

# INTRODUCTION TO DATABASE MANAGEMENT SYSTEM

The slide features a light yellow background with several decorative elements. There are thin vertical lines in light purple and pink. Scattered across the slide are small squares in various colors: cyan, orange, pink, and light green. Some of these squares are solid, while others are hollow outlines.

# What is a Database Management System ?

- **Database** is a collection of related data and data is a collection of facts and figures that can be processed to produce information.
- Mostly data represents recordable facts. Data aids in producing information, which is based on facts. For example, if we have data about marks obtained by all students, we can then conclude about toppers and average marks.
- A **database management system** stores data in such a way that it becomes easier to retrieve, manipulate, and produce information.

# Characteristics of a DBMS

- **Real-world entity** – A modern DBMS is more realistic and uses real-world entities to design its architecture. It uses the behavior and attributes too. For example, a school database may use students as an entity and their age as an attribute.
- **Relation-based tables** – DBMS allows entities and relations among them to form tables. A user can understand the architecture of a database just by looking at the table names.
- **Isolation of data and application** – A database system is entirely different than its data. A database is an active entity, whereas data is said to be passive, on which the database works and organizes. DBMS also stores metadata, which is data about data, to ease its own process.

# Characteristics of a DBMS

- **Less redundancy** – DBMS follows the rules of normalization, which splits a relation when any of its attributes is having redundancy in values. Normalization is a mathematically rich and scientific process that reduces data redundancy.
- **Consistency** – Consistency is a state where every relation in a database remains consistent. There exist methods and techniques, which can detect attempt of leaving database in inconsistent state. A DBMS can provide greater consistency as compared to earlier forms of data storing applications like file-processing systems.
- **Query Language** – DBMS is equipped with query language, which makes it more efficient to retrieve and manipulate data. A user can apply as many and as different filtering options as required to retrieve a set of data. Traditionally it was not possible where file-processing system was used.

# Characteristics of a DBMS

- **ACID Properties** – DBMS follows the concepts of **A**tomicity, **C**onsistency, **I**solation, and **D**urability (normally shortened as ACID). These concepts are applied on transactions, which manipulate data in a database. ACID properties help the database stay healthy in multi-transactional environments and in case of failure.
- **Multiuser and Concurrent Access** – DBMS supports multi-user environment and allows them to access and manipulate data in parallel. Though there are restrictions on transactions when users attempt to handle the same data item, but users are always unaware of them.
- **Multiple views** – DBMS offers multiple views for different users. A user who is in the Sales department will have a different view of database than a person working in the Production department. This feature enables the users to have a concentrate view of the database according to their requirements.

# Characteristics of a DBMS

- **Security** – Features like multiple views offer security to some extent where users are unable to access data of other users and departments. DBMS offers methods to impose constraints while entering data into the database and retrieving the same at a later stage. DBMS offers many different levels of security features, which enables multiple users to have different views with different features. For example, a user in the Sales department cannot see the data that belongs to the Purchase department. Additionally, it can also be managed how much data of the Sales department should be displayed to the user. Since a DBMS is not saved on the disk as traditional file systems, it is very hard for miscreants to break the code.

# Types of Users for DBMS

- **Administrators** – Administrators maintain the DBMS and are responsible for administering the database. They are responsible to look after its usage and by whom it should be used. They create access profiles for users and apply limitations to maintain isolation and force security. Administrators also look after DBMS resources like system license, required tools, and other software and hardware related maintenance.
- **Designers** – Designers are the group of people who actually work on the designing part of the database. They keep a close watch on what data should be kept and in what format. They identify and design the whole set of entities, relations, constraints, and views.
- **End Users** – End users are those who actually reap the benefits of having a DBMS. End users can range from simple viewers who pay attention to the logs or market rates to sophisticated users such as business analysts.

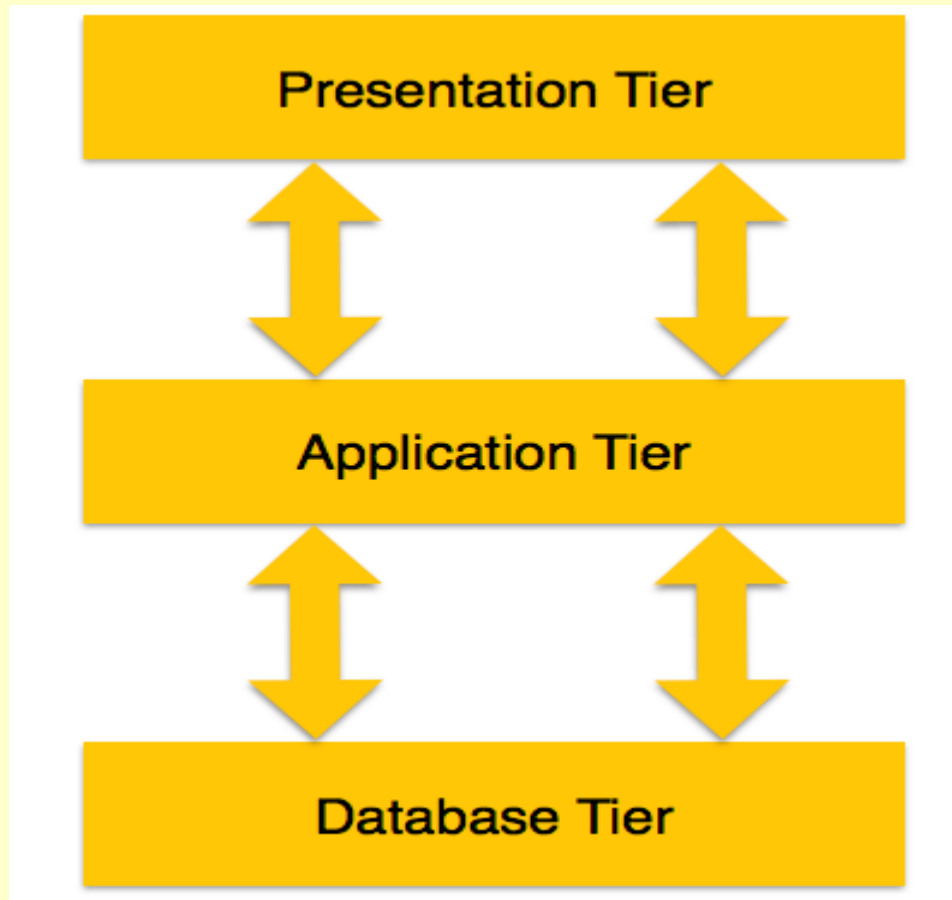
# Architecture of a DBMS

- The **design** of a DBMS depends on its architecture. It can be **centralized** or **decentralized** or **hierarchical**. The architecture of a DBMS can be seen as either single tier or multi-tier. An n-tier architecture divides the whole system into related but independent **n** modules, which can be independently modified, altered, changed, or replaced.
- In **1-tier architecture**, the DBMS is the only entity where the user directly sits on the DBMS and uses it. Any changes done here will directly be done on the DBMS itself. It does not provide handy tools for end-users.
- If the **architecture of DBMS is 2-tier**, then it must have an application through which the DBMS can be accessed. Programmers use 2-tier architecture where they access the DBMS by means of an application. Here the application tier is entirely independent of the database in terms of operation, design, and programming.



# Architecture of a DBMS

- A **3-tier architecture** separates its tiers from each other based on the complexity of the users and how they use the data present in the database. It is the most widely used architecture to design a DBMS.



# Architecture of a DBMS ( 3 – Tier)

- **Database (Data) Tier** – At this tier, the database resides along with its query processing languages. We also have the relations that define the data and their constraints at this level.
- **Application (Middle) Tier** – At this tier reside the application server and the programs that access the database. For a user, this application tier presents an abstracted view of the database. End-users are unaware of any existence of the database beyond the application. At the other end, the database tier is not aware of any other user beyond the application tier. Hence, the application layer sits in the middle and acts as a mediator between the end-user and the database.
- **User (Presentation) Tier** – End-users operate on this tier and they know nothing about any existence of the database beyond this layer. At this layer, multiple views of the database can be provided by the application. All views are generated by applications that reside in the application tier.

# Requirement of a DBMS

- **Data Organization and Management:**

One of the primary needs for a DBMS is data organization and management. DBMSs allow data to be stored in a structured manner, which helps in easier retrieval and analysis. A well-designed database schema enables faster access to information, reducing the time required to find relevant data. A DBMS also provides features like indexing and searching, which make it easier to locate specific data within the database. This allows organizations to manage their data more efficiently and effectively.

- **Data Security and Privacy:**

DBMSs provide a robust security framework that ensures the confidentiality, integrity, and availability of data. They offer authentication and authorization features that control access to the database. DBMSs also provide encryption capabilities to protect sensitive data from unauthorized access. Moreover, DBMSs comply with various data privacy regulations ensuring that organizations can store and manage their data in compliance with legal requirements.

# Requirement of a DBMS

- **Data Integrity and Consistency:**

Data integrity and consistency are crucial for any database. DBMSs provide mechanisms that ensure the accuracy and consistency of data. These mechanisms include constraints, triggers, and stored procedures that enforce data integrity rules. DBMSs also provide features like transactions that ensure that data changes are atomic, consistent, isolated, and durable (ACID).

