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).

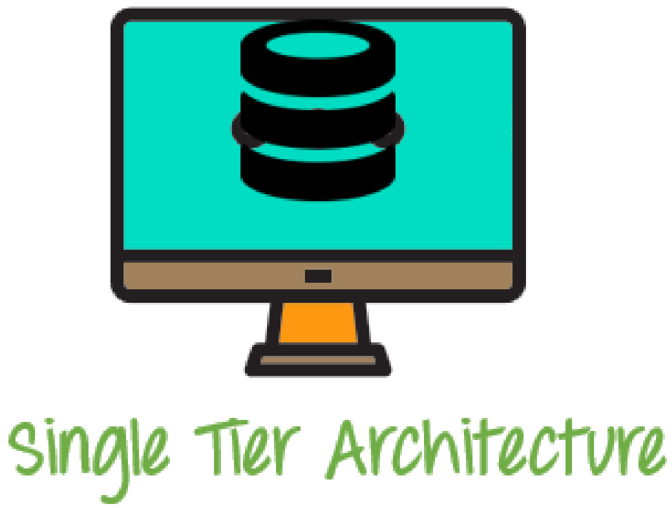
If A and a, B and b, .... are treated in the same way then it is case-insensitive.
**MySQL is case-insensitive**

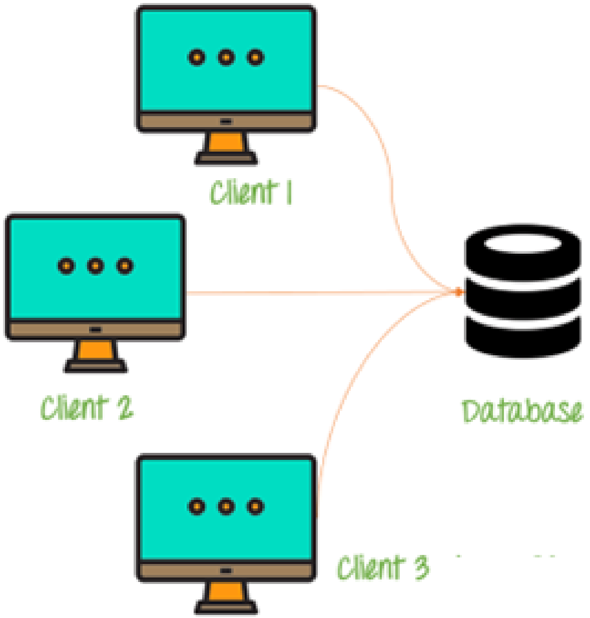
# Introduction

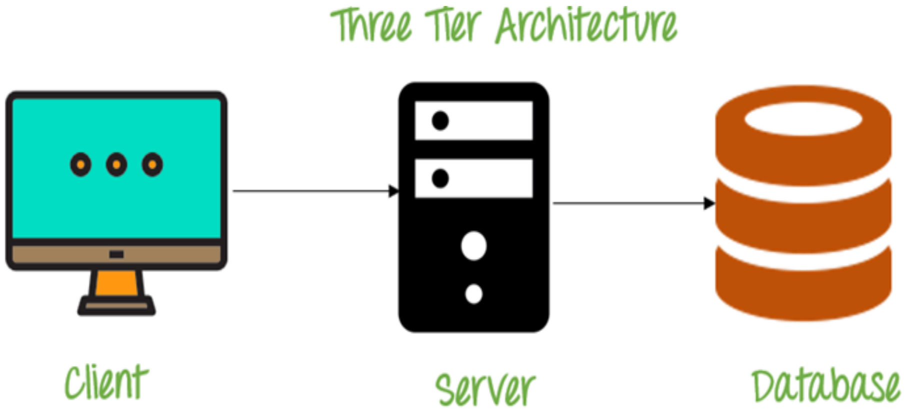
- If anyone who wants to develop a good application
  then he should have the knowledge three major components.

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 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 application (client) 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.)

**A foreign key constraint is also known as a referential constraint or referential integrity constraint.** A foreign key is a column or group of columns in a relational database table that establishes and enforces a link between data in two tables. It references a primary key in another table and can cascade changes or delete related data if the primary key is updated or deleted.

# What is Relation and Relationship?

***Reference / Referential key***

## Remember:

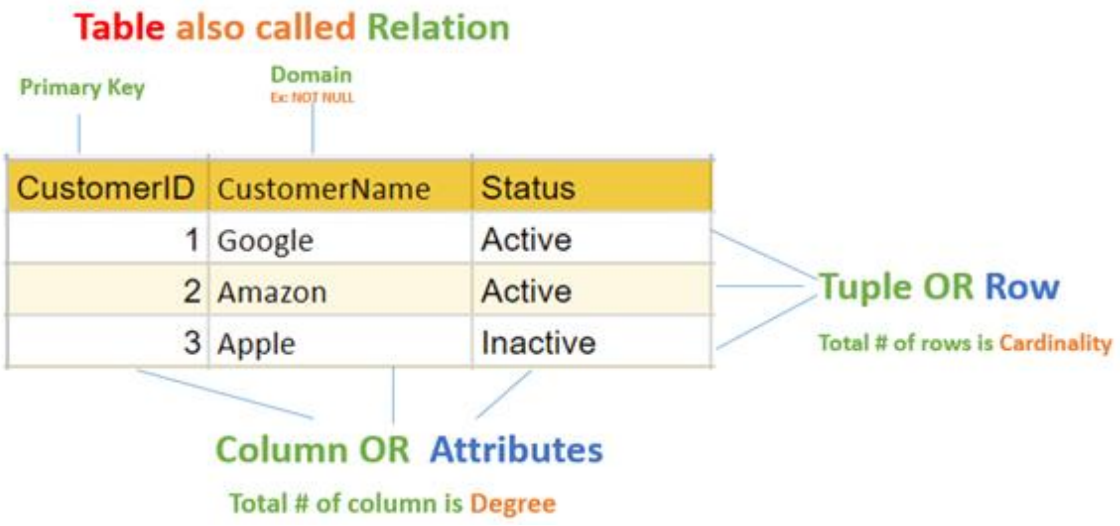
- A ***reference*** is a relationship between two tables where the values in one table refer to the values in another table. This is usually enforced using a foreign key constraint to maintain referential integrity.
- A ***referential key*** is a column or set of columns in a table that refers to the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

# relation and relationship?

**Relation** ***(in Relational Algebra "R" stands for relation)*****:** In Database, a relation represents a **table** or an **entity** than contain attributes. 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.




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

## Remember:

Foreign Key is also known as

- referential constraint
- referential integrity constraint. (Ensures that a foreign key value in one table must always reference an existing primary key value in another table.)

## Note:

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

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

# File System

# vs

# DBMS

# *file-oriented system*

*File Anomalies*

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

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

c:\employee.txt

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

c:\employee.txt

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

c:\employee.txt

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

c:\employee.txt

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

c:\employee.txt

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

# *file-oriented system*

*File Anomalies*

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

c:\employee.txt

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

file attributes

- File Name
- Type
- Location

file permissions

- File permissions
- Share permissions

search empl ID=1

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

search emp_name

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

# 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.

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

# disadvantage of file-oriented system

- **Data redundancy**: It is possible that the same information may be duplicated in different files. This leads to data redundancy results in memory wastage.
  (Suppose a customer having both kind of accounts - **saving** and **current** account. In such a situation a 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.**

- **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

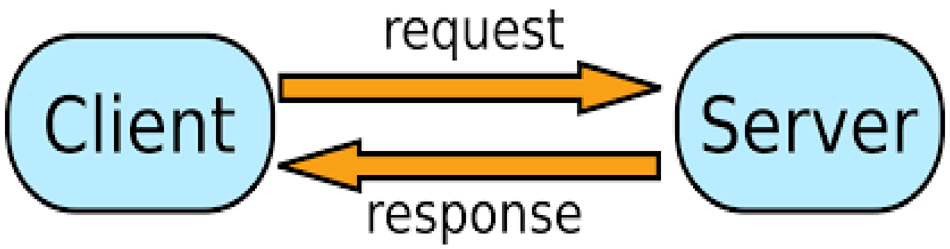
- **database:** Is the collection of **related data** which is **organized,** database can store and retrieve large amount of data easily, which is stored in one or more data files by one or more users, it is called as **structured data.**
- **management system**: it is a software, designed to **define**,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

- CREATE or REPLACE TYPE ***city*** AS VARRAY(3) OF VARCHAR(10);
- CREATE TABLE x (id INT, ename VARCHAR(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 | •••••• |
| 2 | salesman | Ruhan | yyyy-mm-dd | •••••• |
| 5 | 9500 | manager | yyyy-mm-dd | •••••• |
| 5 | Salesman | Rahul Patil | yyyy-mm-dd | •••••• |

## Relational tables have six properties:

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

# What is data?

# *what is data?*

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

Data can be in the form of Text or Multimedia

e.g.

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

Remember:

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

# What is Entity Relationship Diagram?

# Entity Relationship Diagram (ER Diagram)

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

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

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

# What is Entity?

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

# 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**.

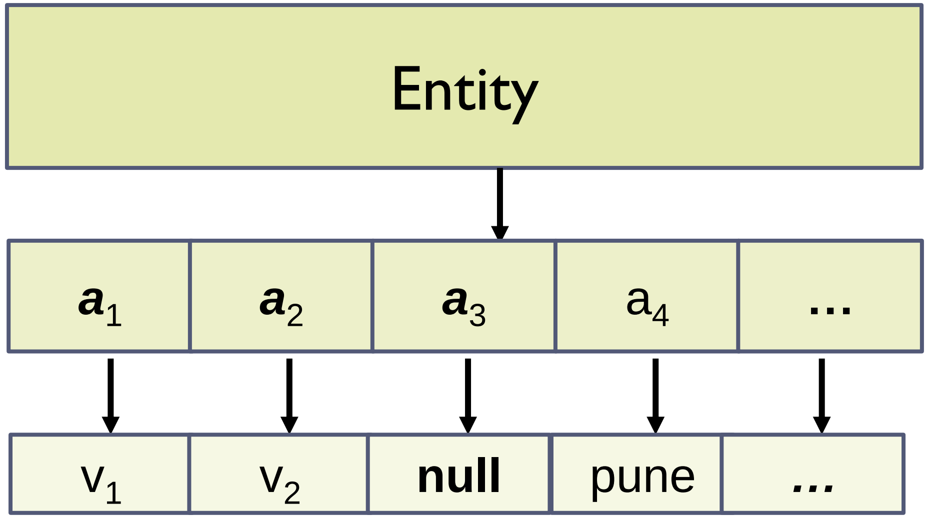
***Every entity has its own characteristics.***

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 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.

