A day without new knowledge is a lost day.

Database Technologies - MySQL

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

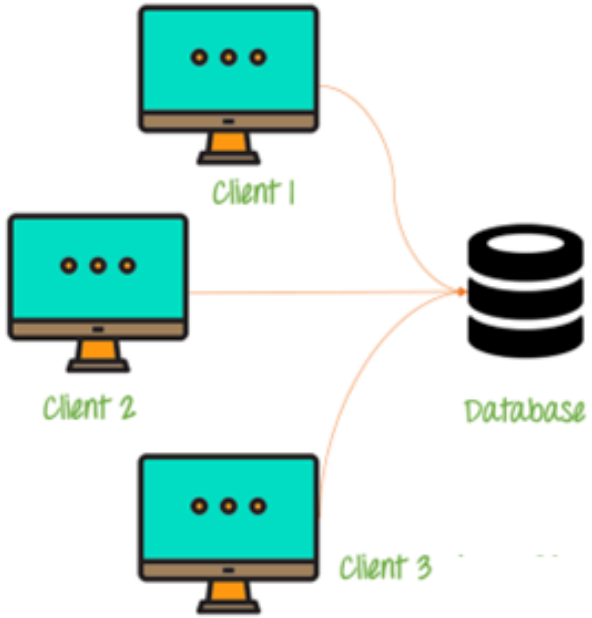
In this module we are going to learn **SQL**, **PL/SQL** and **NoSQL(MongoDB)**

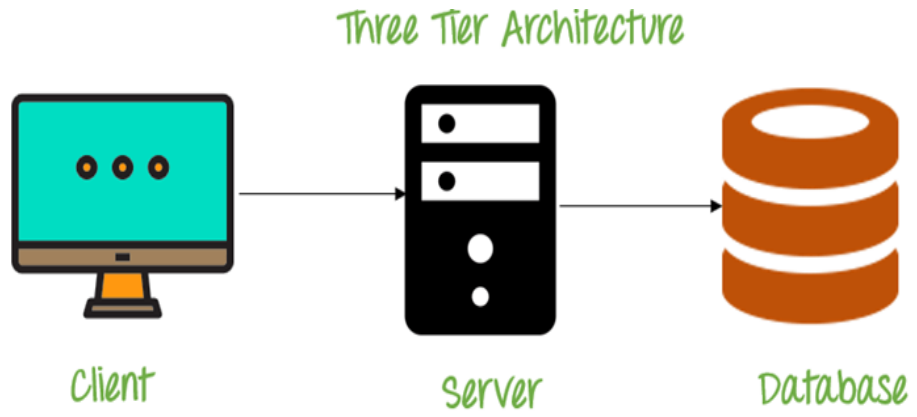
Introduction

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

They are . . . . .

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





Introduction

- A layer refers to pieces of software that are logically separated, but typically live within the same process and machine.

- A tier, instead, refers to pieces of software that live in distinct processes or AppDomains or machines.

- A tier refers to physical separation; a layer is about logical separation.

Introduction

Why do we need databases (Use Case)?

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

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

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

Introduction

4 Important Roles of Database in Industry.

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

What is Relation and Relationship?

Remember:

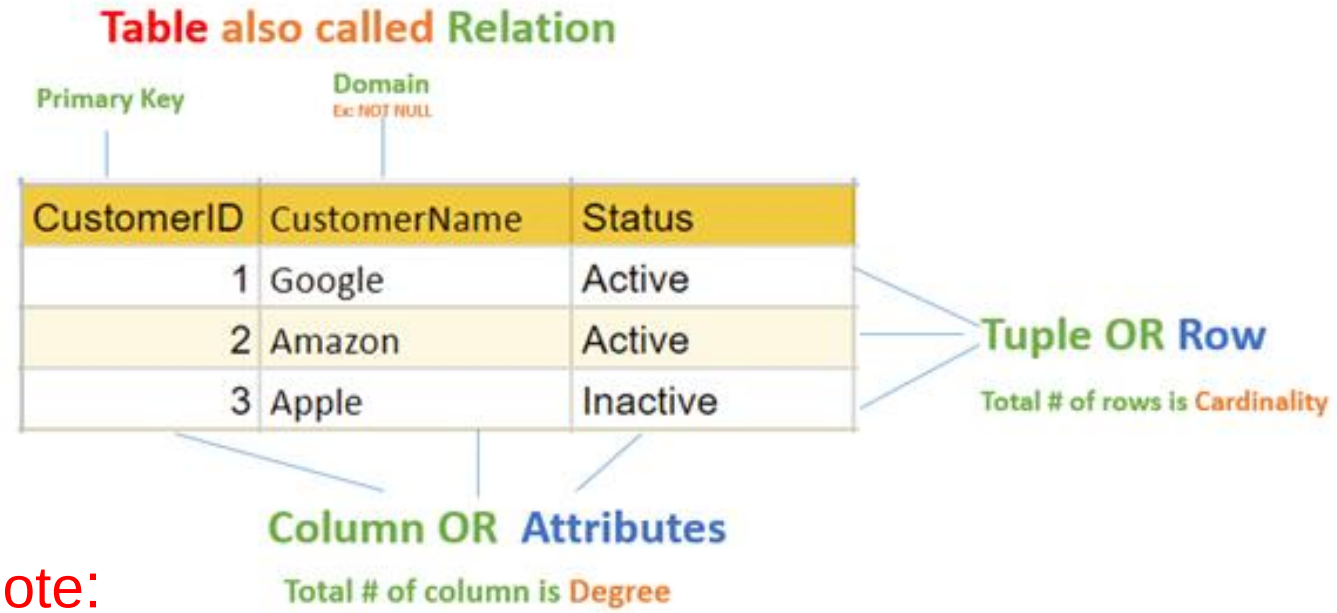
- A ***reference*** is a relationship between two tables where the values in one table refer to the values in another table.
- A ***referential key*** is a column or set of columns in a table that refers to the primary key of another table. It establishes a relationship between two tables, where one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

relation and relationship?

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

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

Primary/Foreign key is used to specify this relationship.



Remember:

Foreign Key is also know as

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

Note:

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

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

File System
vs
DBMS

file-oriented system

File Anomalies

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

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

c:\employee.txt

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

c:\employee.txt

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

c:\employee.txt

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

c:\employee.txt

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

c:\employee.txt

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

file-oriented system

File Anomalies

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

c:\employee.txt

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

file attributes

- File Name
- Type
- Location

file permissions

- File permissions
- Share permissions

search empl ID=1

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

search emp_name

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

file-oriented system

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

1 rajan MG Road Pune MH 34500
2 rahul patil SSG Lane Pune MH 54000
3 suraj raj k Deccan Gymkhana Pune MH 22000

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

1,raj,k,1984-06-12,raj.kumar@gmail.com
2,om,,1969-10-25,om123@gmail.com
3,rajes,kumar,1970-10-25,
4,rahul,patil,1982-10-31,rahul.patil@gmail.com
5,ketan,,,ruhan.bagde@gmail.com

The Zen of Python,

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

advantages & disadvantage of file-oriented system

advantages of file-oriented system

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

Advantage of File-oriented system

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

disadvantage of file-oriented system

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

Disadvantage of File-oriented system

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

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

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

DBMS

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

Difference between File System and DBMS

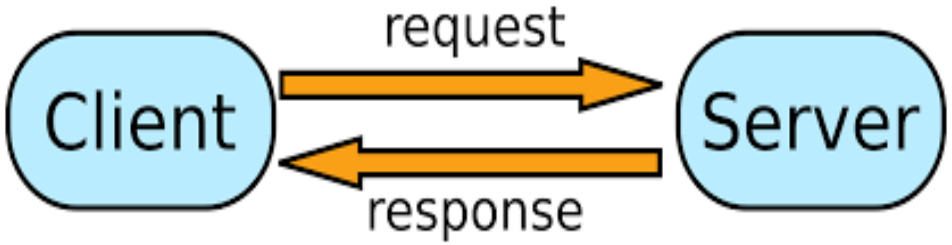
File Management System	Database Management System
• File System is easy-to-use system to store data which require less security and constraints.	• Database Management System is used when security constraints are high.
• Data Redundancy is more in File System.	• Data Redundancy is less in Database Management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

relational database management system?

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

RDBMS supports

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



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

object relational database management system?

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

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

Advantage of ORDBMS

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

difference between dbms and rdbms

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

Codd's Rules

Rule 1: Information Rule

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

Rule 2: Guaranteed Access Rule

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

Rule 3: Systematic Treatment of NULL Values

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

Rule 4: Active Online Catalog

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

Rule 5: Comprehensive Data Sub-Language Rule

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

Rule 6: View Updating Rule

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

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

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

Rule 8: Physical Data Independence

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

Rule 9: Logical Data Independence

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

Rule 10: Integrity Independence

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

Rule 11: Distribution Independence

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

Rule 12: Non-Subversion Rule

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

relational model concepts
and
properties of relational table

relational model concepts

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

- **Tables** – In relational model, relations are saved in the form of Tables. A table has rows and columns.
- **Attribute** – Attributes are the properties that define a relation. **e.g.** (**roll_no, name, address, phone and age)**
- **Tuple** – A single row of a table, which contains a single record for that relation is called a tuple.
- **Relation schema** – A relation schema describes the relation name (table name) with its attribute (column) names.

  e.g. student(prn, name, address, phone, DoB, age, hobby, email, status) is relation schema for student relation.
- **Attribute domain** – An attribute domain specifies the data type, format, constraints of a column, and defines the range of values that are valid for that column.

Remember:

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

properties of relational table

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

Relational tables have six properties:

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

What is data?

what is data?

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

Data can be in the form of Text or Multimedia

e.g.

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

Remember:

- A **Binary Large Object (BLOB)** is a MySQL data type that can store binary data such as multimedia, and PDF files.
- A **Character Large Object(CLOB)** is 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, department, …)
- Place(classroom, building, …) --a particular position or area
- Thing(computer, lab equipment, …) --an object that is not named
- Concept(course, batch, student's attendance, …) -- an idea,

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

Every entity has its own characteristics.

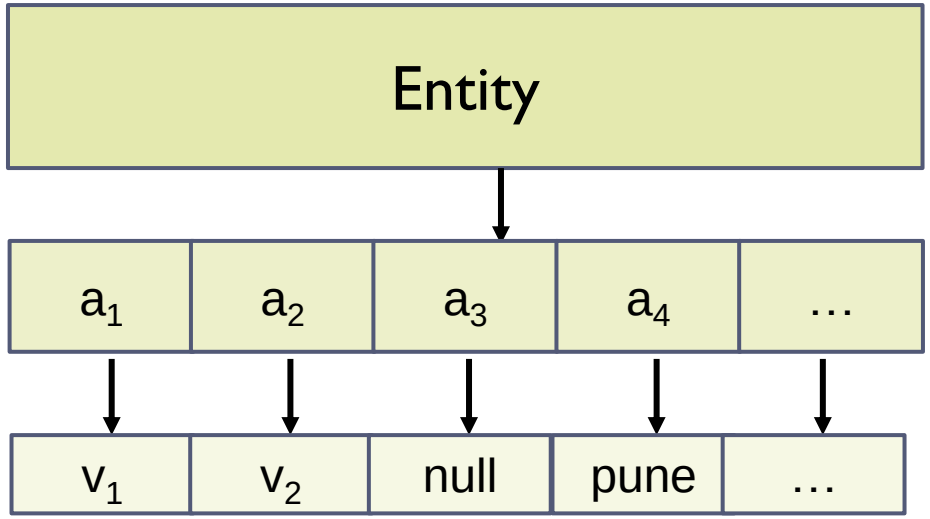
When you are designing attributes for your entities, **you will sometimes find that an attribute does not have a value**. For example, you might want an attribute for a person's middle name, but you can't require a value because some people have no middle name. **For these, you can define the attribute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

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

What is an Attribute?