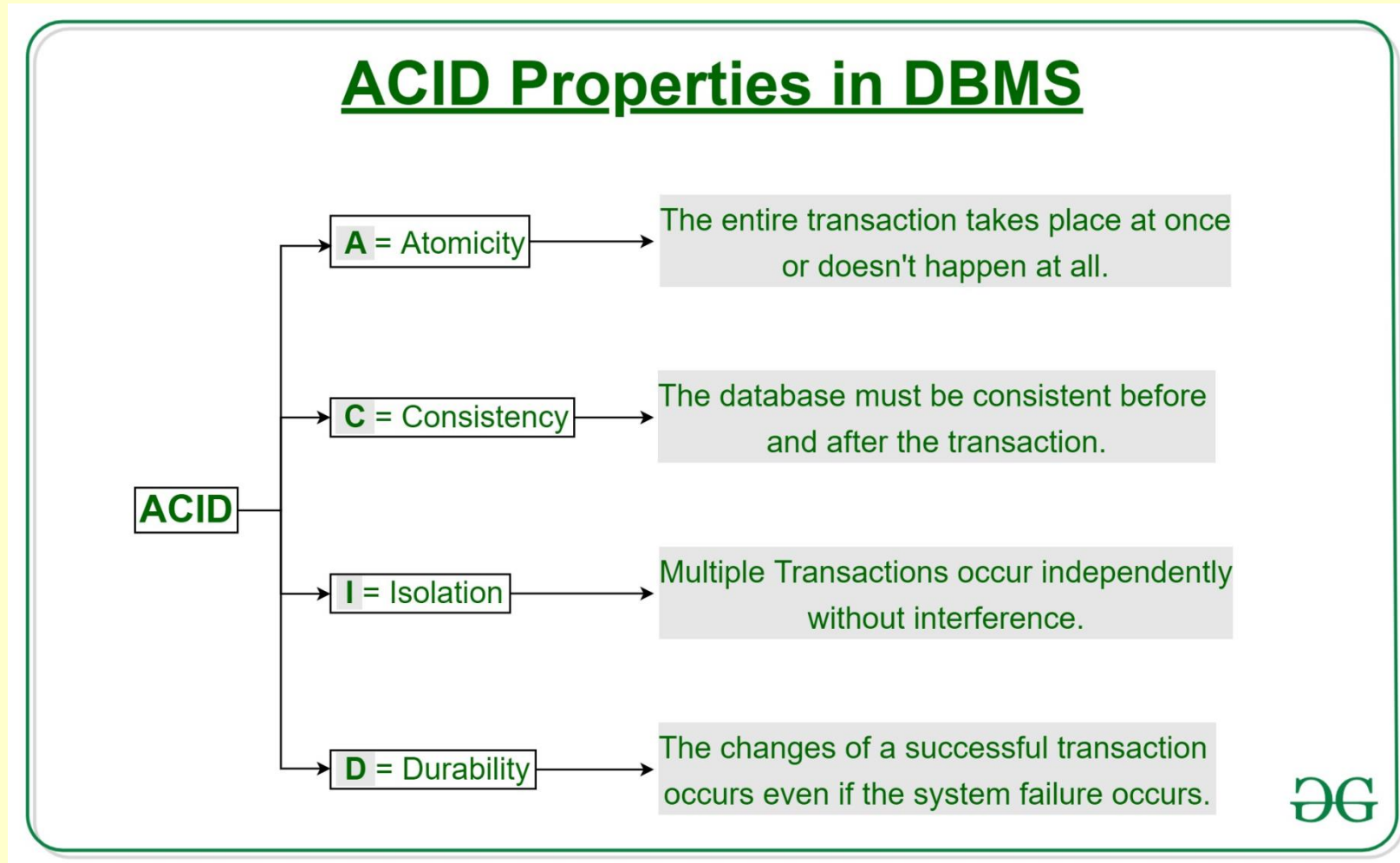
- **Concurrent Data Access:**

A DBMS provides a concurrent access mechanism that allows multiple users to access the same data simultaneously. This is especially important for organizations that require real-time data access. DBMSs use locking mechanisms to ensure that multiple users can access the same data without causing conflicts or data corruption.

# ACID Properties

A transaction is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions access data using read and write operations. In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.

Property	Responsibility for maintaining properties
Atomicity	Transaction Manager
Consistency	Application programmer
Isolation	Concurrency Control Manager
Durability	Recovery Manager



# Requirement of a DBMS

- **Data Analysis and Reporting:**

DBMSs provide tools that enable data analysis and reporting. These tools allow organizations to extract useful insights from their data, enabling better decision-making. DBMSs support various data analysis techniques such as OLAP, data mining, and machine learning. Moreover, DBMSs provide features like data visualization and reporting, which enable organizations to present their data in a visually appealing and understandable way.

- **Scalability and Flexibility:**

DBMSs provide scalability and flexibility, enabling organizations to handle increasing amounts of data. DBMSs can be scaled horizontally by adding more servers or vertically by increasing the capacity of existing servers. This makes it easier for organizations to handle large amounts of data without compromising performance. Moreover, DBMSs provide flexibility in terms of data modeling, enabling organizations to adapt their databases to changing business requirements.

# Requirement of a DBMS

- **Cost-Effectiveness:**

DBMSs are cost-effective compared to traditional file-based systems. They reduce storage costs by eliminating redundancy and optimizing data storage. They also reduce development costs by providing tools for database design, maintenance, and administration. Moreover, DBMSs reduce operational costs by automating routine tasks and providing self-tuning capabilities.

# Data Abstraction in a DBMS

- Database systems comprise complex data structures. In order to make the system efficient in terms of retrieval of data, and reduce complexity in terms of usability of users, developers use abstraction i.e. hide irrelevant details from the users. This approach simplifies database design.
- **Level of Abstraction in a DBMS**
  - **There are mainly 3 levels of data abstraction:**
    - Physical or Internal Level
    - Logical or Conceptual Level
    - View or External Level



# Data Abstraction in a DBMS

- **Physical or Internal Level** - This is the lowest level of data abstraction. It tells us how the data is actually stored in memory. Access methods like sequential or random access and file organization methods like B+ trees and hashing are used for the same. Usability, size of memory, and the number of times the records are factors that we need to know while designing the database.  
Suppose we need to store the details of an employee. Blocks of storage and the amount of memory used for these purposes are kept hidden from the user.
- **Logical or Conceptual Level** - This level comprises the information that is actually stored in the database in the form of tables. It also stores the relationship among the data entities in relatively simple structures. At this level, the information available to the user at the view level is unknown.  
We can store the various attributes of an employee and relationships, e.g. with the manager can also be stored.

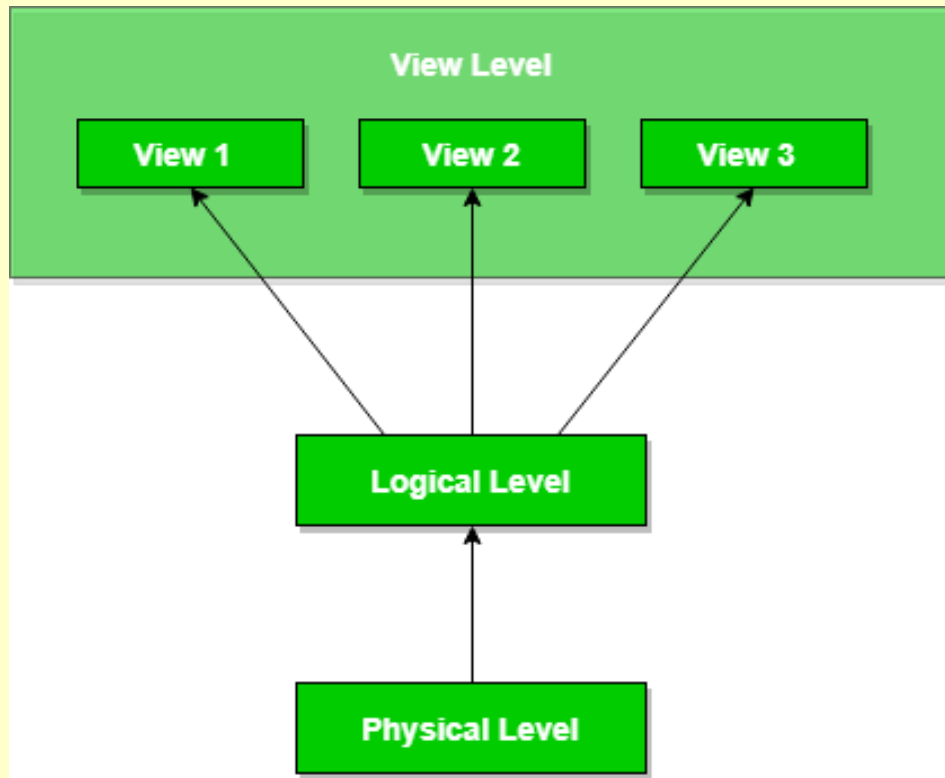
# Data Abstraction in a DBMS

- The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as physical data independence. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.
- **View or External Level** - This is the highest level of abstraction. Only a part of the actual database is viewed by the users. This level exists to ease the accessibility of the database by an individual user. Users view data in the form of rows and columns. Tables and relations are used to store data. Multiple views of the same database may exist. Users can just view the data and interact with the database, storage and implementation details are hidden from them. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system

# Data Abstraction in a DBMS

**Example:** In case of storing customer data,

- **Physical level** – it will contains block of storages (bytes, GB, TB, etc)
- **Logical level** – it will contain the fields and the attributes of data.
- **View level** – it works with [GUI](#) access of database

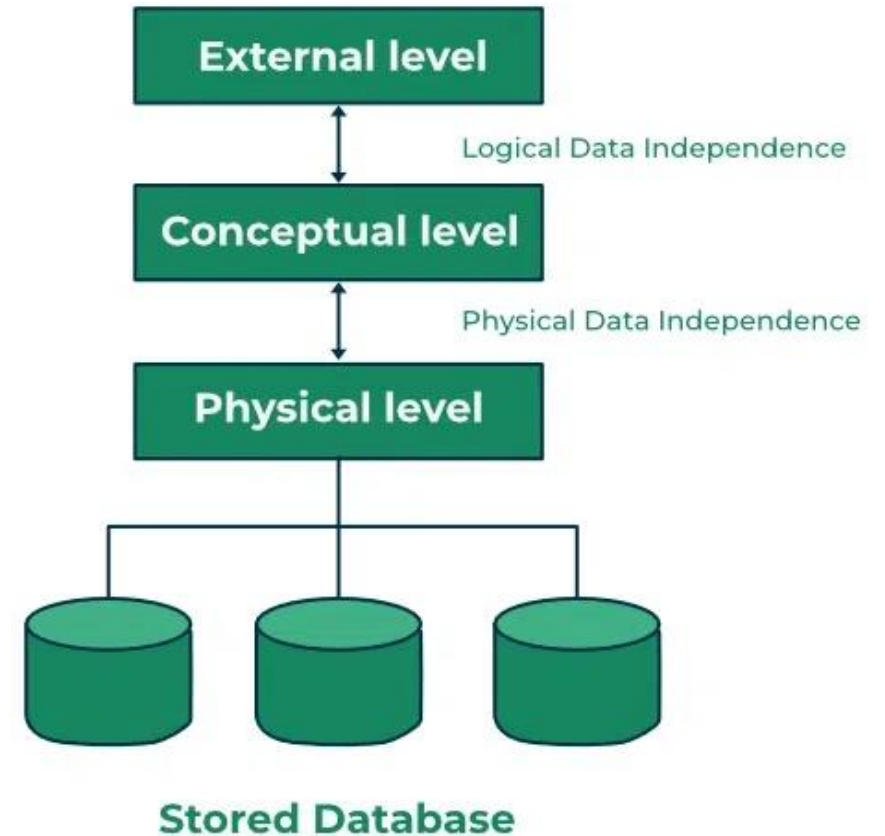


The main purpose of data abstraction is to achieve data independence in order to save the time and cost required when the database is modified or altered.

# Data Independence in a DBMS

- **Data Independence** is mainly defined as a property of DBMS that helps you to change the database schema at one level of a system without requiring to change the schema at the next level. it helps to keep the data separated from all program that makes use of it. We have namely two levels of data independence arising from these levels of abstraction:
  - **Physical level data independence**
  - **Logical level data independence**

## Data Independence in DBMS



# Data Independence in a DBMS

- **Physical Level Data Independence** - It refers to the characteristic of being able to modify the physical schema without any alterations to the conceptual or logical schema, done for optimization purposes, e.g., the Conceptual structure of the [database](#) would not be affected by any change in storage size of the database system server. Changing from sequential to random access files is one such example. These alterations or modifications to the physical structure may include:
  - Utilizing new storage devices.
  - Modifying data structures used for storage.
  - Altering indexes or using alternative file organization techniques etc.
- **Logical Level Data Independence** - It refers characteristic of being able to modify the logical schema without affecting the external schema or application program. The user view of the data would not be affected by any changes to the conceptual view of the data. These changes may include insertion or deletion of attributes, altering table structures entities or relationships to the logical schema, etc.