# Composite **VS** Multi Valued Attribute

## composite / multi valued attributes

## Composite Attribute

### Person Entity

- *Name* attribute: (`firstName` + `middleName` + `lastName` )
- *PhoneNumber* attribute: (`countryCode` + `cityCode` + `phoneNumber` )
- *Date* attribute: (`Day` + `Month` + `Year`)
- *Dimensions* attribute: (`Length` + `Width` + `Height`)

```
{Address}
{street, city, state, postal-code}
{street-number, street-name, apartment-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, . . . ]

# What is an Prime, Non-Prime Attribute?

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

***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).

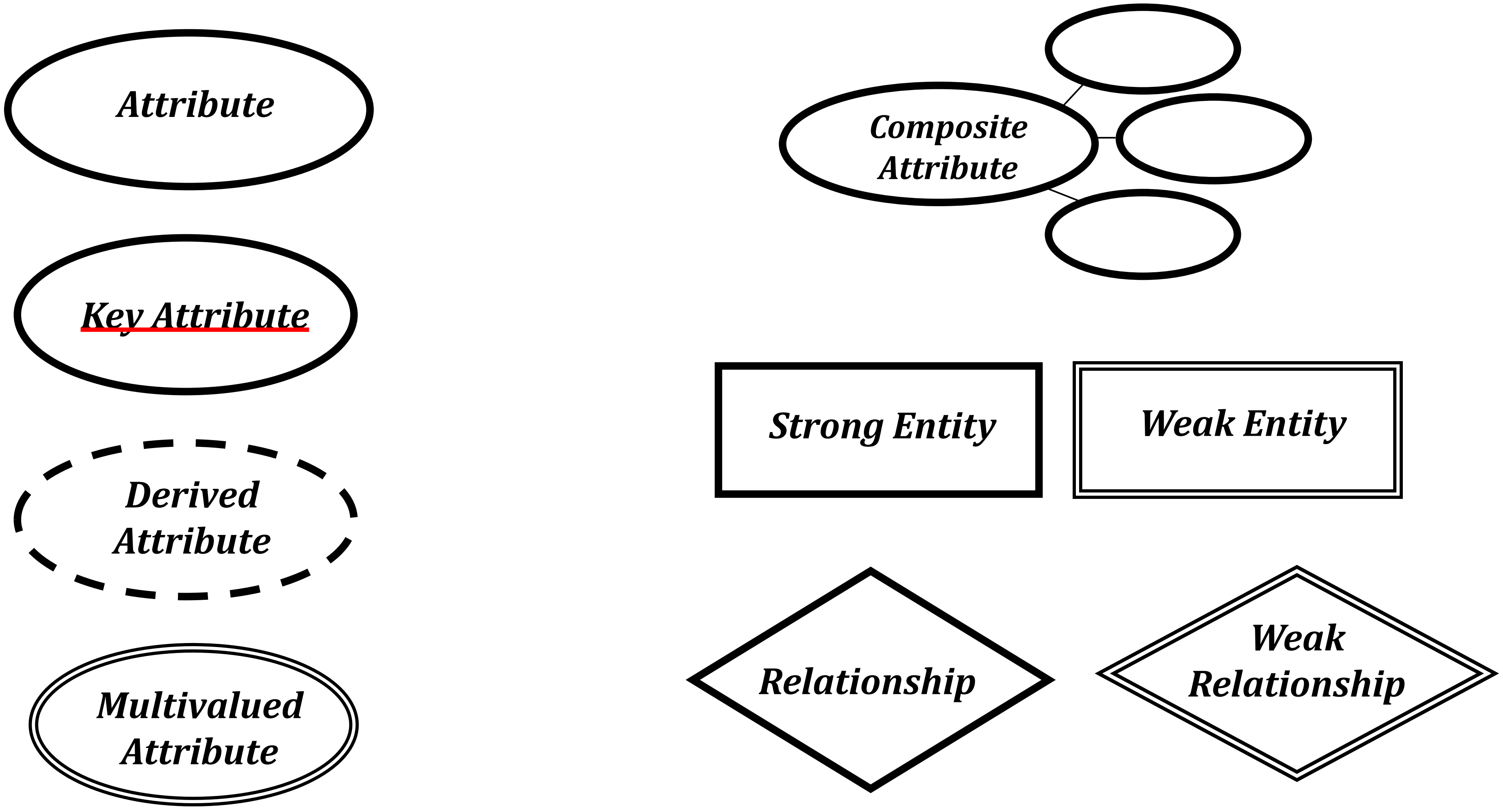
**Non-prime attribute:-** An attribute, which is **not a part of the prime-key** **(**candidate key**)**, is said to be a non-prime attribute.

***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

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

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

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

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

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

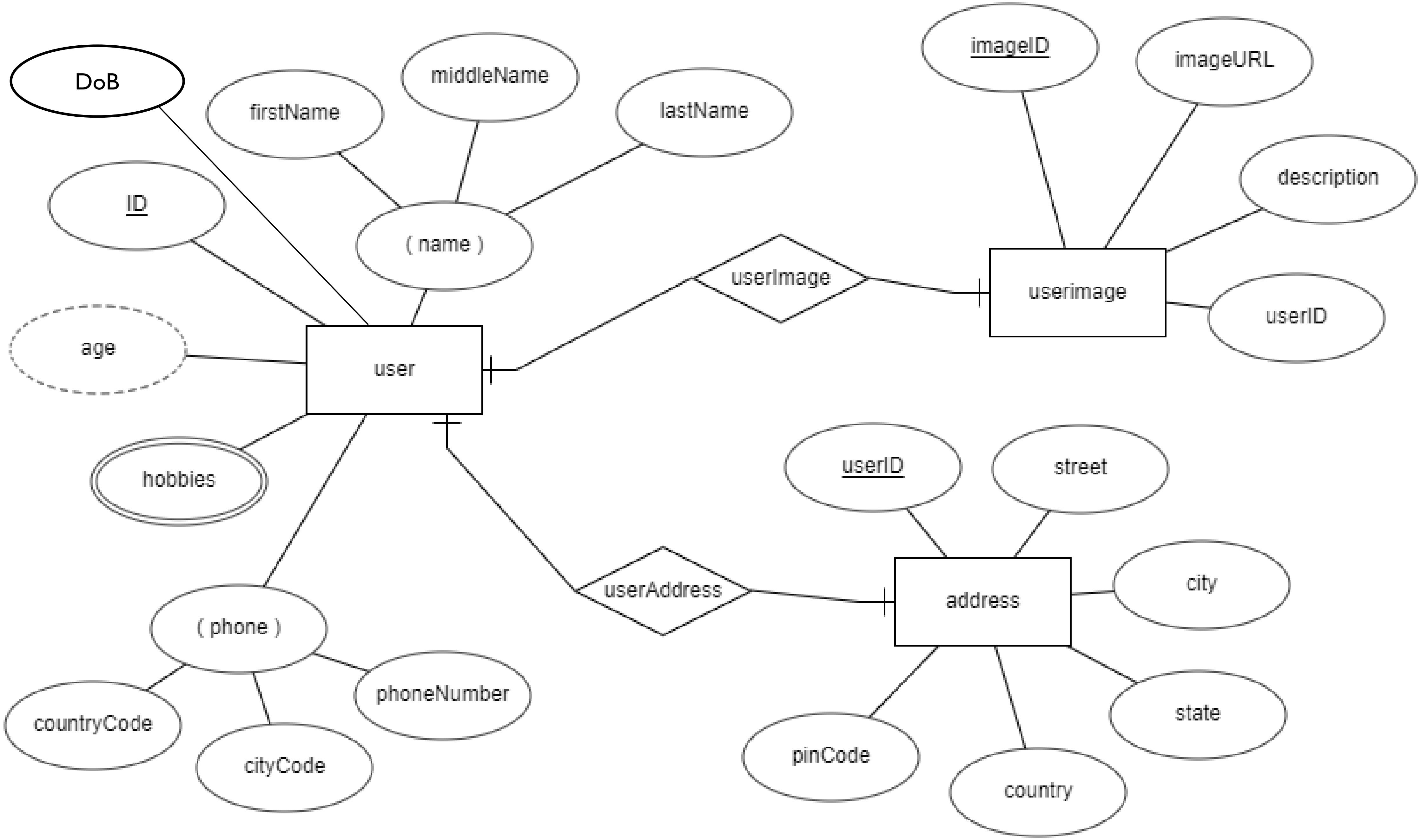
**Example 3 –** A customer address entity can not be created for the customer if the customer doesn’t exist

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

# *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) |
| | |
| | |
| | |
| | |
| | |

# 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 $\mathbf{d}(R)$.

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

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

- **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.

# 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 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 |
| 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') |

# 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

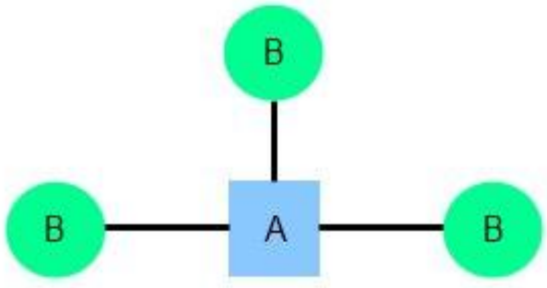
# Common relationships

# *relationships*

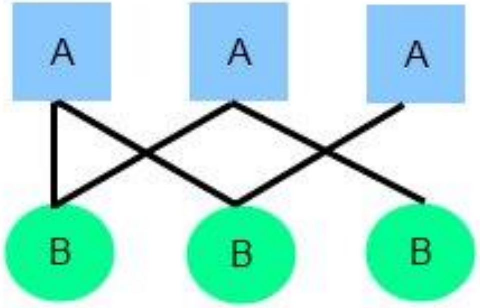
Common relationship

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

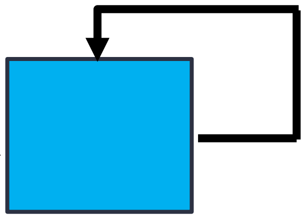

2. one-to-many (1:M)




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




4. Self-Referencing (Recursive)

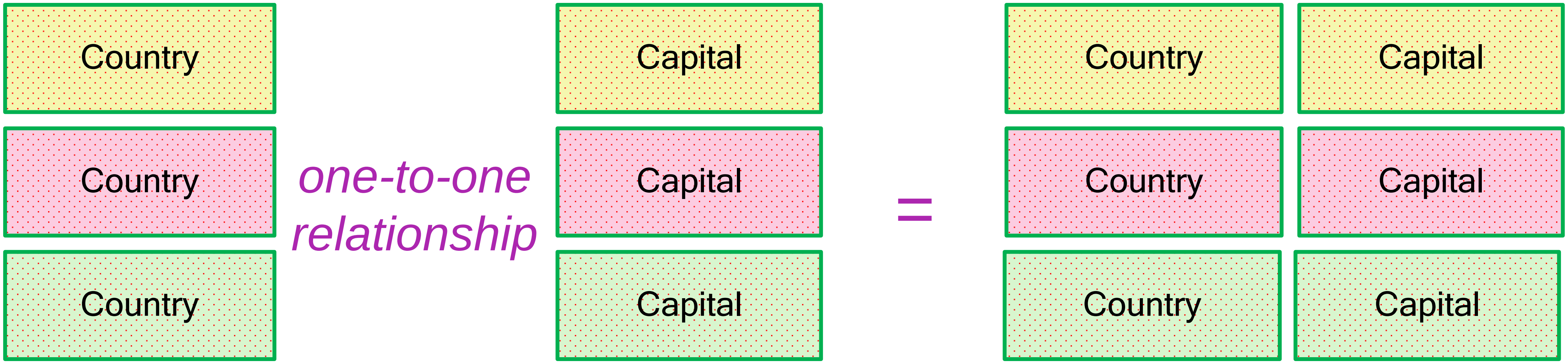


one-to-one relationship

# one-to-one relationship

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

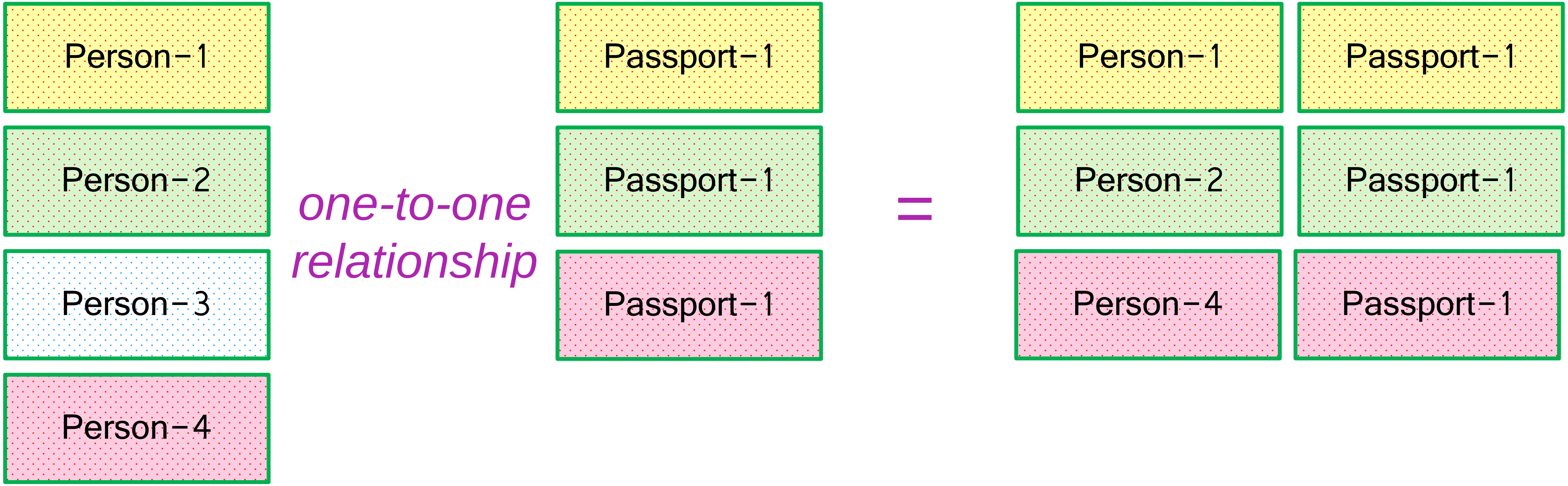
A *one-to-one* relationship is a type of cardinality that refers to the relationship between two entities ***R*** and ***S*** in which one element of entity ***R*** may only be linked to zero/one element of entity ***S***, and vice versa.



# *one-to-one relationship*

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

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

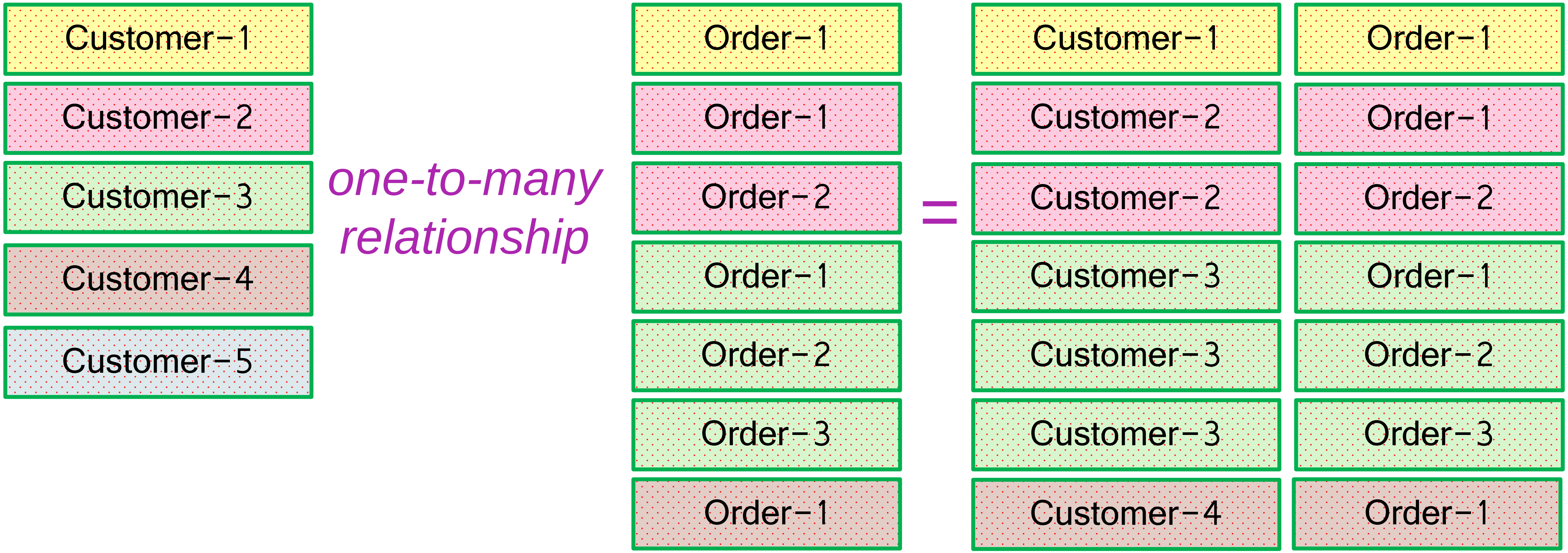


# one-to-many relationship

# one-to-many relationship

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

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



# 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***.

| |
|---|
| Invoice−1 |
| Invoice−2 |
| Invoice−3 |
| Invoice−4 |

one-to-many relationship

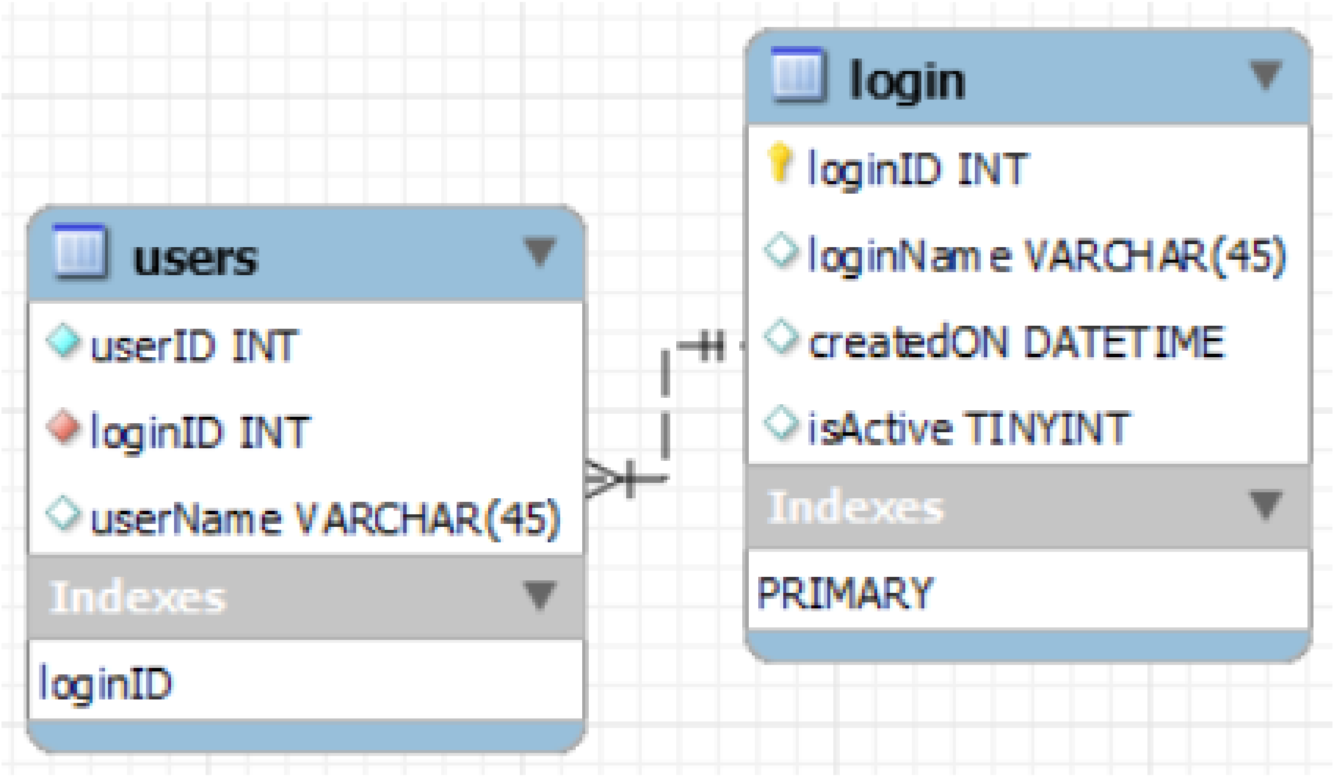
| |
|---|
| Invoice_Item−1 |
| Invoice_Item−1 |
| Invoice_Item−2 |
| Invoice_Item−1 |
| Invoice_Item−2 |
| Invoice_Item−2 |
| Invoice_Item−1 |

=

| | |
|---|---|
| Invoice−1 | Invoice_Item−1 |
| Invoice−2 | Invoice_Item−1 |
| Invoice−2 | Invoice_Item−2 |
| Invoice−3 | Invoice_Item−1 |
| Invoice−3 | Invoice_Item−2 |
| Invoice−3 | Invoice_Item−3 |
| Invoice−4 | Invoice_Item−1 |

# many-to-one relationship

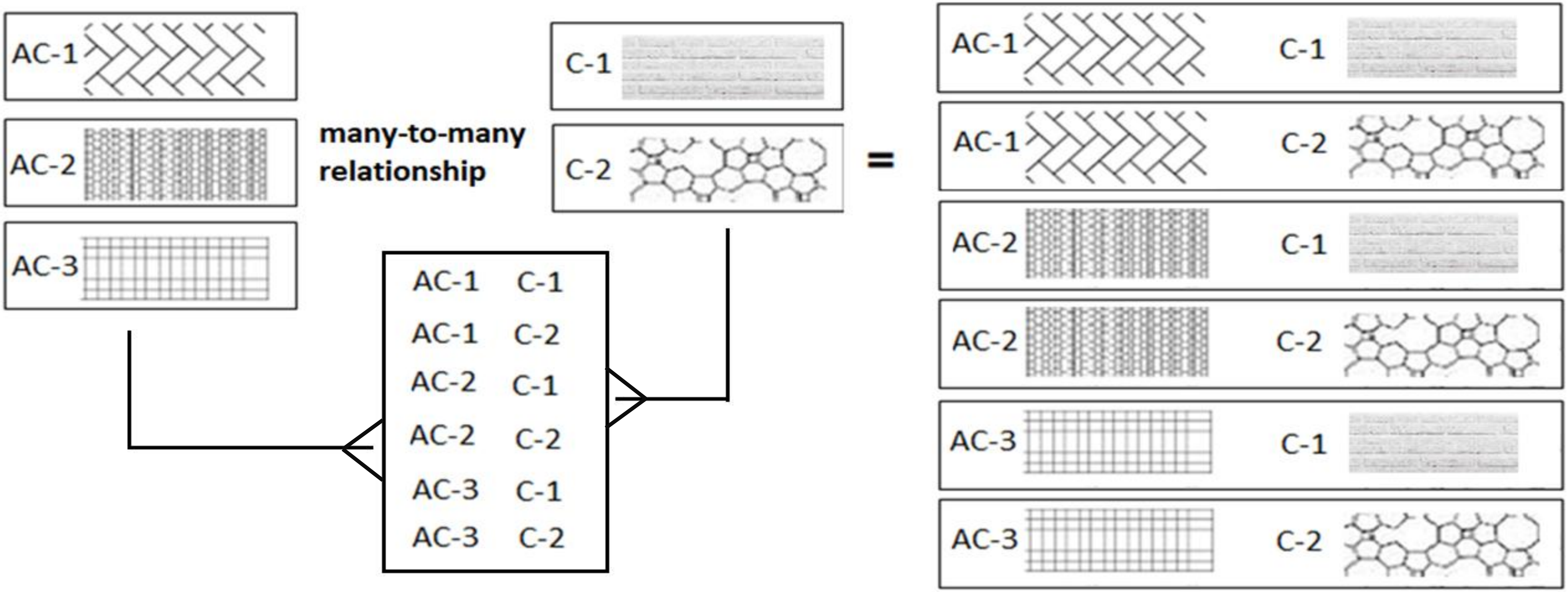
*many-to-one relationship*



# many-to-many relationship

# many-to-many relationship

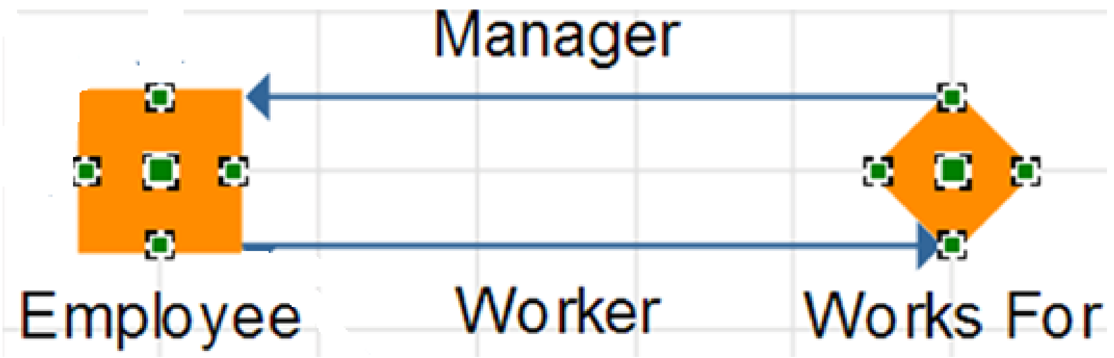
A *many-to-many* relationship is a type of cardinality that refers to the relationship between two entities ***R*** and ***S*** in which ***R*** may contain a parent instance for which there are many children in ***S*** and vice versa.



# self-referencing relationship

# *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

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

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

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

# What is SQL?

Remember:

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

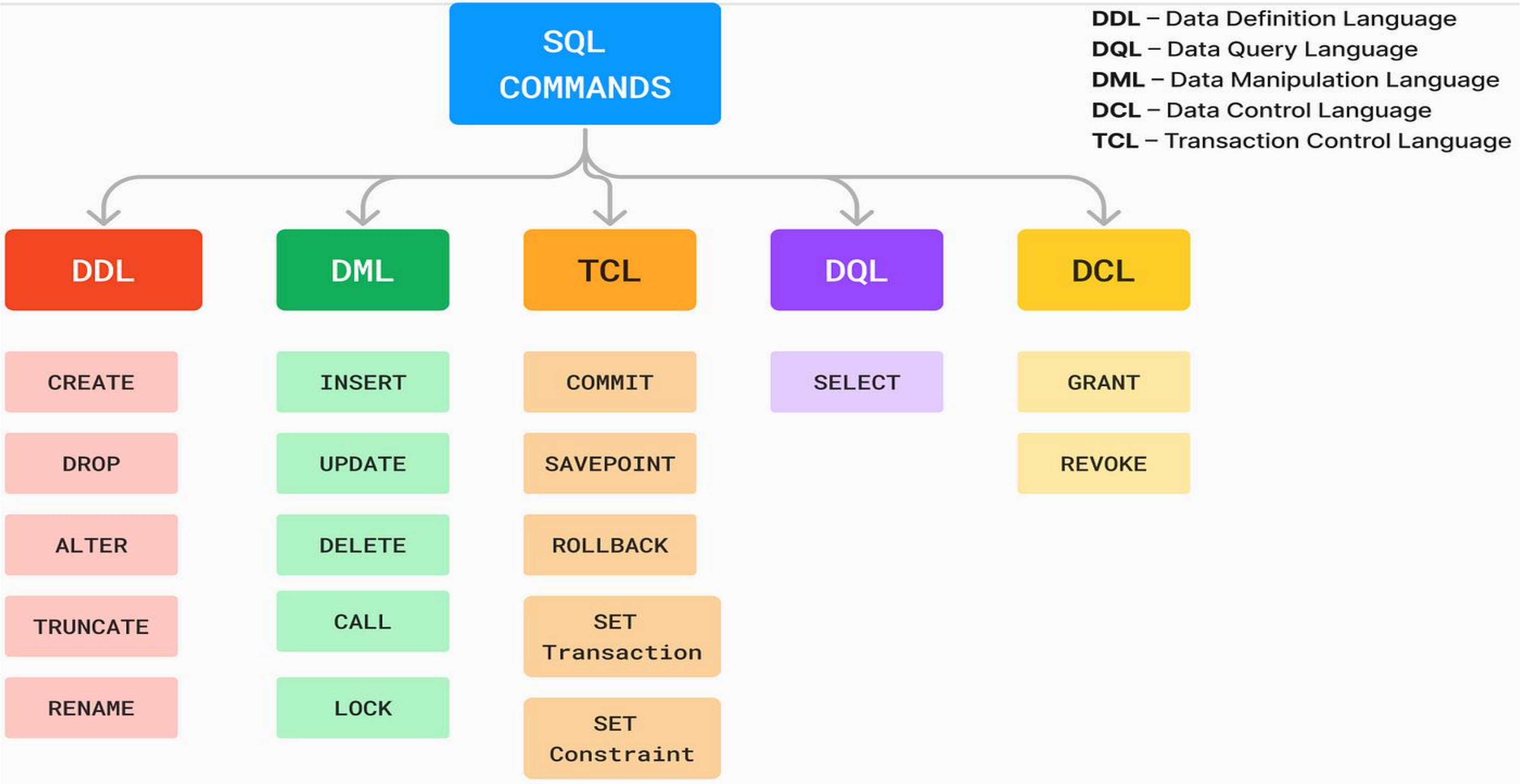
## what is sql?

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

## Remember:

*what is sql?*

- An ***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*]

Windows Command Processor

```
Microsoft Windows [Version 10.0.18363.900]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Windows\system32>mysql -h127.0.0.1 -P3306 -uroot -proot_
```

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

```
CREATE TABLE temp (c1 CHAR(10), c2 VARCHAR(10));

INSERT INTO temp VALUES('SALEEL', 'SALEEL');

SELECT * FROM temp WHERE c1 LIKE 'SALEEL';
```

# *datatypes*

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ENAME CHAR (10) | S | A | L | E | E | L | | | | | LENGTH -> 10 |
| ENAME VARCHAR2(10) | S | A | L | E | E | L | | | | | LENGTH -> 6 |

In MySQL

When CHAR values are retrieved, the trailing spaces are removed (unless the ***PAD_CHAR_TO_FULL_LENGTH*** SQL mode is enabled)

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

Note:

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

# *datatype - string*

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

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

- *SET sql_mode = '';*
- *SET sql_mode = 'PAD_CHAR_TO_FULL_LENGTH';*

# example of char and varchar

| Datatypes | Size | Description |
|---|---|---|
| CHAR [(length)] | 0-255 | |
| VARCHAR (length) | 0 to 65,535 | The maximum row size (65,535 bytes, which is shared among all columns. |

Try Out

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

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

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

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

# datatype - numeric

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

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

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

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

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

Note:

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

## NOTE:

# *datatype – enum*

- 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.

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

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

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

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

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

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

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

Note:

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

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

# datatype – set

NOTE:

- 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);
```

IMP:

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

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

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

## Remember:

- Max 4096 columns per table provided the row size <= 65,535 Bytes.
- 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.

## Remember:

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

## Note:

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

# create table

## syntax

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    (*create_defineation*, . . .)
    [table_options]
    [partition_options]

***create_definition:***
    col_name ***column_definition***

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

table_options:
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 r(
  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 r VALUES();
- INSERT INTO r VALUES(-1, DEFAULT, DEFAULT);
- INSERT INTO r VALUES(-2, DEFAULT(c2), DEFAULT(c3));
- INSERT INTO r 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** is used to add a single or multiple tuple to a relation. We must specify the relation name and a list of values for the tuple. **The values should be listed in the same order in which the corresponding attributes were specified in the CREATE TABLE command.**
- A second form of the **INSERT** statement allows the user to specify explicit attribute names that correspond to the values provided in the **INSERT** command. This is useful if a relation has many attributes but only a few of those attributes are assigned values in the new tuple. However, the values must include all attributes with **NOT NULL** specification and no default value. Attributes with **NULL** allowed or **DEFAULT** values are the ones that can be left out.

# insert rows using values

# dml- insert … values

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

INSERT [IGNORE] [INTO] tbl_name [PARTITION (*partition_name* [, *partition_name*] ...)] [ (*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

# 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 ]

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

e.g.

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

insert multiple rows

# dml- insert … values

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

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

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

e.g.

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

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

# SELECT statement...

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

# *SELECT CLAUSE*

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

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

## *Capabilities of SELECT Statement*

1. SELECTION
2. PROJECTION
3. JOINING

# Capabilities of
# SELECT Statement

## ➢SELECTION

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

***R***

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

# Capabilities of SELECT Statement

## ➢ PROJECTION

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

***R***

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

**Table DEPARTMENTS**

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

Projection
Selection

**Table EMPLOYEES**

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

# Capabilities of SELECT Statement

## ➢ JOINING

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

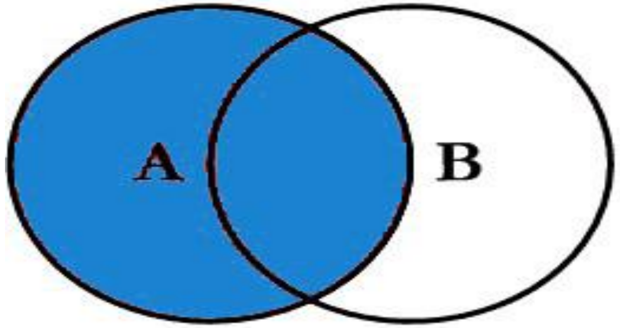
***R***

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

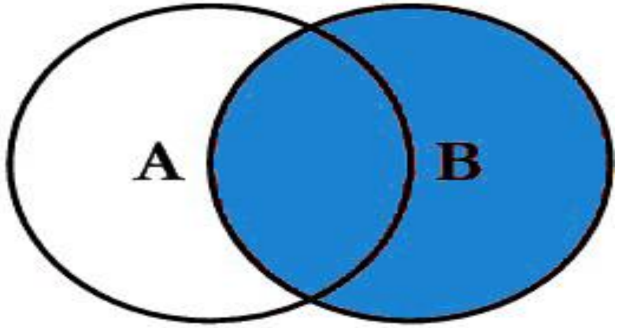
***S***

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

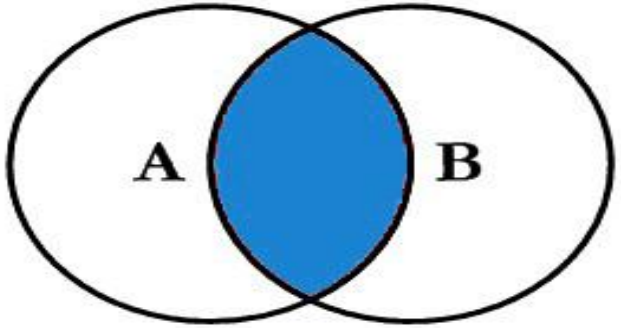
# SQL JOINS



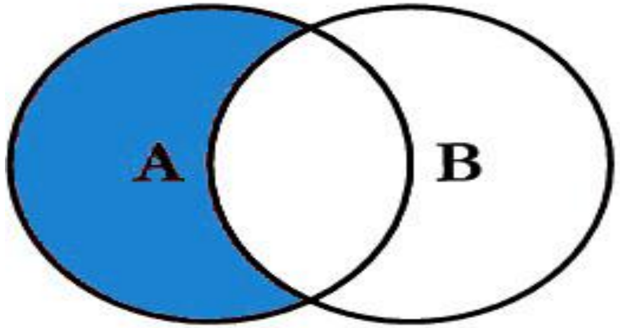

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



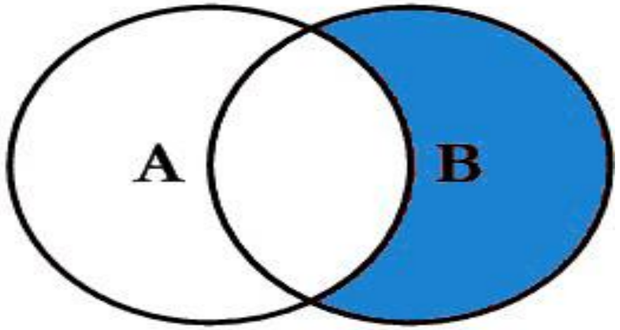

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



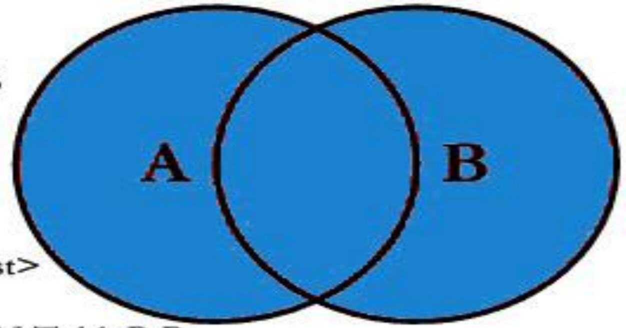

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



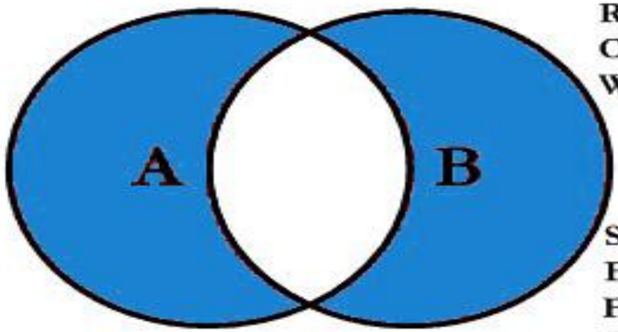

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
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)

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

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

Note:

Here c1 column is a Primary Key

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

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

# single-table update

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

- SET @x := 0;
- UPDATE emp SET id = @x := @x + 1;

- UPDATE t, (SELECT isactive, COUNT(isactive) r1 FROM emp GROUP BY isactive) a SET t.c2 = a.r1 WHERE t.c1 = a.isactive;

```
mysql> SELECT * FROM t;
+-------+--------+        +-------+------+
| c1    | c2     |        | c1    | c2   |
+-------+--------+        +-------+------+
|   0   | NULL |          |   0   |   6  |
|   1   | NULL |          |   1   |  14  |
+-------+--------+        +-------+------+
```

e.g.

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

# single-table update

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

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

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

- UPDATE duplicate SET id = ( SELECT @cnt := @cnt + 1 );

# single-table delete

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

# *delete can violate only in referential integrity.*

Important:

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

# *single-table delete*

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

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

Note:

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

# auto_increment column

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

# *auto_increment*

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

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

## Remember:

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

---

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

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

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

## Note:

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

# virtual column - generated always

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

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

## Note:

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

| 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 |

```
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
);
```

# visible / invisible columns

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

Note:

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

# *invisible column*

*col_name data_type* INVISIBLE

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

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

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

- `ALTER TABLE employee MODIFY total INT VISIBLE;`
- `ALTER TABLE employee MODIFY total INT INVISIBLE;`

# varbinary column

TODO

Note:

- TODO
- TODO
- TODO

# varbinary column

*col_name* VARBINARY

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

- INSERT INTO login(userName, password) VALUES('ram', 'ram@123');
- INSERT INTO login(userName, password) VALUES('pankaj', 'pankaj');
- INSERT INTO login(userName, password) VALUES('rajan', 'rajan');
- INSERT INTO login(userName, password) VALUES('ninad', 'ninad');
- INSERT INTO login(userName, password) VALUES('omkar', 'omkar');

- SELECT * FROM login;
- SELECT username, CAST(password as CHAR) FROM login;

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

# constraints

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

CONSTRAINTS can be classified into two types –

- *Column Level*
- *Table Level*

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

## Remember:

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

# constraints

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

## Note:

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

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

## types of Keys?

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

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

Remember:

- 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.