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

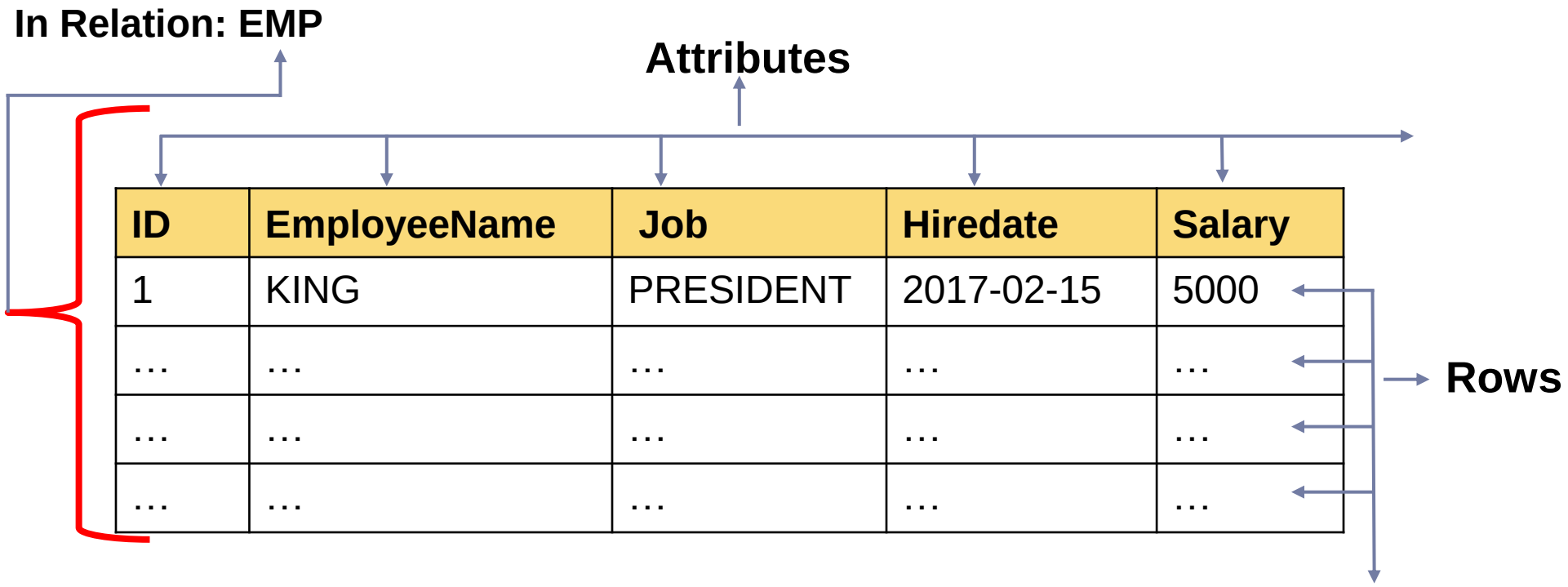
In some cases, you might not want a specific attribute to contain a null value, but you don't want to require that the user or program always provide a value. In this case, a default value might be appropriate. **A default value is a value that applies to an attribute if no other valid value is available.**



attributes

A table has rows and columns

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



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

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

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

Remember:

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

attributes

- **Simple / Atomic Attribute** --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 has only single value 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.

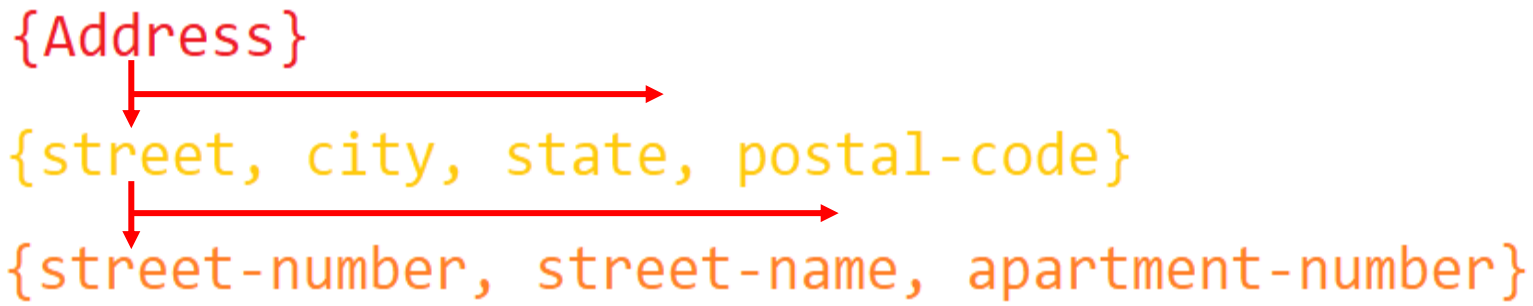
Composite **VS** Multi Valued Attribute

composite / multi valued attributes

Composite Attribute

Person Entity

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



Multi Valued Attribute

Person Entity

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

What is an Prime, Non-Prime Attribute?

Prime attribute (*Entity integrity*)

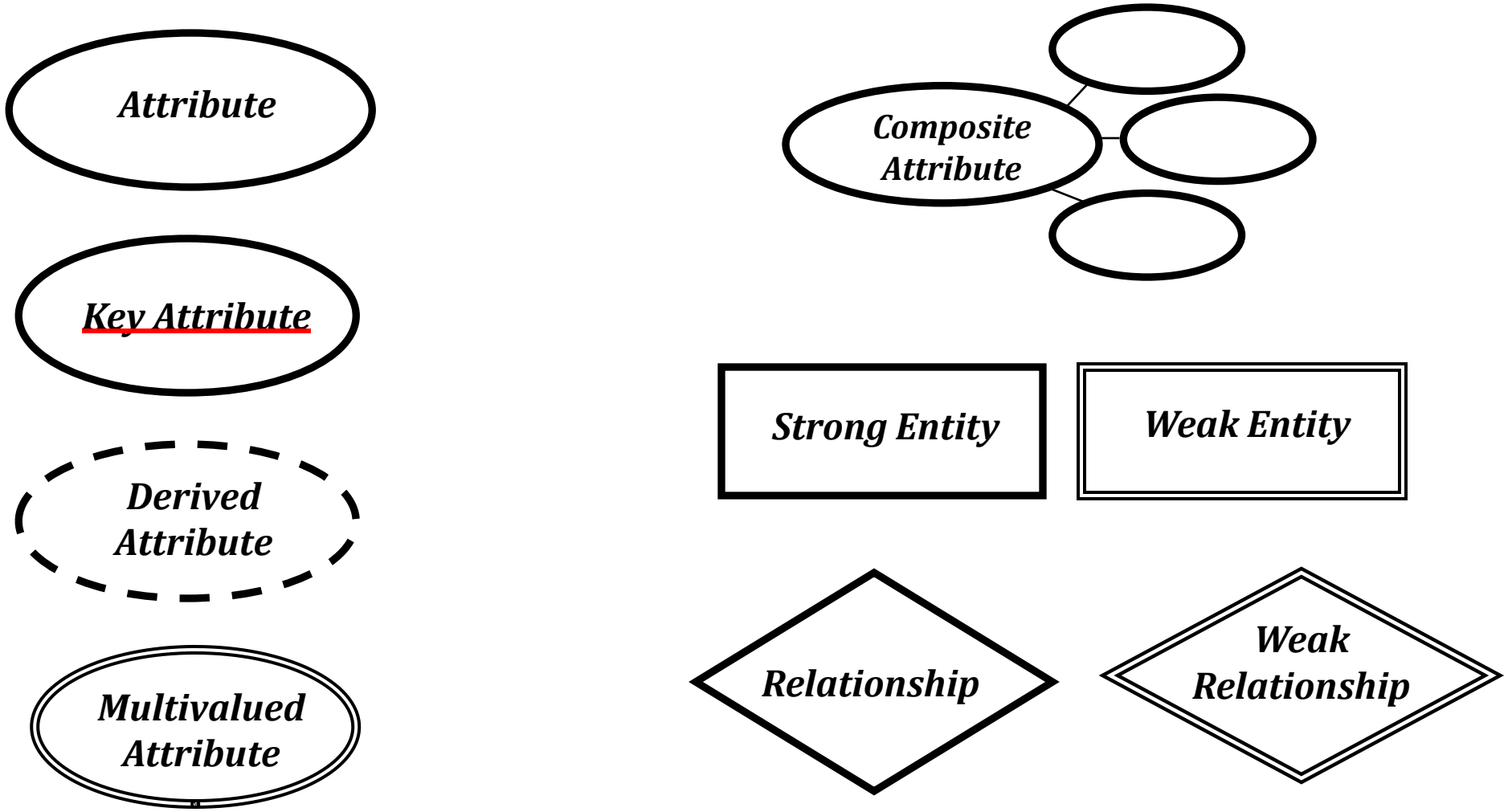
An attribute, which is a **part of the prime-key** **(**candidate key**)**, is known as a prime attribute.

Non-prime attribute

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

Entity Relationship Diagram Symbols

entity relationship diagram symbols



strong and weak entity

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

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

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

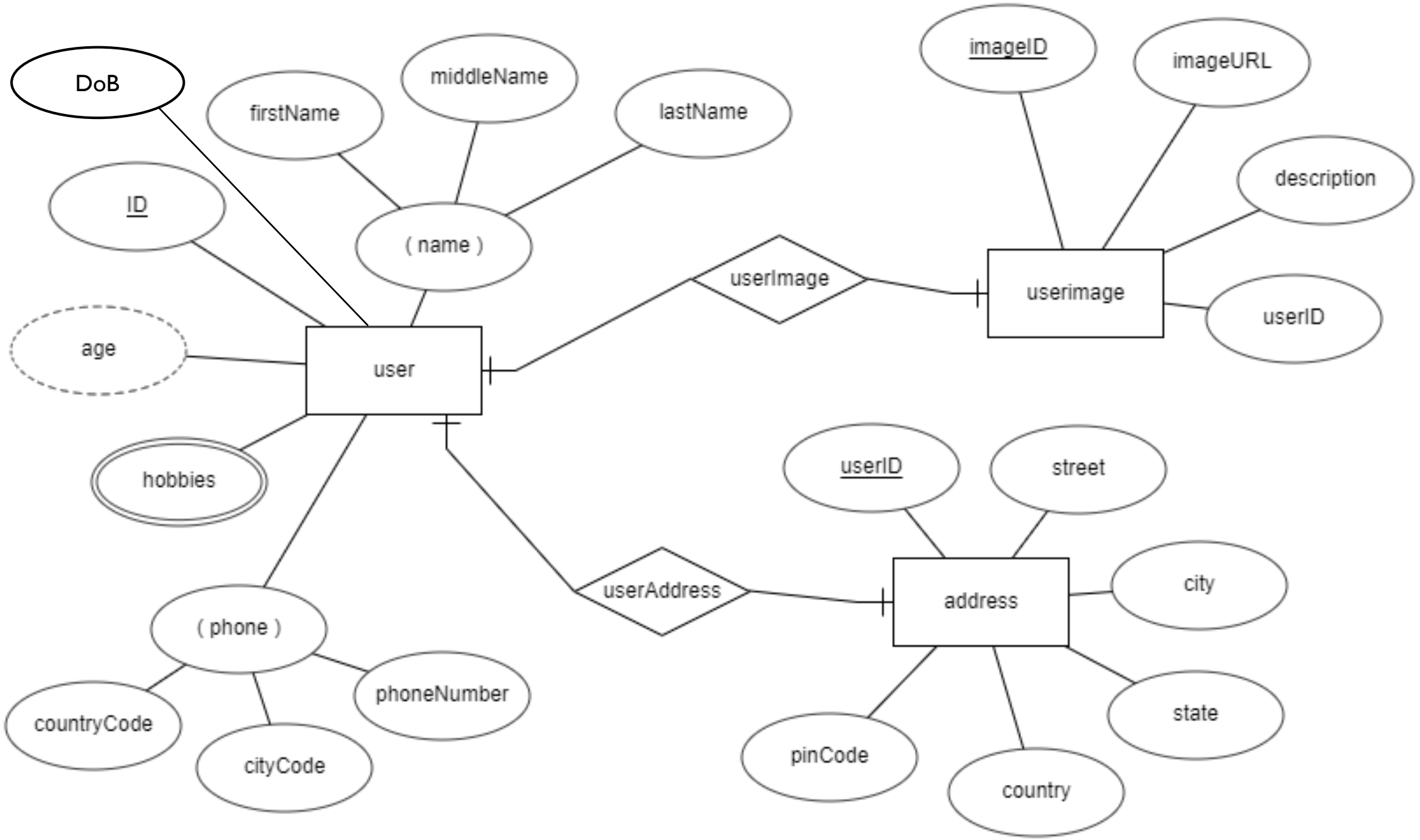
Example 1 – A loan entity can not be created for a customer if the customer doesn't exist

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

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

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

entity relationship diagram



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

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

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

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

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

- **Domain**: A domain is a set of values that can be stored in a column of a database table. A domain is usually defined by a column's data type, which determines the kind of values that can be stored in the column. Total range of accepted values for an attribute of the relation **is called the domain of the attribute**. (**Data Type(size)**)

- **Union Compatibility**: Two relations ***R*** and ***S*** are set to be Union Compatible to each other if and only if:
  1. They have the **same degree $\mathbf{d(R)}$**.
  2. Domains of the respective attributes should also be same.

What is domain constraint and types of data integrity constraints?

Data integrity refers to the correctness and completeness of data.

