A day without new knowledge is a lost day.

Database Technologies - MySQL

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

- `sudo apt install build-essential`

MySQL is case-insensitive

Case Sensitivity in Table Names: By default, MySQL's case sensitivity for table names depends on the operating system. On Linux, table names are case-sensitive, whereas on Windows, they are case-insensitive.

Case Sensitivity in Column Names: Column names in MySQL are case-insensitive by default.

Case Sensitivity in Data: By default, string comparisons are case-insensitive because MySQL uses the utf8_general_ci collation (Unicode Transformation Format where "ci" stands for case-insensitive).

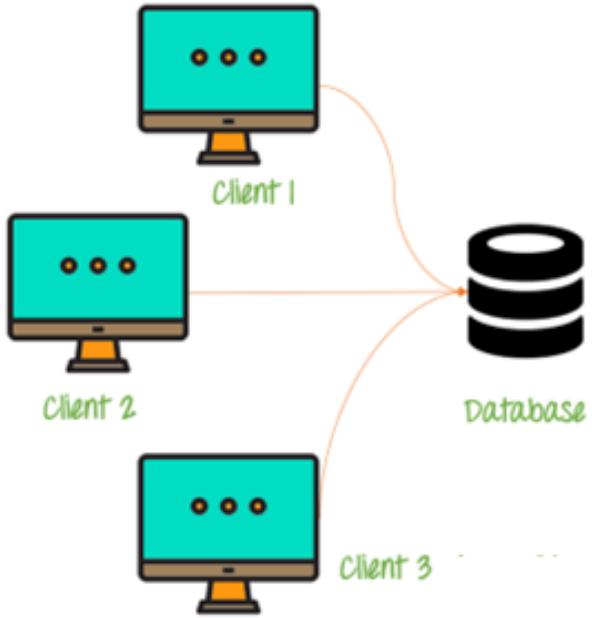
If A and a, B and b, ... are treated in the same way then it is case-insensitive.
MySQL is case-insensitive

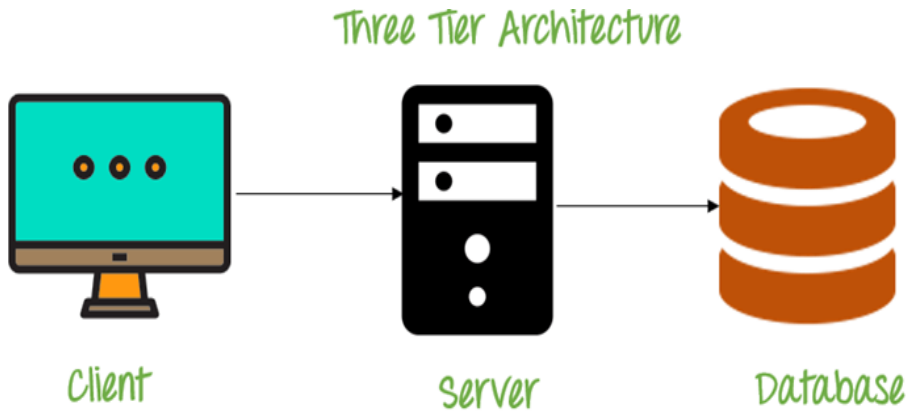
Introduction

- 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 }





Types of Database Architecture

Single-Tier Architecture

1. The database and application (UI, Business Logic, and Data are all combined) reside on the same system.
2. No network communication is required since everything runs locally.
3. Used for small-scale applications.

Two-Tier Architecture (Client-Server)

1. The (client) application (Frontend/UI + Business Logic) communicates with the database server.
2. The client sends queries, and the server processes them and returns results.
3. Used in medium-scale applications.

Three-Tier Architecture

1. Introduces a middle layer (Application Server) between the client and database.
2. The middle layer handles business logic, security, and processing before accessing the database.
3. Used in large-scale web applications.

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 (the ideas, customs, and social behaviour of a particular people or society), etc.)

Term	Simple Meaning
Referential Key	The *column* that holds a reference (like `EMP.deptno`)
Referential Integrity Constraint	The *rule* that makes sure that reference is valid

What is Relation and Relationship?

Reference / Referential key

Remember:

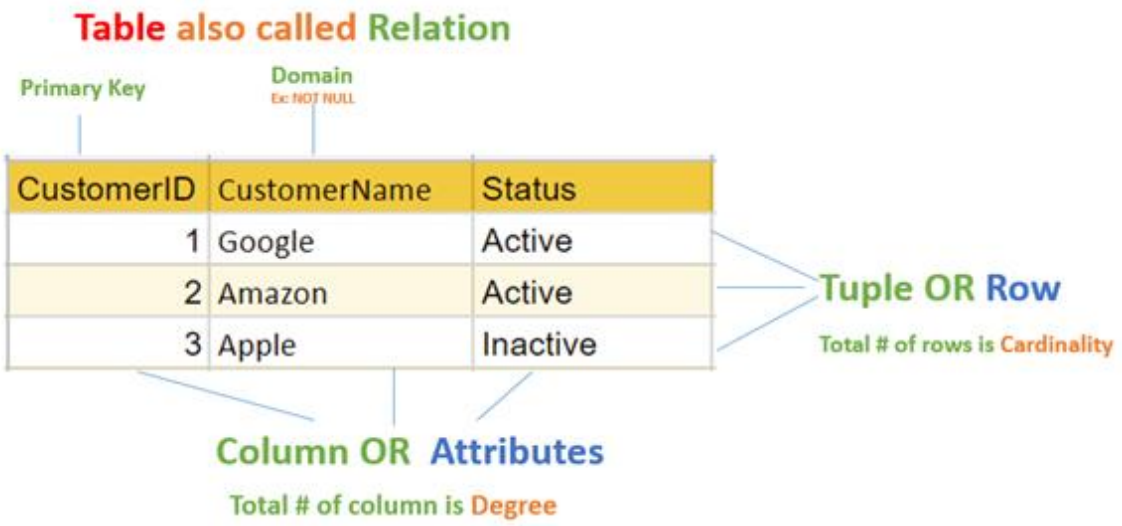
- A ***reference*** is a relationship between two tables where the values in one table refer to the values in another table. This is usually enforced using a ***foreign key*** constraint to maintain referential integrity.
  a) ***The referencing column is called the Foreign Key.***
  b) ***The referenced column is usually the Primary Key of the parent table.***
- A ***referential key*** is a column or set of columns in a table that refers to the ***primary key*** column 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. In Relational Algebra, a relation is a table with rows and columns, just like in a Relational Database Management System (RDBMS). It represents a set of tuples (records) that share the same structure. Relation is a Logical Instantiation/Model of a TABLE.

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

Primary/Foreign key is used to specify this relationship.



Remember:

Foreign Key is also known as

- referential constraint
- referential integrity constraint. (Referential Integrity is a constraint (rule) in a relational database that ensures the relationship between two tables remains consistent.)

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

A File-Oriented System is the traditional way of storing and managing data before databases (like MySQL, Oracle, . . .) were introduced.

- Data is stored in separate files (text files, spreadsheets, binary files).
- Each application manages its own files independently.
- There is no central control over data.