# clustered and non-clustered index

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

MySQL allows two different types of Indexing:

- Clustered Index
- Non-Clustered Index

## Clustered Index:-

- Clustered index is used to optimize the speed of most common lookups and DML operations like INSERT, UPDATE, and DELETE command.
- Clustered indexes sort and store the data rows in the table based on their key (primary key) values that can be sorted in only one direction.
- If the table column contains a **primary key** or **not null** and **unique key**, MySQL creates a clustered index.
- Data retrieval is faster than non-cluster index.
- Slower inserts/updates if the indexed column values change frequently, as it may require rearranging rows to maintain the order.

# clustered and non-clustered index

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

MySQL allows two different types of Indexing:

- Clustered Index
- Non-Clustered Index

## Non-Clustered Index:-

- The indexes other than PRIMARY indexes (clustered indexes) called a non-clustered index.
- The non-clustered indexes are also known as secondary indexes.
- A secondary index may be created on one or more virtual columns or on a combination of virtual columns and regular columns.
- The non-clustered index and base table data are both stored in different places, so it does not affect the physical row order in the base table.
- It is not sorted (ordering) the table data.
- Extra space is required to store logical structure
- Data update is faster than clustered index

```
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:

- 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.

# constraints – add unique key

*col_name data_type* UNIQUE KEY

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

- ```
  CREATE TABLE clients (
     client_id INT,
     first_name VARCHAR(50),
     last_name VARCHAR(50),
     company_name VARCHAR(255),
     email VARCHAR(255) UNIQUE
  );
  ```

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

- ```
  SHOW  INDEX FROM clients;
  ```

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

- ALTER TABLE 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.

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

A referential constraint could be violated in following cases.

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

Note:

- PARENT and CHILD tables must use the same storage engine,
- and they cannot be defined as temporary tables.
- If we don’t give constraint name. System will automatically generated the constraint name and will assign to foreign key constraint. **e.g. login_ibfk_1, login_ibfk_2, …..**

# anomaly – (primary key/foreign key)

Remember:

Student (parent) Table

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

student_course (child) Table

| RollNo | CourceDuration | CourceName |
|---|---|---|
| 1 | 1.5 month | RDBMS |
| 2 | 1.2 month | NoSQL |
| 3 | 2 month | Networking |
| 1 | 2 month | Java |
| 2 | 2 month | .NET |

## Insertion anomaly:

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

## Updation and Deletion anomaly:

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

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

Remember:

Parent Table

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

Child Table

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

DDL command could be violated in following cases.

**Alter command:**

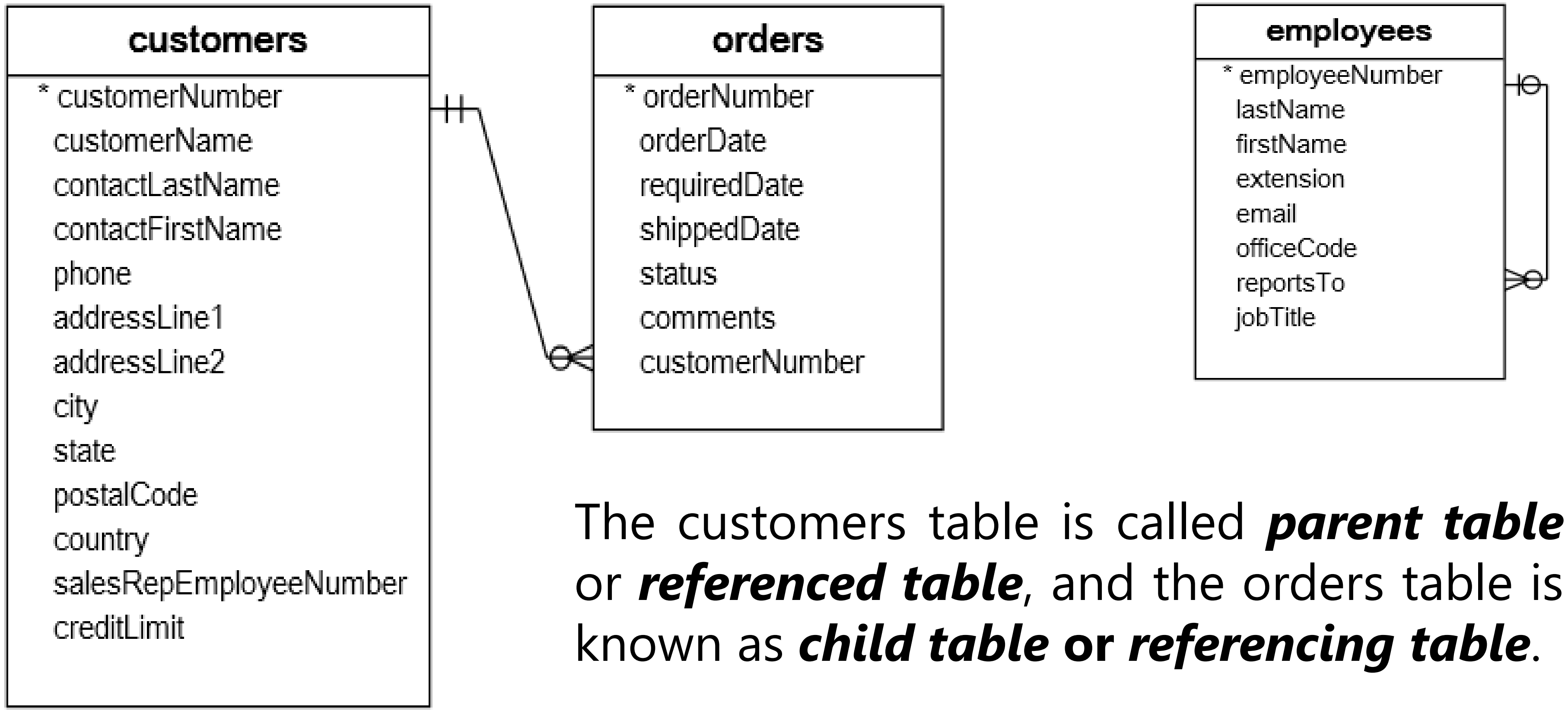
- If we try to modify datatype of RollNo in Student or Student_Course table with VARCHAR, it will not allow.
- If we try to apply auto_increment to RollNo in Student table, it will not allow
- If we try to drop RollNo column from Student table , it will not allow.

**Drop command:**

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

# constraints – foreign key

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




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

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

# Add Foreign Key Constraint using Alter

# constraints – add foreign key using alter

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

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

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

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

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

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

# Drop Foreign Key Constraint using Alter

# constraints – drop foreign key

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

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

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

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

- ALTER TABLE login DROP FOREIGN KEY fk_userID;
- ALTER TABLE login DROP FOREIGN KEY login_ibfk_1; **// login_ibfk_1 is the default constraint name.**
- SELECT table_name, constraint_name, constraint_type FROM information_schema.table_constraints WHERE table_schema = 'DB2';

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

// ERROR

SUM(SAL) MIN(SAL) COUNT(*)

AVG(SAL) MAX(SAL) COUNT(JOB)

# Check Constraint

## CHECK condition expressions must follow some rules.

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

---

## Note:

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

## Remember:

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

- table_name_chk_n

# constraints – check

*col_name data_type* CHECK(*expr*)

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

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

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

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

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

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

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

# Add / Drop Check Constraint using Alter

# constraints – add check using alter

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

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

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

- ALTER TABLE users ADD CHECK(ratings > 50);
- ALTER TABLE users ADD constraint chk_ratings CHECK(ratings > 50);

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

# drop check constraint

# constraints – drop check key

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

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

- `ALTER TABLE users DROP CHECK chk_ratings;`
- `ALTER TABLE users DROP constraint chk_ratings;`
- `ALTER TABLE users DROP CHECK users_chk_1;`

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

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

- `SELECT table_name, constraint_name, constraint_type FROM information_schema.table_constraints WHERE table_schema = 'DB2' AND (table_name LIKE 'U%' OR table_name LIKE 'L%');`

# 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 tbl_name

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

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

## Remember:

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

## Note:

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

```
ALTER TABLE tbl_name ENGINE = InnoDB;

ALTER TABLE tbl_name ADD col1 INT, ADD col2 INT;

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

# add column

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

alter_specification

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

# add column

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

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

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

- ALTER TABLE vehicles
    ADD ID INT UNIQUE auto_increment first,
    ADD model VARCHAR(100) NOT NULL,
    ADD color VARCHAR(50),
    ADD note VARCHAR(255);

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

| | ID | vehicleID | year | make | model | color | note |
|---|---|---|---|---|---|---|---|
| ▶ | 1 | 111 | 2000 | Honda | | NULL | NULL |
| | 2 | 112 | 2002 | Hyundai | | NULL | NULL |
| | 3 | 113 | 2000 | Jeep | | NULL | NULL |
| | 4 | 114 | 2005 | Toyota | | NULL | NULL |
| * | NULL | NULL | NULL | NULL | NULL | NULL | NULL |

# modify column

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

alter_specification

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

# modify column

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

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

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

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

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

# rename column

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

alter_specification

- RENAME COLUMN old_col_name TO new_col_name

# rename column

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

- ```
  ALTER TABLE vehicles
   RENAME COLUMN year TO model_year
  ```

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

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

# change column

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

alter_specification

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

# change column

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

- ```
  ALTER TABLE vehicles
   CHANGE year model_year INT,
   CHANGE color model_color VARCHAR(20),
   CHANGE note vehicleCondition VARCHAR(150);
  ```

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

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

## *change column*

- ```sql
  CREATE TABLE users (
     ID INT PRIMARY KEY,
     userName VARCHAR(40),
     password VARCHAR(25),
     email VARCHAR(255)
  );
  ```

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

- INSERT INTO users VALUES (1, 'rajan', 'ranaj123', 'rajan447.gmail.com');
- INSERT INTO users VALUES (2, 'raj', 'raj', 'raj.gmail.com');

- INSERT INTO login VALUES (1, 1, curdate(), curtime());
- INSERT INTO login VALUES (2, 1, curdate(), curtime());
- INSERT INTO login VALUES (3, 2, curdate(), curtime());
- INSERT INTO login VALUES (4, NULL, curdate(), curtime());

- ALTER TABLE users CHANGE ID userID INT;
- ALTER TABLE login CHANGE userID UID INT;

- INSERT INTO login VALUES (5, NULL, curdate(), curtime());

# drop column

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

*alter_specification*

- DROP [COLUMN] *col_name*

# drop column

```
CREATE TABLE vehicles(
    vehicleID INT,
    model_year SMALLINT,
    make VARCHAR(150),
    model VARCHAR(100),
    model_color VARCHAR(20),
    vehicleCondition VARCHAR(150)
);
```

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

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

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

# alter table

## Sample table

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

## Add new columns to a table

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

## Modify columns

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

## Rename columns

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

## DROP columns

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

# drop table

Remember:

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

# *drop table*

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

Note:

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

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

# create table using different engines

show engines;
set default_storage_engine = memory;

# *create table with memory engine*

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

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

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

re-start mysql server.

- SELECT * FROM temp;

show engines;
set default_storage_engine = csv;

# create table with csv engine

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

e.g.
```
CREATE TABLE x(
    ID INT NOT NULL,
    ename VARCHAR(10) NOT NULL,
    job VARCHAR(10) NOT NULL,
    sal INT NOT NULL) ENGINE = CSV;
```

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

## Note:

- ERROR 1194 (HY000): Table 'x' is marked as crashed and should be repaired.
- mysql> REPAIR TABLE x;

show engines;
set default_storage_engine = blackhole;

# *create table with blackhole engine*

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

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

- INSERT INTO temp(c2, c3, c4) VALUES(100, 200, 300);
- SELECT * FROM temp;

- DROP TRIGGER IF EXISTS triggername;

```
delimiter $$
CREATE TRIGGER triggername BEFRE INSERT ON temp FOR EACH ROW
begin
   INSERT INTO temp1 VALUES (NEW.c1, NEW.c2);
end $$
delimiter ;
```

# *create temporary table*

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

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

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

# table partitioning

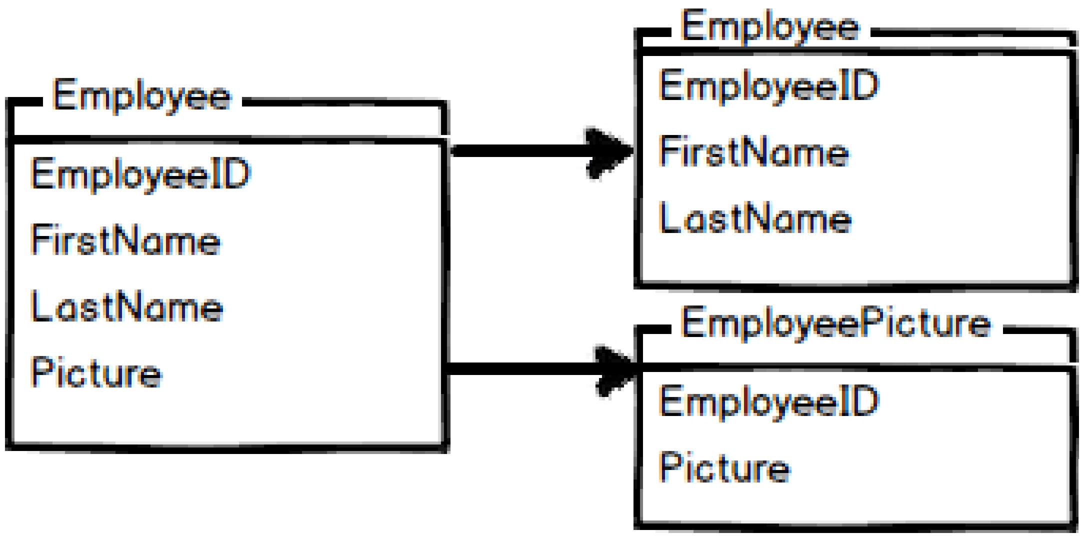
Partitioning separates data into logical units.

# table partitioning

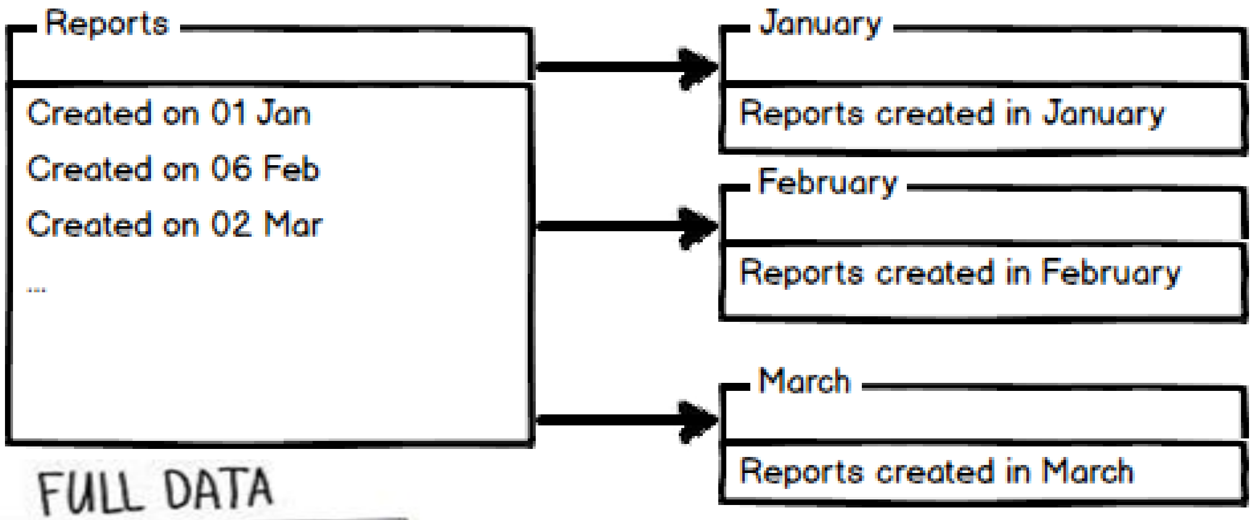
## What is a database table partitioning?

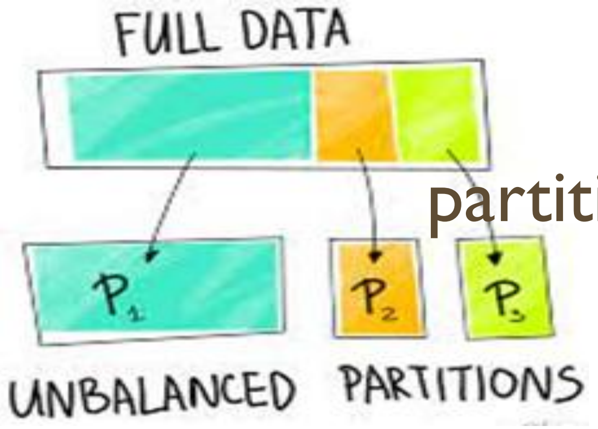
Partitioning is the database process where very large tables are divided into multiple smaller parts. By splitting a large table into smaller, individual tables. The main of goal of partitioning is to aid in maintenance of large tables and to reduce the overall response time to read and load data for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

## Vertical Partitioning




## Horizontal Partitioning




partition skew

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

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

*table partitioning*

## Original Table

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

## Vertical Partitions

### VP1

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

### VP2

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

## Horizontal Partitions

### HP1

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

### HP2

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

# *partitioning by range / list*

## RANGE Partitioning

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

## LIST Partitioning

```
PARTITION BY LIST (COLUMNS)
  (
    PARTITION part_name1 VALUES IN (int_value_list),
    PARTITION part_name2 VALUES IN (int_value_list),
    PARTITION part_name3 VALUES IN (int_value_list)
  )
```

## RANGE Partitioning

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

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

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

## LIST Partitioning

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

# *alter / drop partitioning by range / list*

## Alter Partitioning

### RANGE Partitioning

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

- MAXVALUE can only be used in last partition definition

### LIST Partitioning

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

## DROP Partitioning

ALTER TABLE a ADD PARTITION p3;

# create temporary table

Note:

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

# *temporary table*

Remember:

- You can use the *TEMPORARY* keyword when creating a table.

- A *TEMPORARY* table is visible only to the current session, and is dropped automatically when the session is closed.

- Use *TEMPORARY* table with the same name as the original can be useful when you want to try some statements that modify the contents of the table, without changing the original table.

- The permanent (original) table becomes hidden (inaccessible) to the client who creates the *TEMPORARY* table with same name as the original.

- If you issue a DROP TABLE statement, the *TEMPORARY* table is removed and the original table reappears, it is possible, only when then original *tbl_name* and temporary *tbl_name* are same.

- The original table also reappears if you rename the *TEMPORARY* table.

  e.g. ALTER TABLE **dept** RENAME TO d;

  Temporary table_name

# *temporary table*

e.g.

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

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

# *create temporary table ... like*

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

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

- CREATE TEMPORARY TABLE tempEmployee LIKE employee;

Remember:

- LIKE works only for base tables, not for VIEWS.
- You can use the TEMPORARY keyword when creating a table. A TEMPORARY table is visible only to the current session, and is dropped automatically when the session is closed.
- Use TEMPORARY table with the same name as the original can be useful when you want to try some statements that modify the contents of the table, without changing the original table.

- CREATE TEMPORARY TABLE *new_tbl* SELECT * FROM *orig_tbl* LIMIT 0;