# Database Languages

## Data Definition Language

**DDL** is the short name for Data Definition Language, which deals with database schemas and descriptions, of how the data should reside in the database.

- **CREATE:** to create a database and its objects like (table, index, views, store procedure, function, and triggers)
- **ALTER:** alters the structure of the existing database
- **DROP:** delete objects from the database
- **TRUNCATE:** remove all records from a table, including all spaces allocated for the records are removed
- **COMMENT:** add comments to the data dictionary
- **RENAME:** rename an object

# Database Languages

## Data Manipulation Language

**DML** is the short name for Data Manipulation Language which deals with data manipulation and includes most common SQL statements such SELECT, INSERT, UPDATE, DELETE, etc., and it is used to store, modify, retrieve, delete and update data in a database.

- **SELECT:** retrieve data from a database
- **INSERT:** insert data into a table
- **UPDATE:** updates existing data within a table
- **DELETE:** Delete all records from a database table
- **MERGE:** UPSERT operation (insert or update)
- **CALL:** call a PL/SQL or Java subprogram
- **EXPLAIN PLAN:** interpretation of the data access path
- **LOCK TABLE:** concurrency Control

# Database Languages

## Data Control Language

**DCL** is short for Data Control Language which acts as an access specifier to the database.(basically to grant and revoke permissions to users in the database

- **GRANT:** grant permissions to the user for running DML(SELECT, INSERT, DELETE,...) commands on the table
- **REVOKE:** revoke permissions to the user for running DML(SELECT, INSERT, DELETE,...) command on the specified table



# Database Languages

## Transactional Control Language

**TCL** is short for Transactional Control Language which acts as an manager for all types of transactional data and all transactions. Some of the command of TCL are

- **Roll Back:** Used to cancel or Undo changes made in the database
- **Commit:** It is used to apply or save changes in the database
- **Save Point:** It is used to save the data on the temporary basis in the database

## Data retrieval language

**DRL** is short for Data Retrieval Language which is used for retrieval of data. It can also be said as **DML**.

- **SELECT:** Used for extracting the required data.

# Advantages of DBMS

- **Data organization**: A DBMS allows for the organization and storage of data in a structured manner, making it easy to retrieve and query the data as needed.
- **Data integrity**: A DBMS provides mechanisms for enforcing data integrity constraints, such as constraints on the values of data and access controls that restrict who can access the data.
- **Concurrent access**: A DBMS provides mechanisms for controlling concurrent access to the database, to ensure that multiple users can access the data without conflicting with each other.
- **Data security**: A DBMS provides tools for managing the security of the data, such as controlling access to the data and encrypting sensitive data.
- **Backup and recovery**: A DBMS provides mechanisms for backing up and recovering the data in the event of a system failure.
- **Data sharing**: A DBMS allows multiple users to access and share the same data, which can be useful in a collaborative work environment.

# Disadvantages of DBMS

- **Complexity**: DBMS can be complex to set up and maintain, requiring specialized knowledge and skills.
- **Performance overhead**: The use of a DBMS can add overhead to the performance of an application, especially in cases where high levels of concurrency are required.
- **Scalability**: The use of a DBMS can limit the scalability of an application, since it requires the use of locking and other synchronization mechanisms to ensure data consistency.
- **Cost**: The cost of purchasing, maintaining and upgrading a DBMS can be high, especially for large or complex systems.
- **Limited Use Cases**: Not all use cases are suitable for a DBMS, some solutions don't need high reliability, consistency or security and may be better served by other types of data storage.

# Types of DBMS

- **Relational DBMS (RDBMS)**: An RDBMS stores data in tables with rows and columns, and uses SQL (Structured Query Language) to manipulate the data.
- **Object-Oriented DBMS (OODBMS)**: An OODBMS stores data as objects, which can be manipulated using object-oriented programming languages.
- **NoSQL DBMS**: A NoSQL DBMS stores data in non-relational data structures, such as key-value pairs, document-based models, or graph models.

The background is a solid light yellow. It features several vertical lines of varying heights and colors (pink, purple, blue). Scattered throughout are small squares in various colors including teal, orange, pink, and blue. Some squares are solid, while others are outlined.

THANK YOU

# Models

Entity-Relationship model

Relational model

# Entity-Relationship model

The background features four vertical lines in light purple. Scattered across the page are several small squares in teal, orange, pink, and light green. A horizontal blue line underlines the title, starting with a short black segment on the left.

# Preview

- What is an ER Diagram
- What is an ER Model
- History of ER models
- Why Use ER Diagrams in DBMS
- Symbols Used in ER Diagrams
- Components of ER Diagram
- How to Draw an ER Diagram
- Conclusion



# What is an ER Model and ER Diagram ?

- The Entity Relational Model is a model for identifying entities to be represented in the database and representation of how those entities are related. The ER data model specifies enterprise schema that represents the overall logical structure of a database graphically.
- The Entity Relationship Diagram explains the relationship among the entities present in the database. ER models are used to model real-world objects like a person, a car, or a company and the relation between these real-world objects. In short, the ER Diagram is the structural format of the database.

## Why Use ER Diagrams In DBMS?






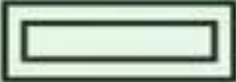
- ER diagrams are used to represent the E-R model in a database, which makes them easy to convert into relations (tables).
- ER diagrams provide the purpose of real-world modeling of objects which makes them intently useful.
- ER diagrams require no technical knowledge and no hardware support.
- These diagrams are very easy to understand and easy to create even for a naive user.
- It gives a standard solution for visualizing the data logically.

# Symbols Used in ER Model

ER Model is used to model the logical view of the system from a data perspective which consists of these symbols:

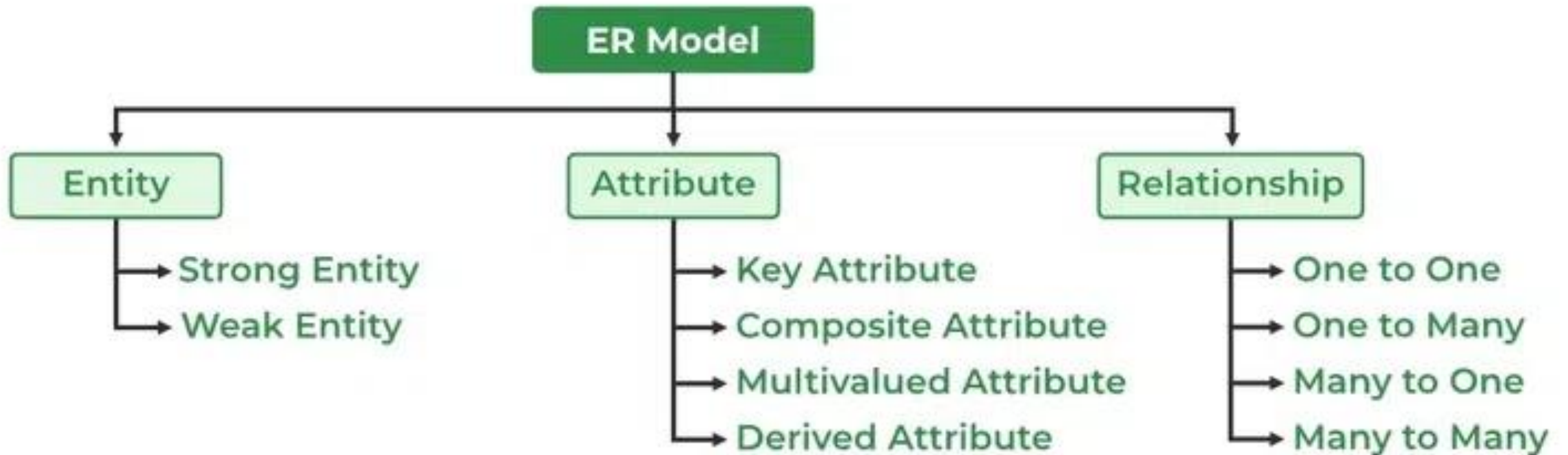
- **Rectangles:** Rectangles represent Entities in the ER Model.
- **Ellipses:** Ellipses represent Attributes in the ER Model.
- **Diamond:** Diamonds represent Relationships among Entities.
- **Lines:** Lines represent attributes to entities and entity sets with other relationship types.
- **Double Ellipse:** Double Ellipses represent Multi-Valued Attributes.
- **Double Rectangle:** Double Rectangle represents a Weak Entity.

# Symbols Used in ER Model

Figures	Symbols	Represents
Rectangle		Entities in ER Model
Ellipse		Attributes in ER Model
Diamond		Relationships among Entities
Line		Attributes to Entities and Entity Sets with Other Relationship Types
Double Ellipse		Multi-Valued Attributes
Double Rectangle		Weak Entity

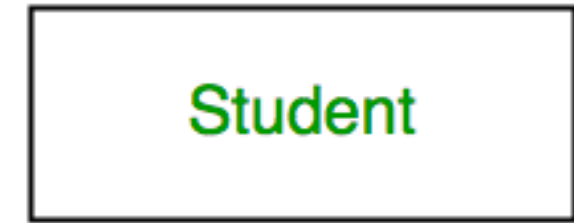
# Components of ER Diagram

**ER Model consists of Entities, Attributes, and Relationships among Entities in a Database System.**

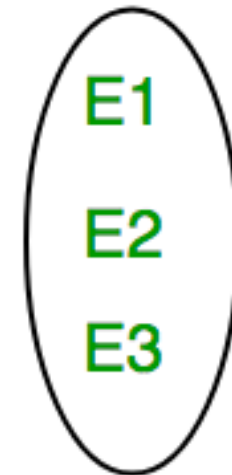


# Components of ER Diagram

- **Entity** - An Entity may be an object with a physical existence – a particular person, car, house, or employee – or it may be an object with a conceptual existence – a company, a job, or a university course.
- **Entity Set:** An Entity is an object of Entity Type and a set of all entities is called an entity set. For Example, E1 is an entity having Entity Type Student and the set of all students is called Entity Set. In ER diagram, Entity Type is represented as:



Entity Type



Entity Set

# Components of ER Diagram

## **Strong Entity**

- A Strong Entity is a type of entity that has a key Attribute. Strong Entity does not depend on other Entity in the Schema. It has a primary key, that helps in identifying it uniquely, and it is represented by a rectangle. These are called Strong Entity Types.