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
```

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

file-oriented system

File Anomalies

c:\employee.txt

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

file attributes

- File Name
- Type
- Location

file permissions

- File permissions
- Share permissions

search empl ID=1

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

search emp_name

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

advantages & disadvantage of file-oriented system

advantages of file-oriented system

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

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

disadvantage of file-oriented system

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

- **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 customer 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 also if the saving account department wants to share data with loan department, they need to manually copy files, leading to delays because File Systems do not support multi-user environments.
- **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.
- (Suppose a loan data is in one file and account holder data in another, there is no easy way to analyze account holder data with his loan status.)
- **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.

disadvantage of file-oriented system

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

Disadvantage of File-oriented system

- **Data Integrity**: Data integrity refers to the accuracy and consistency of data. In a file-oriented system, enforcing data integrity is difficult because there are no built-in mechanisms to ensure that data is valid or consistent across multiple files.
  (the balance field value must be grater than 5000.)
- **Concurrency Issues:** When multiple users or applications try to access and modify a file at the same time, concurrency problems can arise.
  (if two users attempt to update the same file simultaneously, it can lead to data corruption or loss of data.)
- **Lack of Flexibility:** Modifying the structure of files, such as adding new fields or changing data formats, can be difficult and time-consuming. Changes might require manual updates to each file or even rewriting entire applications that interact with the files.
- **Poor Scalability:** As the amount of data grows, file-based systems become less efficient and more difficult to manage. Searching through large files can be slow, and as more files are added, the complexity of managing the system increases.

***Relation Schema*:** A relation schema represents name of the relation with its attributes, every attribute would have an associated domain.

e.g.

- **Student**(rollNo:INT, name:VARCHAR(20), address:VARCHAR(50), phone:VARCHAR(12), age:INT, PRIMARY KEY(rollNo)) is relation schema for STUDENT
- **Customers**(CustomerID:INT, Name:VARCHAR(50), Email:VARCHAR(100), City:VARCHAR(50), PRIMARY KEY(CustomerID)) is relation schema for CUSTOMERS

DBMS

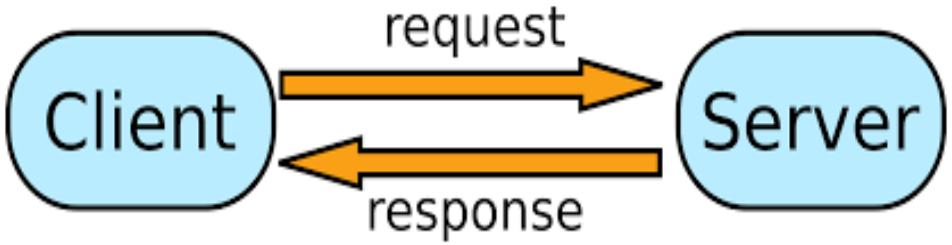
- **database:** Is the collection of **related data** which is **organized,** database can store and retrieve large amount of data easily, which is stored in one or more data files by one or more users, it is called as **structured data.**
- **management system**: it is a software, designed to **define**,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

relational database management system?

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

RDBMS supports

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



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

object relational database management system?

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

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

Advantage of ORDBMS

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

object relational database management system?

- CREATE or REPLACE TYPE ***address*** AS OBJECT(city VARCHAR2(10), state VARCHAR2(2));
- CREATE TABLE person(id INT, name VARCHAR2(10), addr ***address***);
- INSERT INTO person VALUES(1, 'saleel', address('baroda', 'GJ'));
- SELECT id, name, ***n***.addr.city FROM person ***n***;

- CREATE or REPLACE TYPE ***city*** AS VARRAY(3) OF VARCHAR2(10);
- CREATE TABLE x (id INT, ename VARCHAR2(10), c ***city***);
- INSERT INTO x values(1, 'saleel', ***city***('baroda', 'surat', 'bharuch'));
- SELECT n.id, n.ename, nn.column_value FROM x n, TABLE(n.c) nn;

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.

- **Relation (Table)** – In relational model, relations are saved in the form of Tables. A table has rows and columns.
- **Attribute (Column)** – Attributes are the properties that define a relation. **e.g.** (**roll_no, name, address, age, . . .)**
- **Tuple (Row/Record)** – 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), Attributes (Column Names), Domain of Attributes (Data Types & Allowed values), Constraints (Primary Key, Foreign Key, etc.).

  e.g. **Customers**(CustomerID:INT, Name:VARCHAR(50), Email:VARCHAR(100), City:VARCHAR(50), PRIMARY KEY(CustomerID)) is relation schema for CUSTOMERS
- **Attribute domain** – An attribute domain in a relational database refers to the set of allowed values for an attribute (column). It defines the data type and constraints that restrict the values an attribute can take.

Remember:

- In database management systems, **null** (***absence of a value***) 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	ABC
2	salesman	Ruhan	yyyy-mm-dd	••••••
5	9500	manager	yyyy-mm-dd	••••••
5	Salesman	Rahul Patil	yyyy-mm-dd	••••••

Properties of Relational Tables:

- Values are atomic (no multivalued cells).
- Column values are of the same kind. (***Attribute Domain*: Every attribute has some pre-defined datatypes, format, constraints of a column, and defines the range of values that are valid for that column known as attribute domain.)**
- Each row is unique.
- The sequence/order of columns is **irrelevant** – (unimportant).
- The sequence/order of rows is **irrelevant** – (unimportant).
- Each **table name**, attribute/column must have a unique name.
- Attributes may contain **NULL** (unknown/missing values)

What is data?

what is data?

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

Data can be in the form of Text or Multimedia

e.g.

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

Remember:

- A number is a mathematical value used to count, measure, and label.

What is Entity Relationship Diagram?

Entity Relationship Diagram (ER Diagram)

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

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

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

What is Entity?

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

entity

In relation to a database , an entity is a

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

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

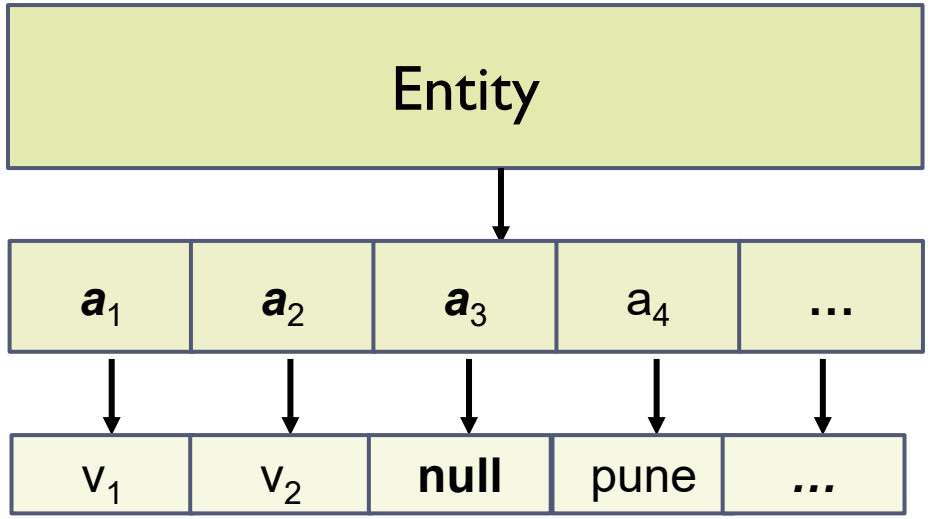
Every entity has its own characteristics.

In database management systems, **null** (***absence of a value***) 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**(***rollNo***:INT, ***name***:VARCHAR(20), ***address***:VARCHAR(50), ***age***:INT)



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 attribute name can be used for two (or more) attributes as long as the attributes are in different relations.

attributes

- **Simple / Atomic Attribute** --VS-- **Composite Attribute**
  (Can't be divided further) (Can be divided further)
- **Single Value Attribute** --VS-- **Multi Valued Attribute**
  (Only One value) (Multiple values)
- **Stored Attribute** --VS-- **Derived Attribute**
  (Only One value) (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 holds exactly one value for a given record at any point in time is known as single valued attribute. Single-valued attributes are typically used to provide a unique identifier for a record.
  e.g. manufactured part can have only one serial number, voter card ID, blood group, branchID 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, TotalPrice can be derived from Quantity × UnitPrice.

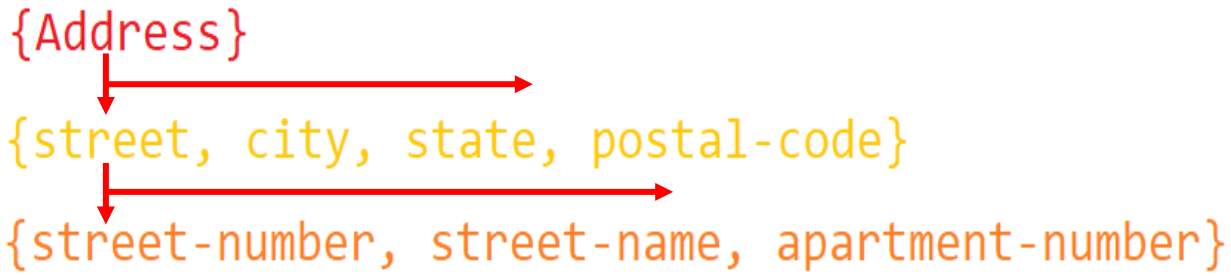
Composite **VS** Multi Valued Attribute