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

# continue with SELECT statement…

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

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

# *SELECT CLAUSE*

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

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

# *Capabilities of SELECT Statement*

1. SELECTION
2. PROJECTION
3. JOINING

# Capabilities of SELECT Statement

## ➢SELECTION

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

***R***

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

# Capabilities of SELECT Statement

## ➢PROJECTION

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

***R***

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

# Capabilities of SELECT Statement

## ➢ JOINING

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

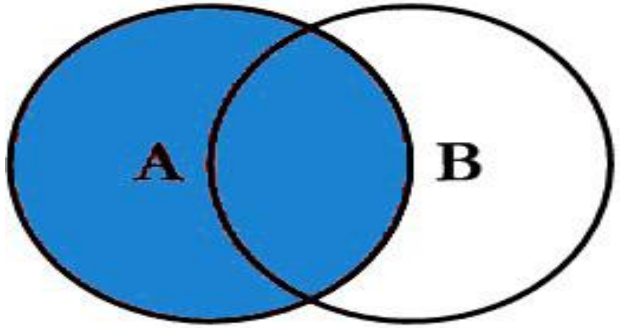
***R***

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

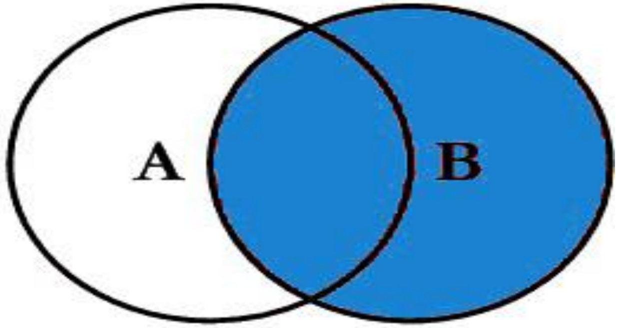
***S***

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

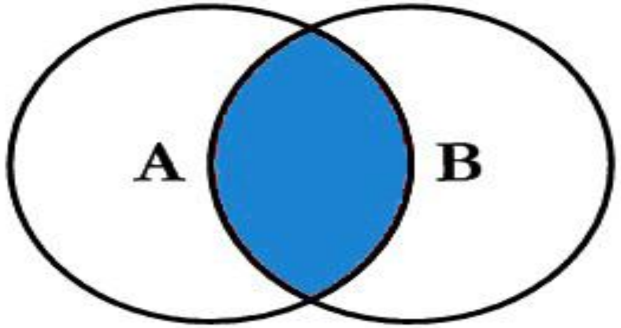
# SQL JOINS



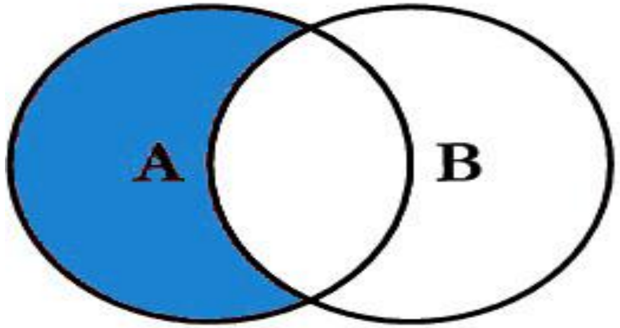

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



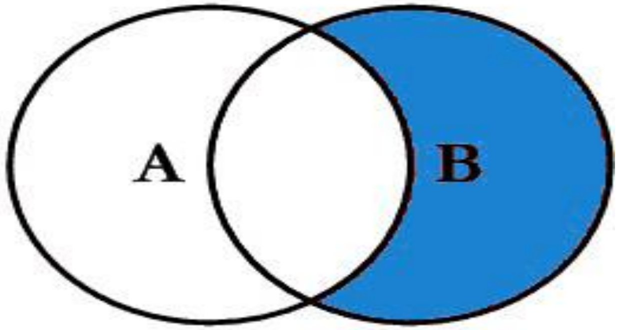

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



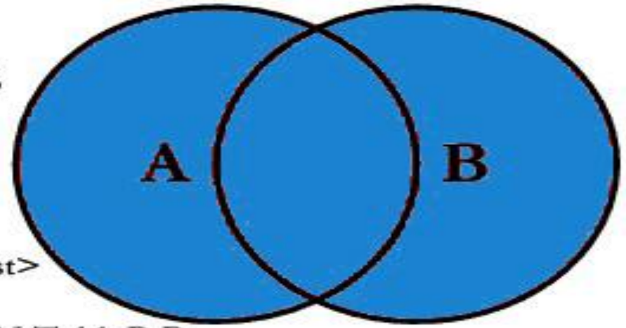

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



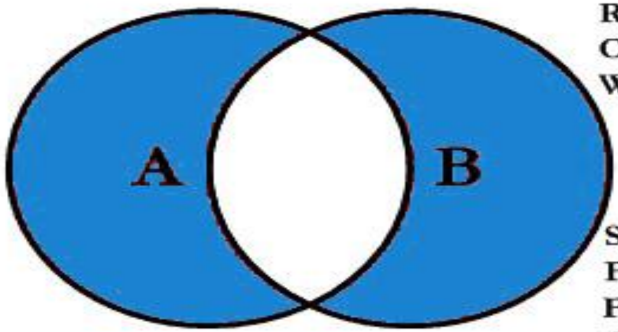

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
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
```

## SELECTION Process

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

## Remember:

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

## PROJECTION Process

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

## Remember:

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

- 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 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;
- SELECT '123abc' = 123;
- SELECT 'abc123' = 0;

Note:

If any expression evaluated with NULL, returns NULL.

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

# identifiers

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

# identifiers

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

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

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

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*)

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

## NULLIF function

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

NULLIF(*expr1*, *expr2*)

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

# control flow functions - case

## CASE function

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

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

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

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

## CASE function

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

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

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

# control flow functions - case

## CASE function

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

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

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

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

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

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

# control flow functions - case

## CASE function

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

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

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

- DATEDIFF(CURDATE(), hiredate) / 365.25

# datetime functions

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

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

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

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

# + 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 | | |

# *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');

# datetime formats

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

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

# datetime formats

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

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

# string functions

## string functions

| Syntax | Result |
|---|---|
| ASCII(*str*) | Returns the numeric value of the leftmost character of the string str. Returns 0 if str is the empty string. Returns NULL if str is NULL.<br>e.g.<br>• 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**.<br>e.g.<br>• 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**.<br>e.g.<br>• SELECT CONCAT('Mr. ' , ename) FROM emp;<br>• SELECT CONCAT('My', NULL, 'SQL'); #op will be NULL |
| ELT(*N, str1, str2, str3, . . .*) | ELT() returns the Nth element of the list of strings: str1 if N = 1, str2 if N = 2, and so on. Returns NULL if N is less than 1 or greater than the number of arguments.<br>e.g.<br>• SELECT ELT(1, 'Bank', 'Of', 'India', 'Kothrud', 'Pune');<br>• SELECT ELT(1, ename, job, sal) FROM emp;<br>• SELECT hiredate, ELT(MONTH(hiredate),'Winter', 'Winter', 'Spring', 'Spring', 'Spring', 'Summer', 'Summer', 'Summer', 'Autumn', 'Autumn', 'Autumn', 'Winter') R1 FROM emp; |

# string functions

| Syntax | Result |
|---|---|
| STRCMP(*expr1, expr2*) | STRCMP() returns 0 if the strings are the same, -1 if the first argument is smaller than the second according to the current sort order, and 1 otherwise. |
| LCASE(*str*) | Returns lower case string. LCASE() **is a synonym for** LOWER(). |
| UCASE(*str*) | Returns upper case string. UCASE() **is a synonym for** UPPER(). |
| LENGTH(*str*) | Returns the length of the string. |
| LPAD(*str, len, padstr*) | Returns the string str, left-padded with the string padstr to a length of len characters. |
| RPAD(*str, len, padstr*) | Returns the string str, right-padded with the string padstr to a length of len characters. |
| REPEAT(*str, count*) | Returns a string consisting of the string str repeated count times. If count is less than 1, returns an empty string. Returns NULL if str or count are NULL. |

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

# string functions

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

- SELECT ename, BINARY ename  FROM emp;

# string functions

| Syntax | Result |
|---|---|
| INSTR(*str, substr*) | Returns the position of the first occurrence of substring substr in string str. |
| REPLACE(*str, from_str, to_str*) | Returns the string str with all occurrences of the string from_str replaced by the string to_str. REPLACE() performs a case-sensitive match when searching for from_str.<br>e.g.<br>• 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().**<br>e.g.<br>• SELECT SUBSTR ('This is the test by IWAY', 6);<br>• 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 \`user name\`, IF(LENGTH(SUBSTR(\`user name\`, INSTR(\`user name\`," "))) = 0,"Weak User", \`user name\`) *R1* FROM emp;
- SELECT LENGTH('saleel') - LENGTH(REPLACE('saleel', 'e', "" ));
- SELECT empno, datePresent, LENGTH(datePresent) - LENGTH(REPLACE(datePresent, ",", "")) + 1 *"Days Present"* FROM emp_attendance;
- SELECT CandidateID,  REPLACE(REPLACE(response, ',', ''), 'n', '') *R1*, LENGTH(REPLACE(REPLACE(response, ',', '') ,'n', '')) *R2* FROM vote_response;
- SELECT c1, c1 / 1, SUBSTR(c1, LENGTH(c1 / 1) + 1) FROM numberString;
- SELECT c1, REVERSE(c1) / 1, LENGTH(REVERSE(c1) / 1), REVERSE(SUBSTR(REVERSE(c1), LENGTH(REVERSE(c1) / 1) + 1)) FROM Stringnumber;
- UPDATE emp SET job := REPLACE(job, 'officers', 'Officers');

## *string functions - examples*

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

# 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),<br>n % m,<br>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)<br>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*

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

e.g.

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

Note:

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

# table statement...

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

# table statement

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

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

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

Remember:

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

# select statement... syntax

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

# syntax

## 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

WHERE → GROUP BY → HAVING → ORDER BY → LIMIT

# select statement… execution

FROM → WHERE → GROUP BY → HAVING → SELECT → DISTINCT → ORDER BY → LIMIT

# select statement... (is checks for)

## Syntax Check

MySQL Database must check each SQL statement for syntactic validity.

```
mysql> SELECT *  FORM emp;

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

## Semantic Check

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

```
mysql> SELECT *  FROM nonexistent_table;

ERROR 1146 (42S02): Table 'db1.nonexistent_table' doesn't exist
```

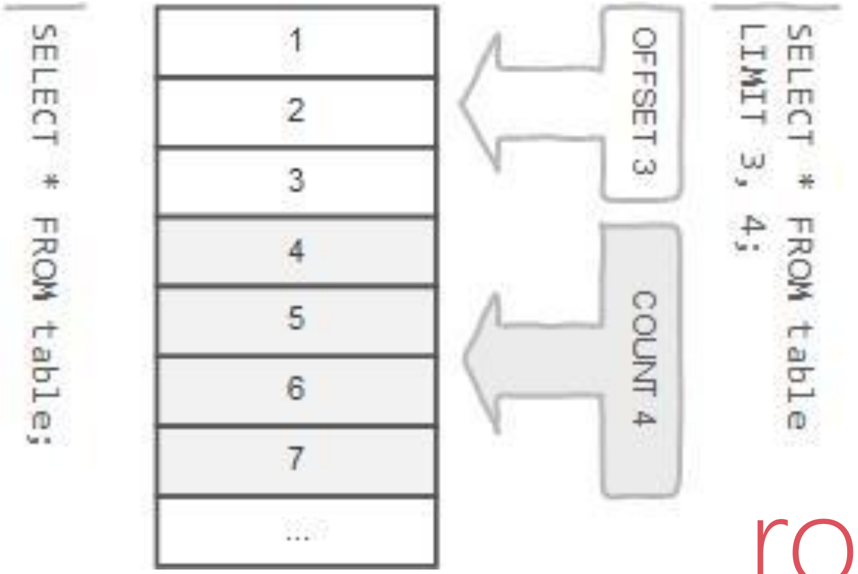
# UUID()/ UUID_SHORT()

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

Remember:

- UUID values in MySQL are **unique** across tables, databases, and servers..

- SELECT UUID() AS R1, UUID_SHORT() AS R2 FROM *tbl_name*;




# row limiting clause

LIMIT is applied after HAVING

## Remember:

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

---

## Note:

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

---

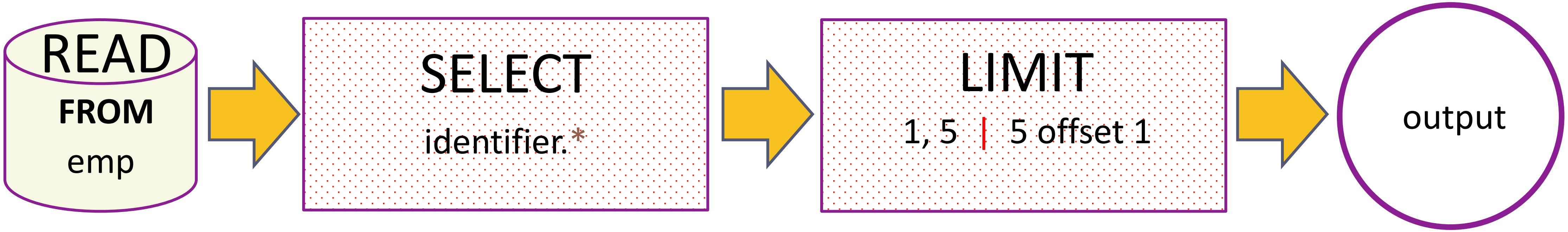
# select - limit

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

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

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

- 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;

# order by clause

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

Remember:

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

# *select - order by*

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

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

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

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

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

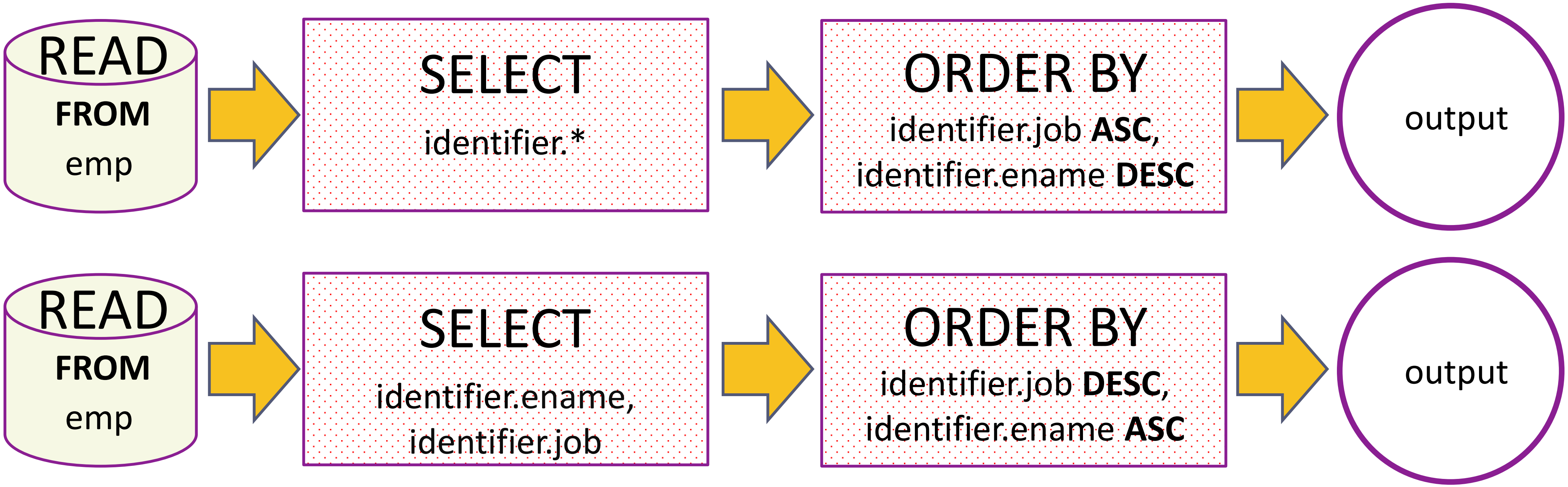
- SELECT * FROM ***r*** ORDER BY *key_part1, key_part2;* // optimizer does not use the index.
- SELECT *key_part1*, *key_part2* FROM ***r*** ORDER BY *key_part1*, *key_part2*; // optimizer uses the index.

# *select - order by*

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

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

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



# select - order by

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

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

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

Remember:

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

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

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

# where clause

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

Note:

**Expressions in WHERE clause can use.**

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

Note:

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

# *select - where*

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

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

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

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

WHERE ($a_1$, $a_2$) <= ($b_1$, $b_2$)

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

Remember:

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

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

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

# select - where

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

### 2. comparison_operator:

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

### 5. logical_operators

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

What will be the result of the query below?

- SELECT 1 = 1;
- SELECT True = 1;
- SELECT True = 2;
- SELECT True = True;
- SELECT 0 = 0;
- SELECT False = False;
- SELECT False = 1;
- SELECT 'a' = 1;
- SELECT 'a' = 0;

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

Note:

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

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

# *select - where*

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

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

WHERE salary > 20000 AND < 30000 –Illegal

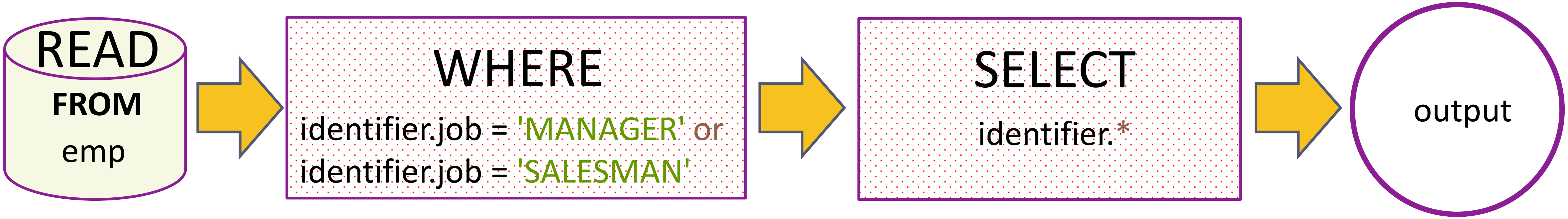
WHERE state NOT = 'CA' --Illegal

| Logical Operators | |
|---|---|
| AND, && | Logical AND e.g. SELECT 1 AND 1; / SELECT 1 AND 0;<br>SELECT 0 AND NULL; / SELECT NULL AND 0;<br>SELECT 1 AND NULL; / SELECT NULL AND 1; |
| OR, \|\| | Logical OR e.g. SELECT 1 OR 1; / SELECT 1 OR 0;<br>SELECT 0 OR NULL; / SELECT NULL OR 0;<br>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 * FROM emp WHERE job = 'MANAGER' OR job = 'SALESMAN';




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

# combining and & or - where

Note:

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

- SELECT * FROM andor WHERE ename = 'saleel' AND city = 'pune' OR city = 'baroda';
- SELECT * FROM andor WHERE ename = 'saleel' AND (city = 'pune' OR city = 'baroda');
- SELECT ename, job, comm FROM emp WHERE comm = 0 OR comm IS NULL AND job = 'CLERK';
- SELECT ename, job, comm FROM emp WHERE (comm = 0 OR comm IS NULL) AND job = 'CLERK';

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

# *select - where*

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

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

- *operand* IS [NOT] *NULL*

### 3. *boolean_ predicate:*

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

# is null / is not null

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

Note:

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

Remember:

# is null / is not null

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

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

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

- SELECT empno, ename, job, sal, comm FROM emp WHERE comm is TRUE;

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

- 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

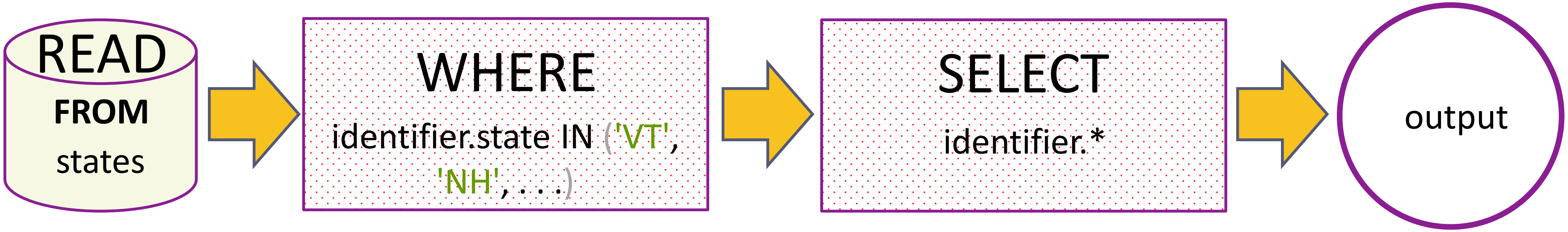
- BOOL and BOOLEAN are synonym of TINYINT(1)

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.) |

# in

*syntax*

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

## Remember:

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

---

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

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

## Problem with NOT IN:

*not in*

*a*

| c1 | c2 |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 3 | 1 |
| 4 | 1 |
| 5 | 1 |

*b*

| c1 | c2 |
|---|---|
| 1 | 7 |
| NULL | 7 |
| 3 | 7 |

- SELECT * FROM a WHERE c1 NOT IN(1, 2, NULL);
- SELECT * FROM a WHERE c1 NOT IN(SELECT c1 FROM b );

  **Empty set (0.00 sec)**

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

Remember:

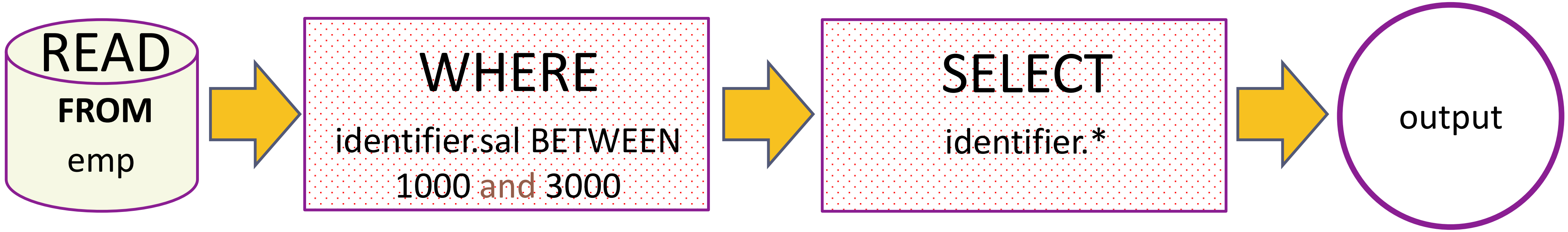
*in*

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

## What will be the result of the query below?

- `SELECT * FROM emp WHERE deptno IN (10);`
- `SELECT * FROM emp WHERE deptno IN (10, 20);`
- `SELECT * FROM emp WHERE False IN (10, 20, 0);`
- `SELECT * FROM emp WHERE True IN (10, 20, 1);`
- `SELECT * FROM emp WHERE 10 IN (10, 20);`
- `SELECT * FROM emp WHERE 7788 IN (empno, mgr);` ←
- `SELECT * FROM emp WHERE 1 IN (10, 20, True, False);`
- `SELECT * FROM emp WHERE deptno IN (10, 20) OR True;`
- `SELECT * FROM emp WHERE deptno IN (10, 20) AND True;`
- `SELECT * FROM emp WHERE deptno IN (SELECT deptno FROM dept);`
- `SELECT * FROM emp WHERE deptno IN (SELECT deptno FROM dept WHERE dname='accounting');`
- `SELECT * FROM emp WHERE deptno IN (TABLE deptno); # ERROR 1241 (21000): Operand should contain 1 column(s)`

- `SELECT * FROM emp WHERE 1 NOT IN (NULL);`
- `SELECT * FROM emp WHERE deptno IN (10, 20);`
- `SELECT * FROM emp WHERE False IN (10, 20, 0);`
- `SELECT * FROM emp WHERE True IN (10, 20, 1);`
- `SELECT * FROM emp WHERE 10 IN (10, 20);`




**4. *predicate*:**

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

# between

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

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

# between

WHERE salary BETWEEN ( 20000 AND 30000 ) – Illegal

*syntax*

column | expression BETWEEN *start_expression* AND *end_expression*

## Remember:

- The BETWEEN operator returns TRUE if the expression to test is greater than or equal to the value of the start_expression and less than or equal to the value of the end_expression.
- You can use the greater than or equal to (**>=**) and less than or equal to (**<=**) to substitute the BETWEEN operator.

---

## Note:

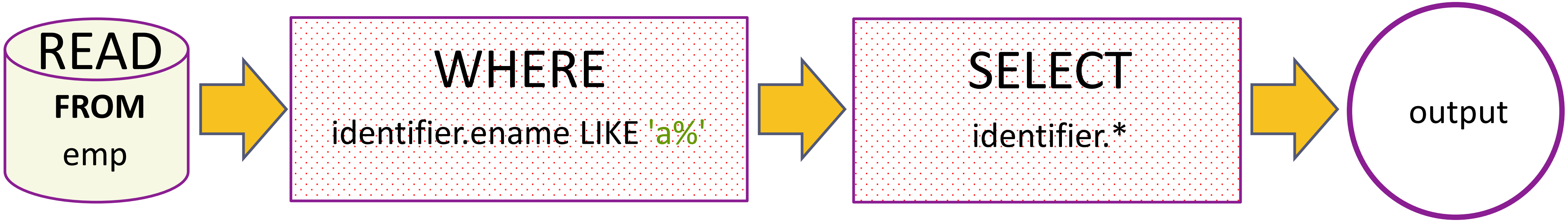
- if any input to the BETWEEN or NOT BETWEEN is NULL, then the result is UNKNOWN.

e.g.

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

---

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




**4. *predicate*:**

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

like

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

# like - string comparison functions

*syntax*

column | expression LIKE 'pattern' [ESCAPE *escape_character*]

## Remember:

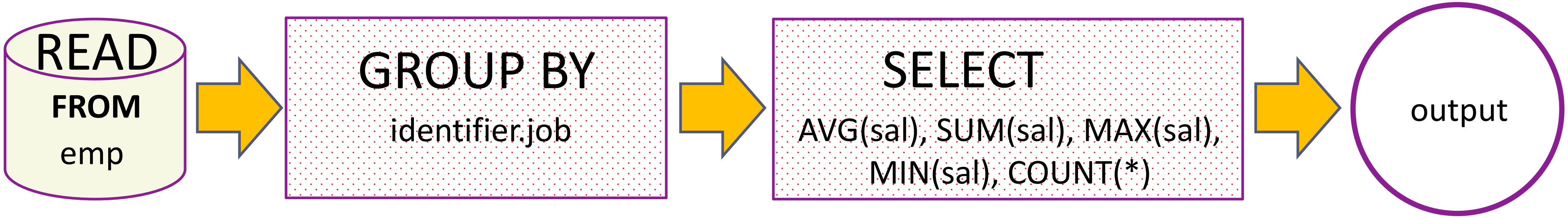
- **%** matches any number of characters, even zero characters. ("%" represents any sequence of characters.)
- _ matches exactly one character. ("_" represents a single character.)
- If we use **default escape character '\'**, then don't use ESCAPE keyword.

## 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;




# aggregate functions

## SUM, AVG, MAX, MIN, COUNT, and GROUP_CONCAT

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

**this is invalid**

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

- SET *SQL_MODE* = ' ';
- SET *SQL_MODE* = IGNORE_SPACE;

## Remember:

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

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

# aggregate functions

Remember:

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

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

Note:

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

  e.g.

  ```
  SELECT MAX(COUNT(*)) FROM emp GROUP BY deptno;
  ```
- Blank space between aggregate functions like (SUM, MIN, MAX, COUNT) are not allowed.

  e.g.

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

Things to… Remember: *aggregate function*

**TODO**

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

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

e.g.

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

Things to… Remember: *aggregate function*

**TODO**

SUM([DISTINCT] *expr*) [*over_clause*]

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

e.g.

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

# Things to… Remember: aggregate function

**TODO**

SUM([DISTINCT] *expr*) [*over_clause*]

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

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

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

Things to… Remember: *aggregate function*

**TODO**

MAX([DISTINCT] *expr*) [*over_clause*]

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

e.g.

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

Things to… Remember: *aggregate function*

**TODO**

MIN([DISTINCT] *expr*) [*over_clause*]

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

e.g.

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

Things to… Remember: *aggregate function*

**TODO**

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

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

Note:

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

Things to… Remember: *aggregate function*

**TODO**

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

e.g.

- SELECT COUNT(*) *"R1"*;
- SELECT COUNT(NULL) *"R1"*;
- SELECT COUNT(*) *"R1"* WHERE True;
- SELECT COUNT(*) *"R1"* WHERE False;
- SELECT COUNT(0) FROM emp;
- SELECT COUNT(1) FROM emp;
- SELECT COUNT(*) FROM emp WHERE empno = -1;
- SELECT COUNT(comm) *"R1"* FROM emp;
- SELECT job, COUNT(*) *"R1"* FROM emp GROUP BY job;
- SELECT CASE WHEN sal <= 1500 THEN 'low' WHEN sal > 1501 and sal < 3000 THEN 'medium' WHEN sal >= 3000 THEN 'high' END *"R1"*, COUNT(*) FROM emp GROUP BY *R1*;

Things to… Remember: *aggregate function*

**TODO**

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

e.g.