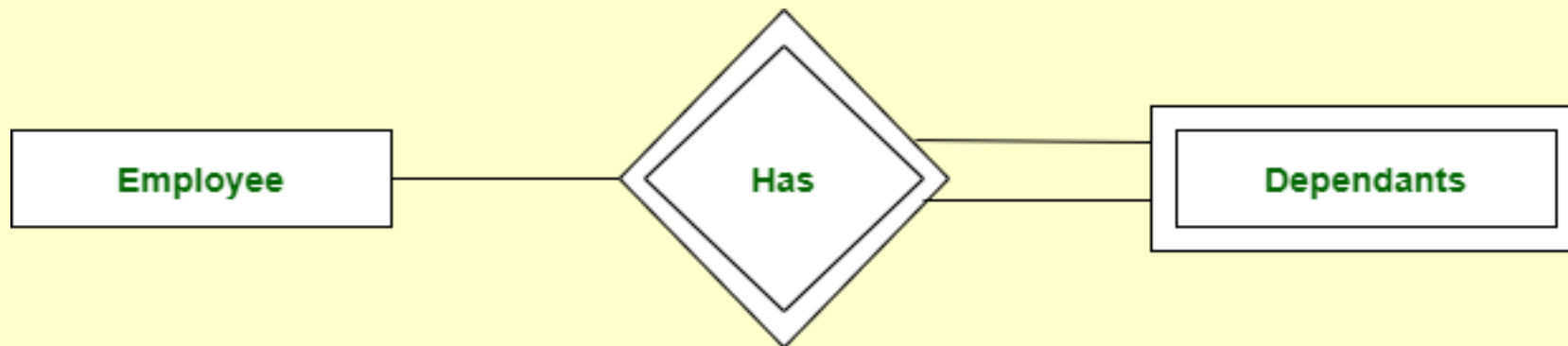
## **Weak Entity**

- An Entity type has a key attribute that uniquely identifies each entity in the entity set. But some entity type exists for which key attributes can't be defined. These are called Weak Entity types.



# Components of ER Diagram

- **For Example**, A company may store the information of dependents (Parents, Children, Spouse) of an Employee. But the dependents don't have existed without the employee. So Dependent will be a **Weak Entity Type** and Employee will be Identifying Entity type for Dependent, which means it is **Strong Entity Type**.
- A weak entity type is represented by a Double Rectangle. The participation of weak entity types is always total. The relationship between the weak entity type and its identifying strong entity type is called identifying relationship and it is represented by a double diamond.





# Components of ER Diagram

## Attributes

- Attributes are the properties that define the entity type. For example, Roll\_No, Name, DOB, Age, Address, and Mobile\_No are the attributes that define entity type Student. In ER diagram, the attribute is represented by an oval.



## Key Attribute

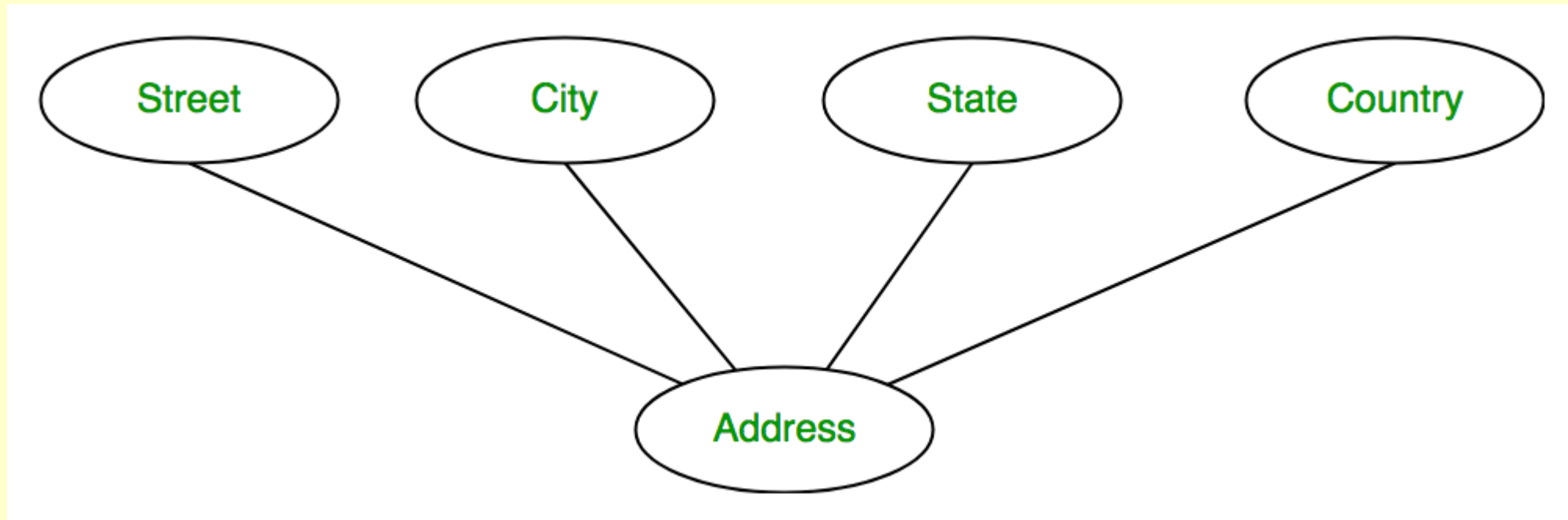
- The attribute which **uniquely identifies each entity** in the entity set is called the key attribute. For example, Roll\_No will be unique for each student. In ER diagram, the key attribute is represented by an oval with underlying lines.



# Components of ER Diagram

## Composite Attribute

- An attribute **composed of many other attributes** is called a composite attribute. For example, the Address attribute of the student Entity type consists of Street, City, State, and Country. In ER diagram, the composite attribute is represented by an oval comprising of ovals.



# Components of ER Diagram

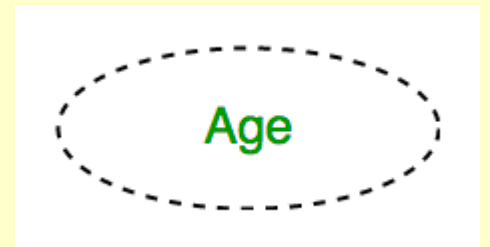
## Multivalued Attribute

- An attribute consisting of more than one value for a given entity. For example, Phone\_No (can be more than one for a given student). In ER diagram, a multivalued attribute is represented by a double oval.



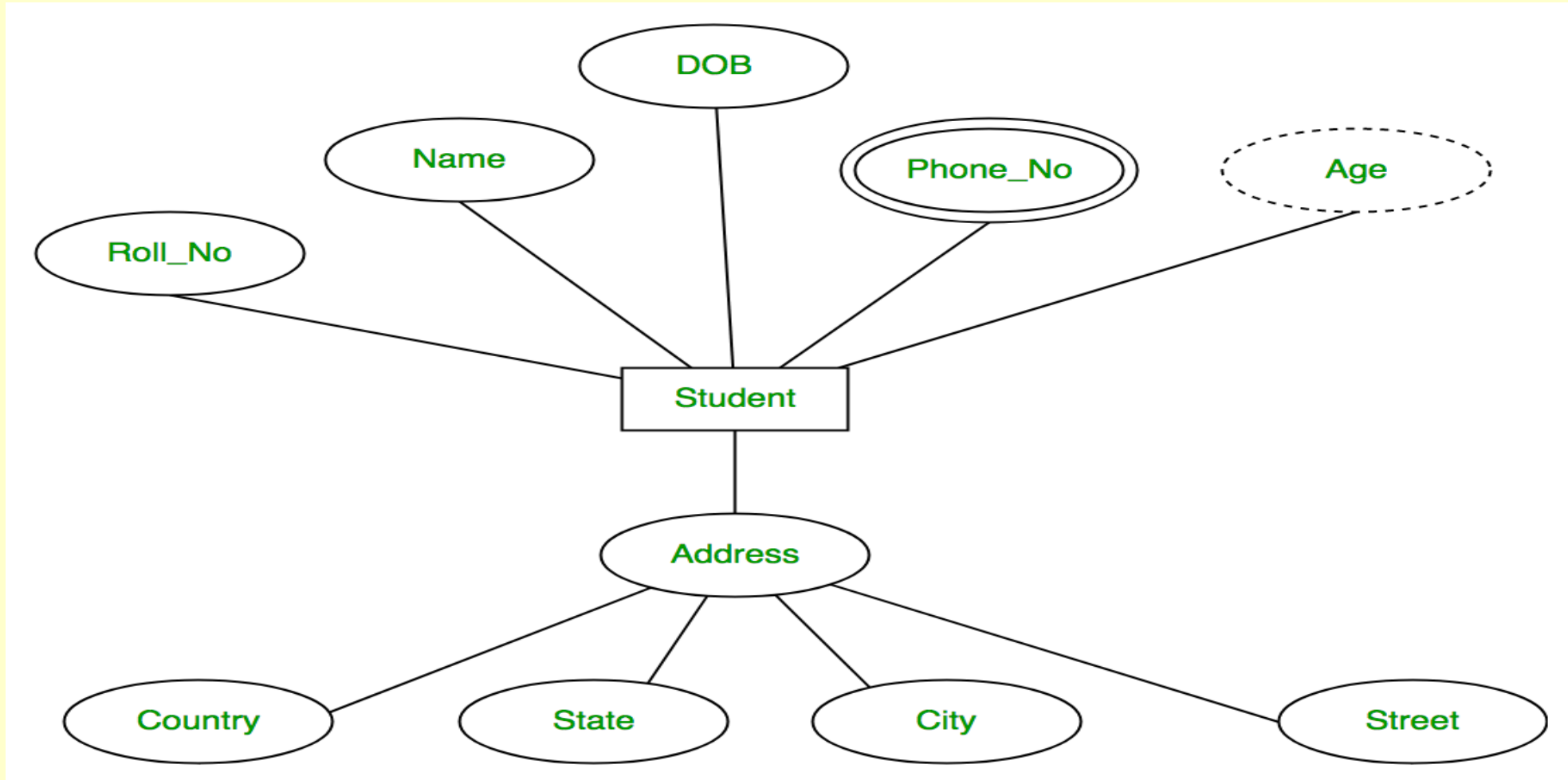
## Derived Attribute

- An attribute that can be derived from other attributes of the entity type is known as a derived attribute. e.g.; Age (can be derived from DOB). In ER diagram, the derived attribute is represented by a dashed oval.