Composite Attribute

composite / multi valued attributes

Person Entity

- *Name* attribute: (`firstName` + `middleName` + `lastName`)
- *PhoneNumber* attribute: (`countryCode` + `cityCode` + `phoneNumber`)
- *Date* attribute: (`Day` + `Month` + `Year`)
- *productDimensions* attribute: (`Length` + `Width` + `Height`)
- *carRegistration* attribute: (`State_Code` + `Series` + `Number`)



Multi Valued Attribute

Person Entity

- *Hobbies* attribute: [reading, hiking, hockey, skiing, photography, . . .]
- *SpokenLanguages* attribute: [Hindi, Marathi, Gujarati, English, . . .]
- *Degrees* attribute**:** [10^{th}, 12^{th}, BE, ME, PhD, . . .]
- *emailID* attribute**:** [saleel@gmail.com, salil@yahoomail.com, . . .]
- *Skills* attribute: [MySQL, Oracle, Redis, MongoDB, Java, . . .]

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 or EmployeeID + deptno 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. Ensures data uniqueness in many-to-many relationships. e.g. in order_details table we can have multiple products OrderID + ProductID

What is a Prime, Non-Prime Attribute?

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

Consider a relation Student(StudentID, Name, Email, Phone).

- ***Candidate Keys:*** {StudentID}, {Email}, {Phone}
- ***Prime Attributes:*** StudentID, Email, Phone (since they are part of a Candidate Key).

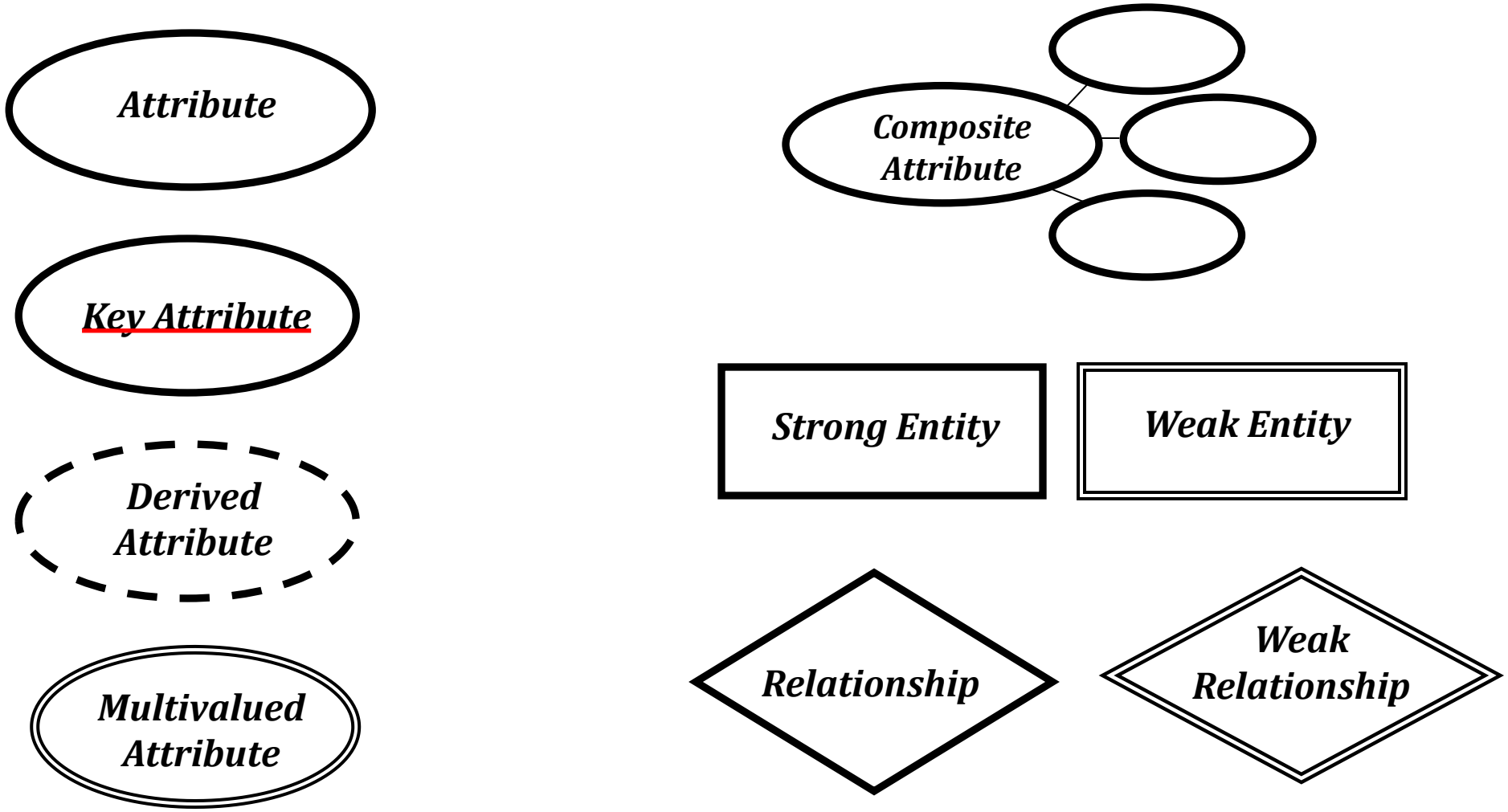
Non-prime attribute:- An attribute, which is **not a part of the prime-key** (candidate key), is said to be a non-prime attribute.

In the Student(StudentID, Name, Email, Phone) relation:

- ***Candidate Keys:*** {StudentID}, {Email}, {Phone}
- ***Prime Attributes:*** StudentID, Email, Phone
- ***Non-Prime Attribute:*** Name (because it is not part of any Candidate Key).

Entity Relationship Diagram Symbols

entity relationship diagram symbols



strong and weak entity

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

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

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

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

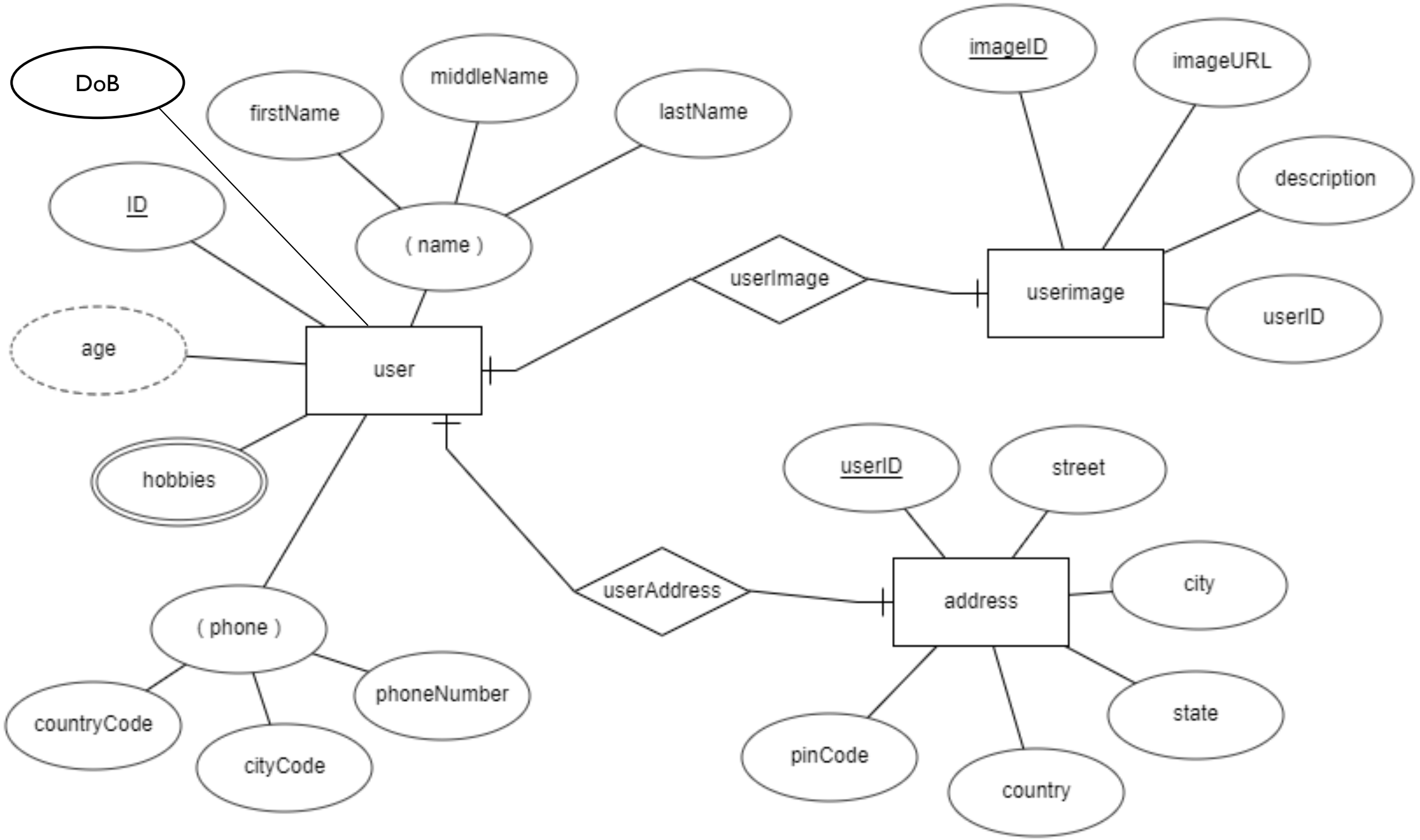
Example 3 – A customer address entity can not be created for the customer if the customer doesn't exist

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

strong and weak entity

Strong Entity	Weak Entity
— Order (OrderID)	— OrderItem (ItemID, OrderID)
— University (UniID)	— Scholarship (ScholarshipID, UniID)
— Patient (PatientID)	— MedicalRecord (RecordID, PatientID)
— Account (AccountID)	— Transaction (TransactionID, AccountID)
— Student (StudentID)	— Grade (GradeID, StudentID)
— Vehicle (VehicleID)	— InsurancePolicy (PolicyID, VehicleID)
— Hotel (HotelID)	— RoomBooking (BookingID, HotelID)
— Product (ProductID)	— WarrantyClaim (ClaimID, ProductID)
— Student (StudentID)	— AttendanceRecord (RecordID, StudentID)
TODO	TODO
TODO	TODO
TODO	TODO
TODO	TODO

entity relationship diagram



What is a degree, cardinality and union in database?

What is a degree, cardinality 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 $\mathbf{d(}\boldsymbol{R}\mathbf{)}$.
- **Cardinality |R|**: Total number of **tuples/rows** present in a relation/table, **is called cardinality of a relation** and is denoted by $|\boldsymbol{R}|$, which changes dynamically as rows are inserted or deleted.

  Cardinality is the numerical relationship between rows of one table and rows in another. Common cardinalities include *one-to-one*, *one-to-many*, and *many-to-many*.
- **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 $\mathbf{d(R)}$**.
  2. Domains of the respective attributes should also be same.
  3. Column names don't need to match, but positions must match.

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 KEY 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 t_2 relationship then it should always reference a corresponding value in the second table t_1 :- $t_1[PK] = t_2[FK]$ or it should be null.

- **Domain integrity:** A domain is a set of values of the same type (data type, range, and format).

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)

Domain integrity is enforced using the following constraints:

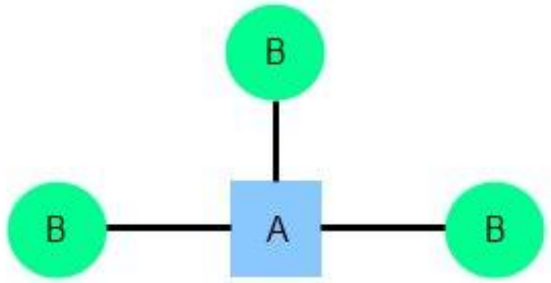
Constraint	Description	Example
Data Type	Ensures that values match a specific type (e.g., INT, VARCHAR, DATE, . . .).	age INT NOT NULL (Only integers allowed)
NOT NULL	Prevents null (empty) values in a column.	name VARCHAR(50) NOT NULL
PRIMARY	Unique and Not Null; identifies rows	id INT PRIMARY KEY
UNIQUE	No duplicate values	id INT UNIQUE
CHECK	Restricts values based on a condition.	salary DECIMAL(10, 2) CHECK (salary > 0)
DEFAULT	Sets a default value if none is provided.	status VARCHAR(10) DEFAULT 'Active'
ENUM	Limits a column to predefined values.	gender ENUM('Male', 'Female', 'Other')
SET	Allows multiple predefined values.	roles SET('Admin', 'Editor', 'User')

Common relationships

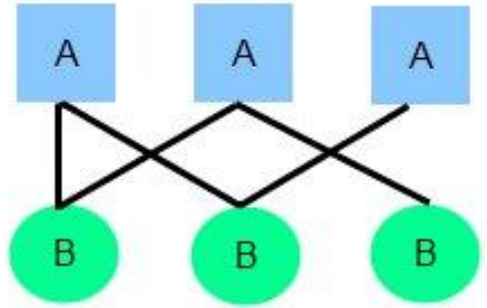
Common relationship

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

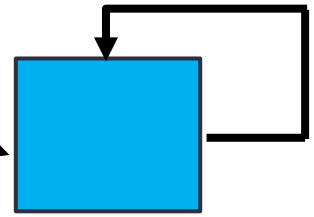
2. one-to-many (1:M)



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



4. Self-Referencing (Recursive)



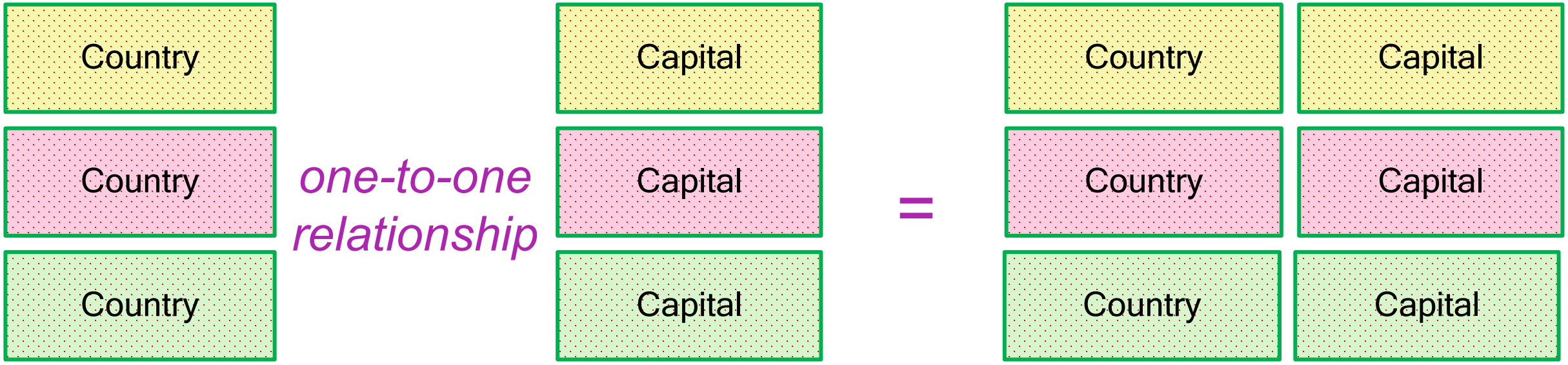
one-to-one relationship

One row in **Table A** corresponds to **one and only one** row in **Table B**.

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.

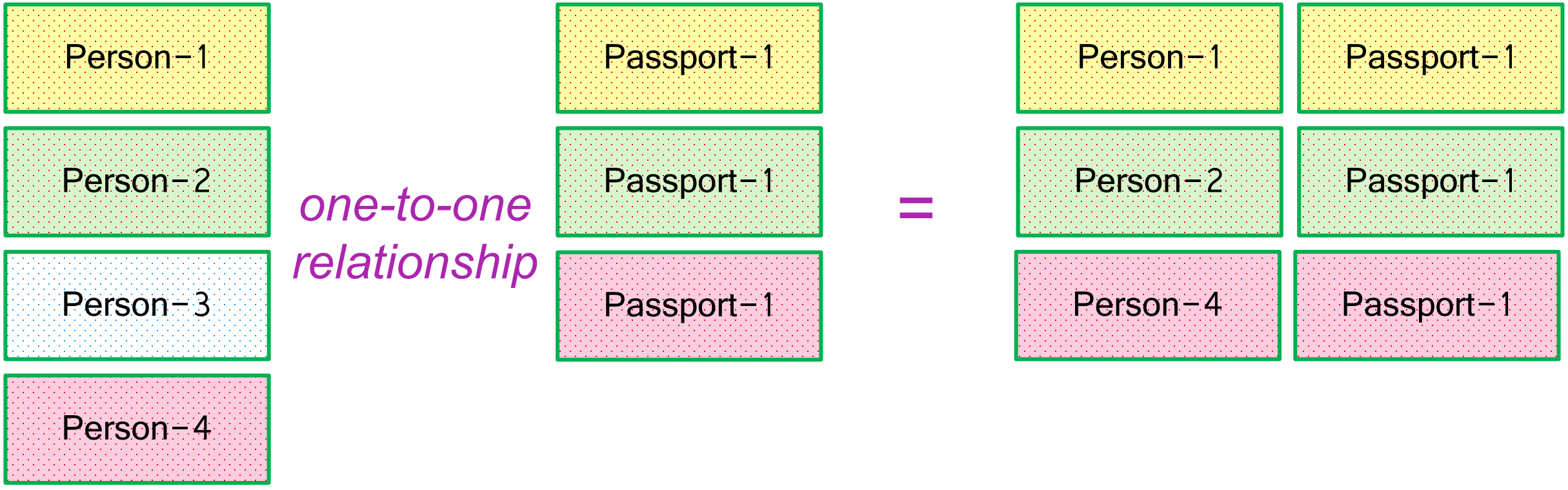
A *one-to-one* relationship is a type of cardinality that refers to the relationship between two entities ***R*** and ***S*** in which one element of entity ***R*** may only be linked to zero/one element of entity ***S***, and vice versa.



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.



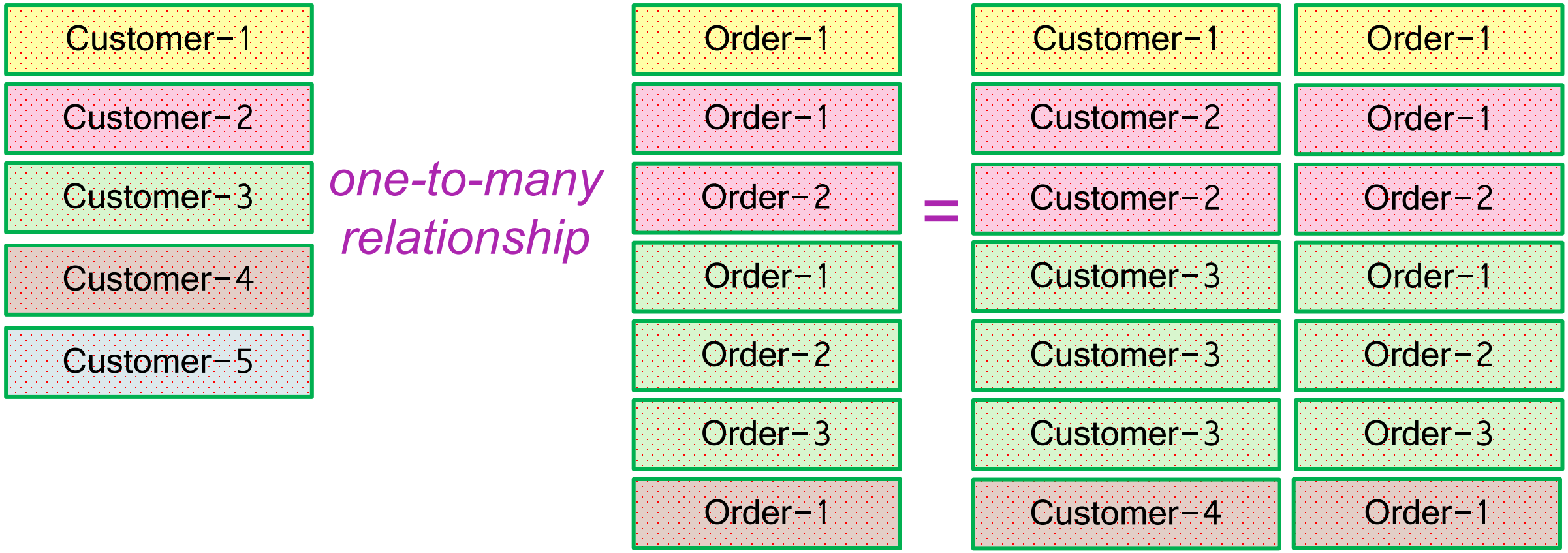
one-to-many relationship

One row in **Table A** can relate to **many rows in Table B**, but **many rows in Table** B **belongs to only one Table A**.

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.

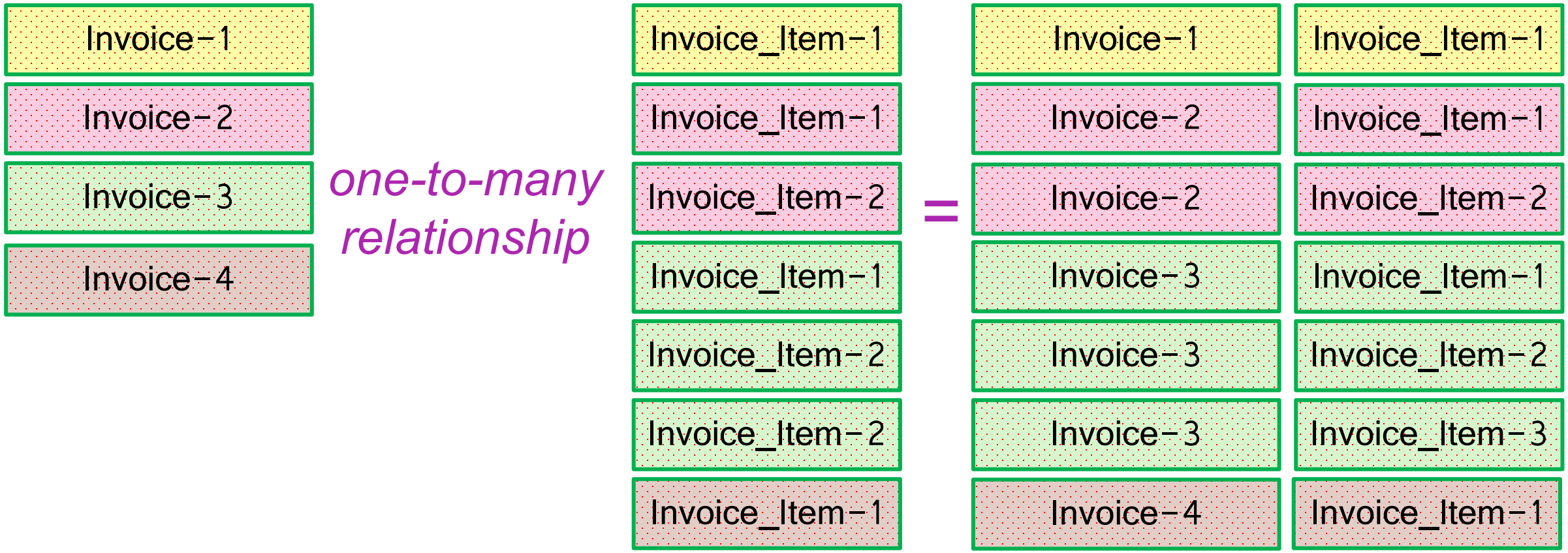
a *one-to-many* relationship is a type of cardinality that refers to the relationship between two entities ***R*** and ***S*** in which an element of ***R*** may be linked to many elements of ***S***, but a member of ***S*** is linked to only one element of ***R***.



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***.

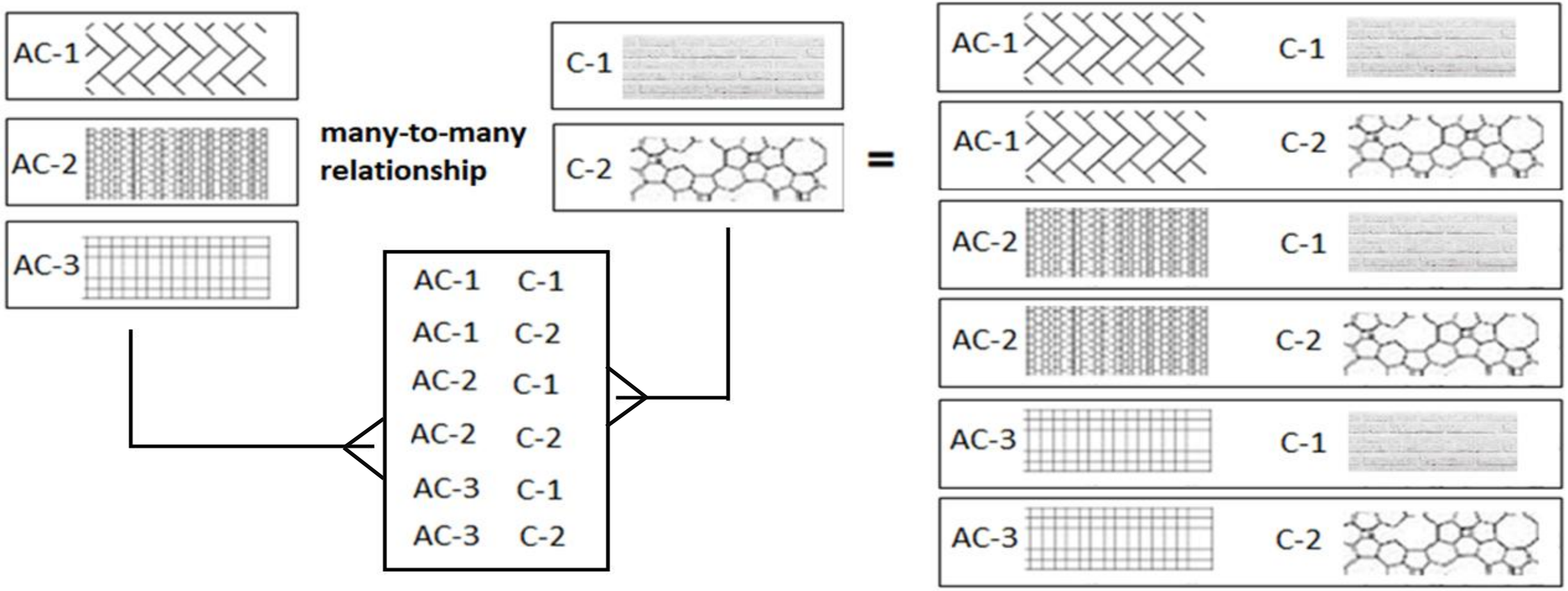


many-to-many relationship

One row in **Table A** can relate to **many rows in Table B**, and one row in **Table B** can also relate to **many rows in Table A**

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.

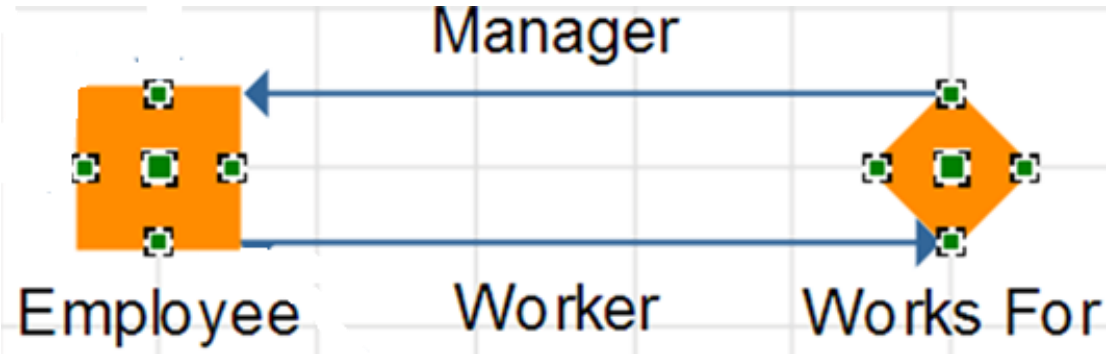


self-referencing relationship

An entity (table) is **related to itself** using a **foreign key** that points to its own primary key.

self-referencing relationship

A *"self-referencing"* or *"recursive"* relationship in databases or data structures means that a record within a table can reference another record in the same table.



Product Categories and Subcategories

CategoryID	CategoryName	ParentCategoryID
1	Electronics	NULL
2	Phones	1
3	Laptops	1
4	Smartphones	2
5	Gaming Laptops	3

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

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

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

What is SQL?

Remember:

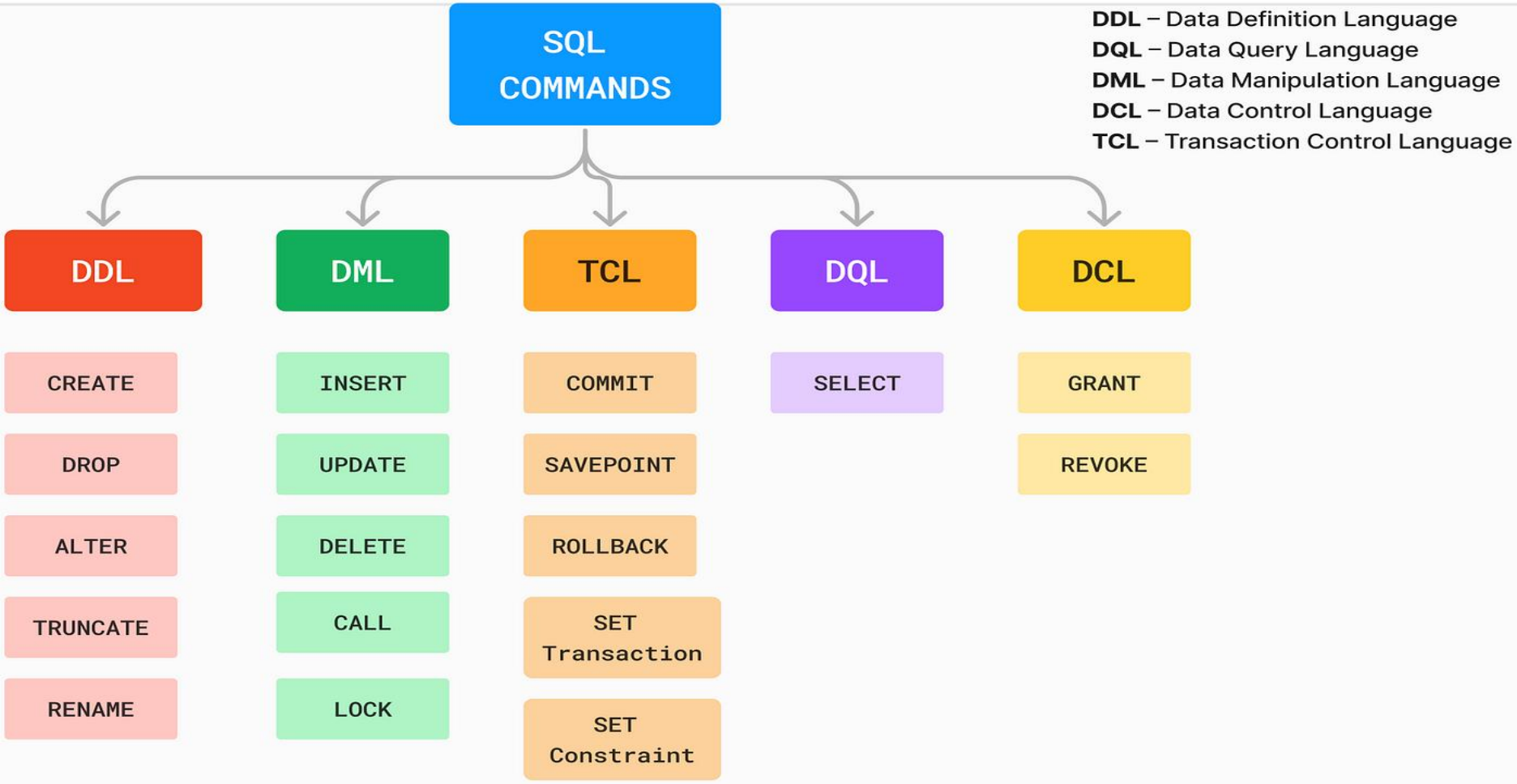
what is sql?

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

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

Remember:

- An ***implicit commit*** occurs ***automatically*** in MySQL ***without the need of COMMIT command***. This means changes made by the SQL statement are immediately saved to the database and ***cannot be rolled back***.
- An ***explicit commit*** is done by the user issuing a ***COMMIT*** command to ***manually save all changes*** made in the current transaction.



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.

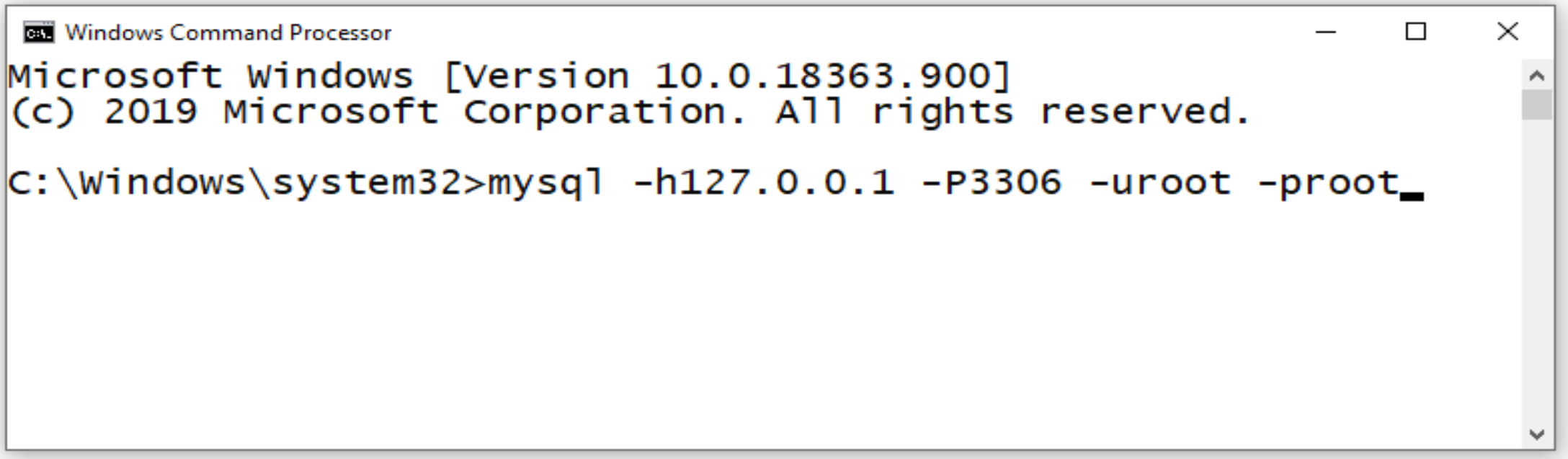
Reconnect to the server	**\r**
Execute a system shell command	**\!**
Exit mysql	**\q**
Change your mysql prompt.	**prompt str or \R str**

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*]



SHOW DATABASES

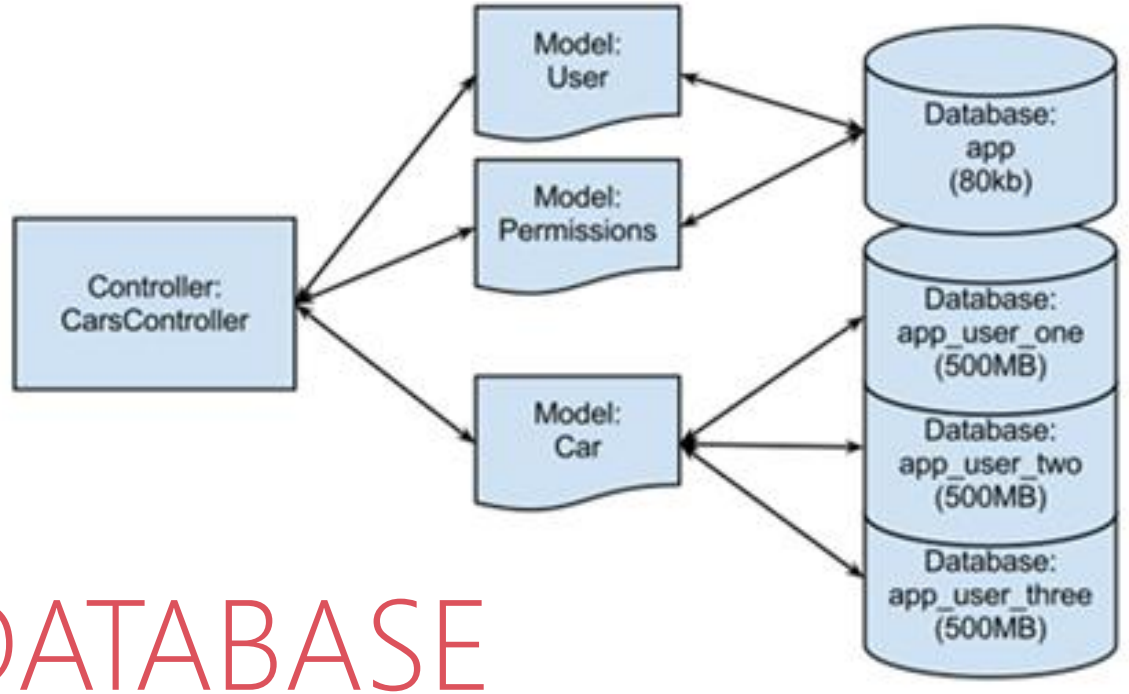
SHOW DATABASES Syntax