A domain constraint and types of data integrity constraints

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

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

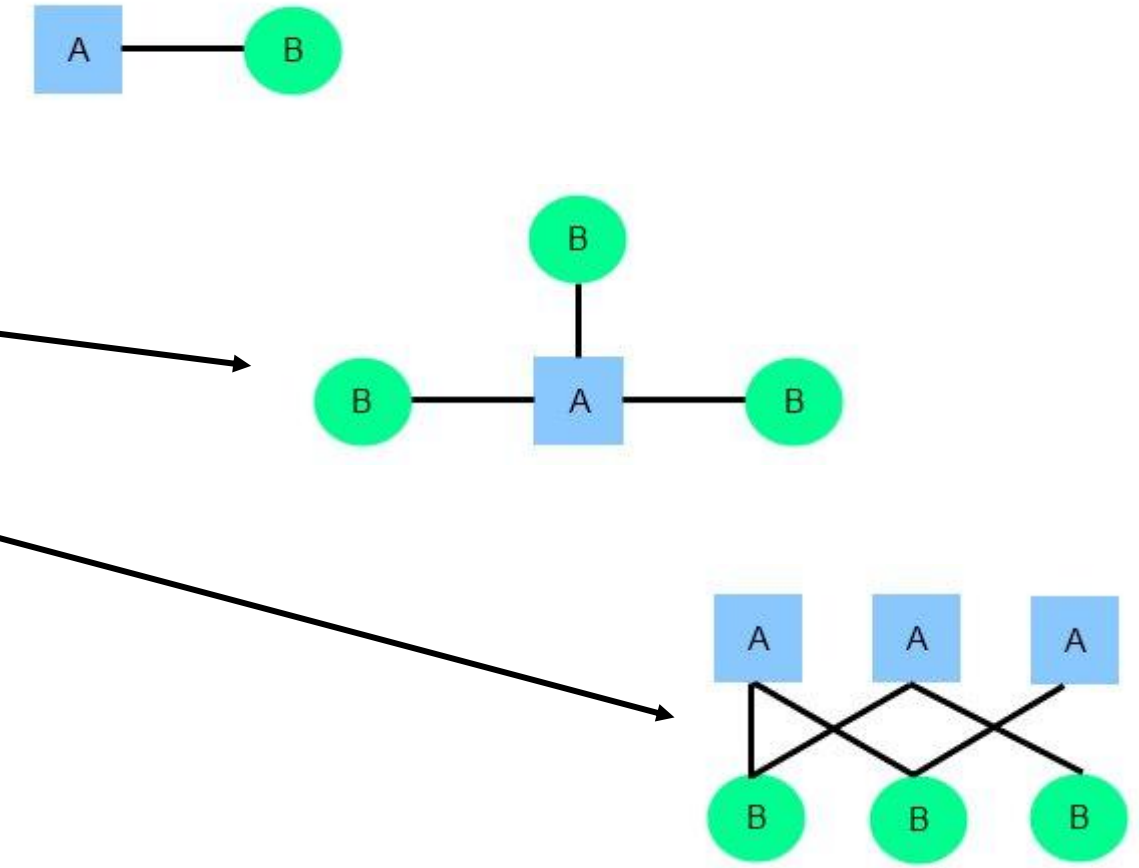
- **Entity integrity:** Entity Integrity Constraint is used to ensure the uniqueness of each record the table. There are primarily two types of integrity constraints that help us in ensuring the uniqueness of each row, namely, UNIQUE constraint and PRIMARY KEY constraint.
-
- **Referential integrity:**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

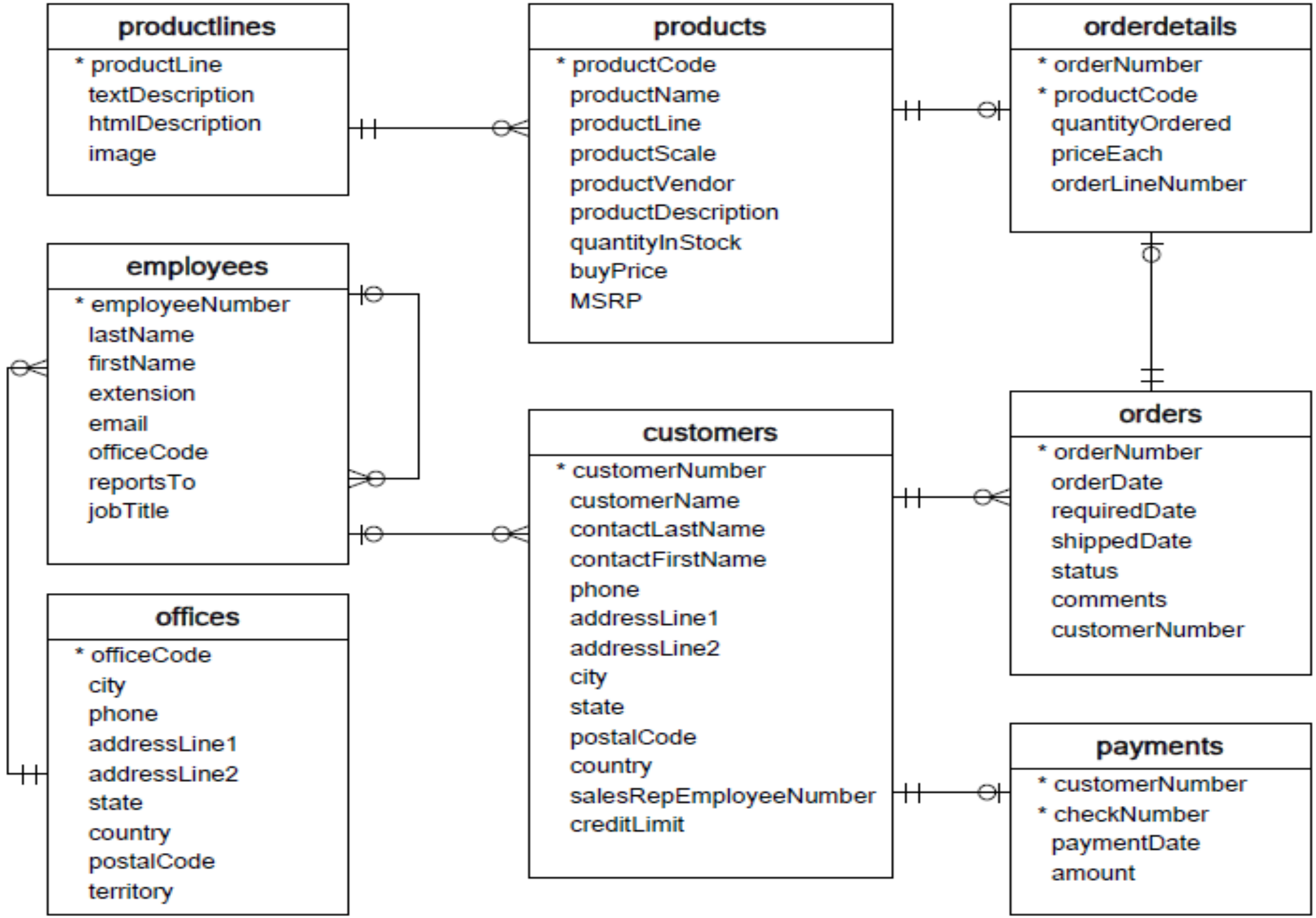
- **Domain integrity:** A domain is a set of values of the same type. For example, we can specify if a particular column can hold null values or not, if the values have to be unique or not, the data type or size of values that can be entered in the column, the default values for the column, etc..

Common relationships

Common relationship

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



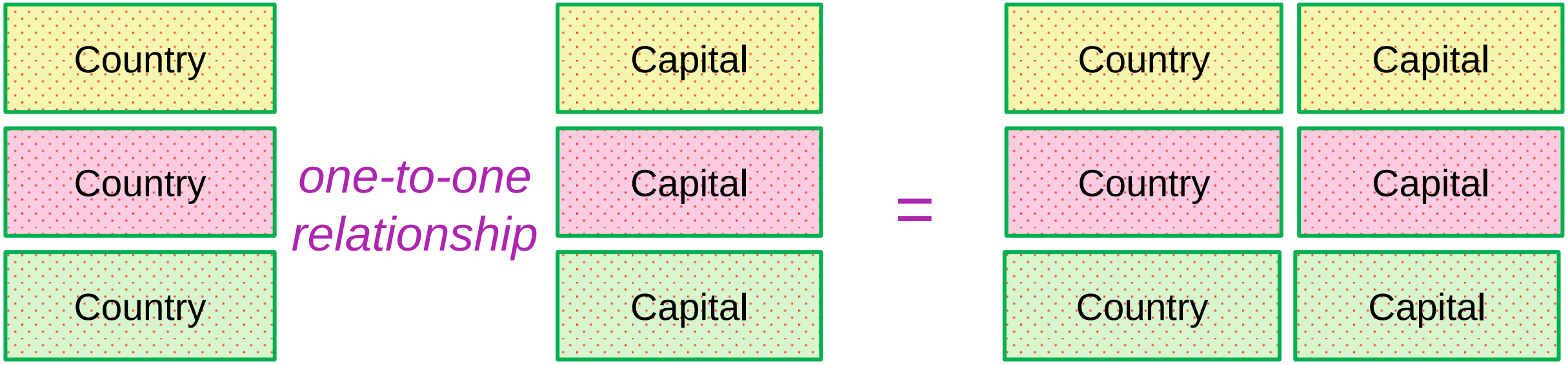

productlines
* productLine
textDescription
htmlDescription
image
products
* productCode
productName
productLine
productScale
productVendor
productDescription
quantityInStock
buyPrice
MSRP
orderdetails
* orderNumber
* productCode
quantityOrdered
priceEach
orderLineNumber
employees
* employeeNumber
lastName
firstName
extension
email
officeCode
reportsTo
jobTitle
customers
* customerNumber
customerName
contactLastName
contactFirstName
phone
addressLine1
addressLine2
city
state
postalCode
country
salesRepEmployeeNumber
creditLimit
orders
* orderNumber
orderDate
requiredDate
shippedDate
status
comments
customerNumber
offices
* officeCode
city
phone
addressLine1
addressLine2
state
country
postalCode
territory
payments
* customerNumber
* checkNumber
paymentDate
amount

one-to-one relationship

one-to-one relationship

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

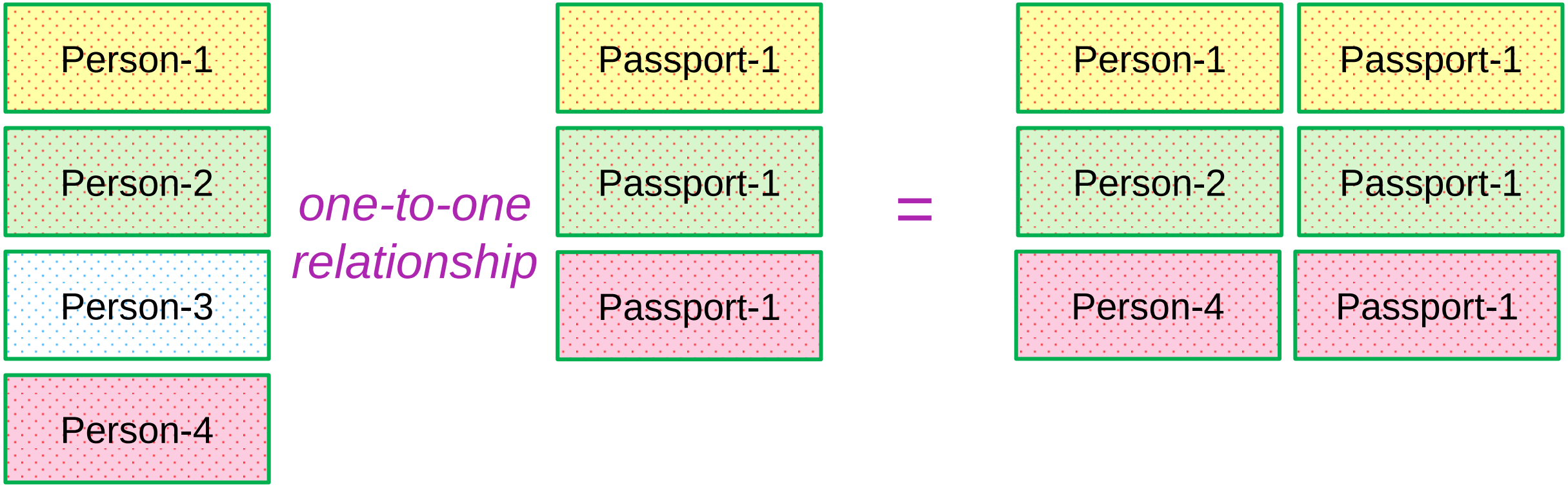
A *one-to-one* relationship is a type of cardinality that refers to the relationship between two entities ***R*** and ***S*** in which one element of entity ***R*** may only be linked to zero/one element of entity ***S***, and vice versa.