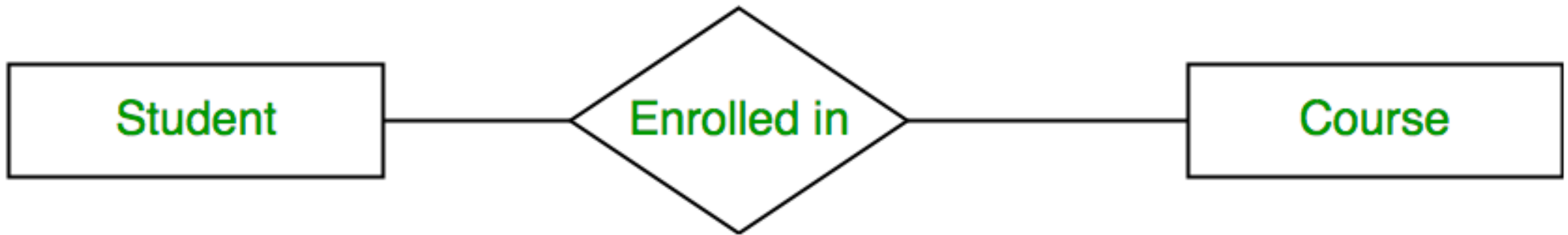
# Components of ER Diagram

The Complete Entity Type Student with its Attributes can be represented as:



# Relationship Type and Relationship Set

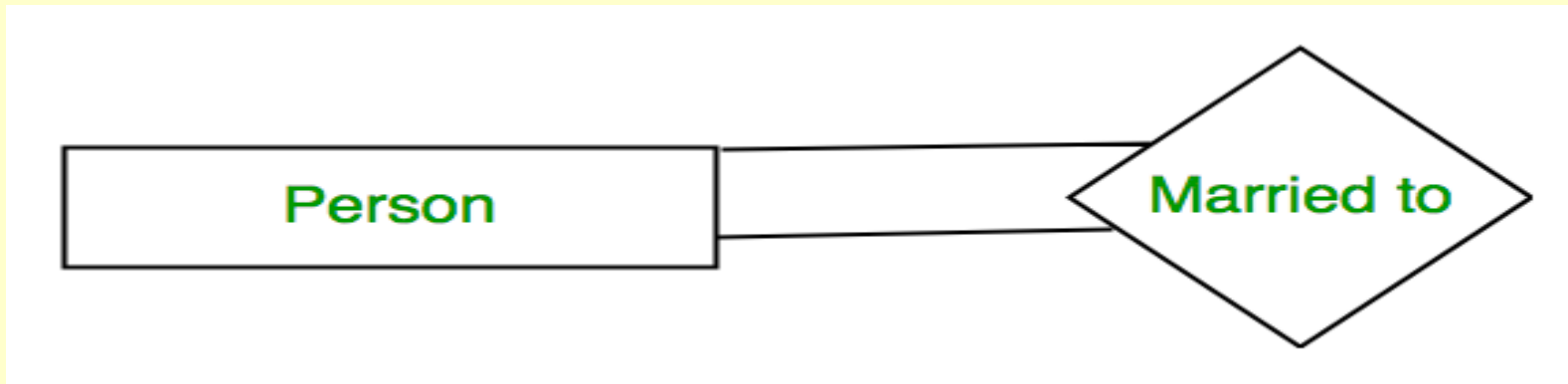
A Relationship Type represents the association between entity types. For example, 'Enrolled in' is a relationship type that exists between entity type Student and Course. In ER diagram, the relationship type is represented by a diamond and connecting the entities with lines.



# Relationship Type and Relationship Set

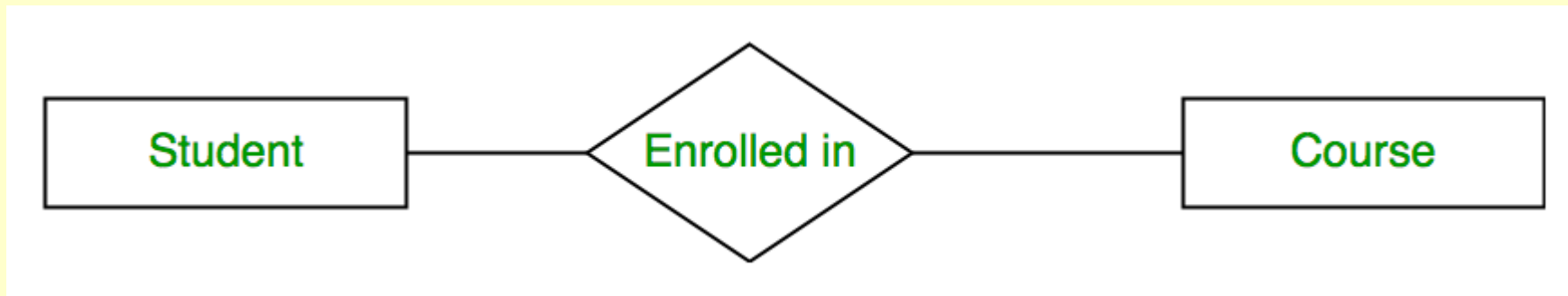
**Degree of a Relationship Set.** The number of different entity sets participating in a relationship set is called the degree of a relationship set.

**Unary Relationship:** When there is only ONE entity set participating in a relation, the relationship is called a unary relationship. For example, one person is married to only one person.



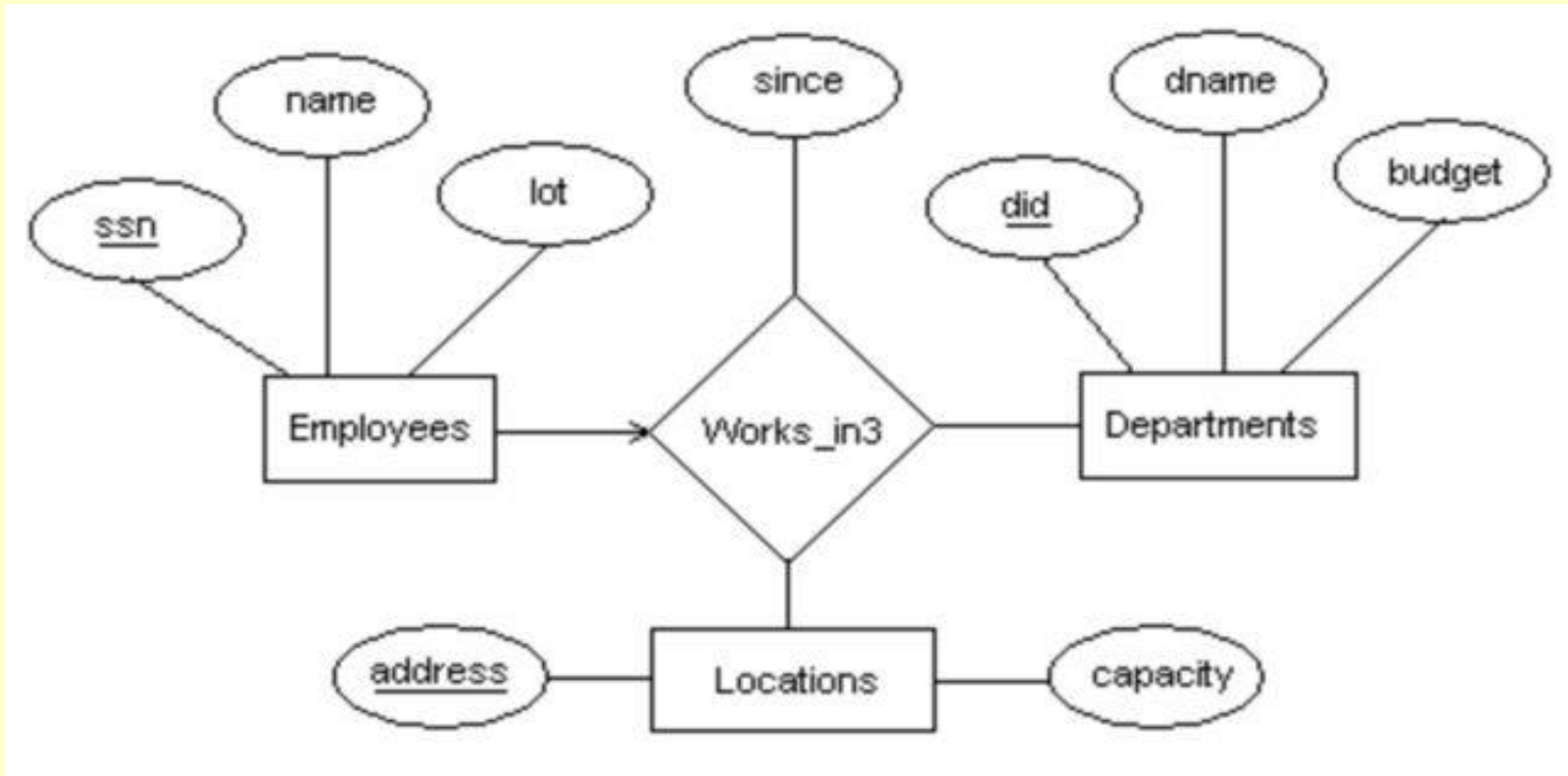
# Relationship Type and Relationship Set

**Binary Relationship:** When there are TWO entities set participating in a relationship, the relationship is called a binary relationship. For example, a Student is enrolled in a Course.



# Relationship Type and Relationship Set

**n-ary Relationship:** When there are n entities set participating in a relation, the relationship is called an n-ary relationship.





