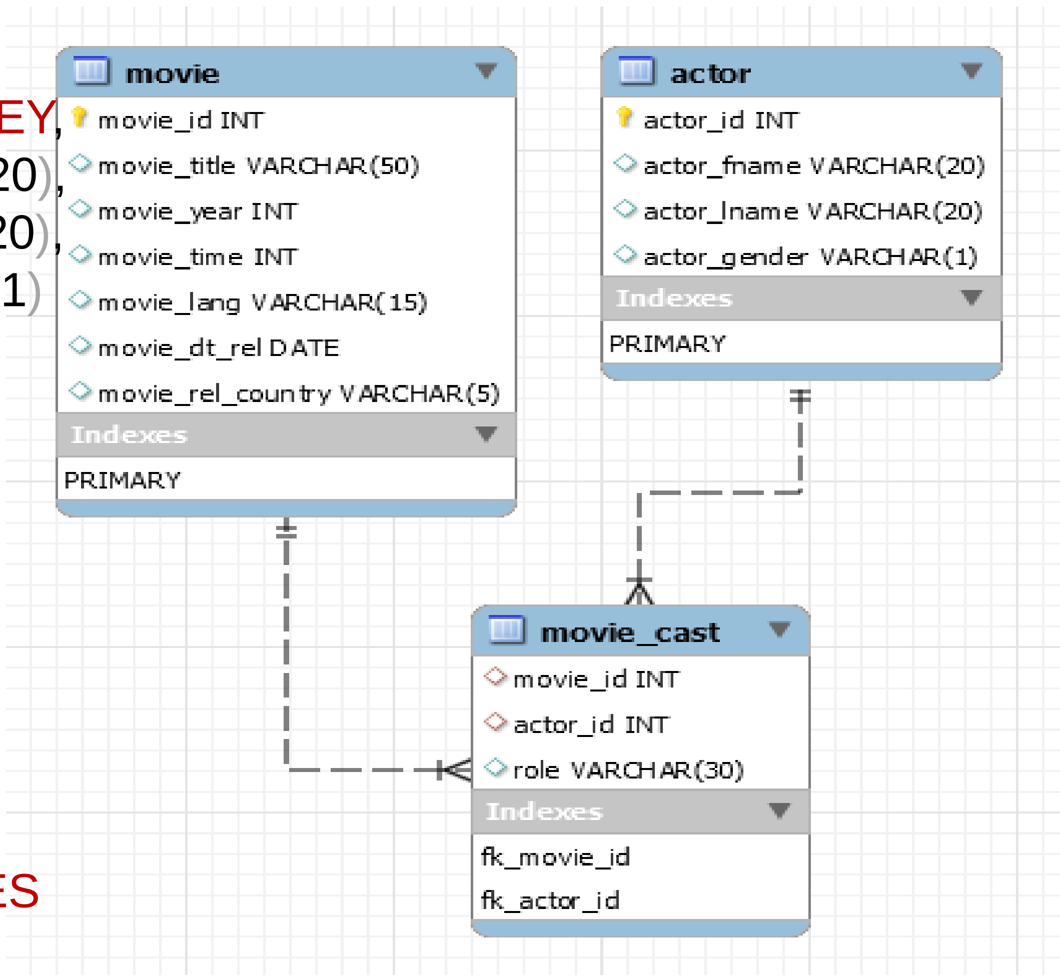
constraints – foreign key

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.

-
  ```
  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,
   FOREIGN KEY(userID) REFERENCES users(ID)
) ;
```