```
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 DATABASE

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 DATABASE 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;
```

SHOW COLUMNS

SHOW COLUMNS Syntax

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

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

SHOW TABLES

SHOW TABLES Syntax

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

- SHOW TABLES;
- SHOW FULL TABLES; // WITH TABLE TYPE
- SHOW TABLES FROM USER01;
- SHOW TABLES WHERE TABLES_IN_USER01 LIKE 'E%' OR TABLES_IN_USER01 LIKE 'B%';
- SHOW TABLES WHERE TABLES_IN_USER01 IN ('EMP');

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.

quantity VARCHAR(20) -- values: 20, '20', 'twenty'

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';*

datatype - numeric

- CREATE TABLE z1 (c1 ZEROFILL); // then Insert few records
- CREATE TABLE z1 (c1 INT(4) ZEROFILL);

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

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

ColumnName 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');`

Note:

- BOOL and BOOLEAN are **synonym of TINYINT(1)**

Remember:

MySQL performs type conversion when comparing values of different types.

- 'b' is a string. When compared to a number (0), MySQL converts the string to a number.
- 'b' → numeric conversion = 0 (since it's not a valid number).

datatype – enum

NOTE:

- An ENUM column can have a maximum of **65,535** distinct elements.
- Each ENUM value is stored as a number internally, starting from 1.
- 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 'C' ); // Valid default value for 'COL2'
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.
- You can use **membership + 0** to get the index value if needed.

```
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
```

NOTE:

datatype – set

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

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

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

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

```
FIND_IN_SET(str, { strlist | Field } )

SELECT FIND_IN_SET('Concert', 'Movie,Music,Concert');

SELECT * FROM clients WHERE FIND_IN_SET('Music', interest);
```

MySQL store them as integers (bit values), but when you insert, you must give string values, not numbers (unless you know the internal bit mapping).

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.

datatype – set

Decimal	Binary	Stored Text
0	0000	(empty set)
1	0001	Red
2	0010	Blue
3	0011	Red,Blue
4	0100	Green
5	0101	Red,Green
6	0110	Blue,Green
7	0111	Red,Blue,Green
8	1000	Yellow
9	1001	Red,Yellow
10	1010	Blue,Yellow
11	1011	Red,Blue,Yellow
12	1100	Green,Yellow
13	1101	Red,Green,Yellow
14	1110	Blue,Green,Yellow
15	1111	Red,Blue,Green,Yellow

- `CREATE TABLE x(c1 INT , c2 SET('Red','Blue','Green', 'Yellow'));`

MySQL store them as integers (bit values), but when you insert, you must give string values, not numbers (unless you know the internal bit mapping).

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

Remember:

- Max 4096 columns per table provided the row size <= 65,535 Bytes.
- The NULL value is different from values such as 0 for numeric types or the empty string for string types.

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.

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

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.

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:
AUTO_INCREMENT = <number> // must be used with AUTO_INCREMENT definition
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;

- Literals, built-in functions (both deterministic and nondeterministic), and operators are permitted.
- Subqueries, parameters, variables, and stored functions are not permitted.
- An expression default value cannot depend on a column that has the AUTO_INCREMENT attribute.

default value

The DEFAULT specifies a default value for the column.

- `CREATE TABLE temp (c1 INT PRIMARY KEY AUTO_INCREMENT, c2 INT DEFAULT(c1 + c2));  // Error`
- `CREATE TABLE temp (c1 INT, c2 INT DEFAULT(c1 < c2)); // Error`
- `CREATE TABLE temp (c1 INT, c2 INT , c3 INT DEFAULT(c1 < c2)); // OK`

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 that exceeds the column maximum length. Data too long for column error will be raise.
- Inserting data into a column than does not exists, then Unknown column error will raise.
- INSERT INTO *tbl_name* (col1,col2) VALUES(15,col1*2); // is legal.
- INSERT INTO *tbl_name* (col1,col2) VALUES(col2*2,15); // is not legal, because the value for col1 refers to col2, which is assigned after col1.

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*] ...)] [(*field_name*, . . .)]
{ VALUES | VALUE } [ROW] ({ *expr* | *DEFAULT* }, . . .), [ROW] (. . .), [ROW] . . . [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

insert multiple rows

dml- insert … values

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

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

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

e.g.

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

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 ename, job, sal, sal * 1.1, sal * 1.25 FROM emp;
```