# Relationship Type and Relationship Set

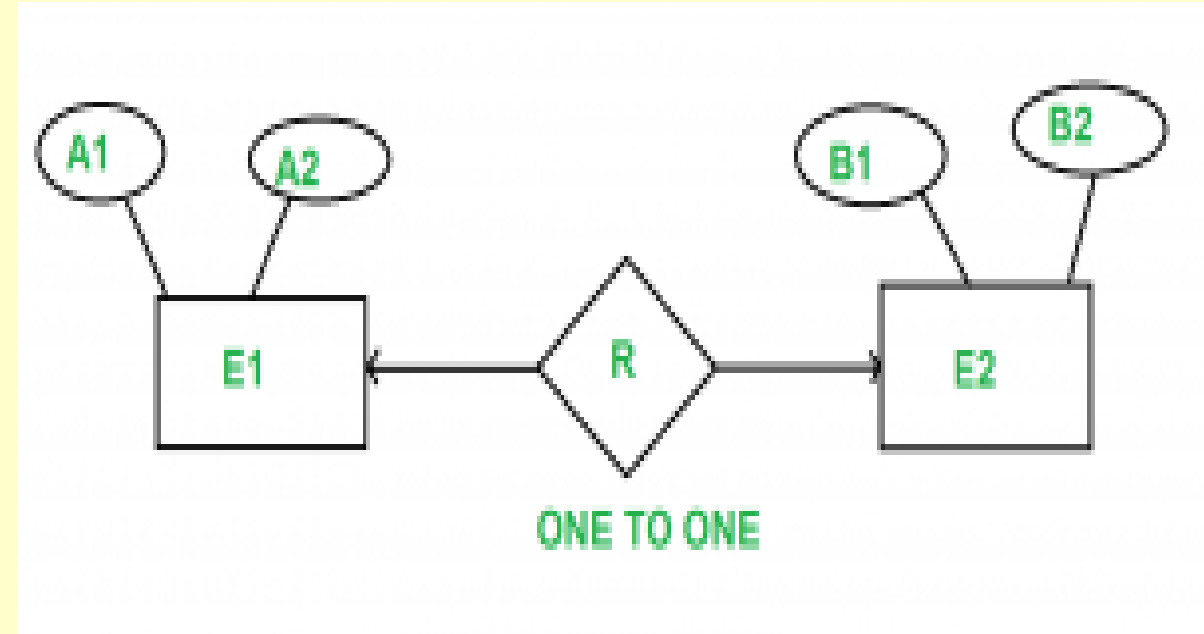
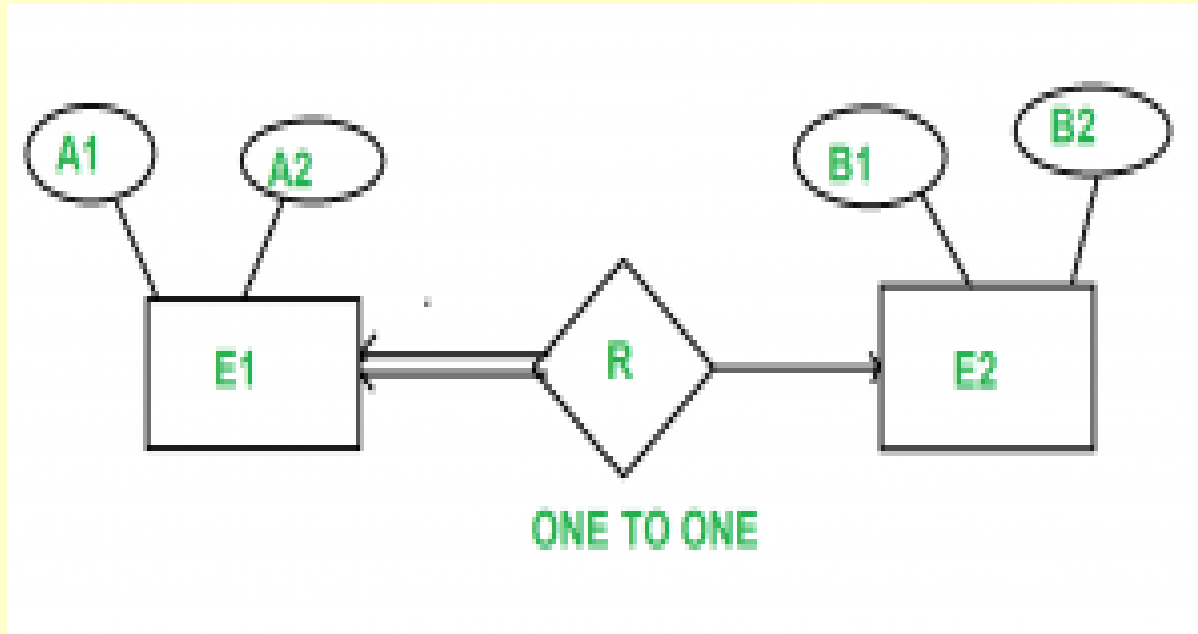
## Cardinality

Cardinality means that what is the number of relationships between the two entity sets in any relationship model. There are four types of cardinality which are mentioned below.

- One-to-One
- One-to-Many
- Many-to-One
- Many-to-Many

# Relationship Type and Relationship Set

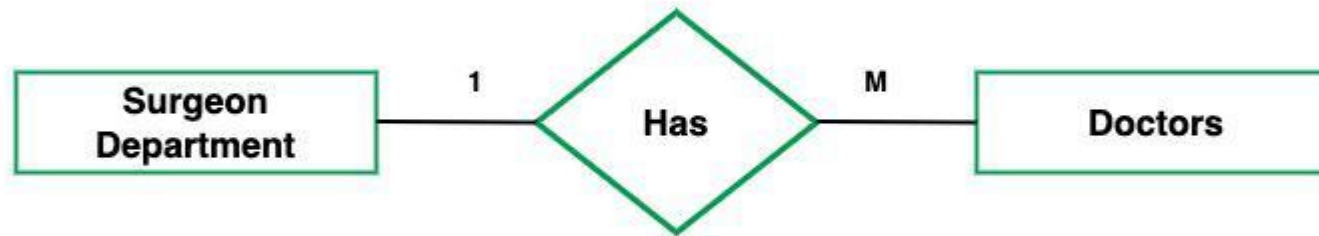
- **One-to-One Cardinality** - One to One Cardinality has two possible cases where we have the case of either total participation or no participation at one end.
- **Total Participation at One End** - Only 1 table is required.
- **No Participation at One End** - A minimum of 2 tables is required.



Own Time Work – Find out two different examples of these cases

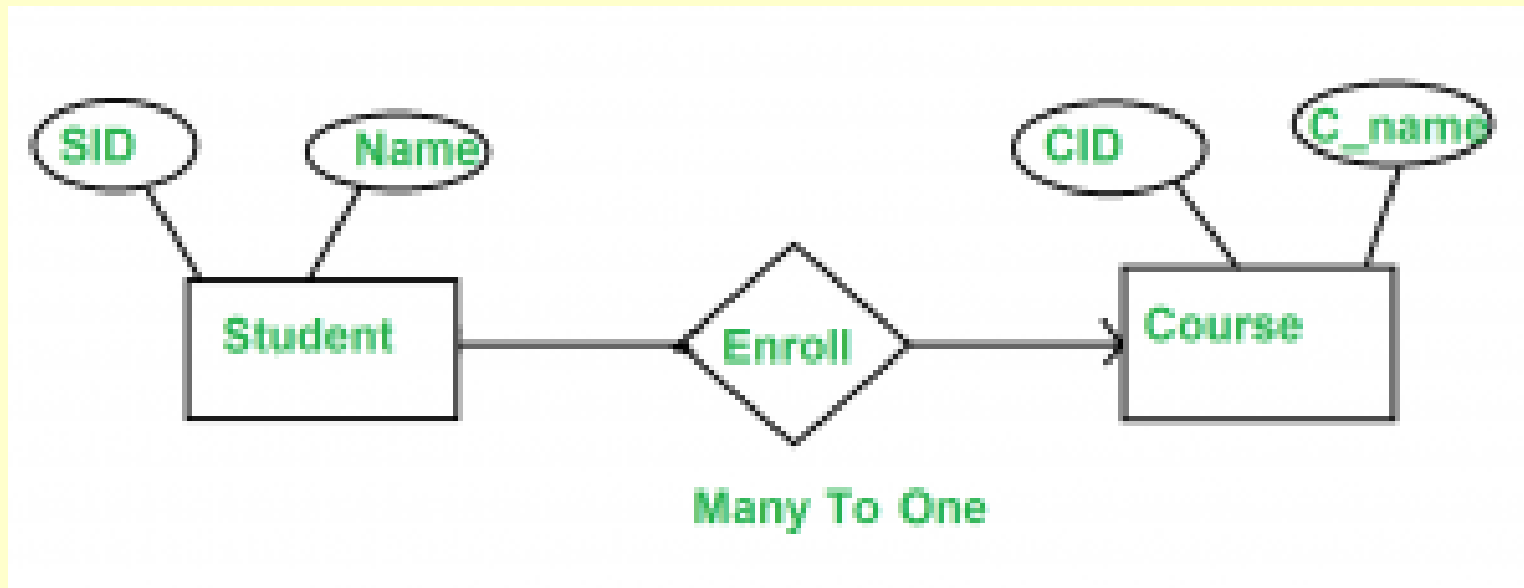
# Relationship Type and Relationship Set

- **One-to-Many:** In one-to-many mapping as well where each entity can be related to more than one relationship and the total number of tables that can be used in this is 2. Let us assume that one surgeon department can accommodate many doctors. So the Cardinality will be 1 to M. It means one department has many Doctors.
- total number of tables that can be used is 3.



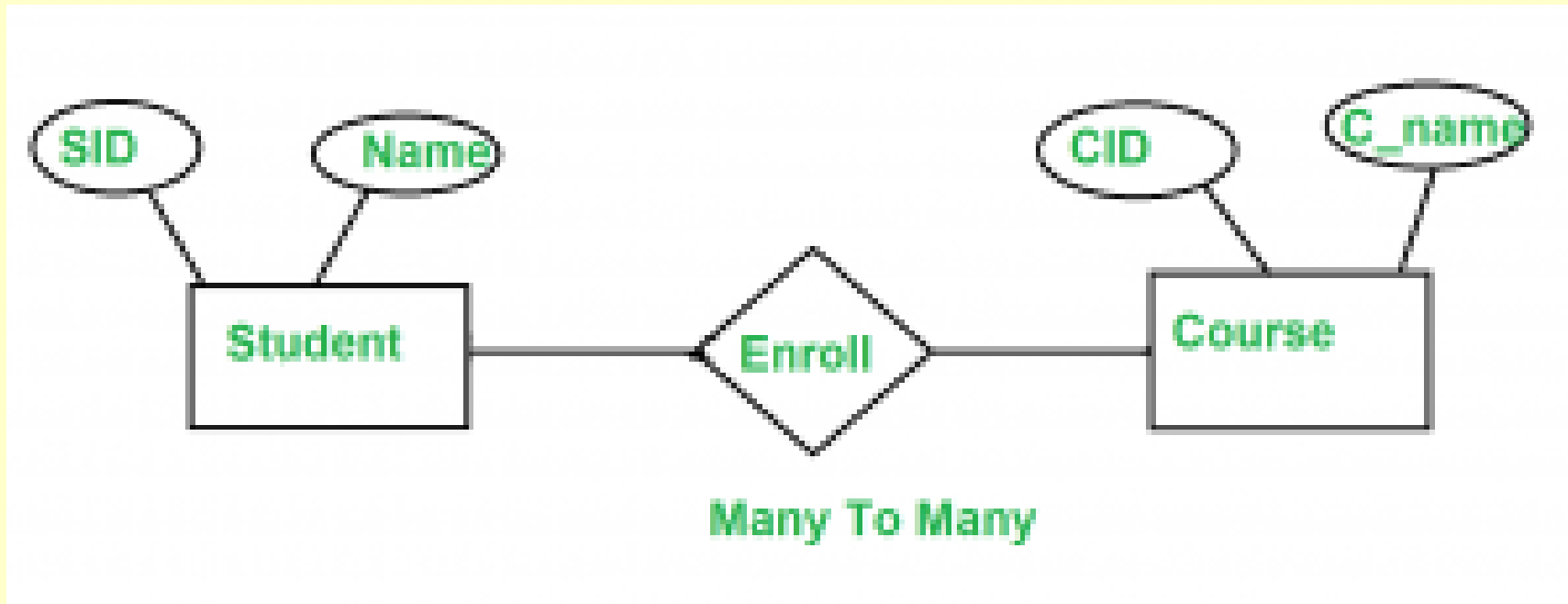
# Relationship Type and Relationship Set

- **Many-to-One:** When entities in one entity set can take part only once in the relationship set and entities in other entity sets can take part more than once in the relationship set, cardinality is many to one. Let us assume that a student can take only one course but one course can be taken by many students. So the cardinality will be  $n$  to 1. It means that for one course there can be  $n$  students but for one student, there will be only one course.
- The total number of tables that can be used in this is 3.