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

```
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

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,
  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,
  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 (conidiation)
```

```
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;

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

- 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%');

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.

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 [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 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 }

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

drop table

Remember:

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

create temporary table

Note:

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

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.”

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**.**

Note:

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

select statement - alias

SELECT $\boldsymbol{A}_1$ [[AS] alias_name], $\boldsymbol{A}_2$ [[AS] alias_name], . . ., $\boldsymbol{A}_N$ FROM ***r*** [[AS] alias_name]

column-name as new-name

table-name as new-name

Remember:

- The alias name can be used in **GROUP BY, HAVING,** or **ORDER BY** clauses.
- Standard SQL **disallows** references to column aliases in a ***WHERE*** clause.
- If the column alias contains spaces, **put it in quotes**.
- Alias name is **max 256 characters**.

comparison functions and operator

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

comparison functions and operator

1. ***arithmetic_operators:***
   * | / | DIV | % |MOD | - | +
2. ***comparison_operator:***
   = | <=> | >= | > | <= | < | <> | !=
3. ***boolean_ predicate:***
   IS [NOT] *NULL*
   | *expr* <=> *null*
4. ***predicate*:**
   expr [NOT] LIKE *expr* [ESCAPE *char*]
   | *expr* [NOT] IN (*expr1, expr2, . . .*)
   | *expr* [NOT] IN (*subquery*)
   | *expr* [NOT] BETWEEN *expr1* AND *expr2*
5. ***logical_operators***
   { AND | && } | { OR | || }
6. ***assignment _operator***
   = (assignment), :=

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

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.

column - expressions

select statement - expressions

Column EXPRESSIONS

SELECT $\boldsymbol{A}_1$, $\boldsymbol{A}_2$, $\boldsymbol{A}_3$, $\boldsymbol{A}_4$, 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;

identifiers

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

identifiers

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

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

- You can refer to a column as
  1. col_name
  2. tbl_name.col_name
  3. db_name.tbl_name.col_name.

control flow functions

control flow functions - ifnull

IFNULL function

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

```
IFNULL(expression1, expression2)
```

- `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;`

control flow functions - if

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)
```

- DATEDIFF(CURDATE(), hiredate) / 365.25

datetime functions

now(), curdate(), curtime()

In MySQL, the **NOW()** function returns a default value for a **DATETIME**.
MySQL inserts the current **date and time** into the column whose default value is NOW().

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

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

+ or - operator

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

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

- SELECT NOW(), NOW() + INTERVAL 1 DAY;
- SELECT NOW(), NOW() + INTERVAL '1-3' YEAR_MONTH;

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

ADDDATE() is a synonym for **DATE_ADD()**

ADDDATE(date, INTERVAL expr unit) / DATE_ADD (date, INTERVAL expr unit)

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

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

SUBDATE() is a synonym for **DATE_SUB()**

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

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

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

extract

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

EXTRACT(unit FROM date)

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

- SELECT EXTRACT(MONTH FROM NOW())

datetime functions

Syntax	Result
DAY(date)	DAY() is a **synonym for DAYOFMONTH().**
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

datetime functions

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

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

datetime formats

datetime formats

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

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

datetime formats

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

- 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. e.g. • SELECT ASCII(ename) FROM emp;
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. **NULL values are skipped**. e.g. • SELECT CHAR(65, 66, 67); **/** SELECT CAST(CHAR(65 66, 67) AS CHAR);
CONCAT(str1, str2, . . .)	Returns the string that results from concatenating the arguments. CONCAT() **returns NULL if any argument is NULL**. e.g. • SELECT CONCAT('Mr. ' , ename) FROM emp; • SELECT CONCAT('My', NULL, 'SQL'); #op will be NULL
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. e.g. • SELECT ELT(1, 'Bank', 'Of', 'India', 'Kothrud', 'Pune'); • SELECT ELT(1, ename, job, sal) FROM emp; • SELECT hiredate, ELT(MONTH(hiredate),'Winter', 'Winter', 'Spring', 'Spring', 'Spring', 'Summer', 'Summer', 'Summer', 'Autumn', 'Autumn', 'Autumn', 'Winter') R1 FROM emp;

string functions

Syntax	Result
STRCMP(expr1, expr2)	STRCMP() 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.
LCASE(str)	Returns lower case string. LCASE() **is a synonym for** LOWER().
UCASE(str)	Returns upper case string. UCASE() **is a synonym for** UPPER().
LENGTH(str)	Returns the length of the string.
LPAD(str, len, padstr)	Returns the string str, left-padded with the string padstr to a length of len characters.
RPAD(str, len, padstr)	Returns the string str, right-padded with the string padstr to a length of len characters.
REPEAT(str, count)	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.