- SELECT job, GROUP_CONCAT(ename) FROM emp GROUP BY job;

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

Things to… Remember:

# *aggregate function examples*

**TODO**

- ```
  SELECT productname,
  SUM(CASE WHEN storelocation = 'North' THEN totalsales END) North,
  SUM(CASE WHEN storelocation = 'South' THEN totalsales END) South,
  SUM(CASE WHEN storelocation = 'East' THEN totalsales END) East,
  SUM(CASE WHEN storelocation = 'West' THEN totalsales END) West,
  SUM(CASE WHEN storelocation = 'Central' THEN totalsales END) Central,
  SUM(totalsales) TotalSales FROM pivot_table GROUP BY productname;
  ```

- ```
  SELECT itemname,
  COUNT(CASE WHEN color = 'white' AND size = 'medium' THEN 1 END) White,
  COUNT(CASE WHEN color = 'dark' AND size = 'medium' THEN 1 END) Dark,
  COUNT(CASE WHEN color = 'pastel' AND size = 'medium' THEN 1 END) Pastel FROM shop GROUP BY itemname;
  ```

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

**Group Attributes**

**Group Function Attributes**

# group by clause

## Remember:

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

## Note:

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

e.g.

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

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

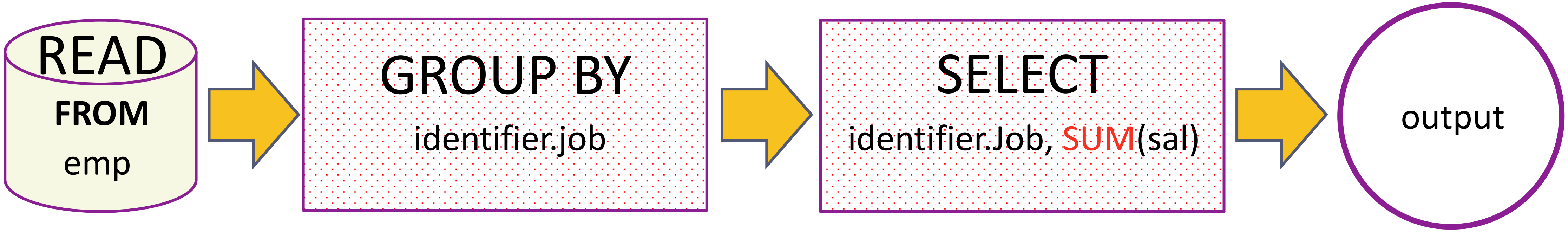
# select - group by

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

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

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

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




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

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

Remember:

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

# having clause

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

Note:

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

e.g.

1. SELECT COUNT(*) FROM emp HAVING deptno=10; *
2. SELECT deptno, COUNT(*) FROM emp GROUP BY deptno HAVING job='manager'; *

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

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

# select - having

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

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

[HAVING *having_condition* ]

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


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

# window function

Note:

MySQL does not support these window function features.

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

# window function

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

- RANK() OVER( [ PARTITION BY *expr1*, *expr2*, . . . ] ORDER BY *expr1* [ ASC|DESC ], . . . )
- DENSE_RANK() OVER( [ PARTITION BY *expr1*, *expr2*, . . . ] ORDER BY *expr1* [ ASC|DESC ], . . . )
- ROW_NUMBER() OVER( [ PARTITION BY *expr1*, *expr2*, . . . ] ORDER BY *expr1* [ ASC|DESC ], . . . )
- LAG(*expr* [, *N* [, *default* ] ] ) OVER( [ PARTITION BY *expr1*, *expr2*, . . . ] ORDER BY *expr1* [ ASC|DESC ], . . . )
- LEAD(*expr* [, *N* [, *default* ] ] ) OVER( [ PARTITION BY *expr1*, *expr2*, . . . ] ORDER BY *expr1* [ ASC|DESC ], . . . )

## Note:

The N and default argument in the function is optional.

- **expr**: It can be a column or any built-in function.
- **N**: It is a positive value which determine number of rows preceding/succeeding the current row. If it is omitted in query then its default value is 1.
- **default**: It is the default value return by function in-case no row precedes/succeedes the current row by N rows. If it is missing then it is by default NULL.

# window function- examples

- SELECT ROW_NUMBER() OVER() *R1*, emp.* FROM emp;
- SELECT RANK() OVER(PARTITION BY job ORDER BY sal) *R1*, ename, sal, job FROM emp;
- SELECT DENSE_RANK() OVER(PARTITION BY job ORDER BY sal) *R1*, ename, sal, job FROM emp;
- SELECT ordid, total, SUM(total) OVER(ORDER BY ordid) FROM ord;
- SELECT * FROM (SELECT ROW_NUMBER() OVER() *R1*, emp.* FROM emp) d WHERE *R1*> (SELECT COUNT(*) - 2 FROM emp); // Print *n* last records
- SELECT id, trainID stationname, timing, TIMEDIFF(LEAD(timing) OVER(PARTITION BY trainid ORDER BY timing), timing) *R2* FROM traintimetable; // train time difference between to stations.
- SELECT id, trainID, stationname *'From Station'*, timing *'Departure Time'* , LEAD(stationname) OVER(PARTITION BY trainid ORDER BY id) *'To Station'*, LEAD(timing) OVER(PARTITION BY trainid ORDER BY id) *'Arrival Time'* FROM traintimetable;

# *window function- examples*

- SELECT custId, type, amount, CASE type WHEN 'd' THEN amount WHEN 'c' THEN amount * -1 END *amount* FROM transactions;
- SELECT year, quarter, amount, SUM(amount) OVER(PARTITION BY year ORDER BY quarter) *R1* FROM quarter_revenue;
- SELECT custId, type, amount, SUM(CASE type WHEN 'd' THEN amount WHEN 'c' THEN amount * -1 END) OVER(PARTITION BY custID ORDER BY _id) *amount* FROM transactions;
- SELECT ordid, custid, total, SUM(total) OVER(PARTITION BY custid ORDER BY ordid) *R1* FROM ord;

# user-defined variables

# user-defined variables

**TODO**

## Remember:

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

  e.g. `SELECT @variable_name;`

# *user-defined variables*

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

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

## Remember:

- for SET, either = or := can be used as the assignment operator.
- You can also assign a value to a user variable in statements (SELECT, ...) other than SET. In this case, the assignment operator must be := and **not** = because latter is treated as the **comparison operator =**.

- set @v1 = 1001, @v2 := 2, @v3 = 'Saleel';
- set @v1 = 1001, @v2 = 2, @v3 := @v1 + @v2;
- SELECT @v1 := MIN(sal), @v2 := MAX(SAL) FROM emp;
- SELECT @v1, @v2, @v3;

Examples:

# common sql statements mistakes

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

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

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

## Remember:

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

# sub-queries

A subquery is a SELECT statement within another statement.

## Note:

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

## subqueries

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

### Remember:

A subquery may occur in:

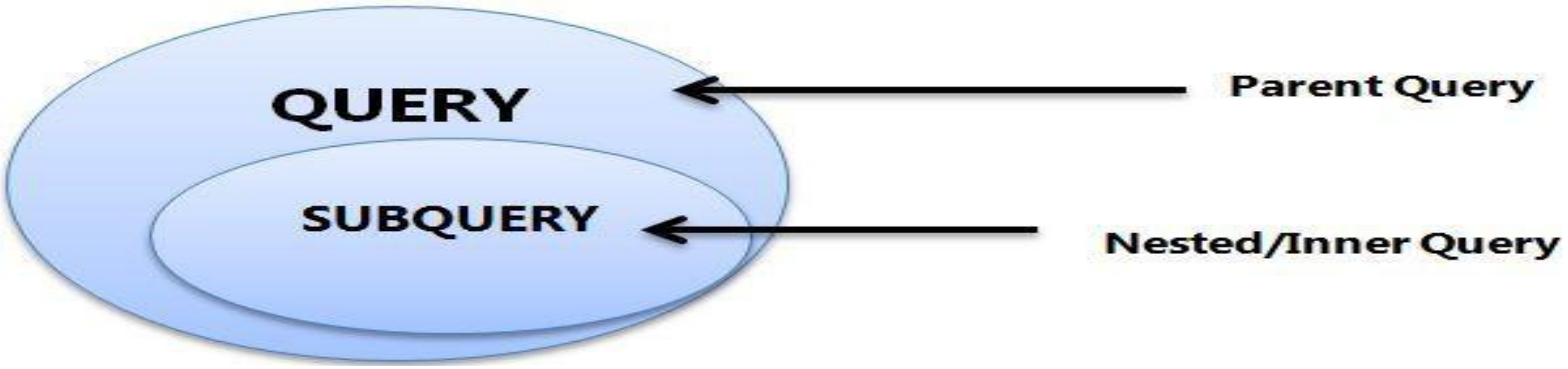
- a SELECT clause
- a FROM clause
- a WHERE clause
- a HAVING clause

### Note:

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

- SELECT
- INSERT
- UPDATE
- DELETE
- CREATE

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



# *types of subqueries*

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

# the subquery as scalar operand

# *the subquery as scalar operand*

TODO

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

## Remember:

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

## Note:

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

## Think:

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

# subquery in the from clause

# subqueries in the from clause

TODO

SELECT $A_1$, $A_2$, $A_3$, (*subquery*) as $A_4$, . . . FROM (*subquery*) [as] name, . . .

Note:

- Every table in a FROM clause must have a name, therefore the [AS] name clause is mandatory.

- SET @x := 0;
  SELECT * FROM (SELECT @x := @x + 1 as *R1*, emp.* FROM emp) DT  WHERE *R1* = 5;
- SELECT * FROM (SELECT @cnt := @cnt + 1 *R1*, MOD(@cnt,2) *R2*, emp.* FROM emp, (SELECT @cnt:=0) DT1) DT2 WHERE *R2* = 0;
- SELECT MIN(*R1*) FROM (SELECT COUNT(job) *R1* FROM emp GROUP BY job) *DT*;
- SELECT MAX(*R1*) FROM (SELECT COUNT(*) *R1* FROM actor_movie GROUP BY actorid) *DT*

| | MAX(R1) |
|---|---|
| | 5 |

- SELECT AVG(SUM(column1)) FROM t1 GROUP BY column1; //ERROR
- SELECT AVG(sum_column1)
  FROM (SELECT SUM(column1) AS *sum_column1*
  FROM t1 GROUP BY column1) AS *t1*;

# comparisons using subquery

# comparisons using subqueries

TODO

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

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

## Remember:

- A subquery can be used before or after any of the comparison operators.
- The subquery can return **at most one value**.
- The value can be the result of an arithmetic expression or a function.

- SELECT * FROM emp WHERE deptno = (SELECT 5 + 5);
- SELECT * FROM emp WHERE sal = (SELECT MAX(sal) FROM emp);
- SELECT MAX(sal) FROM emp WHERE sal < (SELECT MAX(sal) FROM emp);
- SELECT * FROM emp WHERE sal > (SELECT sal FROM emp GROUP BY sal ORDER BY sal DESC limit 3, 1) ORDER BY sal DESC;

## Following statements will raise an error.

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

Remember:

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

# subquery with in, all, any, some

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

The word SOME is an alias for ANY.

- *operand comparison_operator* ANY (*subquery*)
- *operand* IN (*subquery*)
- *operand comparison_operator* SOME (*subquery*)
- *operand comparison_operator* ALL (*subquery*)

- The **ANY** keyword, which must follow a comparison operator, means return TRUE if the comparison is TRUE for ANY of the values in the column that the subquery returns.

- The word **ALL**, which must follow a comparison operator, means return TRUE if the comparison is TRUE for ALL of the values in the column that the subquery returns.

- **IN** and **=ANY** are ***not synonyms*** when used with an expression list. **IN** can take an expression list, but **=ANY** cannot.

- `SELECT * FROM emp WHERE deptno IN (5 + 5, 10 + 10);`
- `SELECT * FROM emp WHERE job =ANY (SELECT job FROM emp WHERE deptno IN(10, 20));`
- `SELECT * FROM emp WHERE deptno =ANY (10, 20);    //error`

# subqueries with in, all, any, some

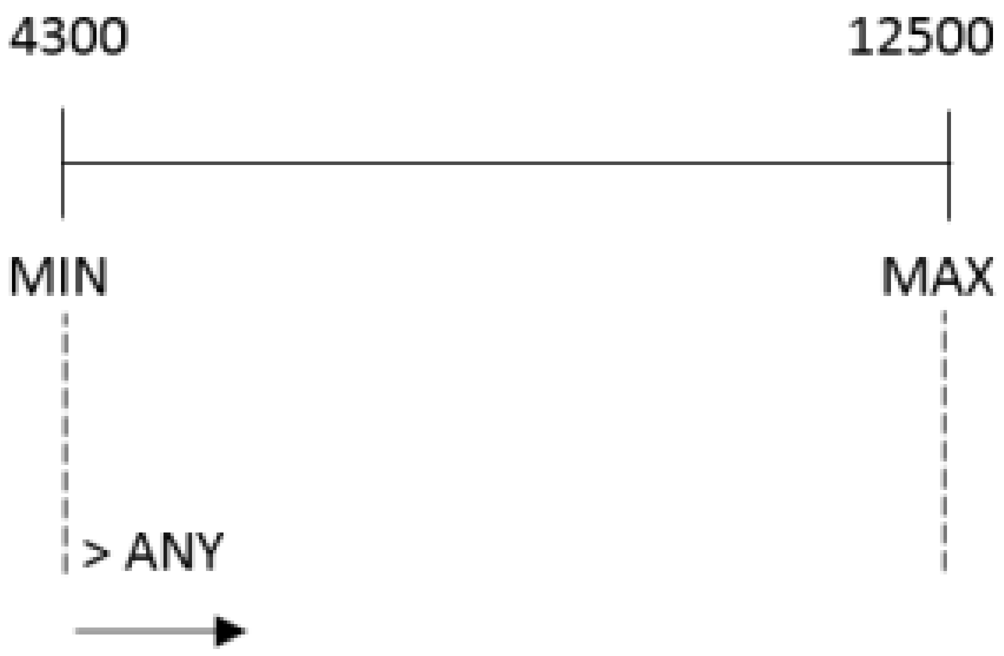
TODO

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

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

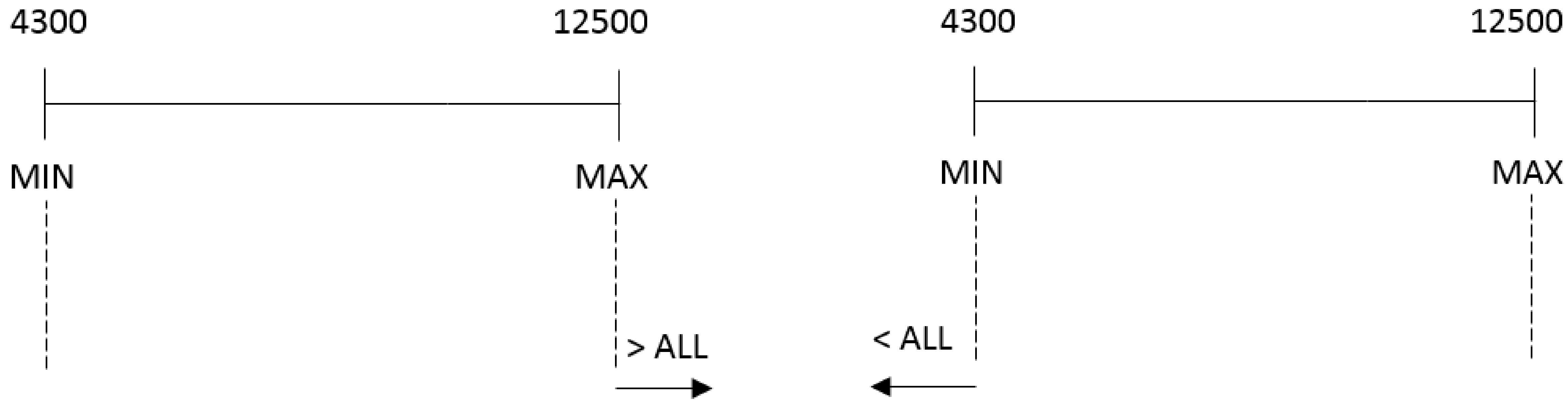
# any / some

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




# *all*

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



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

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

- SELECT * FROM emp WHERE deptno IN (SELECT deptno FROM emp WHERE deptno = 10 OR deptno = 20);
- SELECT * FROM emp WHERE sal >ALL (SELECT sal FROM emp WHERE ename = 'ADAMS' OR ename = 'TURNER');
- SELECT * FROM emp WHERE sal >ANY (SELECT sal FROM emp WHERE ename = 'ADAMS' OR ename = 'TURNER');
- SELECT * FROM emp WHERE sal >SOME (SELECT sal FROM emp WHERE ename = 'ADAMS' OR ename = 'TURNER');
- SELECT * FROM server WHERE id =ANY (SELECT id FROM runningserver); // match all the values in the list
- SELECT * FROM server WHERE id <ALL (SELECT id FROM runningserver); // must be smaller than the smallest value
- SELECT * FROM server WHERE id <ANY (SELECT id FROM runningserver); // smaller than the biggest value

SELECT * FROM d1 WHERE c1 not in (SELECT min(c1) FROM d1 GROUP BY deptno, dname, loc, walletid) ORDER BY deptno;

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

# subquery with exists or not exists

SELECT $A_1, A_2, A_3, A_4, \ldots$ FROM $r$ WHERE [NOT] EXISTS (*subquery*)

# *subqueries with exists or not exists*

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

SELECT $A_1, A_2, A_3, \ldots$
FROM $r$
WHERE [NOT] EXISTS (SELECT $A_1, A_2, A_3, \ldots$ FROM $s$ WHERE $r.A_1 = s.A_1$)

# correlated subquery

# correlated subqueries

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

SELECT $A_1, A_2, A_3, \ldots$
FROM $r$
WHERE <field-name> IN (SELECT $A_1, A_2, A_3, \ldots$ FROM $s$ WHERE $r.A_1 = s.A_1$)

- SELECT * FROM emp *R* WHERE hiredate IN (SELECT hiredate FROM emp *S* WHERE *R*.empno <> *S*.empno); // same hiredate.
- SELECT *R*.* FROM emp *R* WHERE sal IN (SELECT *S*.sal FROM emp *S* WHERE *R*.sal = *S*.sal AND *R*.empno <> *S*.empno); // same salary.
- SELECT *R*.ename, *R*.job, *R*.sal FROM emp *R* WHERE EXISTS (SELECT true FROM emp *S* WHERE *R*.job = *S*.job AND *R*.sal = *S*.sal GROUP BY job, sal HAVING COUNT(*) > 1) ORDER BY job; // same salary jobwise.

# correlated subqueries

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

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

- SELECT * FROM emp *e* WHERE sal > (SELECT AVG(sal) FROM emp WHERE *e*.deptno = emp.deptno) ORDER BY deptno;
- SELECT ename, sal, job FROM emp WHERE sal > (SELECT AVG(sal) FROM emp *e* WHERE *e*.job = emp.job);
- SELECT job, MAX(sal) FROM emp WHERE sal < (SELECT MAX(sal) FROM emp *e* WHERE emp.job = *e*.job GROUP BY e.job) GROUP BY job; // 2nd highest salary jobwise.
- SELECT DISTINCTROW deptno FROM emp WHERE EXISTS (SELECT deptno FROM dept WHERE emp.deptno = dept.deptno); (***Intersect***)
- SELECT DISTINCTROW deptno FROM DEPT WHERE NOT EXISTS (SELECT deptno FROM emp WHERE emp.deptno = dept.deptno); (***except /minus***)

IS NULL
NULL

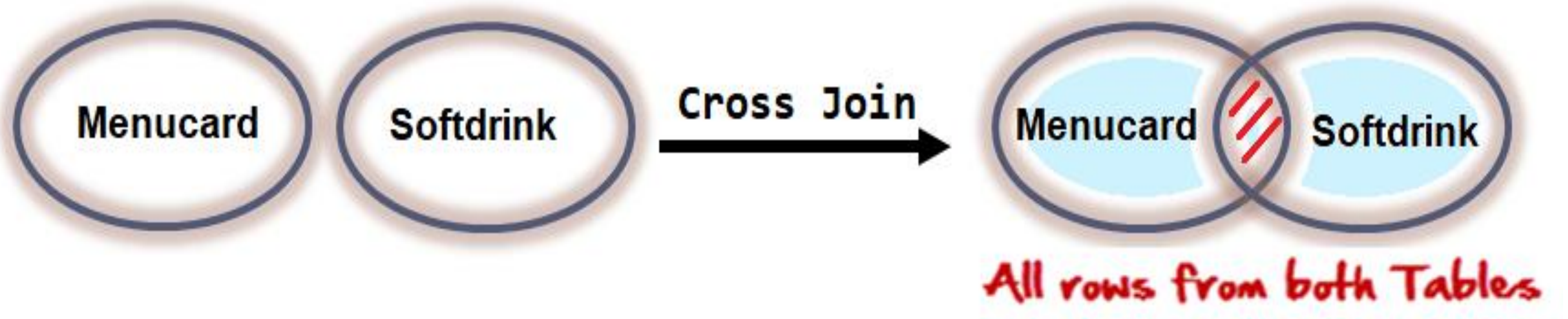
# joins

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

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

## *Type of JOINS*

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




# cartesian or product join (R × S)

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

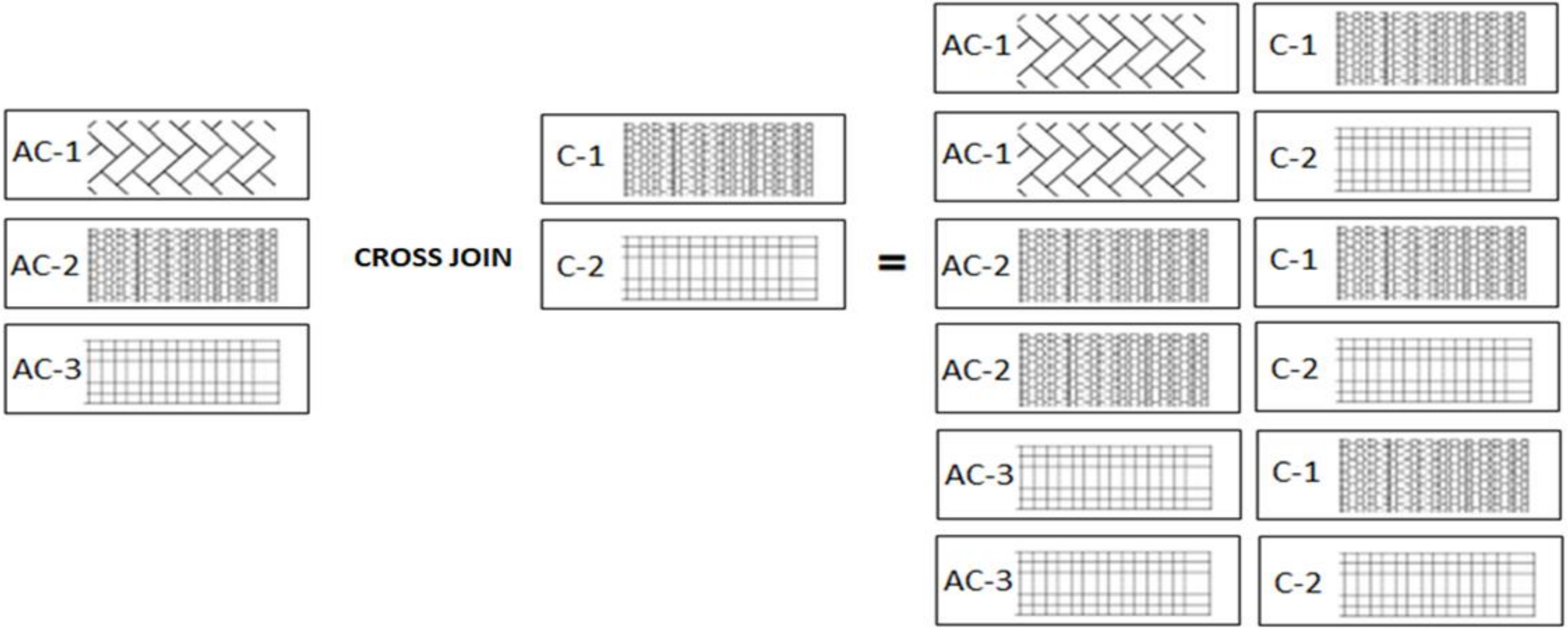
The result of $R(A_1, A_2, \ldots, A_n) \times S(B_1, B_2, \ldots, B_m)$ is a relation Q with degree n + m attributes $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$, in that order.

Degree $d(R \times S) = d(R) + d(S)$.