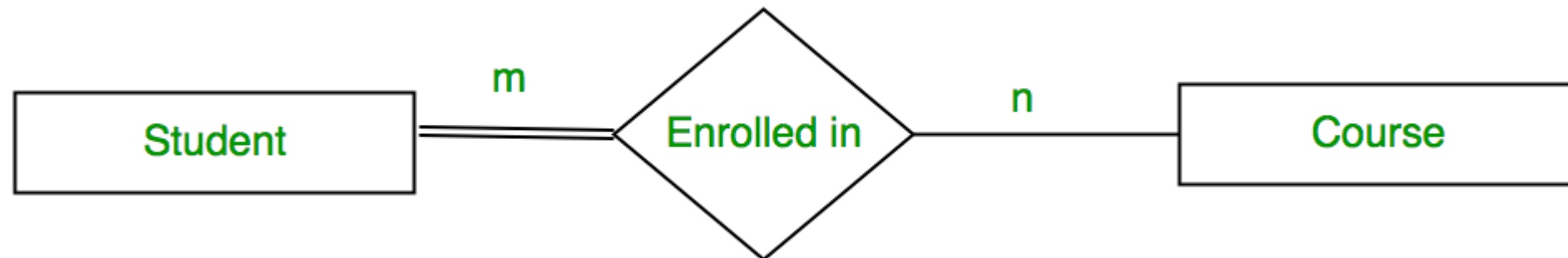
# Relationship Type and Relationship Set

- **Many-to-Many:** When entities in all entity sets can take part more than once in the relationship cardinality is many to many. Let us assume that a student can take more than one course and one course can be taken by many students. So the relationship will be many to many.
- the total number of tables that can be used in this is 3.



# Participation Constraint

- [Participation Constraint](#) is applied to the entity participating in the relationship set.
- **Total Participation** – Each entity in the entity set must participate in the relationship. If each student must enroll in a course, the participation of students will be total. Total participation is shown by a double line in the ER diagram.
- **Partial Participation** – The entity in the entity set may or may NOT participate in the relationship. If some courses are not enrolled by any of the students, the participation in the course will be partial.
- The diagram depicts the 'Enrolled in' relationship set with Student Entity set having total participation and Course Entity set having partial participation.

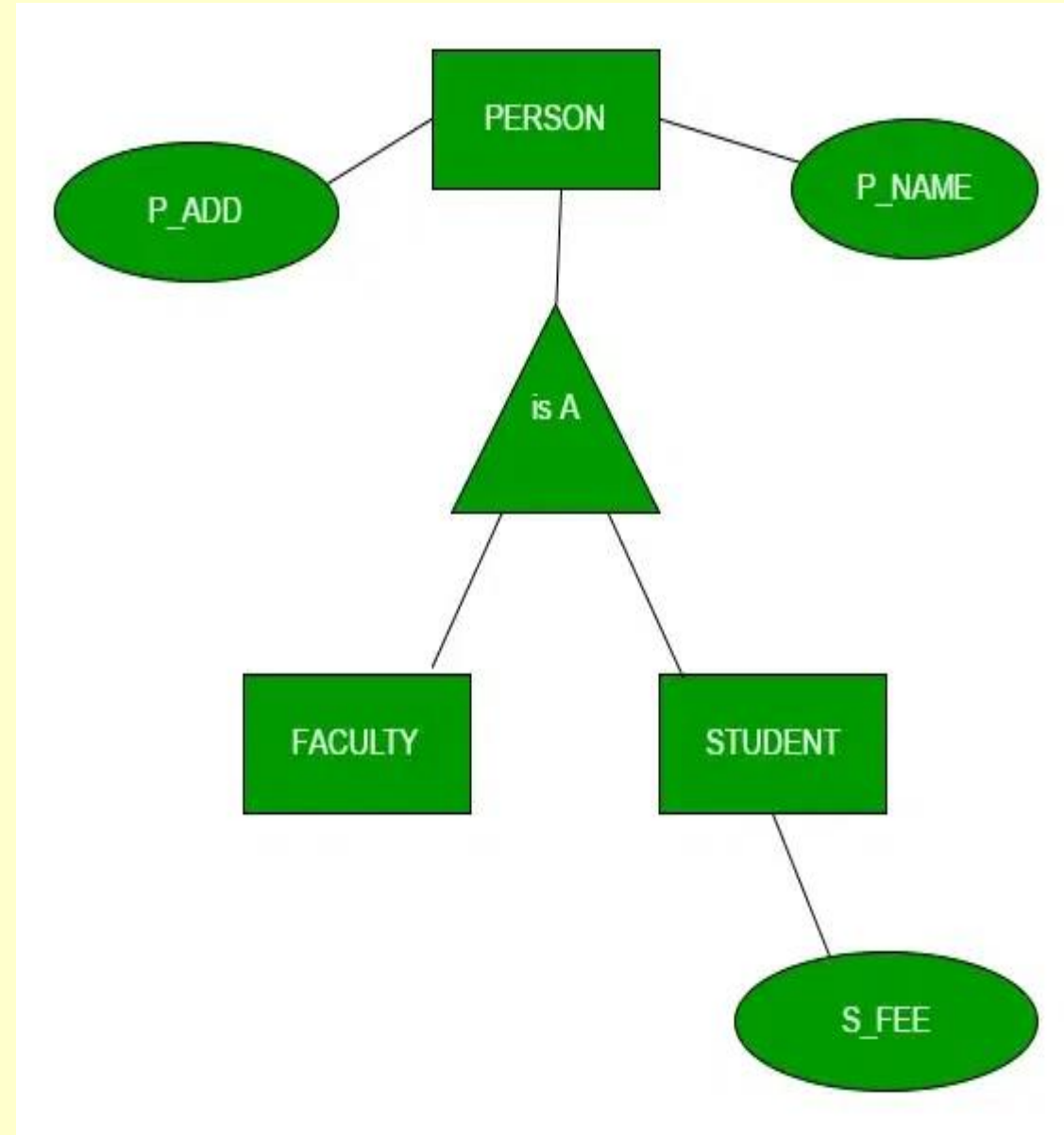


# Enhanced ER - Generalization, Specialization and Aggregation in ER Model

- Using the ER model for bigger data creates a lot of complexity while designing a database model, So in order to minimize the complexity Generalization, Specialization, and Aggregation were introduced in the ER model and these were used for data abstraction in which an abstraction mechanism is used to hide details of a set of objects. Some of the terms were added to the Enhanced ER Model, where some new concepts were added. These new concepts are:
  - Generalization
  - Specialization
  - Aggregation

# Generalization

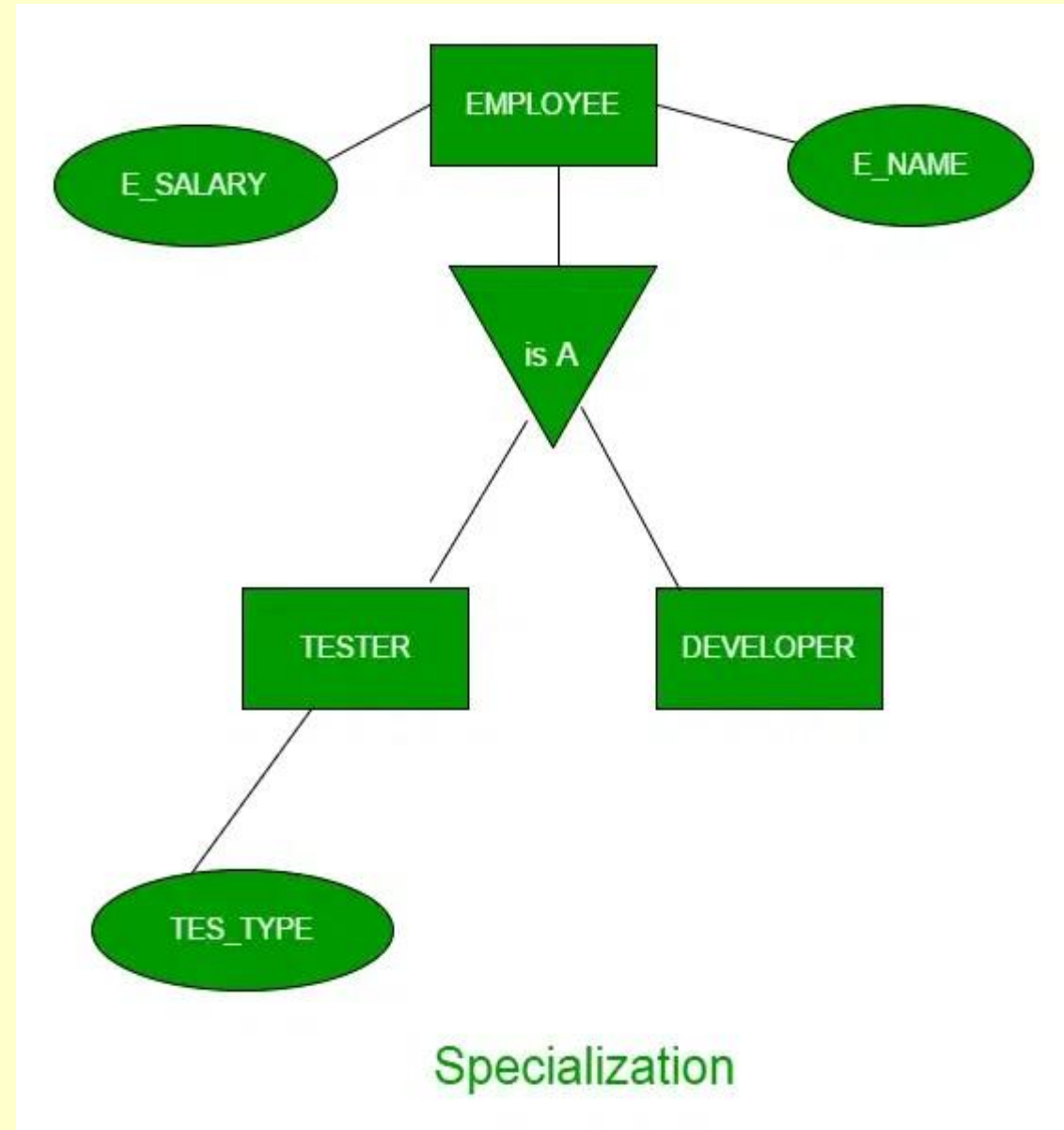
- Generalization is the process of extracting common properties from a set of entities and creating a generalized entity from it. It is a bottom-up approach in which two or more entities can be generalized to a higher-level entity if they have some attributes in common. For Example, STUDENT and FACULTY can be generalized to a higher-level entity called PERSON as shown in Figure 1. In this case, common attributes like P\_NAME, and P\_ADD become part of a higher entity (PERSON), and specialized attributes like S\_FEE become part of a specialized entity (STUDENT).
- Generalization is also called as ‘ Bottom-up approach’.