string functions

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

- 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. e.g. • SELECT REPLACE('Hello', 'l', 'x');
REVERSE(str)	Returns the string str with the order of the characters reversed.
SUBSTR(str, pos, len)	**SUBSTR() is a synonym for SUBSTRING().** e.g. • SELECT SUBSTR ('This is the test by IWAY', 6); • SELECT SUBSTR ('This is the test by IWAY', -4, 4);
MID(str, pos, len)	MID function **is a synonym for** SUBSTRING**.**

mathematical functions

mathematical functions

Syntax	Result
ABS(x)	Returns the absolute value of X.
CEIL(x)	CEIL() is a synonym for CEILING().
CEILING(x)	Returns CEIL value.
FLOOR(x)	Returns FLOOR value.
MOD(n, m), n % m, n MOD m	Returns the remainder of N divided by M. MOD(N,0) returns NULL.
POWER(x, y)	This is a synonym for POW().
RAND()	Returns a random floating-point value

ROUND(x)
ROUND(x, d)
	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.

TRUNCATE(x, d)	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.

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

select statement

modifiers

SELECT [ALL / DISTINCT / DISTINCTROW] identifier.* / identifier.A_1 [[as] alias_name], identifier.A_2 [[as] alias_name], expr1 [[as] alias_name], expr2 [[as] alias_name] . . .

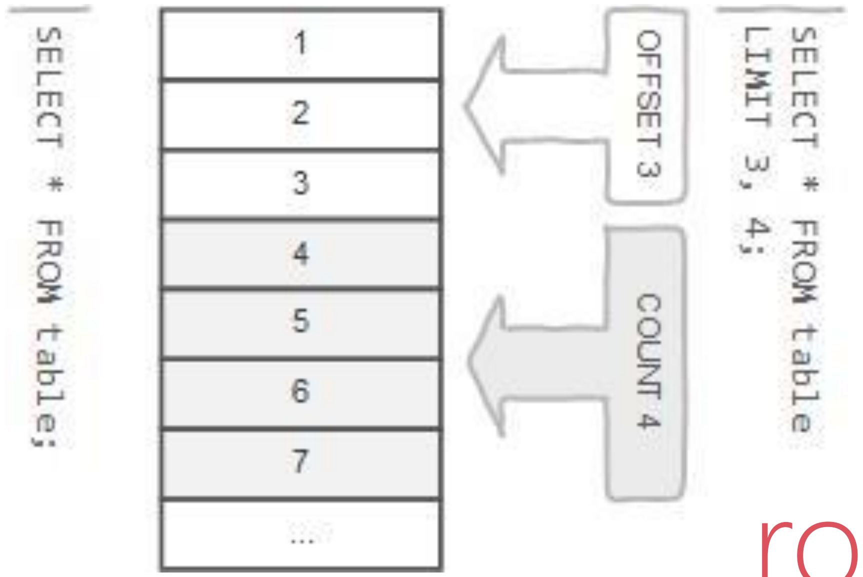
- [FROM <identifier.r_1> [as] alias_name], <identifier.r_2> [as] alias_name], . . .]
- [WHERE < where_condition1 > { and | or } < where_condition2 > . . .]
- [GROUP BY < { col_name | expr | position }, . . . [WITH ROLLUP] >]
- [HAVING < having_condition1 > { and | or } < having_condition2 > . . .]
- [ORDER BY < { col_name | expr | position } [ASC | DESC], . . . >]
- [LIMIT < { [offset,] row_count | row_count OFFSET offset } >]
- [{ INTO OUTFILE '*file_name*' | INTO DUMPFILE '*file_name*' | INTO *var_name* [, *var_name*], . . . }]

sequence of clauses

$$\begin{tikzcd}
\text{WHERE} \arrow[r] & \text{GROUP BY} \arrow[r] & \text{HAVING} \arrow[r] & \text{ORDER BY} \arrow[r] & \text{LIMIT}
\end{tikzcd}$$

select statement… execution

$$\begin{tikzcd}
\text{FROM} \arrow[r] & \text{WHERE} \arrow[r] & \text{GROUP BY} \arrow[r] & \text{HAVING} \arrow[r] & \text{SELECT} \arrow[r] & \text{DISTINCT} \arrow[d, bend left] \\
 & & & & \text{LIMIT} & \text{ORDER BY} \arrow[l]
\end{tikzcd}$$



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***$_3$, . . . 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.***

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.