Salary increased by **10%**

Salary increased by **25%**

SELECT CLAUSE

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

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

Capabilities of SELECT Statement

1. SELECTION
2. PROJECTION
3. JOINING

Capabilities of SELECT Statement

➢ SELECTION

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

R

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

Capabilities of SELECT Statement

➢ PROJECTION

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

R

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

Table DEPARTMENTS

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

Table EMPLOYEES

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

Capabilities of SELECT Statement

➢ JOINING

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

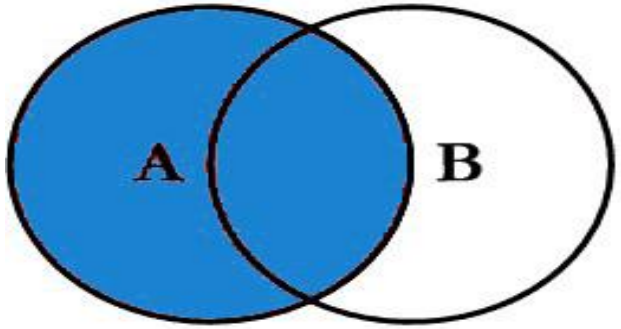
R

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

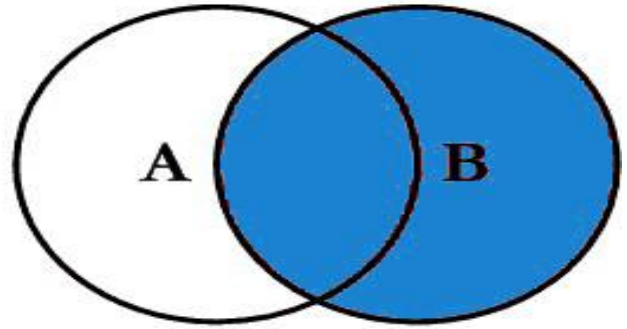
S

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

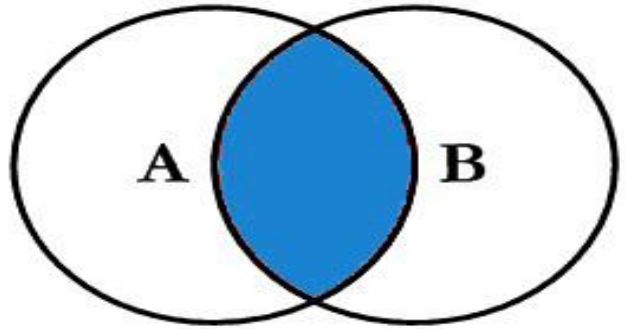
SQL JOINS



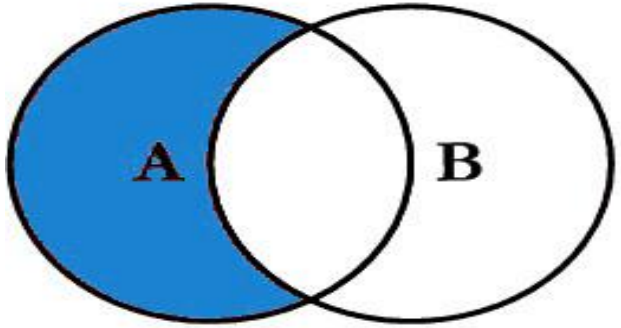

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



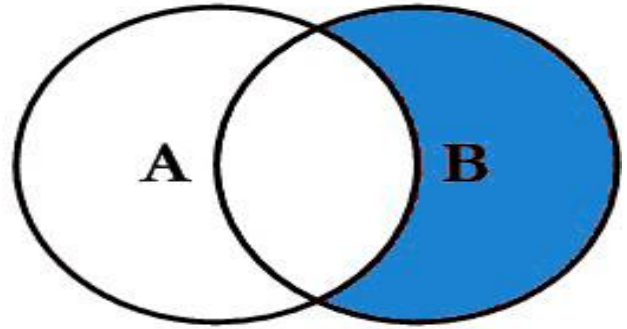

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



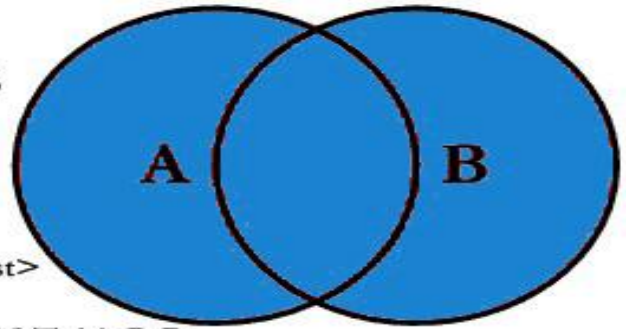

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



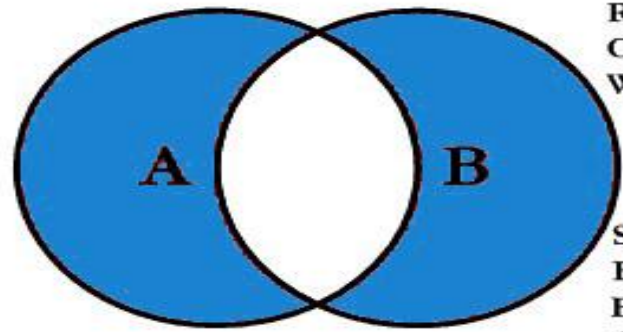

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




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




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




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

select statement

SELECTION Process

SELECT * FROM <table_references>

selection-list | field-list | column-list

Remember:

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

PROJECTION Process

SELECT column-list FROM <table_references>

selection-list | field-list | column-list

Remember:

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

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

```
SELECT ename, job, sal, sal * 1.1, sal * 1.25 FROM emp;
```

Salary increased by **10%**

Salary increased by **25%**

single-table update

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

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* | *query* } [, *col_name2* = { *expr2* | *DEFAULT* | *query* }] . . .
[WHERE *where_condition*] [ORDER BY . . .] [LIMIT *row_count*]

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

single-table delete

DELETE is used to delete tuples from a relation.

delete can violate only in referential integrity.

Important:

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

single-table delete

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

```
DELETE FROM tbl_name
   [PARTITION (partition_name [, partition_name] . . .)]
   [WHERE where_condition]
   [ORDER BY . . .]
   [LIMIT row_count]
```

Note:

- LIMIT clauses apply to single-table deletes, but not multi-table deletes.

- `DELETE FROM temp;`
- `DELETE FROM temp ORDER BY loc LIMIT 2;`
- `DELETE FROM temp WHERE deptno < 50;`
- `DELETE FROM temp WHERE deptno < 50 LIMIT 2;`

auto_increment column

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

Gap occurs when :- INSERT fails or is rolled back

auto_increment

col_name data_type AUTO_INCREMENT [UNIQUE [KEY] | [PRIMARY] KEY]

IDENTITY is a synonym to the ***LAST_INSERT_ID*** variable.

Remember:

- There can be **only one** AUTO_INCREMENT column per table.
- The column defined with AUTO_INCREMENT **must be indexed**.
- it cannot have a DEFAULT value.
- it contains only **positive values**.
- AUTO_INCREMENT works only with **integer types** (TINYINT, SMALLINT, INT, BIGINT . . .)
- when you insert a value of NULL or 0 into AUTO_INCREMENT column, it generates **next value**.
- If a row is deleted or a transaction is rolled back, the auto_increment number is not reused.
- Truncating a table **resets** AUTO_INCREMENT back to 1.
- If you insert multiple rows in one INSERT, each row gets a **different value**.
- *use LAST_INSERT_ID*() function to find the row that contains the most recent AUTO_INCREMENT value.

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

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

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

auto_increment

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

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

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

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

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

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

- **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

A SQL generated column is a type of column that stores values calculated from an expression applied to data in other columns of the same table. The value of a generated column cannot be altered manually and is automatically updated whenever the data it depends on changes.

Remember:

- Stored functions and user-defined functions are not permitted.
- Stored procedure and function with parameters are not permitted.
- Variables (system variables, user-defined variables, and stored program local variables) are not permitted.
- Subqueries are not permitted.
- You cannot use a generated column in a DEFAULT clause.
- The AUTO_INCREMENT attribute cannot be used in a generated column definition.
- You **cannot insert/update** values into a generated column directly.
- 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*) | (CASE WHEN . . . THEN . . . ELSE . . . END)
[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
  ```

generated always as column with case when . . . end

col_name data_type [GENERATED ALWAYS] AS (*expression*) | (CASE WHEN . . . THEN . . . ELSE . . . END)

- ```
  CREATE TABLE loan(
    loanID BIGINT PRIMARY KEY AUTO_INCREMENT,
    bankName VARCHAR(50),
    accountNumber VARCHAR(30),
    amount INT,
    riskStatus VARCHAR(30) GENERATED ALWAYS AS (
    CASE
       WHEN amount < 1000 THEN 'Low Risk'
       WHEN amount BETWEEN 1000 AND 2000 THEN 'Above Average Risk'
       ELSE 'High Risk’
    END));
  ```
  ```

generated always as column with ENUM and case when . . . end

col_name data_type [GENERATED ALWAYS] AS (*expression*) | (CASE WHEN . . . THEN . . . ELSE . . . END)

- ```
  CREATE TABLE patientAdmission(
    admissionID INT PRIMARY KEY,
    patientName VARCHAR(10),
    referred_by ENUM('ref1', 'ref2', 'ref3', 'ref4', 'ref5', 'ref9'),
    admissionType VARCHAR(40) GENERATED ALWAYS AS (
    CASE
       WHEN referred_by = 'ref1' THEN 'Emergency'
       WHEN referred_by = 'ref2' THEN 'Urgent'
       WHEN referred_by = 'ref3' THEN 'Elective'
       WHEN referred_by = 'ref4' THEN 'Newborn'
       WHEN referred_by = 'ref5' THEN 'Trauma Center'
       WHEN referred_by = 'ref9' THEN 'Information Not Available'
    END));
  ```
  ```

generated always as column with ENUM and case when . . . end

col_name data_type [GENERATED ALWAYS] AS (*expression*) | (CASE WHEN . . . THEN . . . ELSE . . . END)

- ```
  CREATE TABLE ticketReservation (
    ticketID INT PRIMARY KEY AUTO_INCREMENT,
    reservationType ENUM('GN', 'TQ','LD','FT', 'HP', 'OQ'),
    description VARCHAR(100) GENERATED ALWAYS AS (
    CASE
       WHEN reservationType = 'GN' THEN 'General Quota'
       WHEN reservationType = 'TQ' THEN 'Tatkal Quota'
       WHEN reservationType = 'LD' THEN 'Ladies Quota'
       WHEN reservationType = 'FT' THEN 'Foreign Tourists Quota'
       WHEN reservationType = 'HP' THEN 'Physically Handicapped'
       WHEN reservationType = 'OQ' THEN 'Headquarters/High Official, Parliament House, Defence, Duty Pass Quota'
   END));
  ```
  ```

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 * queries by default does not include invisible columns.
- Invisible columns can be indexed and used in constraints (PK, UNIQUE, FK).
- You can change visibility with ALTER TABLE.
- Both STORED and VIRTUAL generated columns can also be declared VISIBLE or INVISIBLE.

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 INVISIBLE;
```

varbinary column

VARBINARY is a variable-length binary string column. It's similar to VARCHAR, but it stores binary data, not text.

Note:

- VARBINARY stores binary strings, not character strings.
- Comparisons are based on numeric byte values, not text collation.
- You can index VARBINARY columns.
- You can assign default values but must be valid binary literals.

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;
```

```
CREATE TABLE temp(
  c1 INT PRIMARY KEY,
  c2 VARBINARY(4) DEFAULT 0x89504E47);
```

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

zerofill value

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

Note:

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

zerofill column

col_name data_type ZEROFILL

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

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

- SELECT * FROM employee;
- SELECT ID, LENGTH(ID), salary, LENGTH(salary) FROM employee;

zerofill column

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

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

int

Note:

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

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.

In MySQL, when you run SHOW INDEXES or SHOW CREATE TABLE, you may see MUL under the Key column.

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 (Since it's not unique, the column can contain **duplicate values**.).

constraints

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

Note:

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

Remember: *col_name data_type* PRIMARY KEY

- A primary key cannot be NULL (absence of a value).
- 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.**

Database	Max Columns in Primary Key
MySQL	16
PostgreSQL	32
Oracle	32
SQL Server	16
DB2	16
MariaDB	16

PRIMARY KEY constraint

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

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

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

add / drop Primary Key using Alter

Remember:

col_name data_type UNIQUE KEY

- A unique key can be NULL (absence of a value).
- 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.

- ALTER TABLE users DROP INDEX <COLUMN_NAME>;
- ALTER TABLE users DROP INDEX U_USER_ID; #CONSTRAINT NAME

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

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

add/drop Unique Key using Alter

[CONSTRAINT [*symbol*]] FOREIGN KEY (*col_name*, . . .) REFERENCES *tbl_name* (*col_name*, . . .)

[ON DELETE *CASCADE* | *SET* NULL]
[ON UPDATE *CASCADE* | *SET* NULL]

FOREIGN KEY constraint

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

constraints – foreign key

Remember:

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

Note:

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

```
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);
```

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

add/drop Foreign Key Constraint using Alter

col_name data_type CHECK(*expr*)

Check Constraint

operators like =, >, <, BETWEEN, IN, LIKE. Can use.

- 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)

constraints – check

CHECK condition expressions must follow some rules.

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

Note:

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

Remember:

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

- table_name_chk_n

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

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

Add Check Constraint using Alter

alter table

ALTER TABLE changes the structure of a table.

Note:

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

syntax

alter table

ALTER TABLE tbl_name

[*alter_specification* [, *alter_specification*] . . .