one-to-one relationship

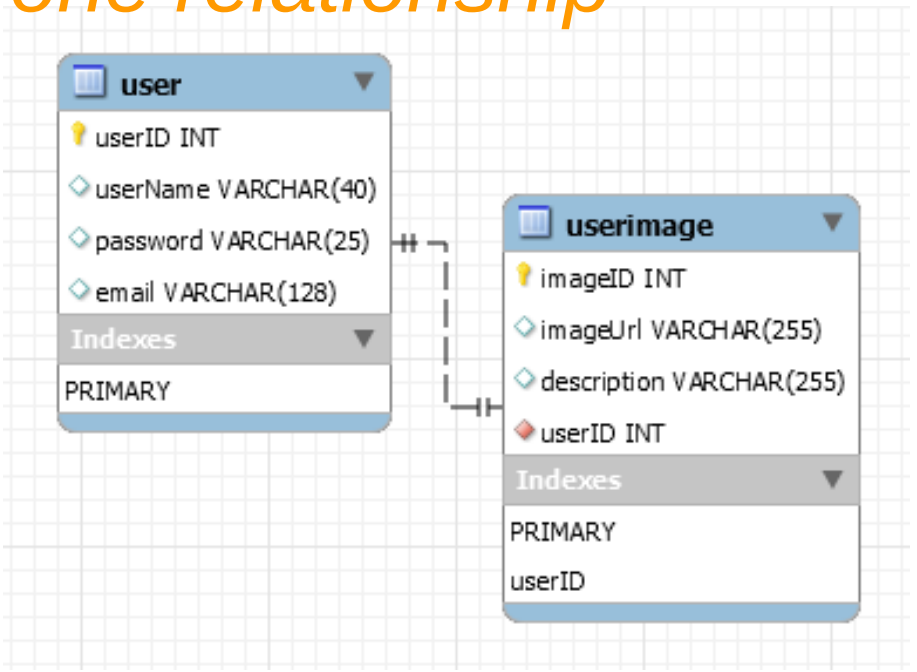
A *one-to-one* relationship between two tables means that a row in one table can only relate to zero/one row in the table on the other side of their relationship. This is the least common database relationship.

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



how to create one-to-one relationship

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

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



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

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

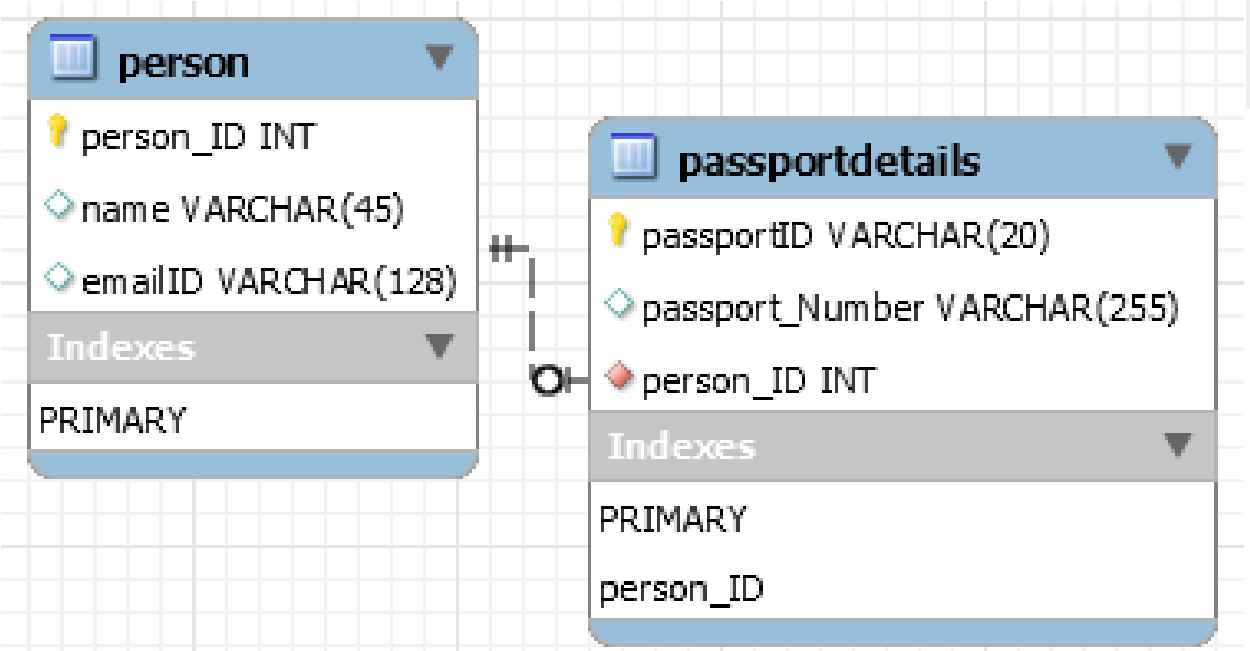
how to create one-to-one relationship

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

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

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

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

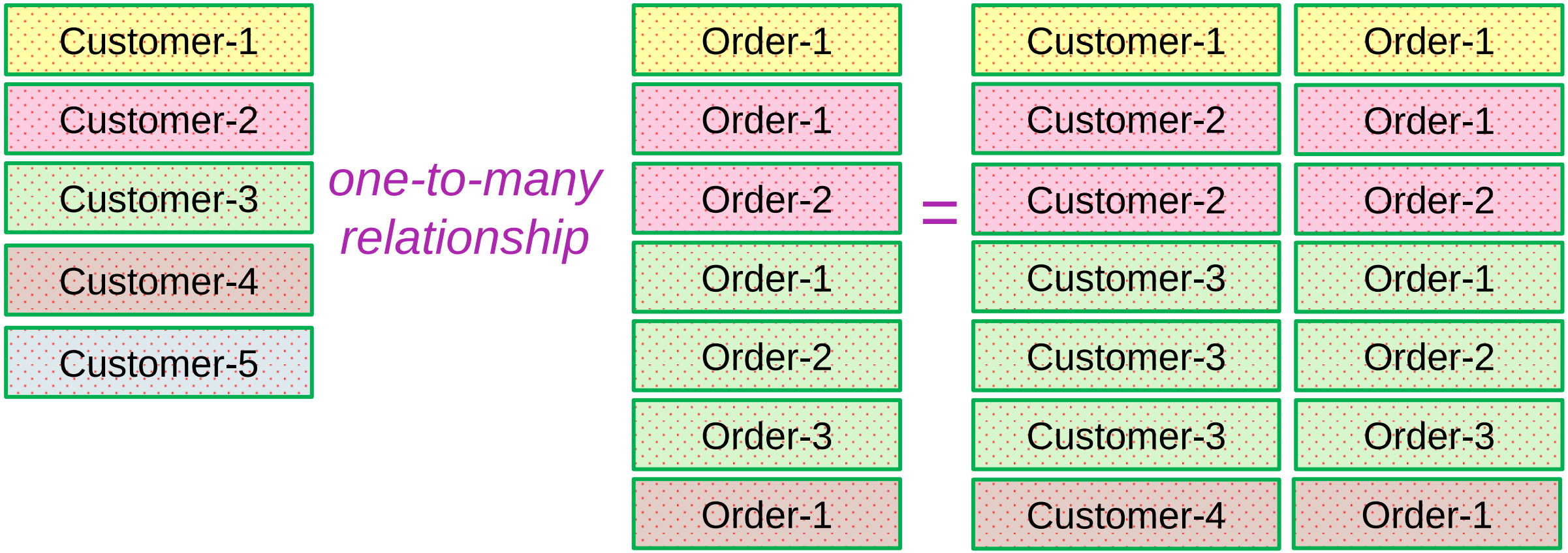


one-to-many relationship

one-to-many relationship

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

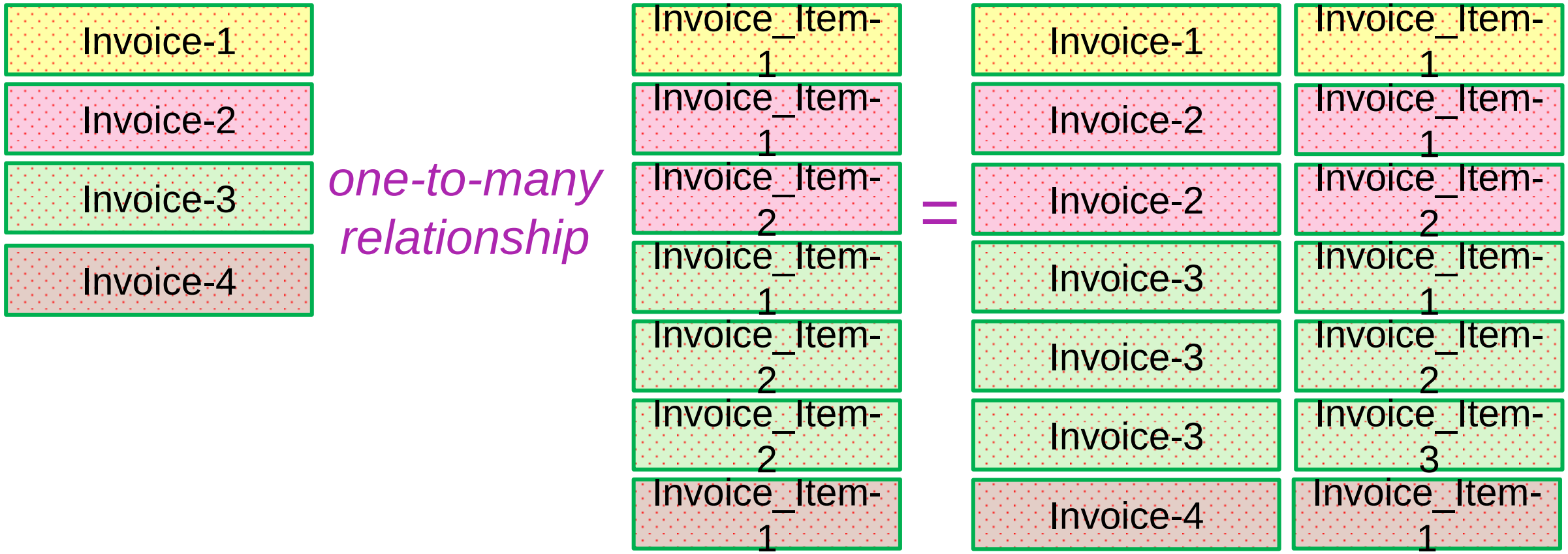
a *one-to-many* relationship is a type of cardinality that refers to the relationship between two entities ***R*** and ***S*** in which an element of ***R*** may be linked to many elements of ***S***, but a member of ***S*** is linked to only one element of ***R***.



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.

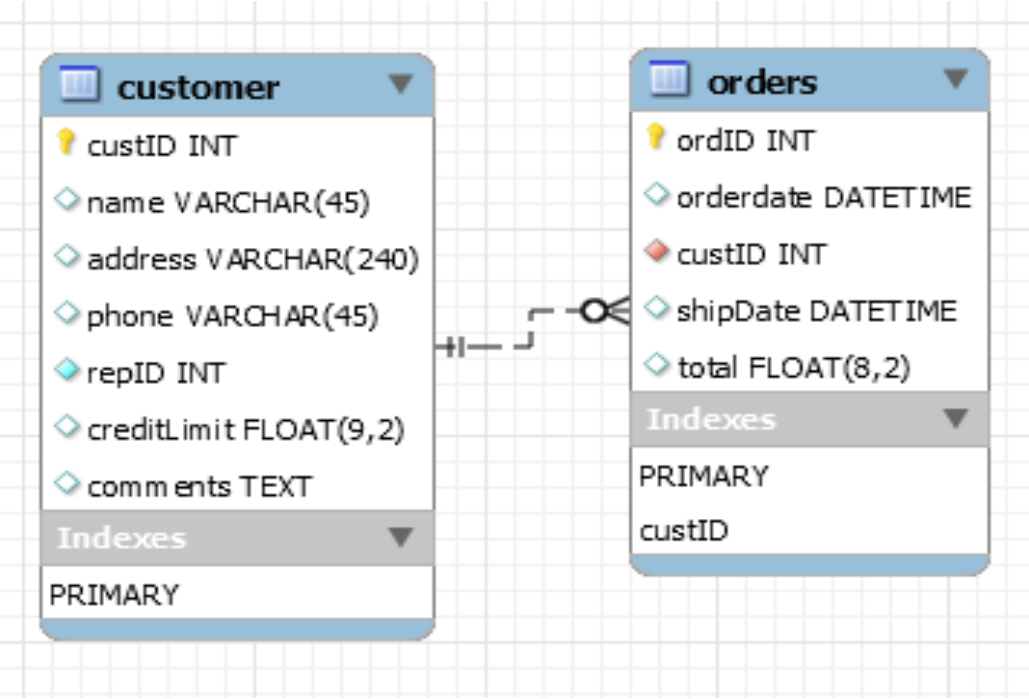
a *one-to-many* relationship is a type of cardinality that refers to the relationship between two entities ***R*** and ***S*** in which an element of ***R*** may be linked to many elements of ***S***, but a member of ***S*** is linked to only one element of ***R***.