Cardinality $|R \times S| = |R| * |S|$

# *joins - cartesian or product*

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



# joins - cartesian or product

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

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

- Warehouse/product
- Product/sales_channel
- Cards

$\boldsymbol{r}_1 = \{1, 2\}$
$\boldsymbol{r}_2 = \{a, b, c\}$

$\boldsymbol{r}_1 \times \boldsymbol{r}_2$

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

$\boldsymbol{r}_1 = \{1, 2, 3\}$
$\boldsymbol{r}_2 = \{a, b\}$

$\boldsymbol{r}_1 \times \boldsymbol{r}_2$

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

$\boldsymbol{r}_1 = \{1, 2\}$
$\boldsymbol{r}_2 = \{a, b, null\}$

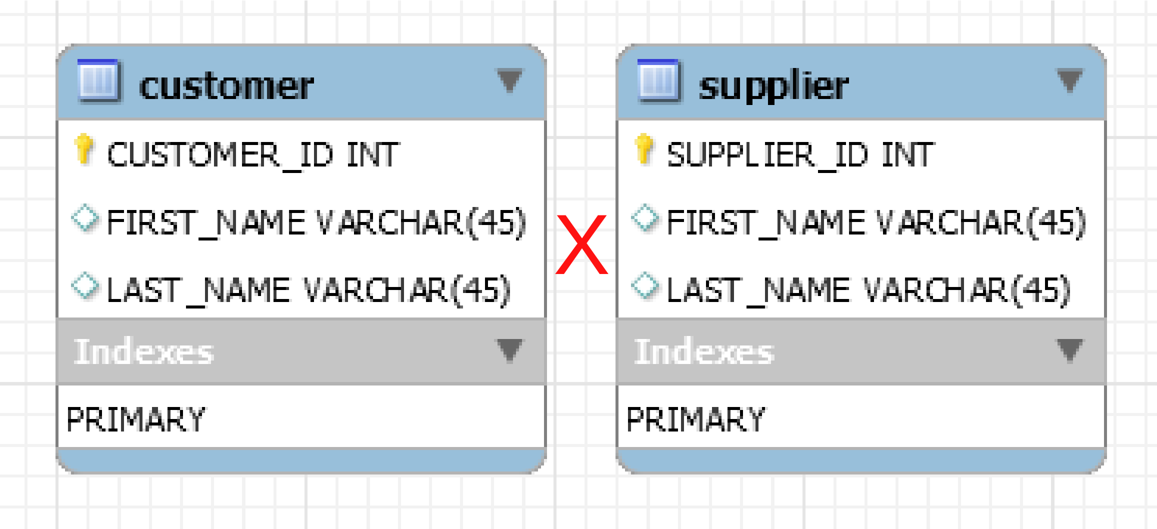
$\boldsymbol{r}_1 \times \boldsymbol{r}_2$

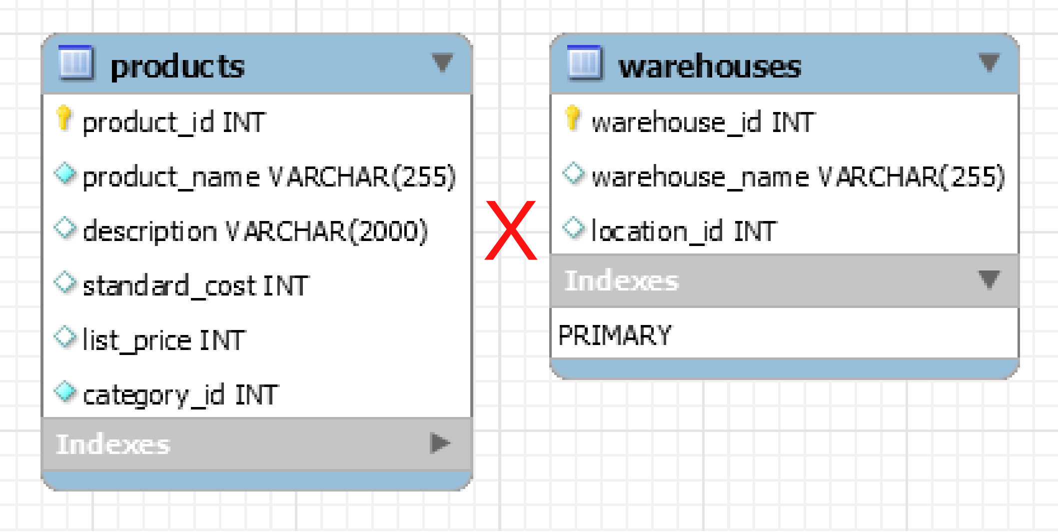
R = { (1, a),
(2, a),
(1, b),
(2, b),
(1, null),
(2, null) }

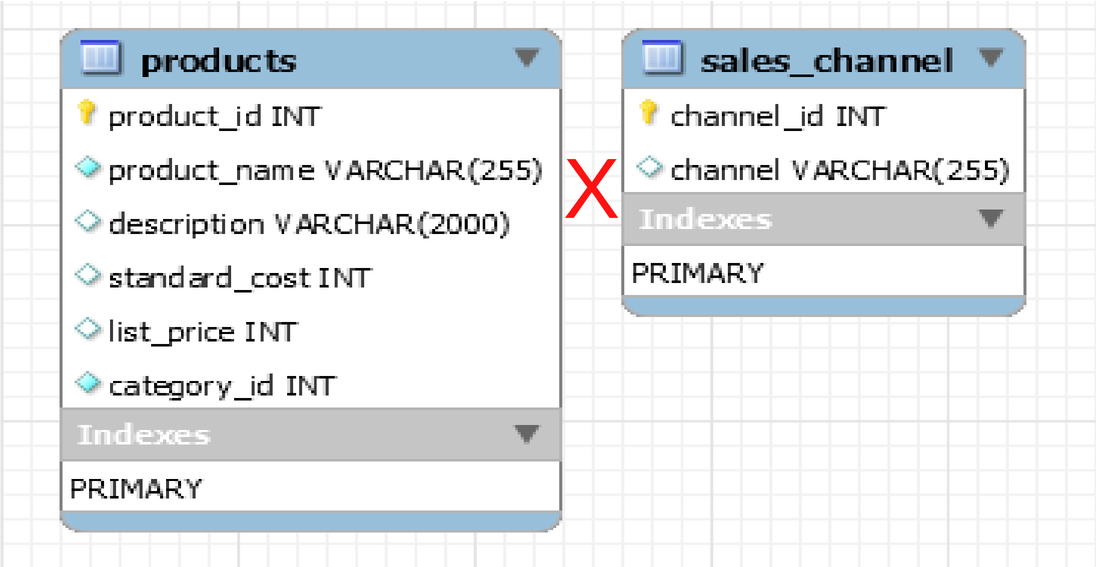
# *joins - cartesian or product*

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

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




- Warehouse/product
- Product/sales_channel
- Cards

# joins - cartesian or product

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

- SELECT * FROM menucard, softdrink;

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

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

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

# joins - cartesian or product

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

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

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

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

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

# joins - cartesian or product example

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

# *joins – cross join*

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

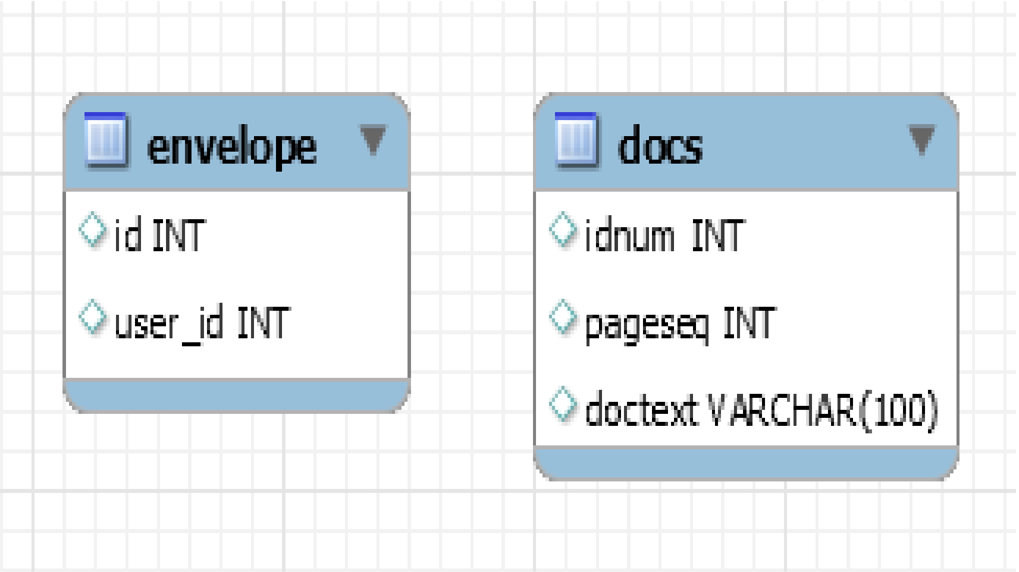
SELECT $A_1, A_2, A_3, \ldots$ FROM $r_1$ CROSS JOIN $r_2, \ldots$

**envelope Table**

| id | user_id |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |

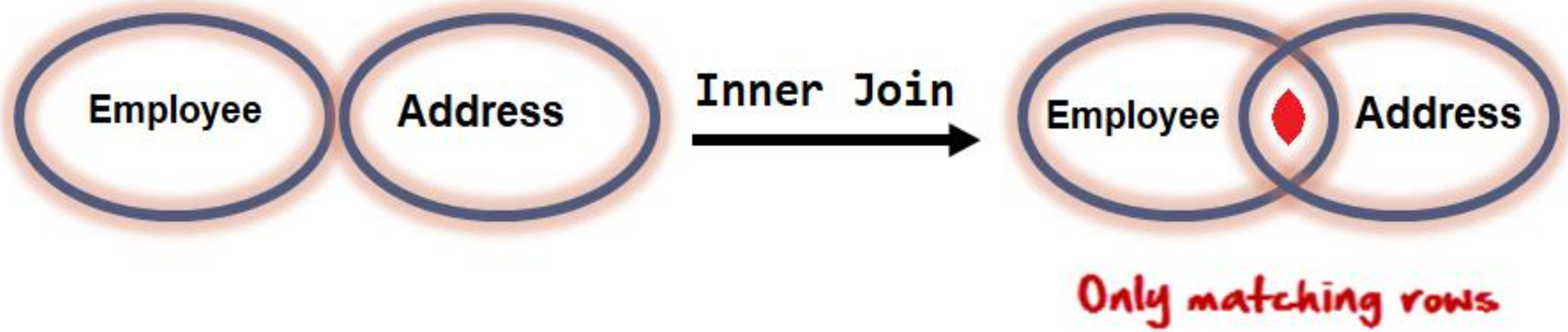
**docs Table**

| idnum | pageseq | doctext |
|---|---|---|
| 1 | 5 | NULL |
| 2 | 6 | NULL |
| NULL | 0 | NULL |




- SELECT * FROM envelope CROSS JOIN docs;

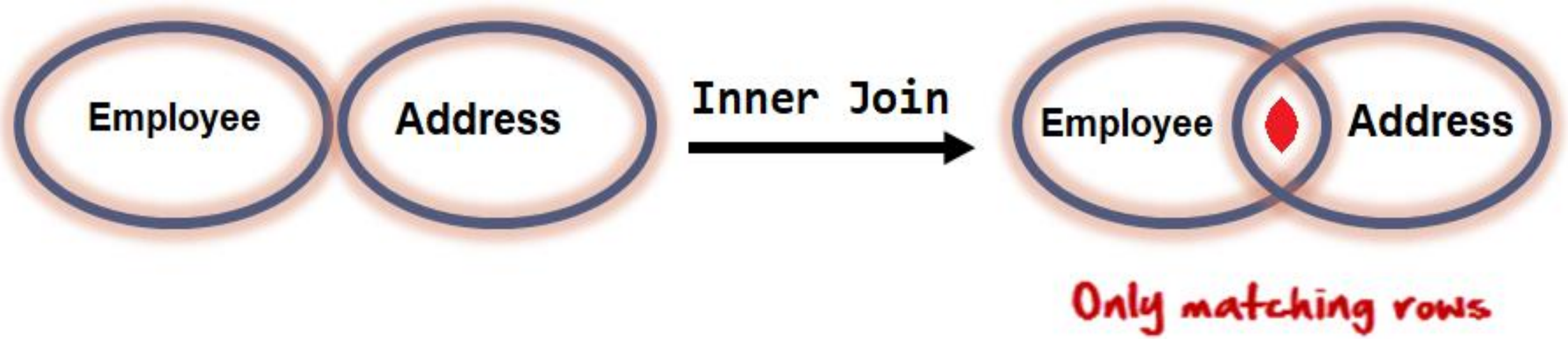
| id | user_id | idnum | pageseq | doctext |
|---|---|---|---|---|
| 1 | 1 | 1 | 5 | NULL |
| 2 | 2 | 1 | 5 | NULL |
| 3 | 3 | 1 | 5 | NULL |
| 1 | 1 | 2 | 6 | NULL |
| 2 | 2 | 2 | 6 | NULL |
| 3 | 3 | 2 | 6 | NULL |
| 1 | 1 | NULL | 0 | NULL |
| 2 | 2 | NULL | 0 | NULL |
| 3 | 3 | NULL | 0 | NULL |




# equi join $(R \bowtie S)$

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

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



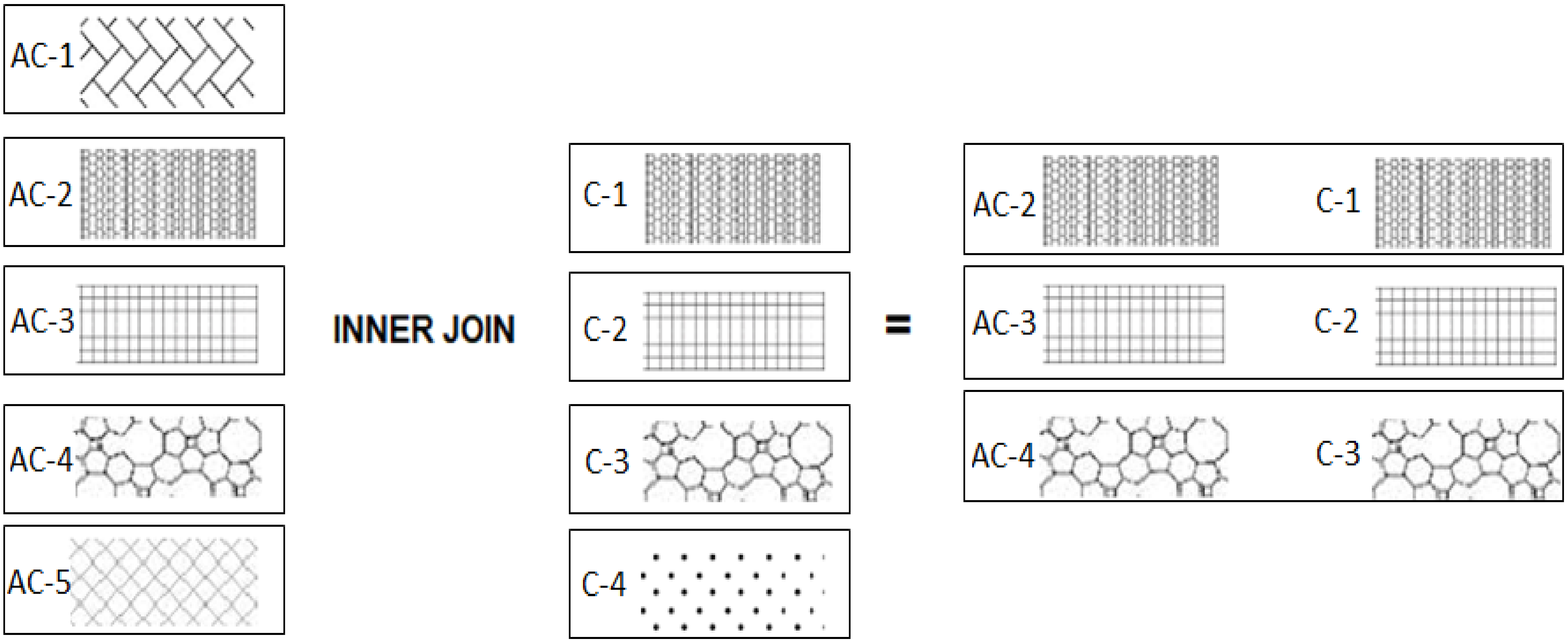

| employeeID | Name | deptNo |
|---|---|---|
| 1 | Saleel | 10 |
| 2 | Sharmin | 20 |
| 3 | Vrushali | 10 |

| deptNo | Name |
|---|---|
| 10 | Sales |
| 20 | Accounting |
| 30 | Manager |

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

# equi join example

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



# *joins – equi join*

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

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

$\boldsymbol{r}_1 = \{ 1, 2, 3, 4 \}$

$\boldsymbol{r}_2 = \{ (1, a), (2, b), (1, c), (3, d), (2, e), (1, f) \}$

$\boldsymbol{r}_1 = \boldsymbol{r}_2$

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

Remember:

A general join condition is of the form <condition> AND <condition> AND . . . AND <condition>, where each <condition> is of the form Ai θ Bj, **Ai is an attribute of R, Bj is an attribute of S**.

# joins – equi join

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

| | EMPNO | ENAME | JOB | MGR | HIREDATE | SAL | COMM | DEPTNO | BONUSID | USER NAME | PWD | isActive | DEPTNO | DNAME | LOC | PWD |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ▶ | 7369 | SMITH | CLERK | 7902 | 1980-12-17 | 800.00 | NULL | 20 | 2 | SMITH | a12recmpm | 0 | 20 | RESEARCH | DALLAS | a12recmpm |
| | 7415 | AARAV | CLERK | 7902 | 1981-12-31 | 3350.00 | NULL | 10 | NULL | AARAV | NULL | 0 | 10 | ACCOUNTING | NEW YORK | r50mpm |
| | 7421 | THOMAS | CLERK | 7920 | 1981-07-19 | 1750.00 | NULL | 10 | 1 | THOMAS | r50mpm | 0 | 10 | ACCOUNTING | NEW YORK | r50mpm |
| | 7499 | ALLEN | SALESMAN | 7698 | 1981-02-20 | 1600.00 | 300.00 | 30 | 4 | ALLEN | sales@2017 | 1 | 30 | SALES | CHICAGO | sales@2017 |
| | 7521 | WARD | SALESMAN | 7698 | 1981-02-22 | 1250.00 | 500.00 | 30 | 1 | WARD | sales@2017 | 1 | 30 | SALES | CHICAGO | sales@2017 |
| | 7566 | JONES | MANAGER | 7839 | 1981-04-02 | 2975.00 | NULL | 20 | 4 | JONES | a12recmpm | 1 | 20 | RESEARCH | DALLAS | a12recmpm |
| | 7654 | MARTIN | SALESMAN | 7698 | 1981-09-28 | 1250.00 | 1400.00 | 30 | 6 | MARTIN | sales@2017 | 1 | 30 | SALES | CHICAGO | sales@2017 |
| | 7698 | BLAKE | MANAGER | 7839 | 1981-05-01 | 2850.00 | NULL | 30 | 1 | BLAKE | sales@2017 | 1 | 30 | SALES | CHICAGO | sales@2017 |
| | 7782 | CLARK | MANAGER | 7839 | 1981-06-09 | 2450.00 | NULL | 10 | 3 | CLARK | r50mpm | 1 | 10 | ACCOUNTING | NEW YORK | r50mpm |
| | 7788 | SCOTT | ANALYST | 7566 | 1982-12-09 | 3000.00 | NULL | 20 | 3 | SCOTT | a12recmpm | 1 | 20 | RESEARCH | DALLAS | a12recmpm |
| | 7839 | KING | PRESIDENT | NULL | 1981-11-17 | 5000.00 | NULL | 10 | 1 | KING | r50mpm | 1 | 10 | ACCOUNTING | NEW YORK | r50mpm |
| | 7844 | TURNER | SALESMAN | 7698 | 1981-09-08 | 1500.00 | 0.00 | 30 | 5 | TURNER | sales@2017 | 1 | 30 | SALES | CHICAGO | sales@2017 |
| | 7876 | ADAMS | CLERK | 7788 | 1983-01-12 | 1100.00 | NULL | 20 | 1 | ADAMS | a12recmpm | 1 | 20 | RESEARCH | DALLAS | a12recmpm |
| | 7900 | JAMES | CLERK | 7698 | 1981-12-03 | 950.00 | NULL | 30 | 2 | JAMES | sales@2017 | 1 | 30 | SALES | CHICAGO | sales@2017 |
| | 7902 | FORD | ANALYST | 7566 | 1981-12-03 | 3000.00 | NULL | 20 | 4 | FORD | a12recmpm | 0 | 20 | RESEARCH | DALLAS | a12recmpm |
| | 7919 | HOFFMAN | MANAGER | 7566 | 1982-03-24 | 4150.00 | NULL | 30 | 3 | HOFFMAN | sales@2017 | 1 | 30 | SALES | CHICAGO | sales@2017 |
| | 7920 | GRASS | SALESMAN | 7919 | 1980-02-14 | 2575.00 | 2700.00 | 30 | 5 | GRASS | sales@2017 | 1 | 30 | SALES | CHICAGO | sales@2017 |
| | 7934 | MILLER | CLERK | 7919 | 982-01-23 | 1300.00 | NULL | 10 | 2 | MILLER | r50mpm | 0 | 10 | ACCOUNTING | NEW YORK | r50mpm |

## Remember:

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

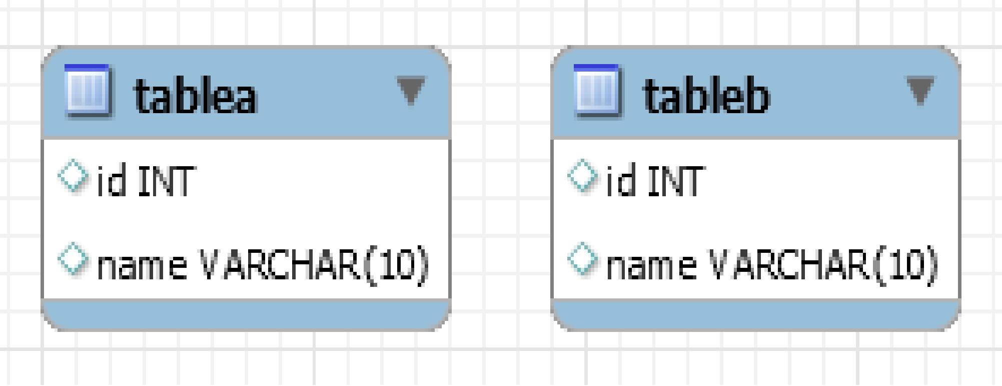
# joins – equi join

tableA Table

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

tableB Table

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




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

| id | name | id | name |
|---|---|---|---|
| 1 | a | 1 | a |
| 1 | NULL | 1 | a |
| 1 | b | 1 | a |
| 2 | b | 2 | x |
| 2 | b | 2 | NULL |
| 5 | aa | 5 | z |
| 5 | NULL | 5 | z |

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

| id | name | id | name |
|---|---|---|---|
| 1 | a | 1 | a |
| 8 | a | 1 | a |
| 2 | b | 4 | b |
| 1 | b | 4 | b |

## ON Contrition

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

# on condition and using attribute

## USING Attribute Contrition

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

Note:

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

# joins – on & using clause example

## The ON clause

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

JOINING CONDITION

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

## The USING clause

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

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

JOINING CONDITION

# inner join $(R \bowtie S)$

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

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

# joins – inner join

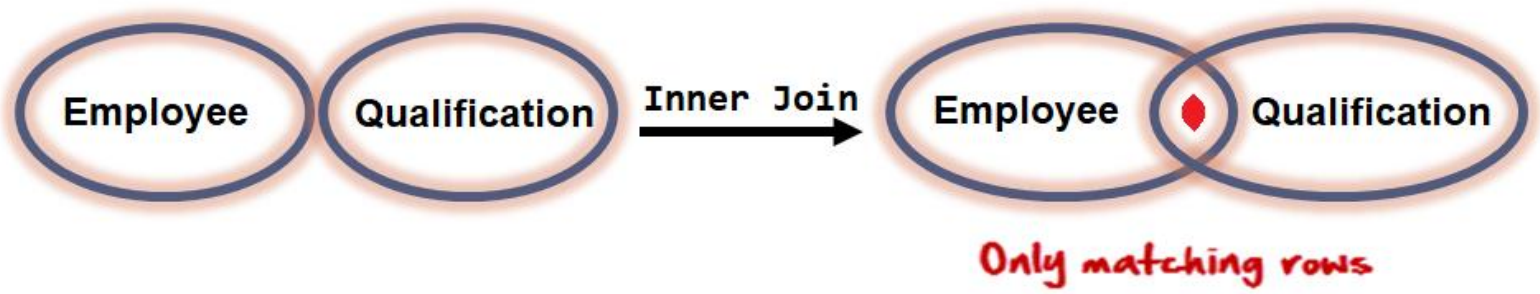
The INNER JOIN selects all rows from both participating tables as long as there is a match between the columns. An SQL INNER JOIN is same as JOIN clause, combining rows from two or more tables.