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 $\boldsymbol{A}_1$, $\boldsymbol{A}_2$, $\boldsymbol{A}_3$, $\boldsymbol{A}_n$ FROM ***r***

[ORDER BY {$\boldsymbol{A}_1$, $\boldsymbol{A}_2$, $\boldsymbol{A}_3$, . . . | expr | position} [ASC | DESC] , . . .]

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 $\boldsymbol{A}_1$, $\boldsymbol{A}_2$, $\boldsymbol{A}_3$, . . . FROM $\boldsymbol{r}_1$, $\boldsymbol{r}_2$, $\boldsymbol{r}_3$, . . . [WHERE ***P***]

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

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

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

- 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

Note:

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

4. ***predicate*:**

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

in

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.

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.

aggregate functions

SUM, AVG, MAX, MIN, COUNT

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

this is invalid

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

aggregate functions

Remember:

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

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

Note:

- The aggregate functions allow you to perform the calculation of a set of rows and **return a *single* value**.
- The WHERE clause cannot refer to aggregate functions. e.g. WHERE SUM(sal) = 5000 # Invalid, Error
- The HAVING clause can refer to aggregate functions. e.g. HAVING SUM(sal) = 5000 # Valid, No Error
- Nesting of aggregate functions are not allowed.
  e.g.
  ```
  SELECT MAX(COUNT(*)) FROM emp GROUP BY deptno;
  ```
- Blank space between aggregate functions like (SUM, MIN, MAX, COUNT) are not allowed.
  e.g.
  ```
  SELECT SUM (sal) FROM emp;
  ```
- The GROUP BY clause is often used with an aggregate function to perform calculation and **return a single value for each subgroup**.
- To eliminate duplicates before applying the aggregate function is available by including the keyword DISTINCT.

Things to… Remember: *aggregate functions*

TODO

AVG([DISTINCT] *expr*)

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

e.g.

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

Things to… Remember: *aggregate functions*

TODO

SUM([DISTINCT] *expr*)

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

e.g.

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

Things to… Remember: *aggregate functions*

TODO

SUM([DISTINCT] *expr*)

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

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

- SELECT SUM(c1) "R1" FROM r;
- SELECT SUM(IF(c1 >= 0, c1, NULL)) FROM r;
- SELECT SUM(IF(c1 < 0, c1, NULL)) FROM r;

- SELECT custId, type, amount, CASE type WHEN 'd' THEN amount WHEN 'c' THEN amount * -1 END amount FROM transactions;

Things to… Remember: *aggregate functions*

TODO

MAX([DISTINCT] *expr*)

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

e.g.

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

Things to… Remember: *aggregate functions*

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;

Things to… Remember:

aggregate functions

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.

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;

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.

group by clause

select - group by

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

SELECT $\boldsymbol{G}_1$, $\boldsymbol{G}_2$, . . . , $\boldsymbol{F}_1(\boldsymbol{A}_1)$, $\boldsymbol{F}_2(\boldsymbol{A}_2)$, . . . FROM $\boldsymbol{r}_1$, $\boldsymbol{r}_2$, $\boldsymbol{r}_3$ . . .

[GROUP BY {$\boldsymbol{G}_1$, $\boldsymbol{G}_2$, . . . | expr | position}, . . . [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.

select - having

SELECT $\boldsymbol{G_1}$, $\boldsymbol{G_2}$, . . . , $\boldsymbol{F_1}(\boldsymbol{A_1})$, $\boldsymbol{F_2}(\boldsymbol{A_2})$, . . . FROM $\boldsymbol{r_1}$, $\boldsymbol{r_2}$, $\boldsymbol{r_3}$ . . .

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

[HAVING having_condition]

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]] . . .])

user-defined variables

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] . . .
```