how to create one-to-many relationship

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

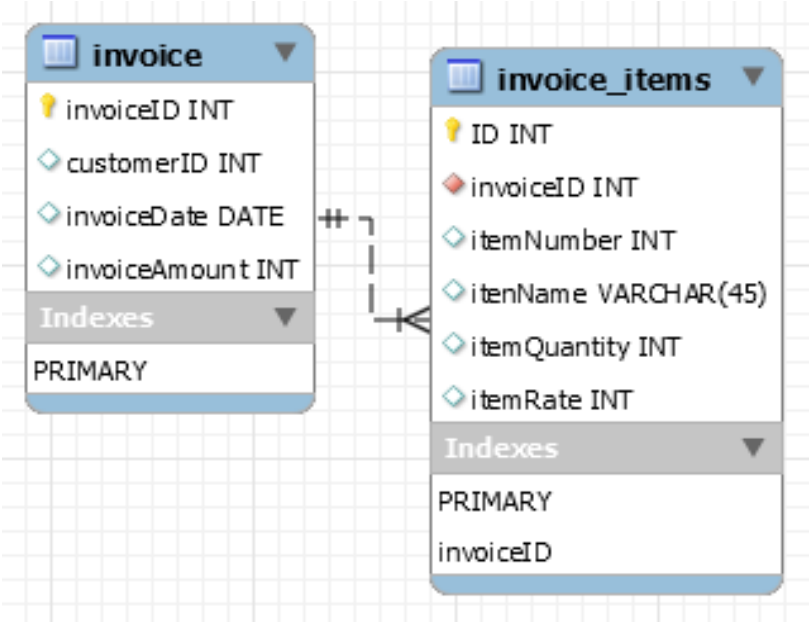
```
CREATE TABLE orders (
    ordID INT PRIMARY KEY,
    orderdate DATETIME,
    custID INT,
    shipDate DATETIME,
    total FLOAT(8,2),
    FOREIGN KEY(custID) REFERENCES customer(custID),
    constraint total_greater_zero CHECK(total >= 0)
);
```



how to create one-to-many relationship

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

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



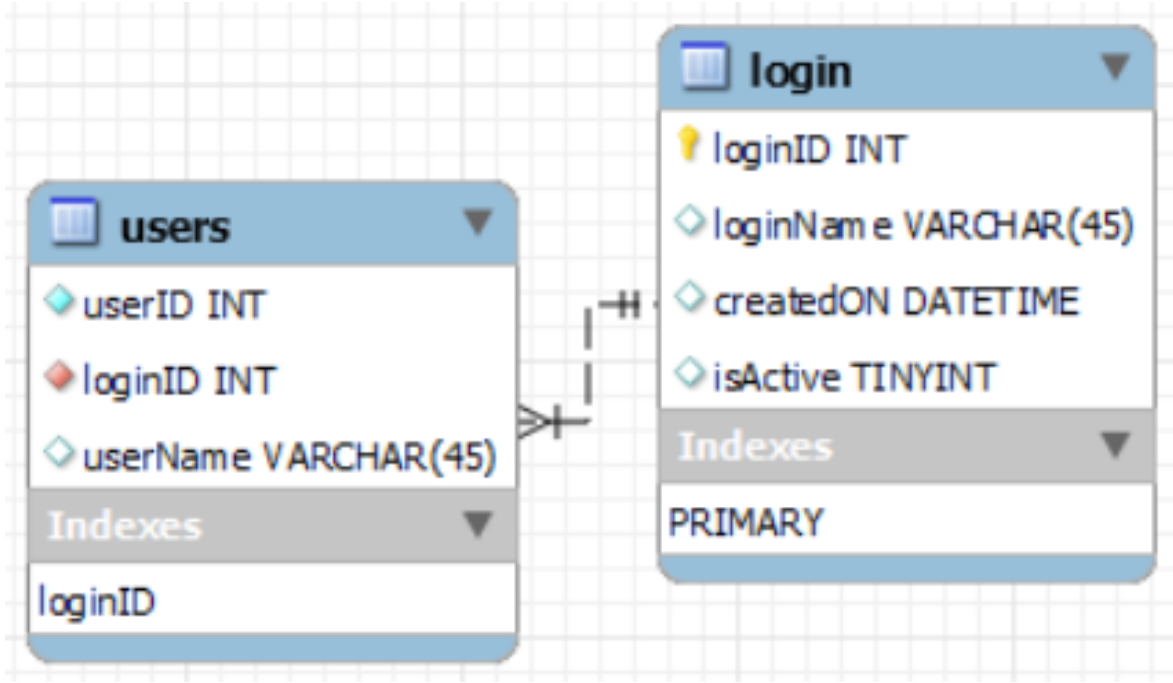

```
CREATE TABLE invoice_items (
   invoiceID INT,
   itemID INT,
   itenName VARCHAR(45),
   itemQuantity INT,
   itemRate INT,
   PRIMARY KEY(invoiceID, itemID),
   FOREIGN KEY(invoiceID) REFERENCES invoice(invoiceID)
);
```

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

many-to-one relationship

many-to-one relationship

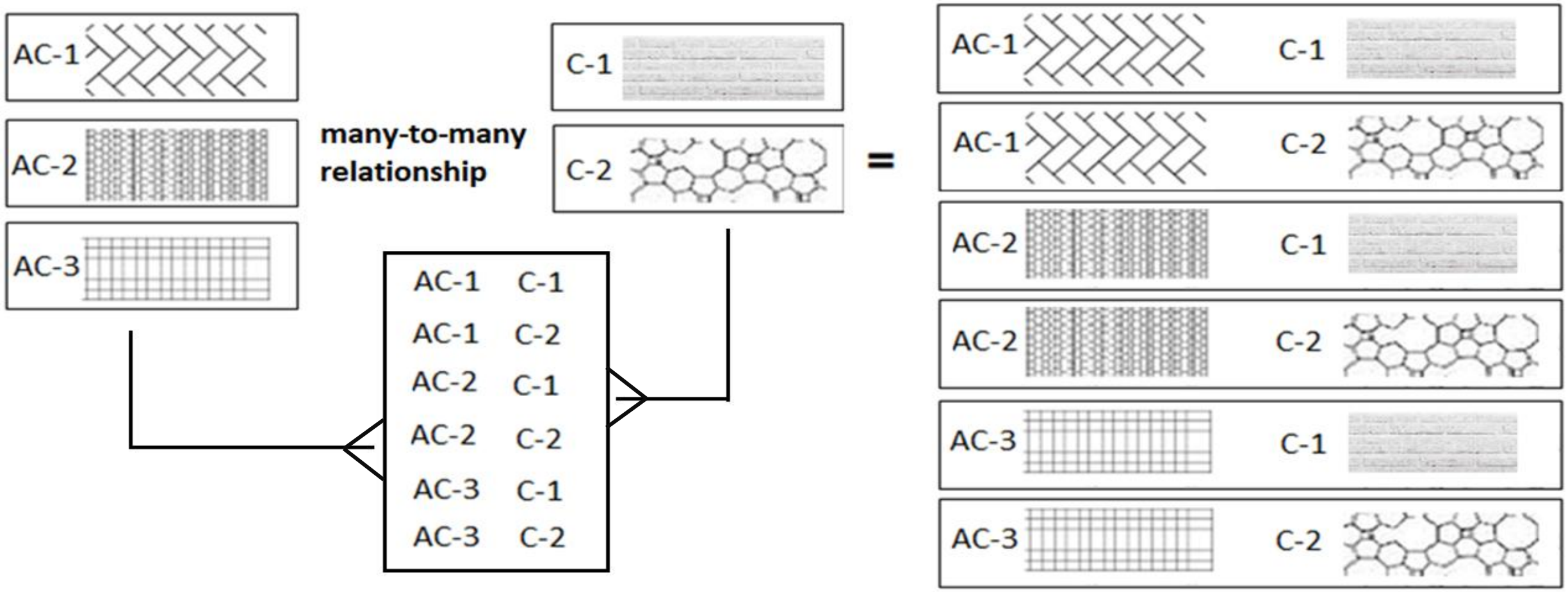
```
CREATE TABLE users (
   userID INT,
   loginID INT,
   userName VARCHAR(45),
   PRIMARY KEY(loginID, userID),
   constraint fk_users_login_loginID1 FOREIGN KEY(loginID)
   REFERENCES login(loginID)
);

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

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



many-to-many relationship

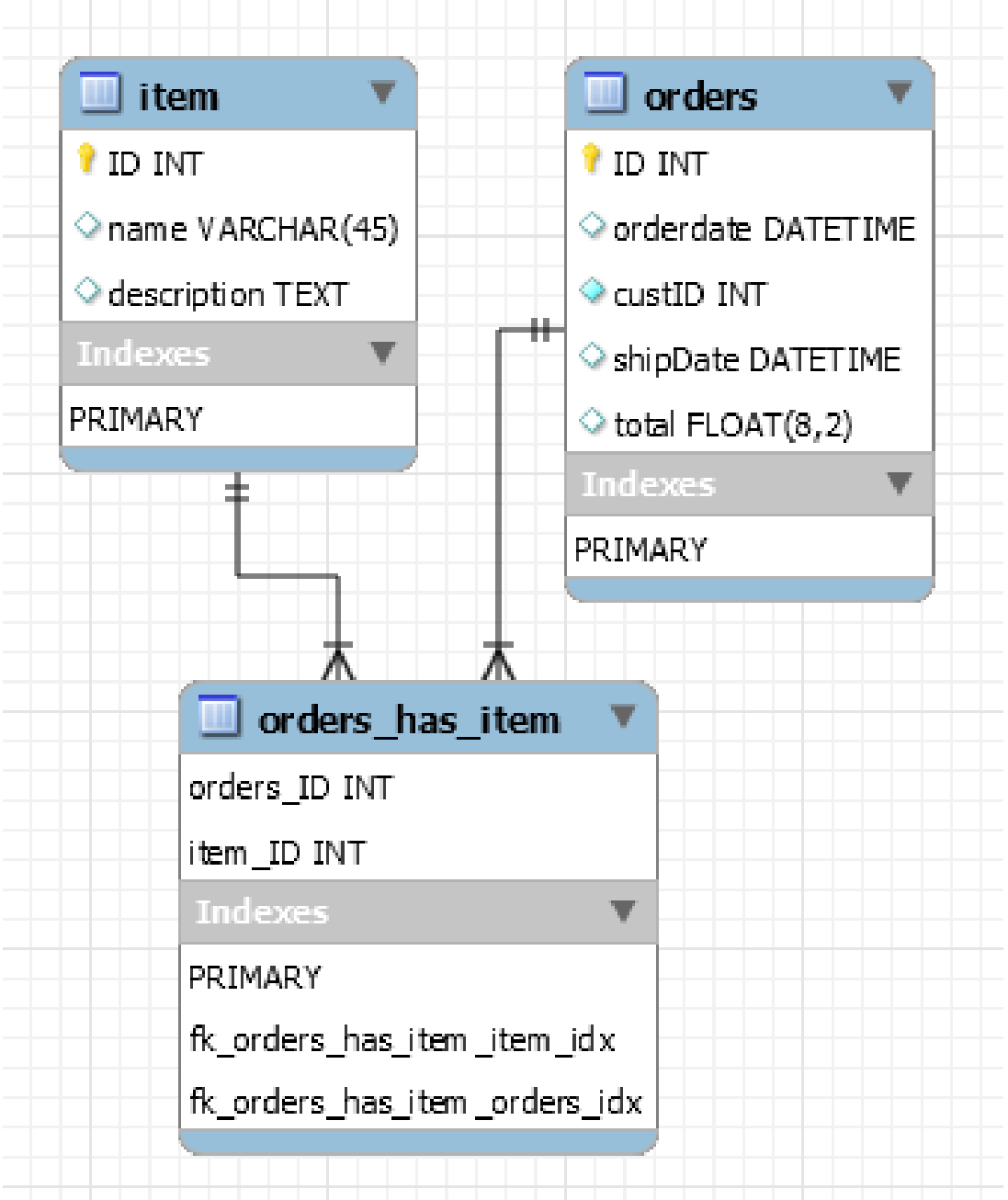
many-to-many relationship

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



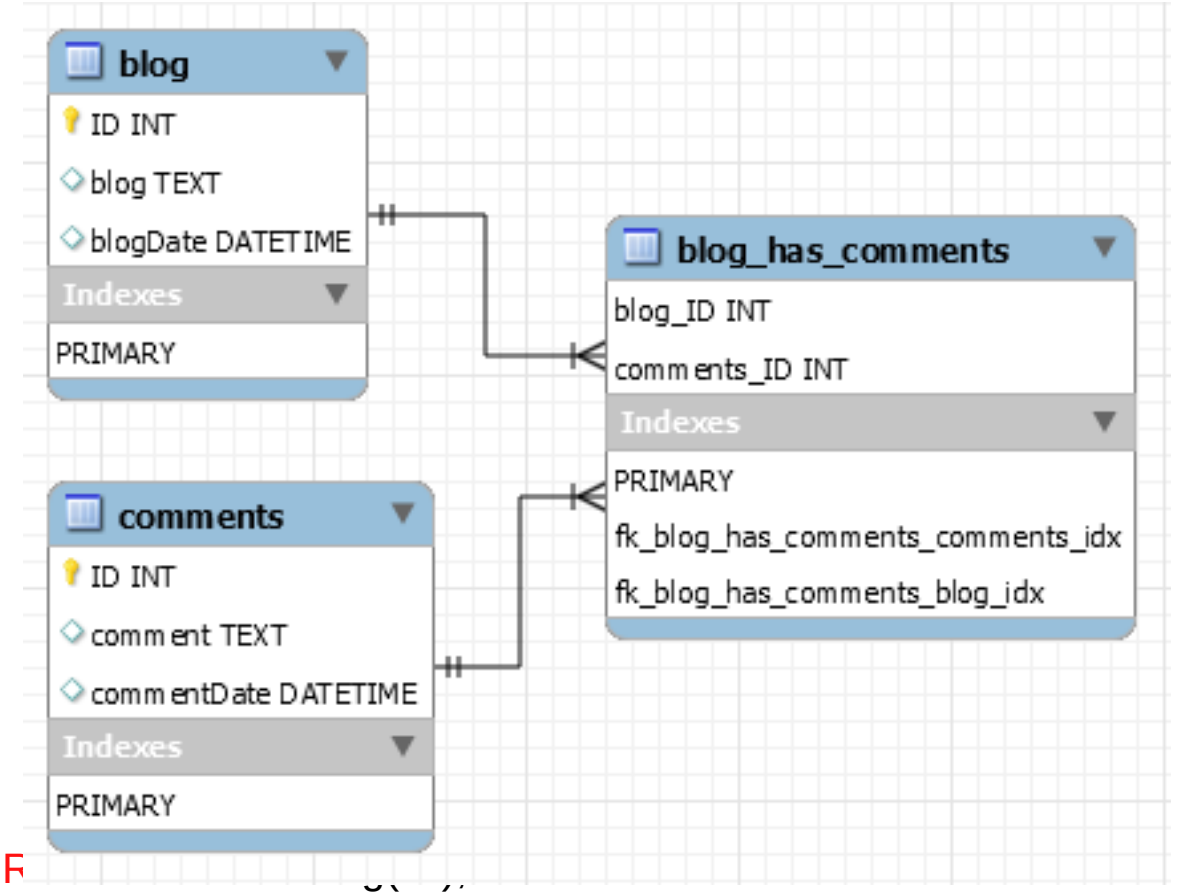
how to create many-to-many relationship