SELECT $A_1, A_2, A_3, \ldots$ FROM $r_1$ [INNER] JOIN $r_2$ ON $r_1.A_1 = r_2.A_1$

- SELECT * FROM employee *emp* INNER JOIN qualification *quali* ON *emp*.id = *quali*.employeeid;

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

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



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

# joins – inner join

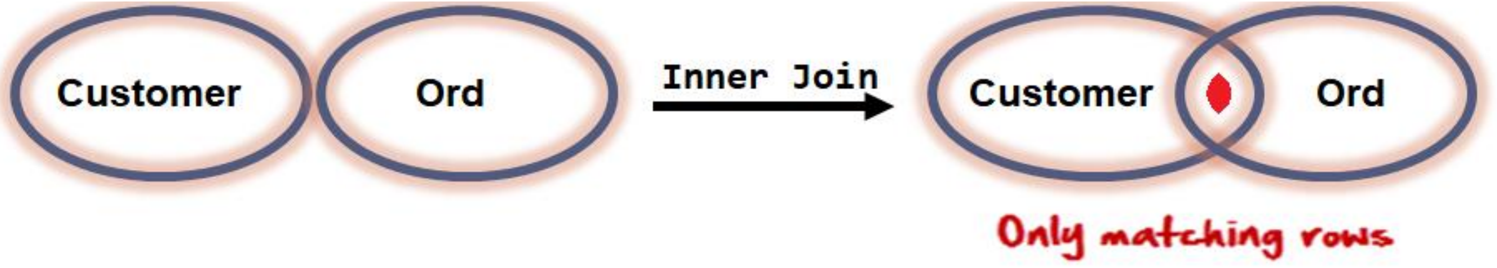
The INNER JOIN selects all rows from both participating tables as long as there is a match between the columns. An SQL INNER JOIN is same as JOIN clause, combining rows from two or more tables.

SELECT $A_1, A_2, A_3, \ldots$ FROM $r_1$ [INNER] JOIN $r_2$ ON $r_1.A_1 = r_2.A_1$

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

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

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



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

# natural join

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

Remember:

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

**e.g.**

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

# joins – natural join

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

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

SELECT $A_1, A_2, A_3, \ldots$ FROM $r_1$ NATURAL [INNER] JOIN $r_2$ NATURAL [INNER] JOIN $r_3 \ldots$

- SELECT * FROM emp NATURAL JOIN dept;

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

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

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

SELECT * FROM EMP NATURAL JOIN DEPT

INNER Vs NATURAL

*inner join vs natural join*

| INNER JOIN | NATURAL JOIN |
|---|---|
| Inner Join joins two table on the basis of the column which is explicitly specified in the ON clause. | Natural Join joins two tables based on same attribute name. |
| In Inner Join, The resulting table will contain all the attribute of both the tables including duplicate columns also | In Natural Join, The resulting table will contain all the attributes of both the tables but keep only one copy of each common column |
| In Inner Join, only those records will return which exists in both the tables | Same as Inner Join |
| SYNTAX:<br>• SELECT * FROM $\mathbf{r_1}$ INNER JOIN $\mathbf{r_2}$ ON $\mathbf{r_1.A_1 = r_2.A_1}$;<br>• SELECT * FROM $\mathbf{r_1}$ INNER JOIN $\mathbf{r_2}$ USING($\mathbf{A_1}$, [$\mathbf{A_2}$]); | SYNTAX:<br>• SELECT * FROM $\mathbf{r_1}$ NATURAL JOIN $\mathbf{r_2}$; |

# simple join

TODO

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

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

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

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

JOINING CONDITION

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

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

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

# outer joins

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

```
CREATE TABLE r1 (id INT, c1 VARCHAR(10));
CREATE TABLE r2 (id INT, c1 VARCHAR(10));

INSERT INTO r1 VALUES (4,'AC-1'), (1,'AC-2'),(2,'AC-3'),(3,'AC-4'),(5,'AC-5');
INSERT INTO r2 VALUES (1,'C-1'), (2,'C-2'),(3,'C-3'),(7,'C-4');
```

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

# left outer joins ($R ⟕ S$)

$r_1$ = {1, 2, 3, 4}
$r_2$ = {(1, a), (2, b), (1, c), (3, d), (2, e), (1, f), (5, z)}

$r_1$ left join $r_2$

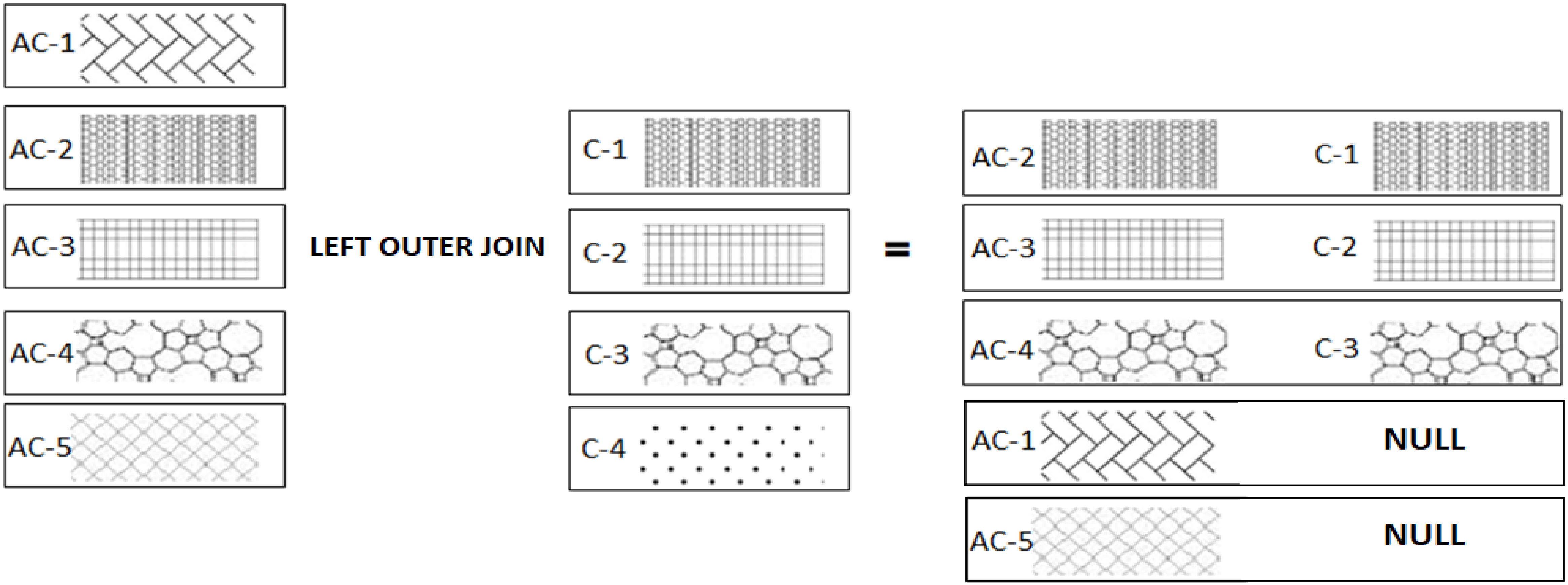
R = {(1, 1, a),
(2, 2, b),
(1, 1, c),
(3, 3, d),
(2, 2, e),
(1, 1, f),
(4, NULL, NULL}

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

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

# joins – left outer join

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

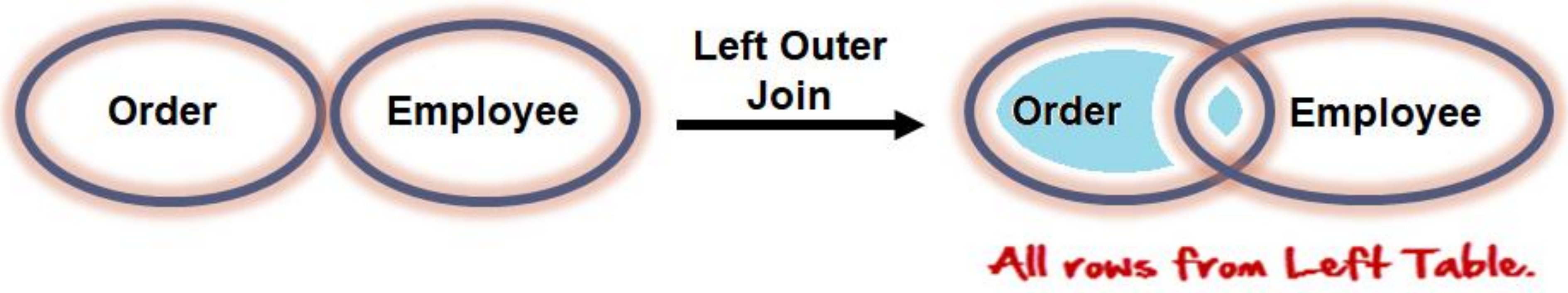


# joins – left outer join

The LEFT JOIN keyword returns all rows from the left table (***r***$_1$), with the matching rows in the right table (***r***$_2$). **The result is NULL in the right side when there is no match.**

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

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



# joins – left outer join

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

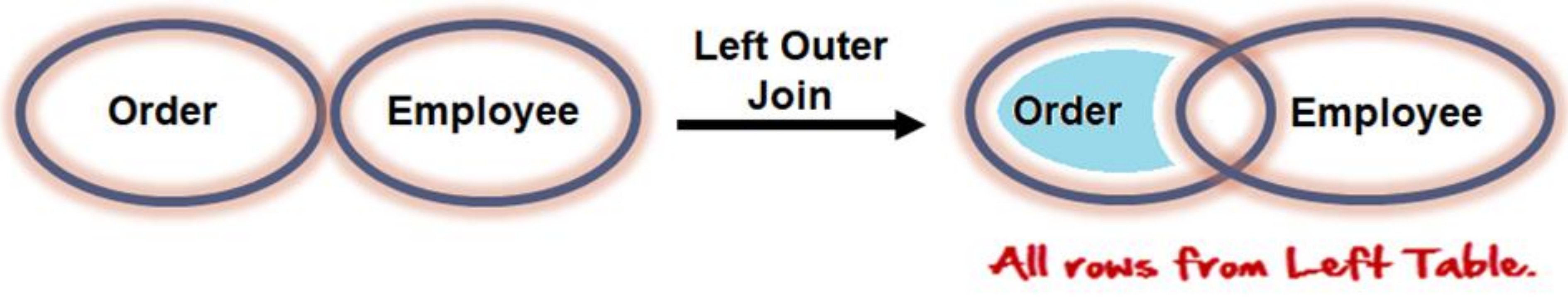
| ID | namefirst | namelast | DOB | emailID | ID | studentID | orderdate | amount |
|---|---|---|---|---|---|---|---|---|
| 6 | lala | prasad | 1980-12-01 | lala.prasad@gmail.com | 26 | 6 | 2019-02-02 | 280 |
| 6 | lala | prasad | 1980-12-01 | lala.prasad@gmail.com | 30 | 6 | 2019-07-10 | 750 |
| 7 |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

# *joins – left outer join*

The LEFT JOIN keyword returns all rows from the left table (***r***$_1$), with the matching rows in the right table (***r***$_2$). **The result is NULL in the right side table when there is no match.**

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

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



# joins – left outer join

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

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

# *joins – left outer join*

The LEFT JOIN keyword returns all rows from the left table (***r***$_1$), with the matching rows in the right table (***r***$_2$). **The result is NULL in the right side table when there is no match.**

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

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

SELECT $\boldsymbol{A}_1, \boldsymbol{A}_2, \boldsymbol{A}_3, \ldots$ FROM $\boldsymbol{r}_1$ NATURAL LEFT [OUTER ] JOIN $\boldsymbol{r}_2$

- SELECT * FROM emp NATURAL LEFT OUTER JOIN dept;

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

# right outer joins ( R ⟖ S )

$r_1$ = {1, 2, 3, 4}
$r_2$ = {(1, a), (2, b), (1, c), (3, d), (2, e), (1, f), (5, z)}

$r_1$ right join $r_2$

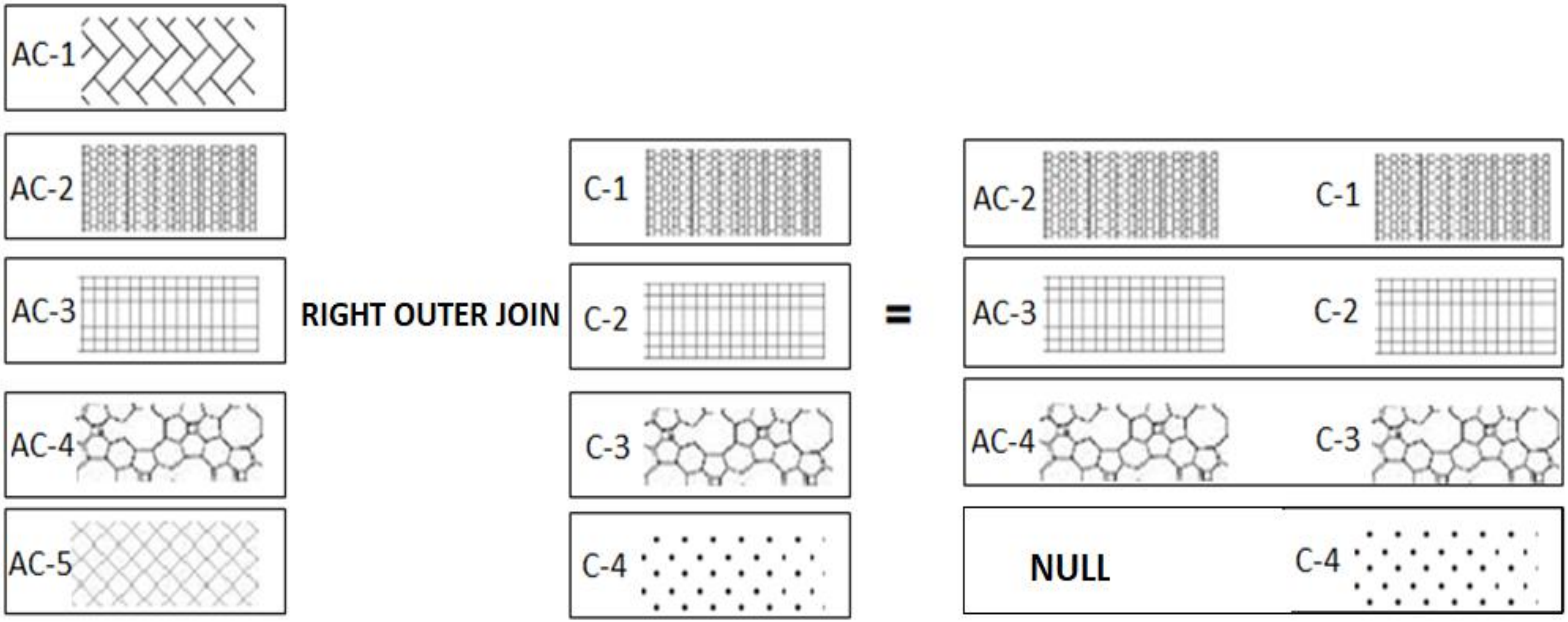
R = {(1, 1, a),
(1, 1, c),
(1, 1, f),
(2, 2, b),
(2, 2, e),
(3, 3, d),
(NULL, 5, z)}

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

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

# *joins – right outer join*

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

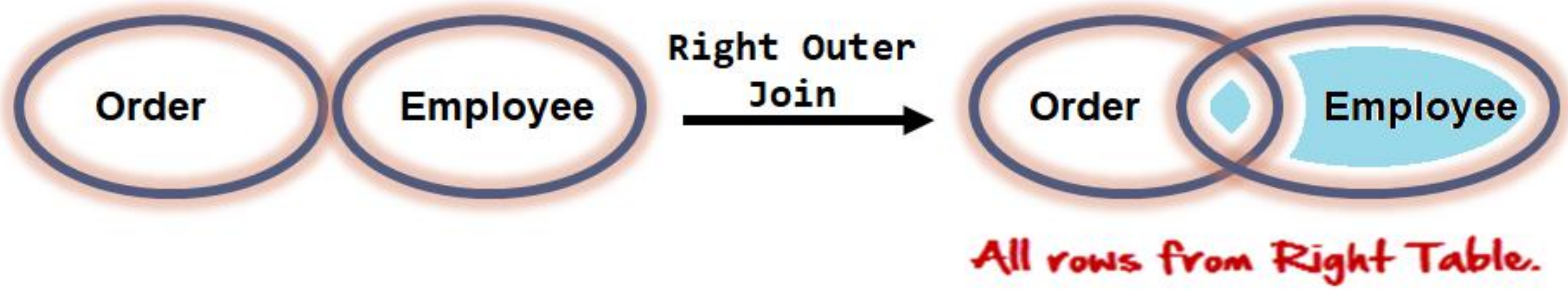


# joins – right outer join

The RIGHT JOIN keyword returns all rows from the right table (***r***$_2$), with the matching rows in the left table (***r***$_1$). **The result is NULL in the left side table when there is no match.**

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

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



# joins – right outer join

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

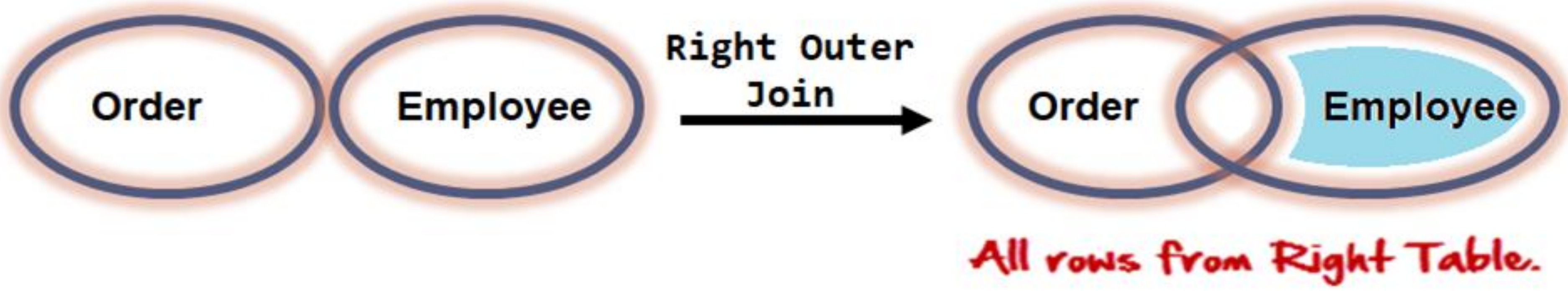
| ID | namefirst | namelast | DOB | emailID | ID | studentID | orderdate | amount |
|---|---|---|---|---|---|---|---|---|
| 6 | lala | prasad | 1980-12-01 | lala.prasad@gmail.com | 7 | 6 | 2019-11-11 | 4000 |
| 1 | saleel | bagde | 1986-12-14 | saleel.bagde@gmail.com | 8 | 1 | 2019-07-19 | 1270 |
| 5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

# *joins – right outer join*

The RIGHT JOIN keyword returns all rows from the right table ($\boldsymbol{r}_2$), with the matching rows in the left table ($\boldsymbol{r}_1$). **The result is NULL in the left side table when there is no match.**

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

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



# joins – right outer join

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

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

# *joins – right outer join*

The RIGHT JOIN keyword returns all rows from the right table (***r***$_2$), with the matching rows in the left table (***r***$_1$).
**The result is NULL in the left side table when there is no match.**

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

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

SELECT $\boldsymbol{A_1}, \boldsymbol{A_2}, \boldsymbol{A_3}, \ldots$ FROM $\boldsymbol{r_1}$ NATURAL RIGHT [OUTER] JOIN $\boldsymbol{r_2}$

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

# joins – left/right outer join examples

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

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

- SELECT server.serverName 'All Servers', IF((server.serverName = runningserver.serverName) = 1 , 'Running Server', 'Server not in use...' ) FROM server LEFT OUTER JOIN  runningserver ON server.id = runningserver.id;

- SELECT COUNT(IF((server.serverName = runningserver.serverName) = 1 , 0, NULL)) 'Running Server', COUNT(IF((server.serverName = runningserver.serverName) = 1 , NULL, 1)) 'Server not is use.....' FROM server LEFT OUTER JOIN  runningserver ON server.id = runningserver.id;

LEFT JOIN Vs RIGHT JOIN

*left join vs right join*

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

Full outer join is like a left or right join except that it contains all rows from both tables. In full outer join, tuples in R that have no matching tuples in S and tuples in S that have no matching tuples in R in their common attribute name.

# full outer joins ( R ⟗ S )

MySQL does not support full outer joins like some database systems. You can get around this by using a UNION operator to join a LEFT OUTER JOIN statement and a RIGHT OUTER JOIN statement.

TODO

# self joins

TODO

# *joins – self join*

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

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

- SELECT DISTINCT *e1*.* FROM emp *e1* , emp *e2* WHERE *e1*.sal = *e2*.sal AND *e1*.empno != *e2*.empno ORDER BY *e1*.sal;
- SELECT *a*.origin, *a*.destination FROM rockets *a*, rockets *b* WHERE (*a*.origin, *a*.destination) = (*b*.destination , *b*.origin) ORDER BY 1;

WITH [*RECURSIVE*]

    cte_name [(*col_name* [, *col_name*] . . .)] AS (*subquery*)
    [, cte_name [(*col_name* [, *col_name*] . . . AS (*subquery*)] . . .

# common table expression ( CTE )

TODO

# common table expression ( CTE )

```
WITH [RECURSIVE]
    cte_name [(col_name [, col_name] . . .)] AS (subquery)
    [, cte_name [(col_name [, col_name] . . . AS (subquery)] . . .
```

- ```
  WITH RECURSIVE a AS (SELECT 1 AS n UNION SELECT n + 1 FROM a WHERE n < 10)
      SELECT n FROM a;
  ```
- ```
  WITH RECURSIVE a AS (SELECT 1 AS n UNION SELECT n + 1 FROM a WHERE n < 10)
      SELECT NOW() + INTERVAL n DAY FROM a;
  ```
- ```
  WITH RECURSIVE a AS (SELECT 1 AS n UNION SELECT n + 1 FROM a WHERE n < 50)
      SELECT ROUND(RAND() * 50) + 50 FROM a;
  ```
- ```
  WITH a AS (SELECT MIN(sal) R1 FROM emp),
      b AS (SELECT MAX(sal) R2 FROM emp)
      SELECT * FROM emp WHERE sal IN ((SELECT R1 FROM a), (SELECT R2 FROM b));
  ```
- ```
  WITH a AS (SELECT job, COUNT(*) R1 FROM emp GROUP BY job)
      SELECT job FROM emp GROUP BY job HAVIN COUNT(job) = (SELECT MIN(R1) FROM a);
  ```
- ```
  WITH a AS (SELECT  CASE WHEN sal <= 1500 THEN 'low' WHEN sal > 1501 AND sal < 3000 THEN 'medium' WHEN sal >=
  3000 THEN 'high' END R1  FROM emp) SELECT *, COUNT(*) FROM a GROUP BY R1;
  ```

# common table expression ( CTE )

WITH [*RECURSIVE*]
cte_name [(*col_name* [, *col_name*] . . .)] AS (*subquery*)
[, cte_name [(*col_name* [, *col_name*] . . . AS (*subquery*)] . . .

- WITH ***a*** AS (SELECT origin, destination FROM rockets),
  ***b*** AS (SELECT origin, destination FROM rockets)
  SELECT ***a***.* FROM ***a***, ***b*** WHERE (***a***.origin, ***a***.destination) = (***b***.destination,***b***.origin) ORDER BY 1;
- WITH ***a*** AS (SELECT channel FROM sales_channel),
  ***b*** AS (SELECT warehouse_Name FROM warehouse),
  ***c*** AS (SELECT channel, warehouse_Name FROM sales_channel, warehouse WHERE sales_channel.channel_ID = warehouse.channel_id)
  SELECT * FROM ***a***, ***b*** WHERE NOT EXISTS (SELECT True FROM ***c*** WHERE (***a***.channel, ***b***.warehouse_Name) = (***c***.channel, ***c***.warehouse_Name)) ORDER BY 1;
- WITH ***a*** AS (SELECT DISTINCT productName FROM pivot_table),
  ***b*** AS (SELECT DISTINCT storeLocation FROM pivot_table)
  SELECT * FROM ***a***, ***b*** WHERE NOT EXISTS (SELECT True FROM pivot_table ***c*** WHERE (***a***.productName, ***b***. storeLocation) = (***c***.productName, ***c***.storeLocation)) ORDER BY 1;

# *common table expression ( CTE )*

WITH [*RECURSIVE*]
    cte_name [(*col_name* [, *col_name*] . . .)] AS (*subquery*)
    [, cte_name [(*col_name* [, *col_name*] . . . AS (*subquery*)] . . .