- | ADD [COLUMN] *col_name column_definition* [FIRST | AFTER *col_name*]
- | ADD [COLUMN] (*col_name column_definition*, . . .)
- | ADD {INDEX|KEY} [index_name] (*index_col_name*, . . .)
- | ADD [CONSTRAINT [*symbol*]] PRIMARY KEY
- | ADD [CONSTRAINT [*symbol*]] UNIQUE KEY
- | ADD [CONSTRAINT [*symbol*]] FOREIGN KEY *reference_definition*
- | CHANGE [COLUMN] *old_col_name new_col_name* column_definition [FIRST|AFTER *col_name*]
- | MODIFY [COLUMN] *col_name column_definition* [FIRST | AFTER *col_name*]
- | DROP [COLUMN] *col_name*
- | DROP PRIMARY KEY
- | DROP {INDEX|KEY} *index_name*
- | DROP FOREIGN KEY *fk_symbol*
- | RENAME [TO|AS] *new_tbl_name*
- | RENAME COLUMN *old_col_name* TO *new_col_name*
- | ALTER [COLUMN] *col_name* { SET DEFAULT {*literal* | (*expr*)} | DROP DEFAULT } | SET { VISIBLE | INVISIBLE }

drop table

Remember:

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

drop table

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

Note:

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

- DROP login;
- DROP TABLE users;
- DROP TABLE login, users;

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

continue with SELECT statement…

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

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

SELECT CLAUSE

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

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

Capabilities of SELECT Statement

1. SELECTION
2. PROJECTION
3. JOINING

Capabilities of SELECT Statement

➢ SELECTION

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

R

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

Capabilities of SELECT Statement

➢ PROJECTION

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

R

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

Capabilities of SELECT Statement

➢ JOINING

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

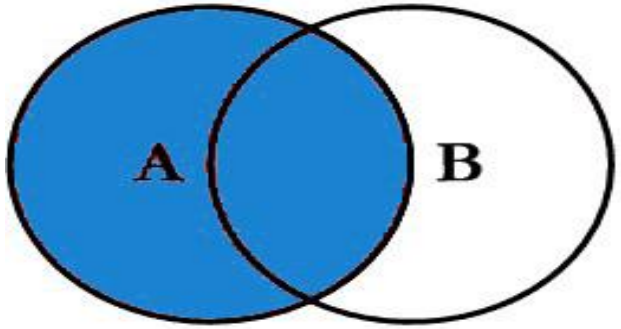
R

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

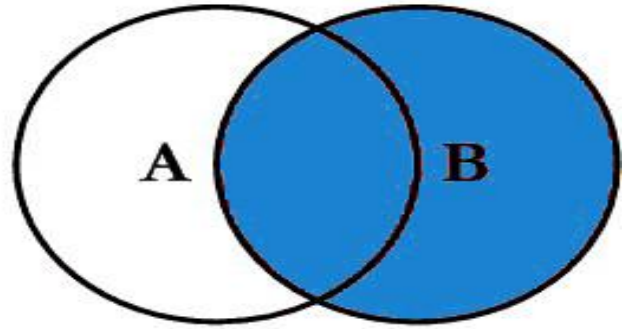
S

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

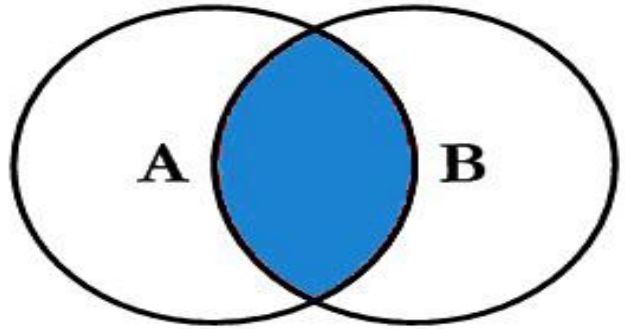
SQL JOINS



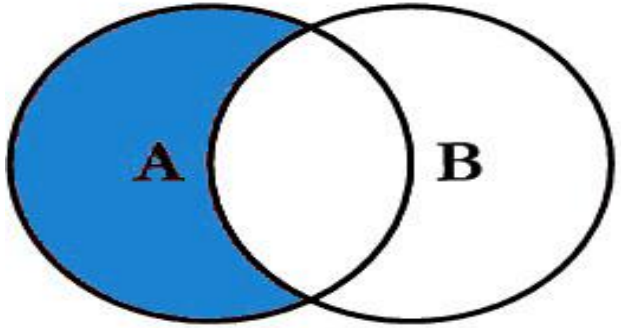

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



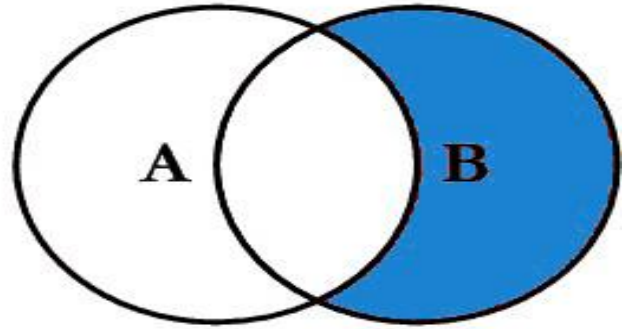

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



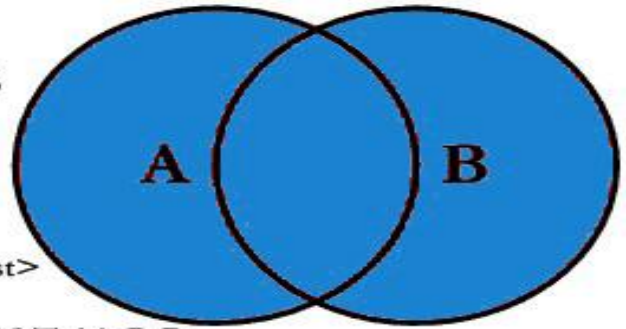

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



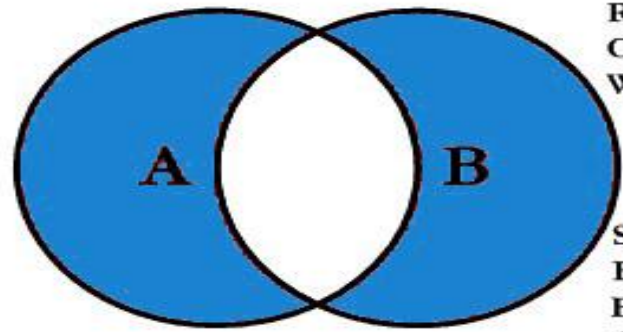

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




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




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




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

select statement

SELECTION Process

SELECT * FROM <table_references>

selection-list | field-list | column-list

When you write a SELECT query, the list of columns or expressions you mention after the SELECT keyword is called the SELECT-LIST (also referred to as FIELD-LIST).

Remember:

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

PROJECTION Process

SELECT column-list FROM <table_references>

selection-list | field-list | column-list

Remember:

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

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

column - alias

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

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

Note:

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

select statement - alias

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

column-name as new-name

table-name as new-name

Remember:

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

- SELECT empno AS *EmployeeID*, ename *EmployeeName* FROM emp;
- SELECT ID AS *'Employee ID'*, ename *"Employee Name"* FROM emp;
- SELECT * FROM emp *employee*;

comparison functions and operator

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

assignment _operator

= (assignment), :=

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

comparison functions and operator

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

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

e.g.

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

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

Note:

- AND has higher precedence than OR.

- `WHERE col * 4 < 16`
- `WHERE col < 16 / 4`

- `SELECT CONCAT(1, "saleel");`

column - expressions

"Strings are automatically converted to numbers (this behavior is known as **implicit type conversion (type coercion)** in MySQL.)."

- If a string starts with a number, MySQL extracts the number and uses it.
- If a string does not start with a number, MySQL converts it to 0.

select statement - expressions

Column EXPRESSIONS

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

- SELECT 1001 + 1;
- SELECT 1001 + '1';
- SELECT '1' + '1' ;
- SELECT '1' + 'a1';
- SELECT '1' + '1a';
- SELECT 'a1' + 1;
- SELECT '1a' + 1;
- SELECT 1 + -1;
- SELECT 1 + -2;
- SELECT -1 + -1;
- SELECT -1 - 1;
- SELECT -1 - -1;
- SELECT 123 * 1;
- SELECT -123 * 1;
- SELECT 123 * -1 ;
- SELECT -123 * -1;
- SELECT 2 * 0;
- SELECT 2435 / 1;
- SELECT 2 / 0;
- SELECT '2435Saleel' / 1;
- SELECT NULL <=> NULL;
- SELECT '20' > 15;
- SELECT 'abc' > 0;
- SELECT 'abc' = 0;
- SELECT '123abc' = 123;
- SELECT 'abc123' = 0;
- SELECT sal, sal + 1000 AS *'New Salary'* FROM emp;
- SELECT sal, comm, sal + comm FROM emp;
- SELECT sal, comm, sal + IFNULL(comm, 0) FROM emp;
- SELECT ename, ename = ename FROM emp;
- SELECT ename, ename = 'smith' FROM emp;
- SELECT c1, c1 / 1 *R1* FROM numberString;

Note:

If any expression evaluated with NULL, returns NULL.

- SELECT 2 + NULL ;
- SELECT 2 - NULL ;
- SELECT 2 * NULL ;
- SELECT 2 / NULL ;
- SELECT 2 % NULL ;

identifiers

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

identifiers

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

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

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

Note:

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

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*)

A year is a leap year if:
- Divisible by **400** → leap year
- Divisible by **4** but **not by 100** → leap year
- Otherwise → **not a leap year**

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

control flow functions - nullif

NULLIF function

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

NULLIF(*expr1*, *expr2*)

- SELECT NULLIF(1, 1) as *R1*;
- SELECT NULLIF(1, 2) as *R1*;

- shipped and actualShipDate must be same.

control flow functions - nullif

todo

NULLIF(*expr1*, *expr2*)

Table:- ord

ordid	. . .	shipdate	actualShipDate	. . .
601	. . .	1986-05-30	1986-06-02	. . .
602	. . .	1986-06-20	NULL	. . .
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
634	. . .	1987-02-22	1987-02-24	. . .
635	. . .	1987-02-23	1987-02-23	. . .

Output

ordid	. . .	shipdate	actualShipDate	. . .
606	. . .	1986-07-30	1986-07-30	. . .
608	. . .	1986-07-25	1986-07-25	. . .
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
632	. . .	1987-03-14	1987-03-14	. . .
635	. . .	1987-02-23	1987-02-23	. . .

- SELECT * FROM ord WHERE NULLIF(shipdate, actualshipdate) IS NULL;
- SELECT * FROM ord WHERE NULLIF(shipdate, actualshipdate) IS NOT NULL;

- SELECT * FROM ord WHERE shipdate = actualshipdate;

- email and userName must be same.

control flow functions - nullif

todo

NULLIF(*expr1*, *expr2*)

Table:- student

studentID	. . .	emailID	userName
1001	. . .	saleel.songs@gmail.com	saleel.songs@gmail.com
1002	. . .	sharmin@gmail.com	sharmin@gmail.com
1003	. . .	vrusha@hotmail.com	vrushali@hotmail.com
. .	. .	. .	. .
1013	. . .	NULL	NULL

- SELECT student.*, NULLIF(emailID, userName) FROM student WHERE NULLIF(emailID, userName) IS NULL;
- SELECT student.*, NULLIF(emailID, userName) FROM student WHERE NULLIF(emailID, userName) IS NOT NULL;

Output

studentID	. . .	emailID	userName
1001	. . .	saleel.songs@gmail.com	saleel.songs@gmail.com
1002	. . .	sharmin@gmail.com	sharmin@gmail.com
. .	. .	. .	. .

see difference

- SELECT student.*, NULLIF(emailID, userName) FROM student WHERE emailID = userName;

control flow functions - case

CASE function

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

```
CASE value WHEN [compare_value] THEN result [WHEN [compare_value] THEN result . . .] [ELSE result] END
```

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

- `SELECT job, SUM(CASE job WHEN 'manager1' THEN 1 ELSE 0 END) R1 FROM emp; # returns 0`
- `SELECT job, SUM(CASE job WHEN 'manager1' THEN 1 END) R1 FROM emp; # returns NULL`

control flow functions - case

CASE function

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

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

CASE WHEN EXISTS (SELECT true FROM R WHERE $R.a_1 = S.a_1$) THEN result [WHEN [*condition*] THEN result . . .] [ELSE *result*] END

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

control flow functions - case

CASE function

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

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

CASE WHEN EXISTS (SELECT true FROM R WHERE $R.a_1 = S.a_1$) THEN result [WHEN [*condition*] THEN result . . .] [ELSE *result*] END

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

```
* ORDER BY CASE
  WHEN filter = 'Debit' THEN 1
  WHEN filter = 'Credit' THEN 2
  WHEN filter = 'Total' THEN 3
  END transactionType;
```

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

```
* ORDER BY CASE status
  WHEN 'active' THEN 1
  WHEN 'approved' THEN 2
  WHEN 'rejected' THEN 3
  WHEN 'submitted' THEN 4
  ELSE 5 END statusType
```

control flow functions - case

CASE function

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

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

CASE WHEN EXISTS (SELECT true FROM R WHERE $R.a_1 = S.a_1$) THEN result [WHEN [*condition*] THEN result . . .] [ELSE *result*] END

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

- DATEDIFF(CURDATE(), hiredate) / 365.25

datetime functions

sysdate(), now(), curdate(), curtime()

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

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

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

sysdate(), now(), curdate(), curtime()

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

- SELECT SYSDATE()
- SELECT NOW()
- SELECT CURDATE()
- SELECT CURTIME()

Result in something like this:

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

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

+ 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());
- SELECT EXTRACT(YEAR_MONTH FROM NOW()) ;

Note:

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

e.g.

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

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

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

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 NOW(), NOW() + INTERVAL 1 DAY, WEEKDAY(NOW() + INTERVAL 1 DAY);`
- `SELECT * FROM emp WHERE DAY(hiredate) = 17;`
- `SELECT YEAR(hiredate), ( YEAR(hiredate) % 4 = 0 AND YEAR(hiredate) % 100 != 0 ) OR YEAR(hiredate) % 400 = 0 R1 FROM emp ;`
- `SELECT STR_TO_DATE('24/05/2022', '%d/%m/%Y');`

datetime formats

datetime formats

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

- SELECT DATE_FORMAT(NOW(), '%a');
- SELECT DATE_FORMAT(CURDATE(), '%Y-%m-01') AS *"First Day"*, LAST_DAY(CURDATE()) AS *"Last Day"*;
- SELECT hiredate, DATE_FORMAT(hiredate, '%Y-%m-01') AS *"First Day"*, LAST_DAY(hiredate) AS *"Last Day"* FROM emp;

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');`

- Print the **first** and the **last** day of every month.

datetime formats

Input

curdate()
2025-06-30

Output

First Date	Last Date
2025-06-01	2025-06-30

- SELECT DATE_FORMAT(CURDATE(), '%Y-%m-01') AS *"First Day"*, LAST_DAY(CURDATE()) AS *"Last Day"*;
- SELECT hiredate, DATE_FORMAT(hiredate, '%Y-%m-01') AS *"First Day"*, LAST_DAY(hiredate) AS *"Last Day"* FROM emp;

- ```
  CREATE TABLE hobbies(
    hobbyid ENUM('1','2','3','4'),
    hobbyName VARCHAR(20) GENERATED ALWAYS AS (ELT(hobbyid, 'Windsurfing', 'Fishing', 'Swimming', 'Shopping'))
  );
  ```

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'); #output 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. • SELECT hiredate, ELT(MONTH(hiredate),'Winter', 'Winter', 'Spring', 'Spring', 'Spring', 'Summer', 'Summer', 'Summer', 'Autumn', 'Autumn', 'Autumn', 'Winter') R1 FROM emp;

- SELECT Name, rnk, ELT(rnk, 'A+', 'A', 'B', 'C', 'D') *grade* FROM student;
- SELECT bankName, accountNumber, riskStatus, CASE WHEN riskStatus = 0 THEN 'No risk' ELSE ELT(riskStatus, 'Low', 'Moderate', 'High', 'Critical') END *riskStatus* FROM bankLoan;

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.

- SELECT UCASE(ename) FROM emp;
- SELECT LPAD(id, 7, '#'), bookName, Type, Cost FROM books;
- SELECT LPAD(prodid, 9, 'x'), Descrip, Price FROM product;
- SELECT CONCAT('TKT', LPAD(id, 7, 0)) *'Ticket ID'* FROM hall_ticket;
- SELECT CASE WHEN status = 'In Process' THEN LPAD(ordid, LENGTH(ordid) + 3, '(I)') WHEN status = 'Delivered' THEN LPAD(ordid, LENGTH(ordid) + 3, '(D)') ELSE 'TODO' END ordid, orderDate, custID, Status, Total FROM ord;

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.
INSTR(*str, substr*)	• *str,* The string that will be searched • *substr,* The string to search for in string1. If string2 is not found, this function returns 0
FIND_IN_SET(*str*, *string_list*)	• *str,* The string to search for • *string_list,* The list of string values to be searched (separated by commas)

FIND_IN_SET:- This function of MySQL is used to find the position of a string within a comma-separated list of strings.