```
CREATE TABLE item (
    ID INT PRIMARY KEY,
    name VARCHAR(45),
    description TEXT
);

CREATE TABLE orders (
    ID INT PRIMARY KEY,
    orderdate DATETIME,
    custID INT NOT NULL,
    shipDate DATETIME,
    total FLOAT(8,2),
    constraint total_greater_zero CHECK(total >= 0)
);

CREATE TABLE orders_has_item (
   orders_ID INT NOT NULL,
   item_ID INT NOT NULL,
   PRIMARY KEY(orders_ID, item_ID),
   constraint fk_orders_has_item_orders FOREIGN KEY(orders_ID)
   REFERENCES orders(ID),
   constraint fk_orders_has_item_item1 FOREIGN KEY(item_ID)
   REFERENCES item(ID)
);
```



how to create many-to-many relationship

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

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

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



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

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

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

What is SQL?

Remember:

what is sql?

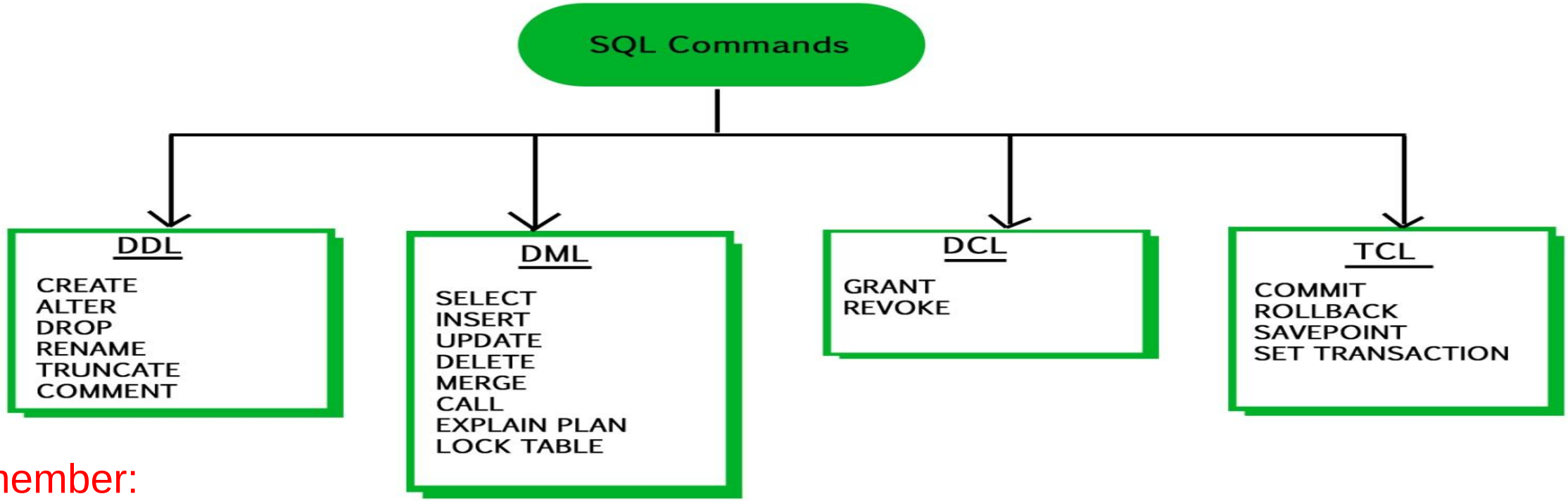
- **EXPLICIT or IMPLICIT commit will commit the data.**

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

Remember:

what is sql?

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



Remember:

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

comments in mysql

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

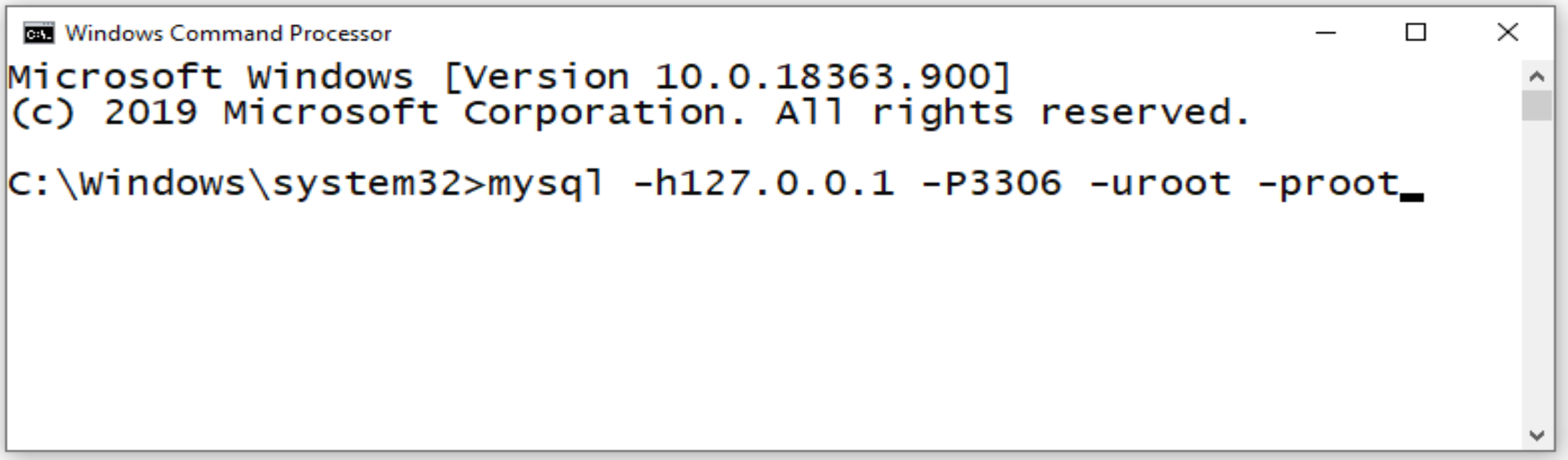
Reconnect to the server	**\r**
Execute a system shell command	**\!**
Exit mysql	**\q**
Change your mysql prompt.	**prompt str or \R str**

Login to MySQL

Default port for MySQL Server: 3306

login

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



SHOW DATABASES

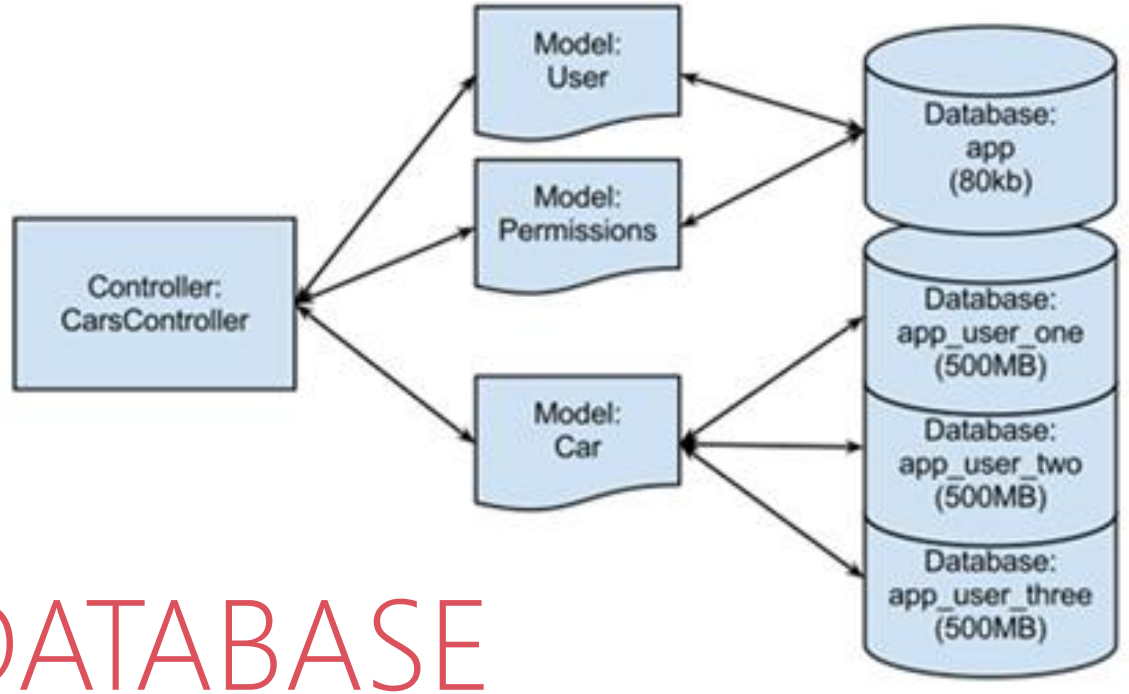
SHOW DATABASES Syntax

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

SHOW SCHEMAS is a synonym for SHOW DATABASES.

```
SHOW DATABASES;
SHOW SCHEMAS;
SHOW DATABASES LIKE 'U%';
SHOW SCHEMAS LIKE 'U%';
```

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



USE DATABASE

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

USE DATABASE Syntax

USE *db_name*
\U *db_name*

Note:

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

USE db1

\U db1

CREATE DATABASE
ALTER DATABASE

create / alter database

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

CREATE { DATABASE | SCHEMA } [IF NOT EXISTS] *db_name*

ALTER { DATABASE | SCHEMA } [*db_name*] READ ONLY [=] { 0 | 1}

CREATE SCHEMA is a synonym for CREATE DATABASE.

- CREATE DATABASE db1;
- CREATE DATABASE IF NOT EXISTS db1;

- ALTER DATABASE db1 READ ONLY = 0; // is in read write mode.
- ALTER DATABASE db1 READ ONLY = 1; // is in read only mode.

Note:

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

DROP DATABASE

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

drop database

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

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

DROP SCHEMA is a synonym for DROP DATABASE.