- WITH ***a*** AS (SELECT ename, deptno, gender FROM emp *e1*  WHERE NOT EXISTS(SELECT True FROM emp *e2* WHERE *e1*.deptno = *e2*.deptno AND *e2*.gender = 'F')),

  ***b*** AS (SELECT ename, deptno, gender FROM emp *e3*  WHERE NOT EXISTS(SELECT True FROM emp *e4* WHERE *e3*.deptno = *e4*.deptno AND *e4*.gender = 'M')),

  ***c*** AS (SELECT * FROM ***a*** UNION SELECT * FROM  ***b***)
  SELECT * FROM emp *e* WHERE NOT EXISTS(SELECT True FROM ***c*** WHERE (*e*.deptno, *e*.gender) = (***c***.deptno, ***c***.gender));

# common table expression ( CTE )

```
WITH [RECURSIVE]
	cte_name [(col_name [, col_name] . . .)] AS (subquery)
	[, cte_name [(col_name [, col_name] . . . AS (subquery)] . . .
```

```
WITH cte AS (SELECT ename, LENGTH(ename) - LENGTH(REPLACE(ename , ' ', '')) R1, REGEXP_INSTR(ename, ' ') firstSpace,
REGEXP_INSTR(ename, ' ', REGEXP_INSTR(ename, ' ‘) + 1 ) secondSpace FROM fullname) SELECT cte.*, CASE WHEN R1 = 0
THEN ename ELSE SUBSTR(ename, 1, firstSpace - 1) END firstName, CASE WHEN R1 <= 1 THEN NULL ELSE SUBSTR(ename,
firstSpace + 1, secondSpace - firstSpace -1) END middleName, CASE WHEN R1 = 2 THEN SUBSTR(ename, secondSpace + 1 )
WHEN R1 = 1 THEN SUBSTR(ename, firstSpace+1) END lastName FROM cte ORDER BY 2;
```

```
WITH cte AS (SELECT sn.id sid, sn.ename sname, tn.id tid, tn.ename tname FROM sourceNames sn LEFT OUTER JOIN
targetNames tn ON sn.id = tn.id UNION ALL SELECT sn.id sid, sn.ename sname, tn.id tid, tn.ename tname FROM
sourceNames sn RIGHT OUTER JOIN targetNames tn ON sn.id=tn.id), cet2 AS ( SELECT cte.*, CASE WHEN (sid,sname)=(tid,
tname) THEN "Common" WHEN sid IS NULL THEN "New in targer" WHEN tid IS NULL THEN "New is source" ELSE
"Mismatch" END R1 FROM cte) SELECT * FROM cet2 WHERE R1 <> 'common';
```

```sql
SELECT C1 FROM T1 UNION SELECT C1 FROM T2
ORDER BY C1

(SELECT C1 FROM T1 ORDER BY C1) UNION
(SELECT C1 FROM T2 ORDER BY C1)

SELECT C1 FROM T1 ORDER BY C1 UNION
SELECT C1 FROM T2 ORDER BY C1   //ERROR
```

(UNION ALL, EXCEPT ALL, INTERSECT ALL)

# set operation in sql

**Set operators** are used to join the results of two (or more) SELECT statements.

There are set union (**UNION ->** ∪), set difference (**EXCEPT ->** -), and set intersection (**INTERSECT ->** ∩) operations. The relations resulting from these set operations are sets of tuples; that is, duplicate tuples are eliminated from the result.

Remember:

- The result set column names are taken from the column names of the first SELECT statement.
- SELECT statement should have the same data type. (Not in MySQL)
- UNION: To apply ORDER BY or LIMIT to an individual SELECT, place the clause inside the parentheses that enclose the SELECT.

e.g. (SELECT . . . ) UNION (SELECT . . .)

- ```sql
  CREATE TABLE Facebook (
    ID INT PRIMARY KEY,
    name VARCHAR(45),
    location VARBINARY(45)
  );
  ```
- ```sql
  CREATE TABLE LinkedIn (
    ID INT PRIMARY KEY,
    name VARCHAR(45),
    location VARBINARY(45)
  );
  ```

```sql
SELECT name FROM students          /* Fetch the union of queries */
UNION
SELECT name FROM contacts;

SELECT name FROM students          /* Fetch the union of queries with duplicates*/
UNION ALL
SELECT name FROM contacts;

SELECT name FROM students          /* Fetch names from students */
EXCEPT                             /* that aren't present in contacts */
SELECT name FROM contacts;

SELECT name FROM students          /* Fetch names from students */
INTERSECT                          /* that are present in contacts as well */
SELECT name FROM contacts;
```

# union

- SELECT DISTINCT * FROM duplicate;
- SELECT * FROM duplicate UNION SELECT * FROM duplicate;
- SELECT deptno, dname, loc, walletid FROM (SELECT ROW_NUMBER() OVER(PARTITION BY deptno) R1, duplicate.* FROM duplicate) T1 WHERE R1=1;

## *syntax*

SELECT . . . UNION [ALL | DISTINCT]
SELECT . . . [UNION [ALL | DISTINCT]
SELECT . . .]

## Note:

- It is used to combine two or more result sets (SELECT statements) into a single set
- it removes duplicate rows between the various SELECT statements.
- each SELECT statement within the UNION operator must have the same number of fields in the result sets.
- default behaviour for UNION is that duplicate rows are removed from the result.

- (SELECT deptno FROM emp LIMIT 1) UNION (SELECT deptno FROM dept LIMIT 1);
- SELECT 'EMP' as *'Table Name'*, COUNT(*) FROM emp UNION SELECT *'DEPT'*, COUNT(*) FROM dept UNION SELECT 'BONUS', COUNT(*) FROM bonus;
- SELECT COUNT(*) FROM customer UNION SELECT COUNT(*) FROM ord;
- SELECT * FROM emp WHERE deptno NOT IN (SELECT deptno FROM emp *m* WHERE *m*.deptno NOT IN (SELECT deptno FROM emp *f* WHERE gender = 'F') UNION SELECT deptno FROM emp *m* WHERE *m*.deptno NOT IN (SELECT deptno FROM emp *f* WHERE gender = 'M' ));

# union

books

| bookID | bookName | Type | Cost |
|---|---|---|---|
| 1 | DS | Hardcover | 950 |
| 1 | DS | Hardcover | 950 |
| 2 | JavaScript | Paperback | 700 |
| 2 | JavaScript | Paperback | 700 |

newbooks

| bookID | bookName | Type | Cost |
|---|---|---|---|
| 1 | Redis | Paperback | 850 |
| 1 | Redis | Paperback | 850 |
| 2 | JavaScript | Paperback | 700 |
| 2 | JavaScript | Paperback | 700 |

Output

| bookID | bookName | Type | Cost |
|---|---|---|---|
| 1 | DS | Hardcover | 950 |
| 2 | JavaScript | Paperback | 700 |
| 1 | Redis | Paperback | 850 |

Duplicate rows not repeated in result set

- ***r*** UNION ***s*** includes distinct/unique rows appearing in both result sets ***r*** and ***s***.

- SELECT * FROM books UNION SELECT * FROM newbooks;
- SELECT * FROM emp WHERE sal IN (SELECT MAX(sal) FROM emp UNION SELECT MIN(sal) FROM emp);

Note:
The following statement will give an error

- SELECT bookName, type FROM books ORDER BY bookname
  UNION
  SELECT bookName, type FROM newbooks;

# union all

books

| | bookID | bookName | Type | Cost |
|---|---|---|---|---|
| ▶ | 1 | DS | Hardcover | 950 |
| | 1 | DS | Hardcover | 950 |
| | 2 | JavaScript | Paperback | 700 |
| | 2 | JavaScript | Paperback | 700 |

newbooks

| | bookID | bookName | Type | Cost |
|---|---|---|---|---|
| ▶ | 1 | Redis | Paperback | 850 |
| | 1 | Redis | Paperback | 850 |
| | 2 | JavaScript | Paperback | 700 |
| | 2 | JavaScript | Paperback | 700 |

Output

| | bookID | bookName | Type | Cost |
|---|---|---|---|---|
| ▶ | 1 | DS | Hardcover | 950 |
| | 1 | DS | Hardcover | 950 |
| | 2 | JavaScript | Paperback | 700 |
| | 2 | JavaScript | Paperback | 700 |
| | 1 | Redis | Paperback | 850 |
| | 1 | Redis | Paperback | 850 |
| | 2 | JavaScript | Paperback | 700 |
| | 2 | JavaScript | Paperback | 700 |

❖ ***r*** UNION ***s*** includes all rows appearing in both result sets ***r*** and ***s***, including duplicates.

Duplicate rows are repeated in result set

- SELECT * FROM books UNION ALL SELECT * FROM newbooks;
- SELECT * FROM emp UNION ALL SELECT * FROM emp;
- SELECT bookname, COUNT(*) FROM (SELECT bookname FROM books UNION ALL SELECT bookname FROM newbooks) *b* GROUP BY bookname;

# intersect / intersect all

It is used to combine two result sets and returns the data which are common in both the result set.

books

| | bookID | bookName | Type | Cost |
|---|---|---|---|---|
| ▶ | 1 | DS | Hardcover | 950 |
| | 1 | DS | Hardcover | 950 |
| | 2 | JavaScript | Paperback | 700 |
| | 2 | JavaScript | Paperback | 700 |

newbooks

| | bookID | bookName | Type | Cost |
|---|---|---|---|---|
| ▶ | 1 | Redis | Paperback | 850 |
| | 1 | Redis | Paperback | 850 |
| | 2 | JavaScript | Paperback | 700 |
| | 2 | JavaScript | Paperback | 700 |

Output

| | bookID | bookName | Type | Cost |
|---|---|---|---|---|
| ▶ | 2 | JavaScript | Paperback | 700 |

Output

| | bookID | bookName | Type | Cost |
|---|---|---|---|---|
| ▶ | 2 | JavaScript | Paperback | 700 |
| | 2 | JavaScript | Paperback | 700 |

❖ ***r*** INTERSECT ***s*** includes only rows appearing in both result sets ***r*** and ***s***.

- SELECT * FROM books INTERSECT SELECT * FROM newbooks;
- SELECT * FROM books INTERSECT ALL SELECT * FROM newbooks;
- SELECT bookName, type FROM books WHERE EXISTS (SELECT bookname, type FROM newbooks WHERE books.bookName = newbooks.bookName);
- SELECT bookName, type FROM books WHERE bookName IN (SELECT bookName FROM newbooks);

# except / except all

It is used to combine two result sets and returns the data from the first result set which is not present in the second result set.

books

| | bookID | bookName | Type | Cost |
|---|---|---|---|---|
| ▶ | 1 | DS | Hardcover | 950 |
| | 1 | DS | Hardcover | 950 |
| | 2 | JavaScript | Paperback | 700 |
| | 2 | JavaScript | Paperback | 700 |

newbooks

| | bookID | bookName | Type | Cost |
|---|---|---|---|---|
| ▶ | 1 | Redis | Paperback | 850 |
| | 1 | Redis | Paperback | 850 |
| | 2 | JavaScript | Paperback | 700 |
| | 2 | JavaScript | Paperback | 700 |

Output

| | bookID | bookName | Type | Cost |
|---|---|---|---|---|
| ▶ | 1 | DS | Hardcover | 950 |

Output

| | bookID | bookName | Type | Cost |
|---|---|---|---|---|
| ▶ | 1 | DS | Hardcover | 950 |
| | 1 | DS | Hardcover | 950 |

❖ ***r*** EXCEPT ***s*** returns only those rows from result set ***r*** which do not appear in ***s***.

- SELECT * FROM books EXCEPT SELECT * FROM newbooks;
- SELECT * FROM books EXCEPT ALL SELECT * FROM newbooks;

- SELECT bookName, type FROM books WHERE NOT EXISTS (SELECT bookName, type FROM newbooks WHERE books.bookName = newbooks.bookName);
- SELECT bookName, type FROM newbooks WHERE NOT EXISTS (SELECT bookName, type FROM books WHERE books.bookName = newbooks.bookName);

Note:

*except*

- There is no MINUS operator in MySQL, you can easily simulate this type of query using either the EXCEPT , NOT IN clause or the NOT EXISTS clause.

1. ```
   SELECT * FROM books          /* Fetch  everything from books */
   EXCEPT                       /* that are not present in newbooks */
   SELECT * FROM newbooks;
   ```

2. ```
   SELECT * FROM newbooks  /* Fetch everything  from newbooks */
   EXCEPT                  /* that are not present in books */
   SELECT * FROM books;
   ```

# create table … like statement

# *create table ... like*

Use CREATE TABLE ... LIKE to create an empty table based on the definition of another table, including any column attributes and indexes defined in the original table.

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

- LIKE works only for base tables, not for VIEWS.
- You cannot execute CREATE TABLE or CREATE TABLE ... LIKE while a LOCK TABLES statement is in effect.

- CREATE TABLE e LIKE emp;

# create table … select statement

# create table ... select

You can create one table from another by adding a SELECT statement at the end of the CREATE TABLE statement.

```
CREATE TABLE new_tbl(column_definition) [as] SELECT * FROM orig_tbl;
```

- `CREATE TABLE e SELECT * FROM emp;`
- `CREATE TABLE e as SELECT * FROM emp;`
- `CREATE TABLE e (a INT, b INT DEFAULT 200) SELECT * FROM emp;   //Add new column('s) with existing columns.`
- `CREATE TABLE e as SELECT 1+1, ename FROM emp;`
- `CREATE TABLE e as SELECT 1+1 "R1", ename FROM emp;`
- `CREATE TABLE e as SELECT * FROM emp WHERE 1=2;`

## Note:

- By default, this statement does not copy all column attributes such as AUTO_INCREMET, . . .

```
TRUNCATE [TABLE] tbl_name
```

# truncate table

Remember:

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

# index

- it is a schema object.
- it is used by the server to speed up the retrieval of rows.
- it reduces disk I/O(input/output) process.
- it helps to speed up select queries where clauses, but it slows down data input, with the update and the insert statements.
- it can be created or dropped with no effect on the data.
- multiple columns index may consist of up to 16 columns.

# *index*

Indexes are used to find rows with specific column values quickly. Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows. If the table has an index for the columns in question, MySQL can quickly determine the position (**from the index file**) to seek to in the middle of the data file without having to look at all the data. This is much faster than reading every row sequentially.

- To find the rows matching a WHERE clause quickly.
- If the table has a multiple-column index, any leftmost prefix of the index can be used by the optimizer to look up rows. For example, if you have a three-column index on (col1, col2, col3), you have indexed search capabilities on (col1), (col1, col2), and (col1, col2, col3).

## Note:

- Index name can have max 64 char.
- It is not possible to create an INDEX on a VIEW.
- If we drop the BASE TABLE, the INDEX will be dropped automatically.

# *create index*

Indexes are used to find rows with specific column values quickly.

```
CREATE [UNIQUE] INDEX index_name
   ON tbl_name (index_col_name, . . .)
```

e.g.

- CREATE INDEX indexOnName ON emp(ename);
- CREATE INDEX indexOnUniversity ON student_qualifications(university);
- CREATE UNIQUE INDEX uniqueIndexOnName ON emp(ename);

# SHOW INDEX Syntax

To get the index of a table, you specify the table name after the FROM keyword. The statement will return the index information associated with the table in the current database.

```
SHOW { INDEX | INDEXES | KEYS }
    { FROM | IN } tbl_name
    [{ FROM | IN } db_name]
    [WHERE expr]
```

e.g.

- SHOW INDEX FROM emp;
- SHOW INDEX FROM student_qualifications;

SHOW INDEX returns table index information.

# drop index

DROP INDEX drops the index named index_name from the table tbl_name.

DROP INDEX index_name ON tbl_name

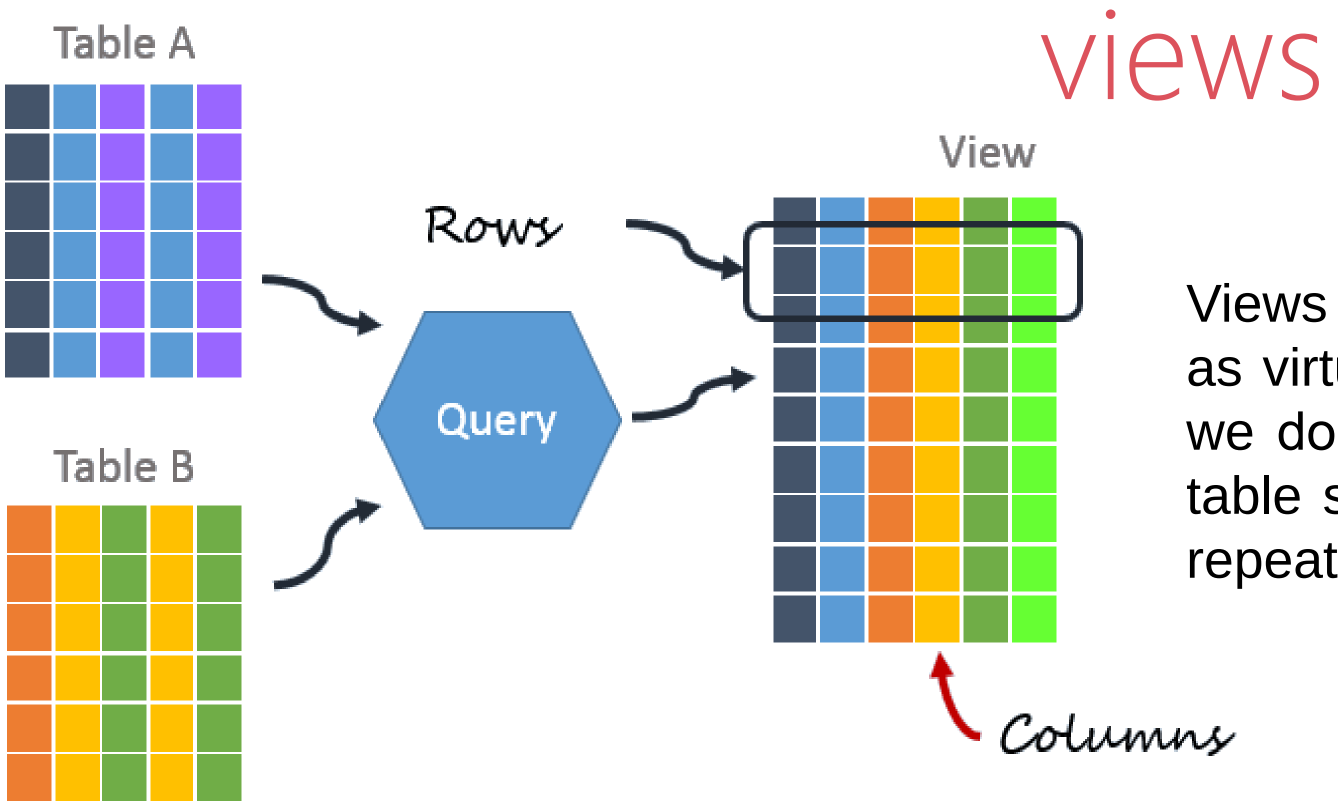
e.g.

- DROP INDEX indexOnName ON emp;
- DROP INDEX indexOnUniversity ON student_qualifications;
- DROP INDEX uniqueIndexOnName ON emp;

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

There are 2 types of Views in SQL:

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




Views are created in order to store your queries as virtual table. If there are a lot of columns and we don't want to use all columns and make the table simpler, we can create a view and reuse it repeatedly.

## Remember:

*view*

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

## Rules:

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

## Note:

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

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

Try this

# create view

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

CREATE [ OR REPLACE ]  VIEW view_name [ (*column_list*) ]
    as *select_statement*  [ WITH CHECK OPTION ]

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

- CREATE VIEW v1 as SELECT * FROM dept;
- CREATE VIEW v1($A_1$, $A_2$) as SELECT deptno, dname FROM dept;
- CREATE or REPLACE VIEW v1 as SELECT * FROM dept WITH CHECK OPTION;

desc INFORMATION_SCHEMA.VIEWS;

# create user and grant/revoke privileges

# *create user*

TODO

CREATE USER [IF NOT EXISTS] '*user_name*' @'*localhost*' IDENTIFIED BY '*password*'

SET PASSWORD FOR '*user_name*'@'*localhost*' = '*auth_string*'

DROP USER [IF EXISTS] '*user_name*' @'*localhost*'

- CREATE USER 'saleel'@'localhost' IDENTIFIED BY 'saleel';
- SET PASSWORD FOR 'saleel'@'localhost' = '*sharmin*';
- DROP USER 'saleel'@'localhost';

# *grant/revoke privileges*

TODO

GRANT *priv_type* ON *object_type* TO '*user_name*'@'*localhost*'

REVOKE *priv_type* ON *object_type* FROM '*user_name*'@'*localhost*'

| Privilege | Privilege |
|---|---|
| ALL [PRIVILEGES] | SELECT |
| CREATE | INSERT |
| ALTER | UPDATE |
| DROP | DELETE |
| EXECUTE | |

- GRANT ALL PRIVILEGES ON db1.* TO 'saleel'@'localhost';
- GRANT ALL PRIVILEGES ON *.* TO 'saleel'@'localhost';

- GRANT INSERT, UPDATE ON emp TO 'saleel';
- REVOKE INSERT, UPDATE ON emp FROM 'saleel';

# select ... into

# *select … into*

An INTO clause should not be used in a nested SELECT because such a SELECT must return its result to the outer context.

The SELECT ... INTO form of SELECT enables a query result to be stored in variables or written to a file.

SELECT . . . INTO *var_list*

SELECT . . . INTO OUTFILE

SELECT . . . INTO DUMPFILE

- Selects column values and stores them into variables.
- Writes the selected rows to a file. Column and line terminators can be specified to produce a specific output format.
- Writes a single row to a file without any formatting.

select … into var_list

# *select … into var_list*

The SELECT ... INTO form of SELECT enables a query result to be stored in variables or written to a file.

SELECT . . . INTO var_list

- The selected values are assigned to the variables.
- The number of variables must match the number of columns.
- The query should return a single row.
- If the query returns no rows, a warning with error code 1329 occurs (No data), and the variable values remain unchanged.
- If the query returns multiple rows, error 1172 occurs (Result consisted of more than one row).
- The statement may retrieve multiple rows, you can use LIMIT 1 to limit the result set to a single row.

# select … into outfile

If not working then do changes in ***my.ini*** file.

```
secure_file_priv = ""
SHOW VARIABLES LIKE "secure_file_priv";
```

# select … into outfile

Writes the selected rows to a file. Column and line terminators can be specified to produce a specific output format.

SELECT . . . INTO OUTFILE

- SELECT identifier.* FROM emp INTO OUTFILE "/tmp/emp.csv" ;
- SELECT identifier.* FROM emp INTO OUTFILE "/tmp/emp.txt" ;
- SELECT identifier.* FROM emp INTO OUTFILE "/tmp/emp.csv" FIELDS TERMINATED BY ',';
- SELECT identifier.* FROM emp INTO OUTFILE "/tmp/emp.csv" FIELDS TERMINATED BY ',';
- SELECT identifier.* FROM emp INTO OUTFILE "/tmp/emp.csv" FIELDS TERMINATED BY ',' LINES TERMINATED by '\n';
- SELECT identifier.* FROM emp INTO OUTFILE "/tmp/emp.csv" FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"' LINES TERMINATED BY '\n';
- SELECT  "DEPT", "DNAME", "LOC", "PWD", "STARTEDON" UNION SELECT * FROM dept;

---

- TABLE emp INTO OUTFILE "/tmp/emp.csv";
- TABLE emp INTO OUTFILE "/tmp/emp.txt";

select ... into dumpfile

# *select … into dumpfile*

If you use INTO DUMPFILE instead of INTO OUTFILE, MySQL writes only one row into the file, without any column or line termination and without performing any escape processing. This is useful if you want to store a BLOB value in a file.

SELECT . . . INTO DUMPFILE

- SELECT identifier.* FROM emp WHERE empno = 7788 INTO DUMPFILE "/tmp/emp.txt";

```
CountryName,CapitalName,CapitalLatitude,CapitalLongitude,CountryCode,ContinentName,remark
Somaliland,Hargeisa,9.55,44.05,NULL,Africa,
South Georgia and South Sandwich Islands,King Edward Point,-54.283333,-36.5,GS,Antarctica,
French Southern and Antarctic Lands,Port-aux-Français,-49.35,70.216667,TF,Antarctica,
Palestine,Jerusalem,31.76666667,35.233333,PS,Asia,
Aland Islands,Mariehamn,60.116667,19.9,AX,Europe,
Nauru,Yaren,-0.5477,166.920867,NR,Australia,
Saint Martin,Marigot,18.0731,-63.0822,MF,North America,
Tokelau,Atafu,-9.166667,-171.833333,TK,Australia,
Western Sahara,El-Aaiún,27.153611,-13.203333,EH,Africa,
```

```
load data infile "D:\abc1.csv" into table a fields terminated by ","
ignore 1 rows (id,name,@dt) SET dt =
STR_TO_DATE(@dt,'%d-%m-%Y');
```

# import .csv / .tsv file

```
CREATE TABLE countries (
CountryName VARCHAR(45),
CapitalName VARCHAR(45),
CapitalLatitude VARCHAR(45),
CapitalLongitude VARCHAR(45),
CountryCode VARCHAR(45),
ContinentName VARCHAR(45),
remark VARCHAR(45)
);
```

```
LOAD DATA INFILE 'path/file-name.csv' INTO TABLE countries
FIELDS TERMINATED BY ','  or FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

- ```
  SELECT  "DEPT", "DNAME", "LOC", "PWD", "STARTEDON" UNION
  SELECT * FROM dept;
  ```
  ```