- If *string* is not found in *string_list*, this function returns 0
- If *string* or *string_list* is NULL, this function returns NULL
- If *string_list* is an empty string (""), this function returns 0

- SELECT ename, BINARY ename FROM emp;
- SELECT CONCAT(UCASE(LEFT(ename, 1)), LCASE(SUBSTRING(ename, 2))) *"Title Case"* FROM emp;

- Find the position of the given name.

string functions - examples

FIND_IN_SET(*str*, *string_list*)

Table:- teams

TeamID	Members
1	Saleel, Sharmin, Vrushali, Gau
2	Kaushal, Natsha, Ruhan
3	Lalu, Brooke, Boy, Naru, Bhavin
4	Bhuru, Baba, Anoop, Meera, Boy
5	Dilu, Fredy
6	Daigo, Harshil, Boy, Raju, Bothyo
7	Bandish, Supriya, Sangita
8	Monika, Karishma, William, Varsha, Jack, Grace
9	Vasant, Neel
10	Nitish, Bipin, Mai, Didi

Output →

Output

```
select members, length(members) -
length(replace(members,',',''))  + 1 from teams;
```

TeamID	Members	Position
1	Saleel, Sharmin, Vrushali, Gau	0
2	Kaushal, Natsha, Ruhan	0
3	Lalu, Brooke, Boy, Naru, Bhavin	3
4	Bhuru, Baba, Anoop, Meera, Boy	5
5	Dilu, Fredy	0
6	Daigo, Harshil, Boy, Raju, Bothyo	3
7	Bandish, Supriya, Sangita	0
8	Monika, Karishma, William, Varsha, Jack, Grace	0
9	Vasant, Neel	0
10	Nitish, Bipin, Mai, Didi	0

- SELECT members, FIND_IN_SET('Boy', REPLACE(members, ' ', '')) *Position* FROM teams;

SUBSTRING_INDEX(*str*, *delimiter*, *count*) **// count can be either +ve or -ve**

string functions

Syntax	Result
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**.**

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

string functions - examples

- SELECT sal, REPEAT('$', sal/100) FROM emp;
- SELECT emailid, REPEAT('*', LENGTH(emailid)) FROM emp;
- SELECT pwd, REPEAT('*', LENGTH(pwd)) *password* FROM emp;
- SELECT c1, CONCAT(REPEAT('0', 10 - LENGTH(c1)) , c1) FROM leading_zeroes;
- SELECT ename, job, IF(ISNULL(phone), '**********', RPAD(LEFT(phone, 4), 10, '*')) FROM emp;
- SELECT 9850884228, LPAD(SUBSTR(9850884228, -4),10,'*') *R1*;
- SELECT `user name`, IF(LENGTH(SUBSTR(`user name`, INSTR(`user name`," "))) = 0,"Weak User", `user name`) *R1* FROM emp;
- SELECT LENGTH('saleel') - LENGTH(REPLACE('saleel', 'e', ""));
- SELECT empno, datePresent, LENGTH(datePresent) - LENGTH(REPLACE(datePresent, ",", "")) + 1 *"Days Present"* FROM emp_attendance;
- SELECT c1, c1 / 1, SUBSTR(c1, LENGTH(c1 / 1) + 1) FROM numberString;
- SELECT c1, REVERSE(c1) / 1, LENGTH(REVERSE(c1) / 1), REVERSE(SUBSTR(REVERSE(c1), LENGTH(REVERSE(c1) / 1) + 1)) FROM Stringnumber;
- UPDATE emp SET job := REPLACE(job, 'officers', 'Officers');

- Count total ***y*** votes.

string functions - examples

Table:- vote_response

CandidateID	response
1	n,n,n,y,y,y,y,n,y,n,y,y,n,n,y,n,n,n,n,n,n,n,n,y,y,y,y,y,y,y,n,y,n,y,y,y,n,n,n,y,y,y,y,n,n,n,n,n,y
2	y,n,n,y,y,y,y,n,y,n,y,y,n,n,y,n,n,y,n,y,n,y,n,y,y,y,y,y,y,y,n,y,n,y,y,y,n,n,n,y,y,y,y,y,y,y,n,y,n,y,n,y,n,n,n,y,y,y,n
3	y,y,y,y,y,y,y,n,y,y,y,y,n,y,y,y,y,y,n,y,y,y,y,y,y,y,y,y,y,y,y,y,y,n,y,y,y,y,y,y,y,n,y,y,y,y,y,n,y,y,y,y,y,y,y,n,y,n
4	n,n,n,y,y,n,n,n,y,n,y,y,n,n,y,n,y,n,n,y,n,y,n,y,n,y,n,y,n,n,n,y,y,n,n

Output

CandidateID	Yes Votes	Total Votes
1	yyyyyyyyyyyyyyyyyyyyyyy	24
2	yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy	36
3	yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy	40
4	yyyyyyyyyyyyyyyyyy	19

Output →

- SELECT candidateID, REPLACE(REPLACE(response, ',', ''), 'n', '') *'Yes Votes'*, LENGTH(REPLACE(REPLACE(response, ',', '') ,'n', '')) *'Total Votes'* FROM vote_response;

string functions - examples

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

-
```
SELECT   SUBSTRING_INDEX('State Bank of India', ' ', 1) AS part1,                    -- 'State'
SUBSTRING_INDEX(SUBSTRING_INDEX('State Bank of India', ' ', 2), ' ', -1) AS part2,   -- 'Bank'
SUBSTRING_INDEX(SUBSTRING_INDEX('State Bank of India', ' ', 3), ' ', -1) AS part3,   -- 'of'
SUBSTRING_INDEX('State Bank of India', ' ', -1) AS part4;                            -- 'India'
```

- Count elements in **enum** data-type

string functions - examples

Table:- teams

TeamID	Members
1	Saleel, Sharmin, Vrushali, Gau
2	Kaushal, Natsha, Ruhan
3	Lalu, Brooke, Boy, Naru, Bhavin
4	Bhuru, Baba, Anoop, Meera, Boy
5	Dilu, Fredy
6	Daigo, Harshil, Boy, Raju, Bothyo
7	Bandish, Supriya, Sangita
8	Monika, Karishma, William, Varsha, Jack, Grace
9	Vasant, Neel
10	Nitish, Bipin, Mai, Didi

Output →

Output

TeamID	Members	**Total Members**
1	Saleel, Sharmin, Vrushali, Gau	4
2	Kaushal, Natsha, Ruhan	3
3	Lalu, Brooke, Boy, Naru, Bhavin	5
4	Bhuru, Baba, Anoop, Meera, Boy	5
5	Dilu, Fredy	2
6	Daigo, Harshil, Boy, Raju, Bothyo	5
7	Bandish, Supriya, Sangita	3
8	Monika, Karishma, William, Varsha, Jack, Grace	6
9	Vasant, Neel	2
10	Nitish, Bipin, Mai, Didi	4

- `SELECT TeamID, members, LENGTH(members) - LENGTH(REPLACE(members,",", "")) + 1 `Total Members` FROM teams;`

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.

mathematical functions

e.g.

- SELECT CEIL(1.23);
- SELECT CEIL(-1.23);
- SELECT FLOOR(1.23);
- SELECT FLOOR(-1.23);
- SELECT ROUND(-1.23);
- SELECT ROUND(-1.58);
- SELECT FLOOR(1 + RAND() * 100); // between 1 to 100
- SELECT FLOOR(20 + RAND() * 56); // between 20 to 75
- SELECT FLOOR(899999 + RAND() * 100000) *OTP*; // 6 digit
- SELECT weight, TRUNCATE(weight, 0) AS *kg*, MID(weight, INSTR(weight, ".") + 1) AS *gms* FROM mass_table;
- SELECT weight, TRUNCATE(weight, 0) AS *kg*, RIGHT(MOD(weight , 1), 2) AS *gms* FROM mass_table;

random characters / number

FLOOR(*start_value* + RAND() * difference between (*start_value* – *end_value*) + 1);

- CAST(CHAR(FLOOR(65 + RAND() * 26)) AS CHAR) → Random uppercase letter (A–Z)
- CAST(CHAR(FLOOR(97 + RAND() * 26)) AS CHAR) → Random lowercase letter (a–z)
- FLOOR(0 + RAND() * 10)) → Random digit (0–9)

random numbers between

- 1 to 100 → FLOOR(1 + RAND() * 100);
- 1 to 14 → FLOOR(1 + RAND() * 14);
- 20 to 75 → FLOOR(20 + RAND() * 56);
- 6 digit OPT number → FLOOR(899999 + RAND() * 100000);

Table:- cNumbers

ID	num
1	1
2	1
3	1
4	2
5	1
6	2
7	2
8	7
9	7
10	7
11	5
12	5
13	2
14	4
15	4
16	4

Output →

OUTPUT

ID	R1	R2
1	1	1
2	2	1
3	3	1
4	4	1
5	5	1
6	1	2
7	2	2
8	3	2
9	4	2
10	5	2
11	1	3
12	2	3
13	3	3
14	4	3
15	5	3
16	1	4

TODO

TODO

- SELECT id, CASE WHEN id%5 = 0 THEN 5 ELSE id%5 END *R1*, CEIL(id/5) *R2* FROM cnumbers;

random characters / number

FLOOR(*start_value* + RAND() * difference between (*start_value* – *end_value*) + 1);

- CAST(CHAR(FLOOR(65 + RAND() * 26)) AS CHAR) → Random uppercase letter (A–Z)
- CAST(CHAR(FLOOR(97 + RAND() * 26)) AS CHAR) → Random lowercase letter (a–z)
- FLOOR(0 + RAND() * 10)) → Random digit (0–9)

- WITH *RECURSIVE* ***cte*** AS (SELECT 1 *n* UNION SELECT *n* + 1 FROM ***cte*** WHERE *n* <= 10) SELECT CAST(CHAR(FLOOR(65 + RAND() * 26)) AS CHAR) *R1* FROM ***cte***; // letter (A–Z)
- WITH *RECURSIVE* ***cte*** AS (SELECT 1 *n* UNION SELECT *n* + 1 FROM ***cte*** WHERE *n* <= 10) SELECT CAST(CHAR(FLOOR(97 + RAND() * 26)) AS CHAR) *R1* FROM ***cte***; // letter (a–z)
- WITH *RECURSIVE* ***cte*** AS (SELECT 1 *n* UNION SELECT n + 1 FROM ***cte*** WHERE *n* <= 10) SELECT FLOOR(0 + RAND() * 10) *R1* FROM ***cte***; // digit (0–9)

- SELECT ELT(FLOOR(1 + RAND() * 14), 'Pending', 'Shipped', 'Delivered', 'Cancelled', 'Processing', 'In Transit', 'Out for Delivery', 'Failed Attempt', 'Returned to Origin', 'Returned', 'Lost', 'Damaged', 'On Hold', 'Ready for Pickup') *Status*;

Remember:

- Here, "*" is known as metacharacter (all columns)
- You can print data in JSON format using built-in JSON functions like JSON_OBJECT(), JSON_ARRAYAGG(), and JSON_ARRAY().

e.g

- SELECT JSON_OBJECT('empno', empno, 'ename', ename, 'job', job, 'sal', sal) AS emp_json FROM emp;
- SELECT JSON_ARRAYAGG(JSON_OBJECT('empno', empno, 'ename', ename, 'job', job, 'sal', sal)) AS emp_data FROM emp;

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

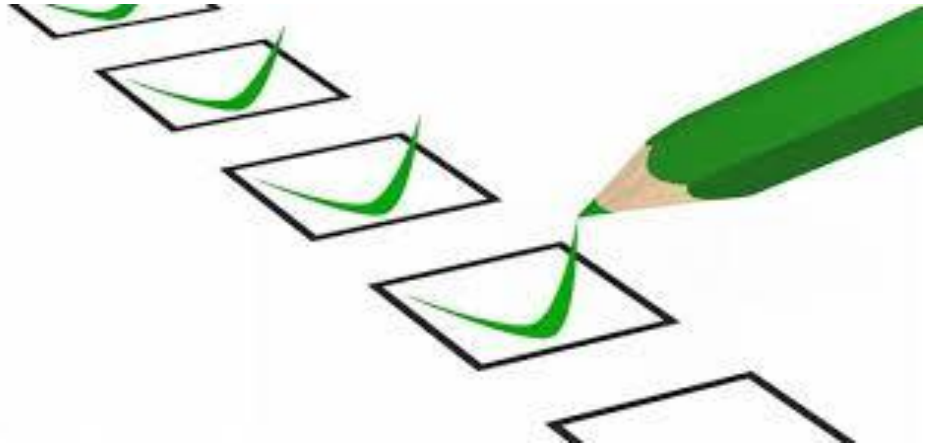
SELECT [ALL / DISTINCT / DISTINCTROW] identifier.* / identifier.A_1 [[as] *alias_name*], identifier.A_2 [[as] *alias_name*], identifier.A_3 [[as] *alias_name*], expression1 [[as] *alias_name*], expression2 [[as] *alias_name*] . . .