```
DROP DATABASE db1;
DROP DATABASE IF EXISTS db1;
```

Source Command

source command

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

\. file_name
source file_name

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

SHOW COLUMNS

SHOW COLUMNS Syntax

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

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

SHOW TABLES

SHOW TABLES Syntax

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

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

SHOW VARIABLES

shows the values of MySQL system variables.

SHOW VARIABLES Syntax

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

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

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

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

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

datatypes

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

In MySQL

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

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

Note:

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

datatype - string

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

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

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

example of char and varchar

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

Try Out

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

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

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

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

datatype - numeric

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

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

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

UNSIGNED prohibits negative values.

datatype – date and time

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

datatype – boolean

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

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

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

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

Note:

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

datatype – enum

- An ENUM column can have a maximum of **65,535** distinct elements.
- ENUM values are sorted based on their index numbers, which depend on the order in which the enumeration members were listed in the column specification.
- Default value, NULL if the column can be NULL, first enumeration value if NOT NULL

```
CREATE TABLE temp (col1 INT, COL2 ENUM('A','B','C'));
INSERT INTO temp (col1, col2) VALUES(1, 1);
INSERT INTO temp(col1) VALUES (1);  // NULL
```

```
CREATE TABLE temp (col1 INT, col2 ENUM('A','B','C') NOT NULL);
INSERT INTO temp(col1) VALUES (1); // First element from the ENUM datatype
```

```
CREATE TABLE temp (col1 INT, col2 ENUM('') NOT NULL);
INSERT INTO temp (col1, col2) VALUES (1,'This is the test'); // NULL
```

```
CREATE TABLE temp (col1 INT, COL2 ENUM('A','B','C') default 'D' ); // Invalid default value for 'COL2'
```

IMP:

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

datatype – enum

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

```
size ENUM('small', 'medium', 'large', 'x-large')
membership ENUM('Silver', 'Gold', 'Diamond', 'Platinum')
interest ENUM('Movie', 'Music', 'Concert')
zone ENUM('North', 'South', 'East', 'West')
season ENUM('Winter', 'Summer', 'Monsoon', 'Autumn')
sortby ENUM('Popularity', 'Price -- Low to High', 'Price -- High to Low', 'Newest First')
status ENUM('active', 'inactive', 'pending', 'expired', 'shipped', 'in-process', 'resolved', 'on-hold', 'cancelled',
'disputed')
```

Note:

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

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

datatype – set

- A SET column can have a maximum of **64** distinct members.

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

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

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

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

IMP:

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

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

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

Remember:

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

create table

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

Note:

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

create table

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

Remember:

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

Note:

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

create table

syntax

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

create_definition:
    col_name column_definition

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

table_options:
ENGINE [=] engine_name
```

e.g.

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

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

default value

The DEFAULT specifies a default value for the column.

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

default value

col_name data_type DEFAULT value

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

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

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

version 8.0 and above.

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

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

default value - insert

The **DEFAULT** example.

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

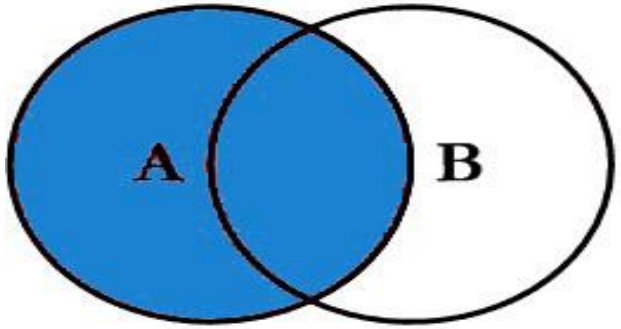
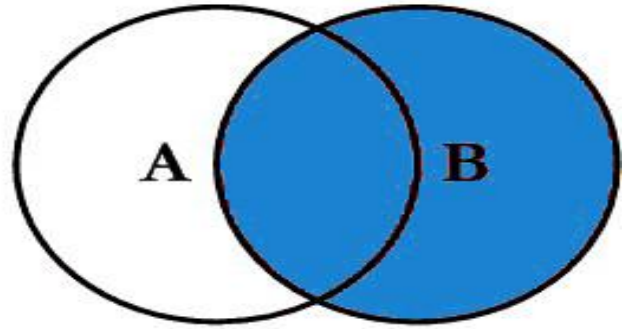
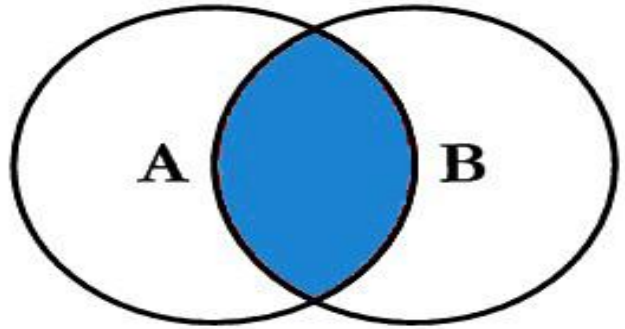
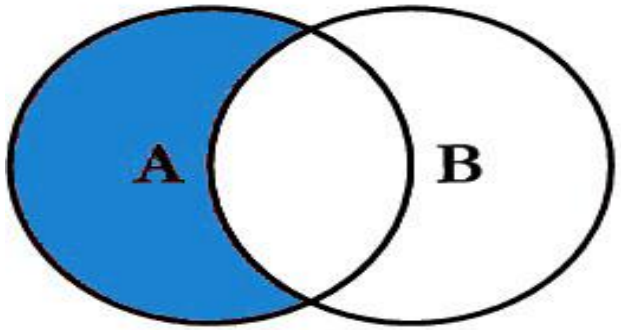
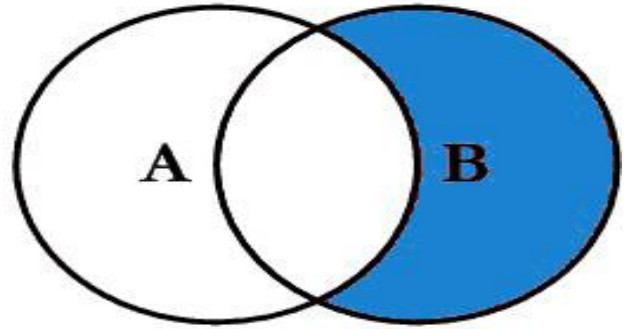
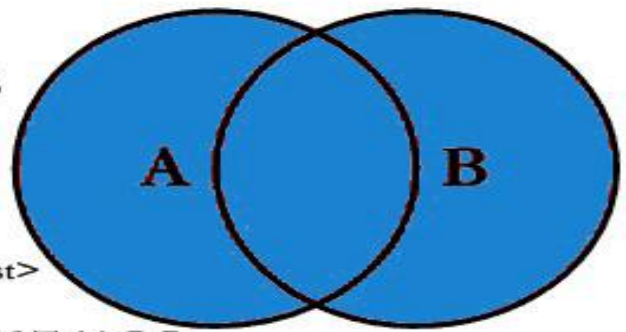
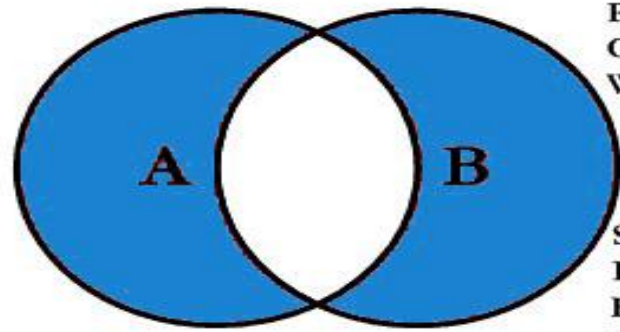
	Field	Type	Null	Key	Default	Extra
▶	c1	int	YES		NULL	
	c2	int	YES		1	
	c3	int	YES		3	

  - INSERT INTO t VALUES();
  - INSERT INTO t VALUES(-1, DEFAULT, DEFAULT);
  - INSERT INTO t VALUES(-2, DEFAULT(c2), DEFAULT(c3));
  - INSERT INTO t VALUES(-3, DEFAULT(c3), DEFAULT(c2));

default value - update

The **DEFAULT** example.

- ```
  CREATE TABLE temp (
    c1 INT,
    c2 INT,
    c3 INT DEFAULT(c1 + c2),
    c4 INT DEFAULT(c1 * c2 )
  );
  ```

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

insert rows

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

You can insert data using following methods:

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

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

Important:

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

INSERT will also fail in following cases.

Important :

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

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

insert rows using values

dml- insert … values

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

INSERT [IGNORE] [INTO] tbl_name [PARTITION (*partition_name* [, *partition_name*] ...)] [(col_name, . . .)] { VALUES | 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 rows into an existing table. The INSERT ... VALUES

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

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

e.g.

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

insert multiple rows

dml- insert … values

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

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

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

e.g.

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

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

- 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.

e.g.

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

single-table update

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

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

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

single-table delete

DELETE is used to delete tuples from a relation.

delete can violate only in referential integrity.

Important:

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

single-table delete

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

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

Note:

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

auto_increment column

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

auto_increment

IDENTITY is a synonym to the *LAST_INSERT_ID* variable.

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

Remember:

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

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

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

auto_increment

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

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

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

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

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

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

auto_increment

- **auto_increment_increment** – is the incremental value, controls the interval between successive column values.
- **auto_increment_offset** – determines the starting value for the AUTO_INCREMENT field; this value is used for the first record inserted into the table.

```
SET @@auto_increment_offset=5;
SET @@auto_increment_increment=10;
ALTER TABLE temp AUTO_INCREMENT = 0;
```

Remember:

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

auto_increment

NO_AUTO_VALUE_ON_ZERO affects handling of *AUTO_INCREMENT* columns. Normally, you generate the next sequence number for the column by inserting either NULL or 0 into it. *NO_AUTO_VALUE_ON_ZERO* suppresses this behaviour for 0 so that only NULL generates the next sequence number.

- *SET sql_mode = '';*
- *SET sql_mode = 'NO_AUTO_VALUE_ON_ZERO';*

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

generated column

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.

- ```
  CREATE TABLE product (
    productCode INT AUTO_INCREMENT PRIMARY KEY,
    productName VARCHAR(45),
    productVendor VARCHAR(45),
    productDescription TEXT,
    quantityInStock INT,
    buyPrice FLOAT,
    stockValue FLOAT GENERATED ALWAYS AS(quantityInStock * buyPrice) VIRTUAL
  );
  ```

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

PROBLEMS

virtual column – MODIFY with ALTER command

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

Note:

- **Virtual column cannot be converted to stored column**.

- CREATE TABLE product1 (
  productCode INT,
  quantityInStock INT,
  buyPrice INT,
  stockValue FLOAT GENERATED ALWAYS AS (quantityInStock * buyPrice)
  VIRTUAL
  );

- ALTER TABLE product1 MODIFY stockValue FLOAT (12, 2); // error: We are trying to convert virtual column [GENERATED ALWAYS AS] to stored column by not giving GENERATED ALWAYS AS

Then What-→ give

- ALTER TABLE product1 MODIFY stockValue FLOAT (12, 2) GENERATED ALWAYS AS (quantityInStock * buyPrice) VIRTUAL ;

PROBLEMS *virtual column – MODIFY with ALTER command*

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

Note:

- **Stored column cannot be converted to virtual column**.

- ```
  CREATE TABLE product1 (
     productCode INT,
     quantityInStock INT,
     buyPrice INT,
     stockValue FLOAT
  );
  ```

- ```
  ALTER TABLE product1 MODIFY stockValue FLOAT GENERATED ALWAYS AS (quantityInStock * buyPrice) VIRTUAL;   // error: We are trying to convert stored column to virtual column by giving GENERATED ALWAYS AS
  ```

visible / invisible columns

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

Note:

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

invisible column

col_name data_type INVISIBLE

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

```
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.

Remember:

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

PRIMARY KEY constraint

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

Note:

- Primary key in a relation is always associated with an **INDEX** object.
- If, we give on a column a combination of **NOT NULL & UNIQUE** key then it behaves like a PRIMARY key.
- If, we give on a column a combination of **UNIQUE key & AUTO_INCREMENT** then also it behaves like a PRIMARY key.
- Stability: The value of the primary key should be stable over time and not change frequently.

clustered and non-clustered index

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

MySQL allows two different types of Indexing:

- Clustered Index
- Non-Clustered Index

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

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

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

constraints – add primary key

col_name data_type PRIMARY KEY

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

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

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

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

- ```
  CREATE TABLE person (
    ID INT NOT NULL UNIQUE,
    lastName VARCHAR(45),
    firstName VARCHAR(45),
    age INT,
    email VARCHAR(255)
  );
  ```

constraints – add composite primary key

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

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

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

constraints – add composite primary key

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

- ```
  CREATE TABLE payments (
    paymentID INT,
    orderID INT,
    amount INT,
    bankDetails VARCHAR(255),
    PRIMARY KEY(paymentID , orderID)
  ) ;
  ```

- ```
  CREATE TABLE order_product (
    orderID INT,
    productID INT,
    qty INT,
    rate INT,
    constraint pk_orderID_productID PRIMARY KEY(orderID, productID)
  ) ;
  ```

Try It

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

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

Add Primary Key using Alter

constraints – add primary key using alter

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

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

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

- ```
  ALTER TABLE vendors ADD PRIMARY KEY(vendor_id);
  ```

- ```
  ALTER TABLE vendors ADD constraint pk_vendor_id PRIMARY KEY(vendor_id);
  ```
  ```

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

Drop Primary Key

constraints – drop primary key

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

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

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

```
ALTER TABLE vendors DROP PRIMARY KEY;
```
```

Remember:

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

UNIQUE KEY constraint

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

Note:

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

constraints – add unique key

col_name data_type UNIQUE KEY

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

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

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

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

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

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

Add Unique Key using Alter

constraints – add unique key using alter

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

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

- ```
  CREATE TABLE shop (
     ID INT,
     shop_name VARCHAR(30)
  );
  ```

- ```
  ALTER TABLE shop ADD UNIQUE(shop_name);
  ```
- ```
  ALTER TABLE shop ADD constraint uni_shop_name UNIQUE(shop_name);
  ```
  ```

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

Drop Unique Key

constraints – drop unique key

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

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

- SELECT table_name, constraint_name, constraint_type FROM information_schema.table_constraints WHERE constraint_schema = 'z' AND table_name IN ('A', 'B');

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

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

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

- ALTER TABLE users DROP INDEX email;

- ALTER TABLE users DROP INDEX uni_email;

[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:

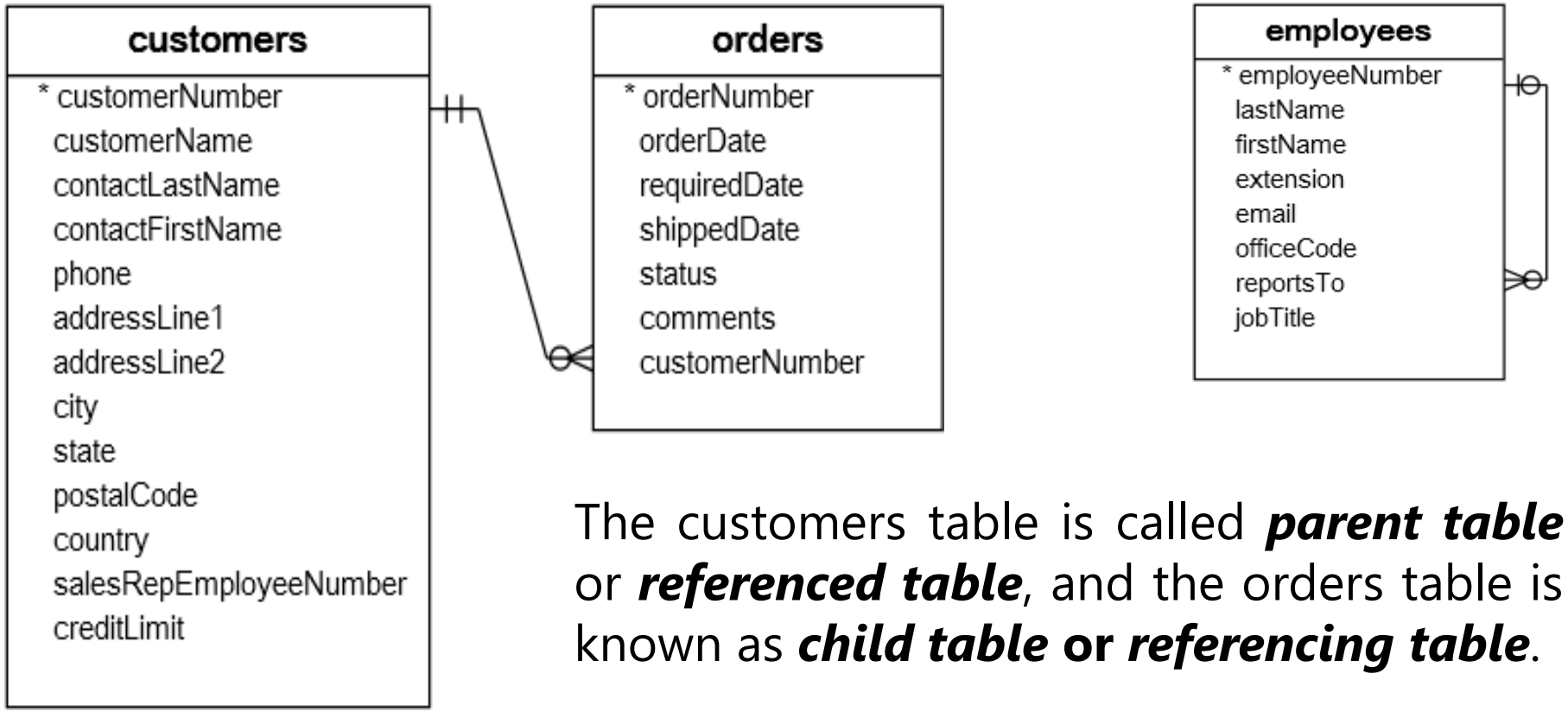
- If we try to modify datatype of RollNo in Student or Student_Course table with VARCHAR, it will not allow.
- If we try to apply auto_increment to RollNo in Student table, it will not allow
- If we try to drop RollNo column from Student table , it will not allow.

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.



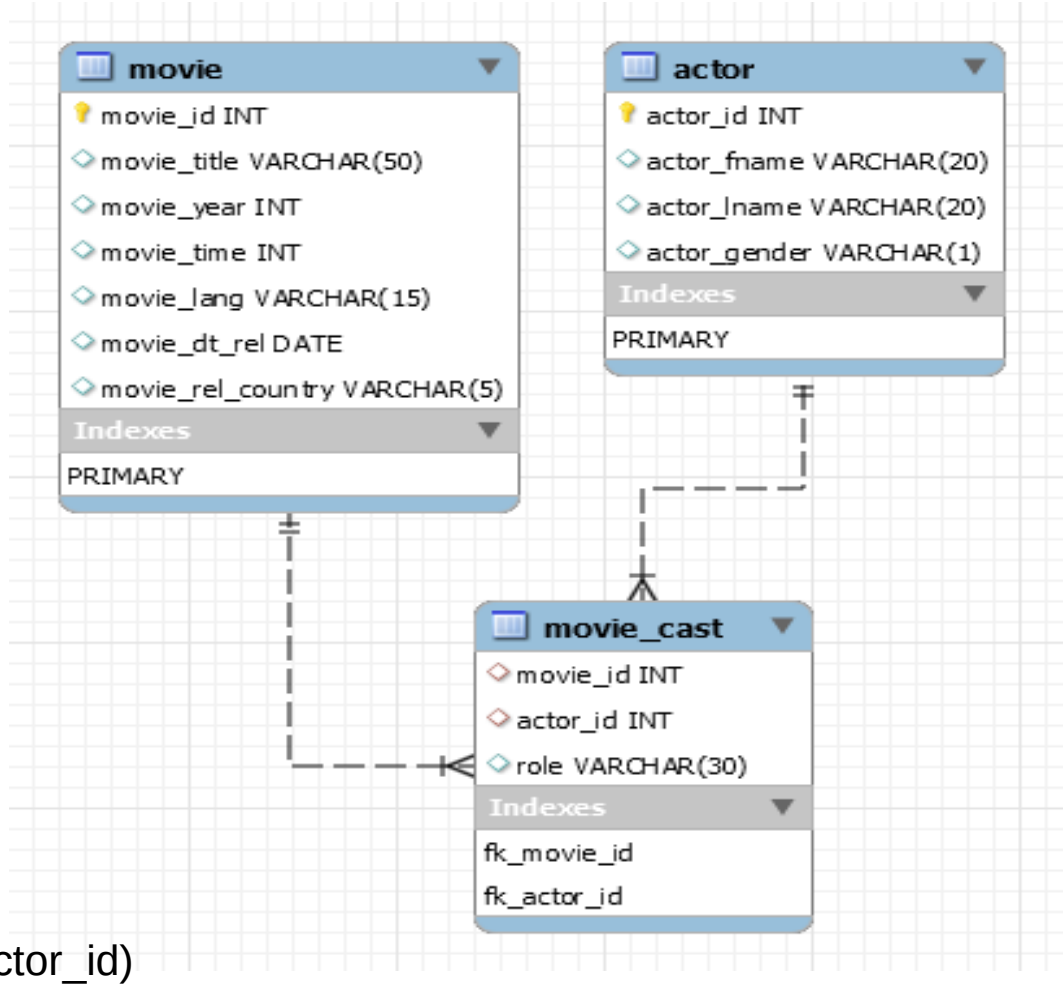
The customers table is called ***parent table*** or ***referenced table***, and the orders table is known as ***child table*** **or** ***referencing table***.

constraints – foreign key

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

- CREATE TABLE movie (
  movie_id INT PRIMARY KEY,
  movie_title VARCHAR(50),
  movie_year INT,
  movie_time INT,
  movie_lang VARCHAR(15),
  movie_dt_rel DATE,
  movie_rel_country VARCHAR(5)
  );

- CREATE TABLE actor (
  actor_id INT PRIMARY KEY,
  actor_fname VARCHAR(20),
  actor_lname VARCHAR(20),
  actor_gender VARCHAR(1)
  );



- CREATE TABLE movie_cast (
  movie_id INT,
  actor_id INT,
  role VARCHAR(30),
  constraint fk_movie_id FOREIGN KEY(movie_id) REFERENCES movie(movie_id),
  constraint fk_actor_id FOREIGN KEY(actor_id) REFERENCES actor(actor_id)
  );

QUESTION – find foreign key columns

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

- ```
  CREATE TABLE owner (
    owner_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    email VARCHAR(255)
  );
  ```

- ```
  CREATE TABLE shop (
    shop_id INT,
    owner_id INT,
    shop_name VARCHAR(30)
  );
  ```

- ```
  CREATE TABLE brands (
    brand_id INT PRIMARY KEY,
    brand_name VARCHAR(30) UNIQUE
   );
  ```

- ```
  CREATE TABLE contacts (
     contact_id INT PRIMARY KEY,
     owner_id INT,
     contact_number VARCHAR(15)
  );
  ```

- ```
  CREATE TABLE shop_brand (
    ID INT PRIMARY KEY,
    shop_id INT,
    brand_id INT
  );
  ```
  ```

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

Add Foreign Key Constraint using Alter

constraints – add foreign key using alter

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

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

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

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

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

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

Drop Foreign Key Constraint using Alter

constraints – drop foreign key

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

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

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

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

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

constraints – foreign key

- [ON DELETE *reference_option*]
- [ON UPDATE *reference_option*]

Cascaded FOREIGN KEY actions do not activate triggers.

reference_option: RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT

Remember:

- When an UPDATE or DELETE operation affects a key value in the parent table that has matching rows in the child table, the result depends on the referential action specified using ON UPDATE and ON DELETE sub clauses of the FOREIGN KEY clause.

- ON DELETE or ON UPDATE that is not specified, the default action is always RESTRICT.

constraints – foreign key

Remember:

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

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

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

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

on delete / on update – foreign key

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

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

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

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

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

// ERROR

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

Check Constraint

constraints – check

CHECK condition expressions must follow some rules.

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

Note:

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

Remember:

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

- table_name_chk_n

constraints – check

col_name data_type CHECK(*expr*)

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

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

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

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

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

constraints – check

col_name data_type CHECK(*expr*)

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

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

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

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

Add Check Constraint using Alter

constraints – add check using alter

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

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

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

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

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

drop check constraint

constraints – drop check key

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

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

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

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

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

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

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

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

- `ALTER TABLE test ADD constraint chk_id CHECK(ID > 10);`
- `ALTER TABLE test DROP CHECK chk_id;`

check with (in, like, and between)

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

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

alter table

ALTER TABLE changes the structure of a table.

Note:

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

syntax

alter table

ALTER TABLE tbl_name

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

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

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	

- ALTER TABLE vehicles
  RENAME COLUMN year TO model_year

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

change column

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

alter_specification

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

change column

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

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

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

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

change column

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

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

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

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

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

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

drop column

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

alter_specification

- DROP [COLUMN] *col_name*

drop column

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

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

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

create table with blackhole engine

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

- **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