- [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* } >]
- [FOR { UPDATE }]
- [{ INTO OUTFILE '*file_name*' | INTO DUMPFILE '*file_name*' | INTO *var_name* [, *var_name*], . . . }]

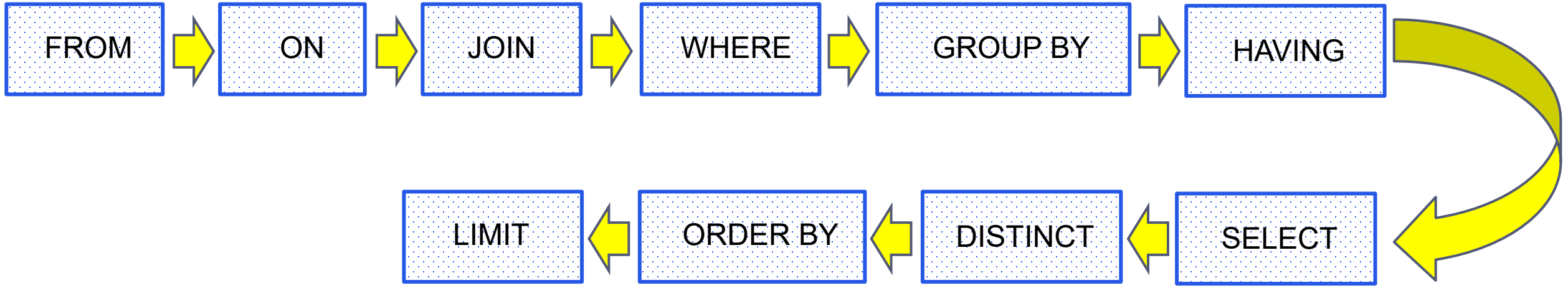
Remember:

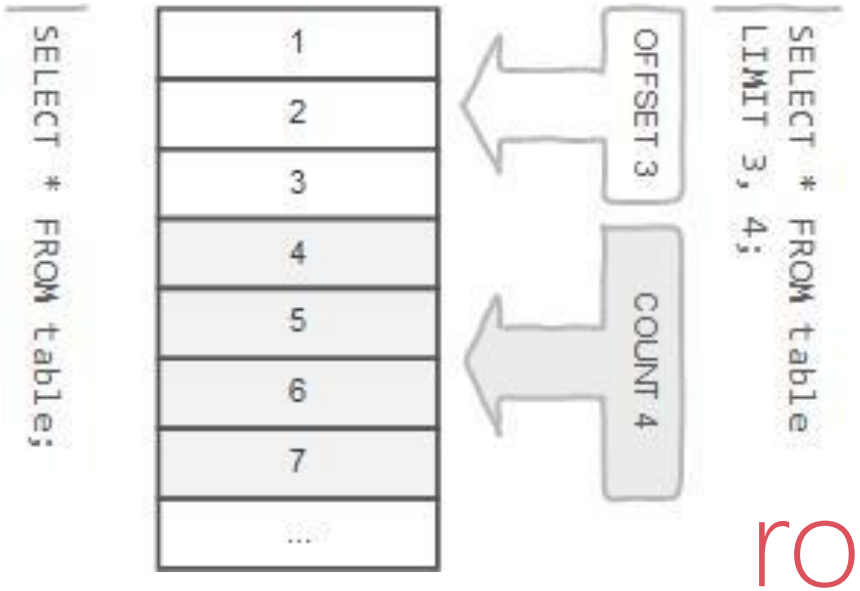
- **ALL** (**modifier is default**) specifies that all matching rows should be returned, including duplicates.
- **DISTINCT** (**modifier**) specifies removal of duplicate rows from the result set.
- **DISTINCTROW** (**modifier**) is a synonym for **DISTINCT**.
- It is an error to specify both modifiers.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

sequence of clauses



select statement... execution





row limiting clause

LIMIT is applied after HAVING

Remember:

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

Note:

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

---

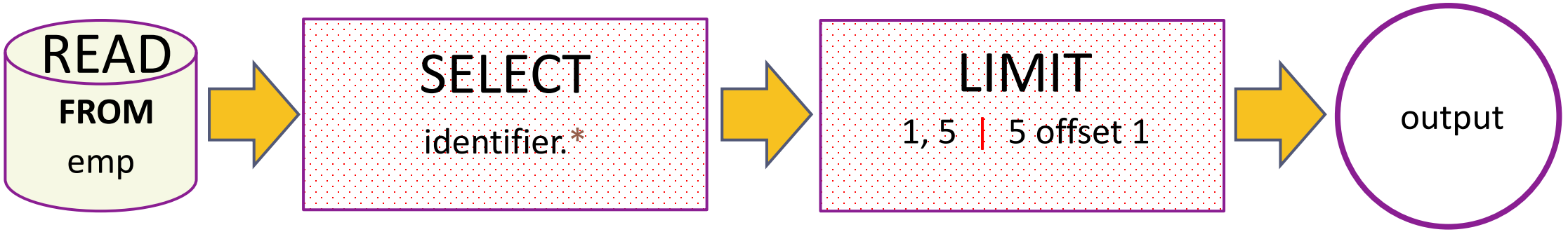
select - limit

SELECT $A_1, A_2, A_3, \ldots$ FROM *r*

[LIMIT { [*offset*,] *row_count* | *row_count* OFFSET *offset* }]

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

- SELECT * FROM emp LIMIT 5 OFFSET 1;



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

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

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

order by clause

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

Remember:

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

select - order by

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

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

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

"Ordered by attributes A_1, A_2, A_3 …"

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

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

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

select - order by

[ORDER BY { $A_1, A_2, A_3, \ldots$ | *expr* | *position* | CASE WHEN *condition* END]) [*over_clause*] } [ASC | DESC] , . . .]

- SELECT * FROM emp ORDER BY comm;
- SELECT * FROM emp ORDER BY comm IS NULL ;
- SELECT * FROM emp ORDER BY comm IS NOT NULL ;
- SELECT * FROM emp ORDER BY 1 + 1;
- SELECT * FROM emp ORDER BY True;
- SELECT sal FROM emp ORDER BY -sal;
- SELECT ename, LENGTH(ename) FROM emp ORDER BY LENGTH(ename), ename DESC ;
- SELECT * FROM emp ORDER BY IF(job = 'manager', 3, IF(job = 'salesman', 2, NULL)) ;
- SELECT onum, paymentType, amt FROM orders ORDER BY FIELD(paymentType, 'Credit Cards', 'Debit Cards', 'Wallets', 'Paytm', 'PhonePe', 'GPay', 'EMI Options', 'COD');
- SELECT * FROM emp ORDER BY ISNULL(comm), comm ;
- SELECT ename `e` FROM emp ORDER BY `e` ;
- SELECT ename `e` FROM emp ORDER BY e ;
- SELECT ename 'e' FROM emp ORDER BY 'e' ;
- SELECT * FROM emp ORDER BY CASE WHEN ename='sharmin' THEN 0 ELSE 1 END, *ename*;
- SELECT ename, job, sal, sal > 2500 FROM emp ORDER BY sal > 2500, sal;
- SELECT * FROM emp ORDER BY NULL;
- SELECT * FROM emp ORDER BY (SELECT NULL);

- swap every pair of rows (1↔2, 3↔4, 5↔6, etc.)

select - order by

swap every pair of row

Table:- swapCity

ID	ename	cityName
1	Saleel	Baroda
2	Sharmin	Rajkot
3	Vrushali	Surat
4	Ruhan	Bharuch
5	Nitish	Jamnagar
6	Deep	Anand
7	Neel	Bhuj

Output →

Output

ID	ename	**cityName**
2	Sharmin	Rajkot
1	Saleel	Baroda
4	Ruhan	Bharuch
3	Vrushali	Surat
6	Deep	Anand
5	Nitish	Jamnagar
7	Neel	Bhuj

- SELECT id, CASE WHEN id % 2 = 1 THEN id + 1 ELSE id END, ename, cityName FROM swapcity;
- SELECT id, ename, cityName FROM swapcity ORDER BY CASE WHEN id % 2 = 1 THEN id + 1 ELSE id - 1 END;

Remember:

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

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

```
SELECT patientID, patientName FROM patient WHERE (SELECT COUNT(*) FROM doctorvisits
WHERE patient.patientID = doctorVisits.patientid GROUP BY doctorvisits.patientID) > 8;
```

* 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*

with OR

- If <*condition1*> is **TRUE**, then MySQL **does not check** <*condition2*>.
- If <*condition1*> is **FALSE**, then MySQL **must check** <*condition2*>.

with AND

- If <*condition1*> is **FALSE**, then MySQL **does not check** <*condition2*>.
- If <*condition1*> is **TRUE**, then MySQL **must check** <*condition2*>.

Note:

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

select - where

SELECT $A_1, A_2, A_3, \ldots$ FROM $r_1, r_2, r_3, \ldots$ [WHERE *P*] CASE WHEN *condition* END

2. *comparison_operator:*

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

5. *logical_operators*

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

What will be the result of the query below?

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

Note:

AND has higher precedence than **OR.**

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

select - where

SELECT $A_1, A_2, A_3, \ldots$ FROM $r_1, r_2, r_3, \ldots$ [WHERE *P*]
CASE WHEN *condition* END

WHERE state = 'NY' OR 'CA' --Illegal
WHERE salary > 20000 AND < 30000 –Illegal
WHERE state NOT = 'CA' --Illegal

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

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

select - where

SELECT $A_1, A_2, A_3, \ldots$ FROM $r_1, r_2, r_3, \ldots$ [WHERE *P*] CASE WHEN *condition* END

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

- ```
  SELECT GREATEST(10, 20, 30),      # 30
         LEAST(10, 20, 30);         # 10
  ```

- ```
  SELECT GREATEST(10, null, 30),    # null
         LEAST(10, NULL , 30);      # null
  ```

- ```
  SELECT * FROM emp WHERE (deptno, pwd) = (SELECT deptno, pwd FROM dept WHERE deptno = 30);
  ```

Remember:

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

select - where

SELECT $A_1, A_2, A_3, \ldots$ FROM $r_1, r_2, r_3, \ldots$ [WHERE *P*] CASE WHEN *condition* END

What will be the output of the following statement?

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

- SELECT * FROM orders WHERE amt > CASE WHEN registered = 1 THEN 2000 ELSE 5000 END;
- SELECT * FROM orders WHERE CASE WHEN paymentType = 'Wallets' THEN amt > 1500 WHEN paymentType = 'COD' THEN amt > 3000 END;
- SELECT * FROM orders WHERE CASE WHEN paymentType IN ('GPay', 'Paytm', 'PhonePe') THEN amt > 1000 WHEN paymentType IN ('Credit Cards', 'Debit Cards', 'Wallets') THEN amt > 2000 END ORDER BY paymentType;

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

- *operand* IS [NOT] *NULL*

3. *boolean_ predicate:*

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

is null / is not null

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

Note:

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

select – boolean

- BOOL and BOOLEAN are synonym of TINYINT(1)

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

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

- SELECT * FROM tasks WHERE completed;
- SELECT * FROM tasks WHERE completed IS True;
- SELECT * FROM tasks WHERE completed = 1;
- SELECT * FROM tasks WHERE completed = True;

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

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

- SELECT * FROM tasks WHERE NOT completed;
- SELECT * FROM tasks WHERE completed IS False;
- SELECT * FROM tasks WHERE completed = 0;
- SELECT * FROM tasks WHERE completed = False;

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

select – boolean

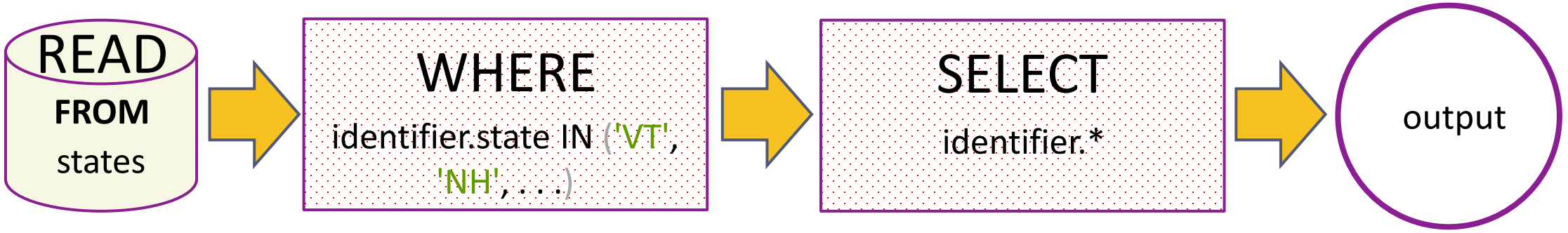
- BOOL and BOOLEAN are synonym of TINYINT(1)

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

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

What will be the result of the query below?

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



4. ***predicate***:

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

in

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

```
SELECT . . . FROM r1 WHERE (
    state = 'VT'  OR
    state = 'NH' OR
    state = 'ME' OR
    state = 'MA' OR
    state = 'CT'   OR
    state = 'RI'
);
```

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

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

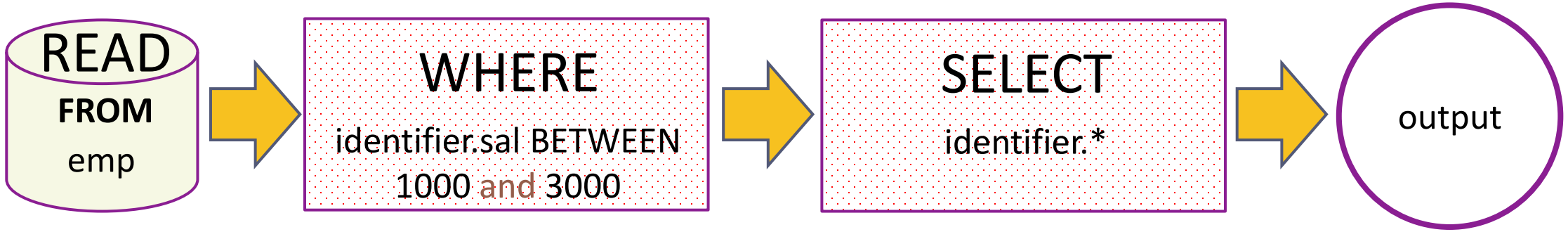
Remember:

in

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

What will be the result of the query below?

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



4. ***predicate***:

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

between

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

A BETWEEN ***B*** AND ***C***	A is between B and C

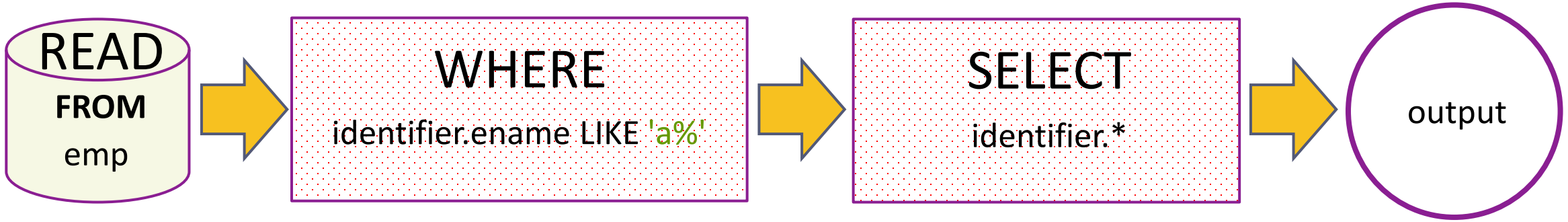
Table:- betweenSalary

id	sal
1	2000

Output

R1	R2
1	1

- SELECT COUNT(CASE WHEN sal BETWEEN 1000 AND 2000 THEN 1 END) *R1*, COUNT(CASE WHEN sal BETWEEN 2000 AND 3000 THEN 1 END) *R2* FROM betweenSalary;



4. ***predicate***:

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

REGEXP_LIKE(*expr*, *pattern*[, match_type])

like

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

like - string comparison functions

syntax

column | expression LIKE 'pattern' [ESCAPE *escape_character*]
REGEXP_LIKE(*expr*, *pattern*[, match_type])

Remember:

- **%** matches any number of characters, even zero characters. ("%" represents any sequence of characters.)
- **_** matches exactly one character. ("_" represents a single character.)
- If we use **default escape character '\'**, then don't use ESCAPE keyword.

REGEXP_LIKE

- ^ [start]
- $ [end]

- SELECT ename FROM emp WHERE regexp_like(ename, "^['sa']");
- SELECT ename FROM emp WHERE regexp_like(ename, "['sa']$");

Note:

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

What will be the result of the query below?

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