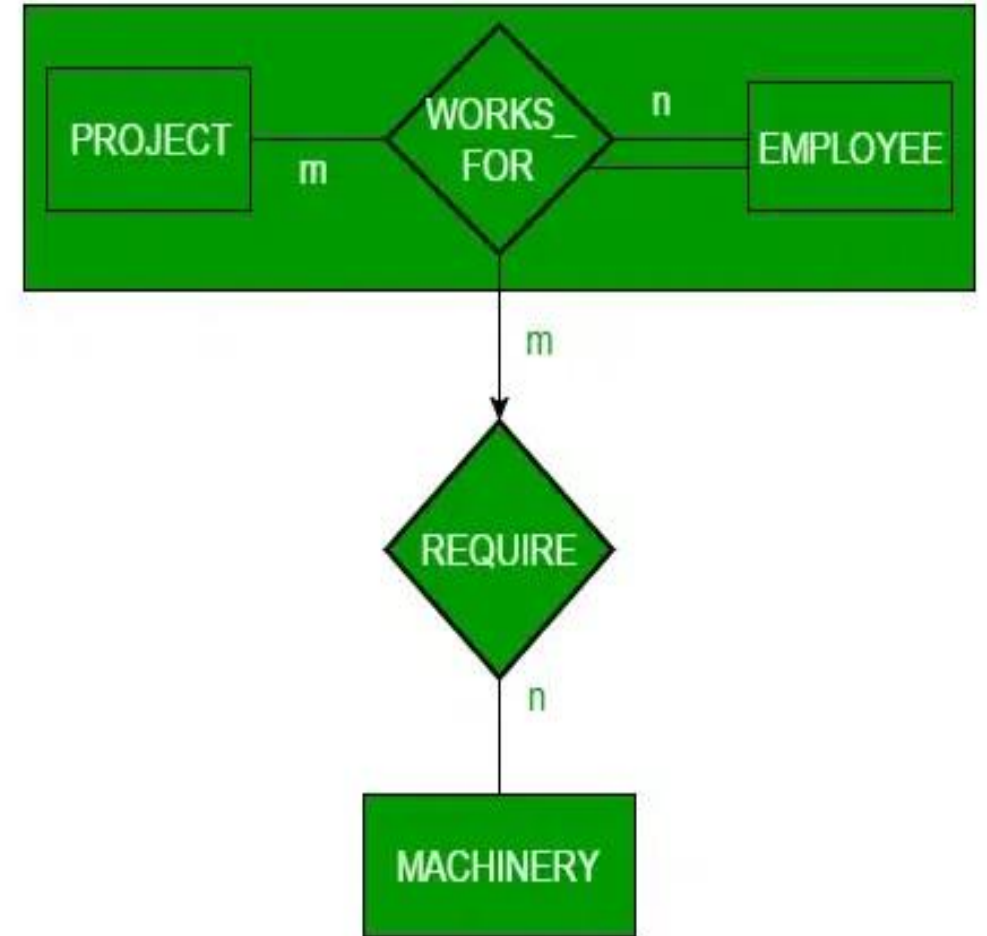
# Specialization

- In specialization, an entity is divided into sub-entities based on its characteristics. It is a top-down approach where the higher-level entity is specialized into two or more lower-level [entities](#). For Example, an EMPLOYEE entity in an Employee management system can be specialized into DEVELOPER, TESTER, etc. as shown in Figure 2. In this case, common attributes like E\_NAME, E\_SAL, etc. become part of a higher entity (EMPLOYEE), and specialized attributes like TES\_TYPE become part of a specialized entity (TESTER).
- Specialization is also called as " Top-Down approach".



# Aggregation

- An ER diagram is not capable of representing the relationship between an entity and a relationship which may be required in some scenarios. In those cases, a relationship with its corresponding entities is aggregated into a higher-level entity. Aggregation is an abstraction through which we can represent relationships as higher-level entity sets.
- For Example, an Employee working on a project may require some machinery. So, REQUIRE relationship is needed between the relationship WORKS\_FOR and entity MACHINERY. Using aggregation, WORKS\_FOR relationship with its entities EMPLOYEE and PROJECT is aggregated into a single entity and relationship REQUIRE is created between the aggregated entity and MACHINERY.



Aggregation

# How to Draw ER Diagram?

- The very first step is Identifying all the Entities, and place them in a Rectangle, and labeling them accordingly.
- The next step is to identify the relationship between them and place them accordingly using the Diamond, and make sure that, Relationships are not connected to each other.
- Attach attributes to the entities properly.
- Remove redundant entities and relationships.
- Add proper colors to highlight the data present in the database.

# Relational model

The background features a light yellow-green gradient. It is decorated with several vertical lines in shades of pink and purple. Scattered throughout are small squares in various colors, including teal, orange, pink, and light green. Some of these squares are solid, while others are outlined.

# Relational Model in DBMS

- E.F. Codd proposed the relational Model to model data in the form of relations or tables. After designing the conceptual model of the Database using [ER diagram](#), we need to convert the conceptual model into a relational model which can be implemented using any [RDBMS](#) language like Oracle SQL, MySQL, etc. So we will see what the Relational Model is.
- The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as relations. The relational model is an example of a record-based model. Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.

# What is the Relational Model?

- The relational model represents how data is stored in Relational Databases. A relational database consists of a collection of tables, each of which is assigned a unique name. Consider a relation STUDENT with attributes ROLL\_NO, NAME, ADDRESS, PHONE, and AGE shown in the table.

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	ARIF	DHAKA	0175123451	18
2	SARAH	COMILLA	0172431543	18
3	TOWHID	SYLHET	0176253131	20
4	MUJIB	DHAKA		18

# Important Terminologies

- **Attribute:** Attributes are the properties that define an entity.  
e.g.; **ROLL\_NO, NAME, ADDRESS**
- **Relation Schema:** A relation schema defines the structure of the relation and represents the name of the relation with its attributes. e.g.; STUDENT (ROLL\_NO, NAME, ADDRESS, PHONE, and AGE) is the relation schema for STUDENT. If a schema has more than 1 relation, it is called Relational Schema.
- **Tuple:** Each row in the relation is known as a tuple. The above relation contains 4 tuples, one of which is shown as:

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	ARIF	DHAKA	0175123451	18

# Important Terminologies

- **Relation Instance:** The set of tuples of a relation at a particular instance of time is called a relation instance. Table 1 shows the relation instance of STUDENT at a particular time. It can change whenever there is an insertion, deletion, or update in the database.
- **Degree:** The number of attributes in the relation is known as the degree of the relation. The **STUDENT** relation defined above has degree 5.
- **Cardinality:** The number of tuples in a relation is known as [cardinality](#). The **STUDENT** relation defined above has cardinality 4.
- **Column:** The column represents the set of values for a particular attribute. The column **ROLL\_NO** is extracted from the relation STUDENT.

ROLL_NO
1
2
3
4



# Important Terminologies

- **NULL Values:** The value which is not known or unavailable is called a NULL value. It is represented by blank space. e.g.; PHONE of STUDENT having ROLL\_NO 4 is NULL.
- **Relation Key:** These are basically the keys that are used to identify the rows uniquely or also help in identifying tables. These are of the following types.
  - **Primary Key** - A primary key constraint depicts a key comprising one or more columns that will help uniquely identify every tuple/record in a table.

## Properties :

- No duplicate values are allowed, i.e. Column assigned as primary key should have UNIQUE values only.
- NO NULL values are present in column with Primary key. Hence there is Mandatory value in column having Primary key.
- Only one primary key per table exist although Primary key may have multiple columns.
- No new row can be inserted with the already existing primary key.
- Classified as : a) Simple primary key that has a Single column 2) Composite primary key has Multiple column.

# Important Terminologies

- **Foreign Key** - A foreign key is a column or a group of columns used to identify a row uniquely of a different table. The table that comprises the foreign key is called the referencing table or child table. And the table to that the foreign key references is known as the referenced table or parent table. A table can possess multiple foreign keys according to its relationships with other tables.
- **Composite Key** - A composite key is made by the combination of two or more columns in a table that can be used to uniquely identify each row in the table when the columns are combined uniqueness of a row is guaranteed, but when it is taken individually it does not guarantee uniqueness, or it can also be understood as a primary key made by the combination of two or more attributes to uniquely identify every row in a table.

# Constraints in Relational Model

- While designing the Relational Model, we define some conditions which must hold for data present in the database are called Constraints. These constraints are checked before performing any operation (insertion, deletion, and updation ) in the database. If there is a violation of any of the constraints, the operation will fail.
- **Domain Constraints** - These are attribute-level constraints. An attribute can only take values that lie inside the domain range. e.g.; If a constraint  $AGE > 0$  is applied to STUDENT relation, inserting a negative value of AGE will result in failure.
- **Key Integrity** - Every relation in the database should have at least one set of attributes that defines a tuple uniquely. Those set of attributes is called keys. e.g.; ROLL\_NO in STUDENT is key. No two students can have the same roll number. So a key has two properties:
  - It should be unique for all tuples.
  - It can't have NULL values.

# Constraints in Relational Model

- **Referential Integrity**
- When one attribute of a relation can only take values from another attribute of the same relation or any other relation, it is called [referential integrity](#). Let us suppose we have 2 relations

ROLL_NO	NAME	ADDRESS	PHONE	AGE	BRANCH_CODE
1	ARIF	DHAKA	0175123451	18	CS
2	SARAH	COMILLA	0172431543	18	CS
3	TOWHID	SYLHET	0176253131	20	ECE
4	MUJIB	DHAKA		18	IT

BRANCH_CODE	BRANCH_NAME
CS	COMPUTER SCIENCE
IT	INFORMATION TECHNOLOGY
ECE	ELECTRONICS AND COMMUNICATION ENGINEERING
CV	CIVIL ENGINEERING

**BRANCH\_CODE** of STUDENT can only take the values which are present in **BRANCH\_CODE** of BRANCH which is called referential integrity constraint. The relation which is referencing another relation is called **REFERENCING RELATION** (STUDENT in this case) and the relation to which other relations refer is called **REFERENCED RELATION** (BRANCH in this case).

# Codd's Twelve Rules of Relational Database

- Codd rules were proposed by E.F. Codd which should be satisfied by the [relational model](#). Codd's Rules are basically used to check whether DBMS has the quality to become [Relational Database Management System \(RDBMS\)](#). But, it is rare to find that any product has fulfilled all the rules of Codd. They generally follow the 8-9 rules of Codd. E.F. Codd has proposed 13 rules which are popularly known as Codd's 12 rules. These rules are stated as follows:
- **Rule 0: Foundation Rule**— For any system that is advertised as, or claimed to be, a relational database management system, that system must be able to manage databases entirely through its relational capabilities.
- **Rule 1: Information Rule**— Data stored in the Relational model must be a value of some cell of a table.
- **Rule 2: Guaranteed Access Rule**— Every data element must be accessible by the table name, its primary key, and the name of the attribute whose value is to be determined.

# Codd's Twelve Rules of Relational Database

- **Rule 3: Systematic Treatment of NULL values**– NULL value in the database must only correspond to missing, unknown, or not applicable values.
- **Rule 4: Active Online Catalog**– The structure of the database must be stored in an online catalog that can be queried by authorized users.
- **Rule 5: Comprehensive Data Sub-language Rule**- A database should be accessible by a language supported for definition, manipulation, and transaction management operation.
- **Rule 6: View Updating Rule**- Different views created for various purposes should be automatically updatable by the system.
- **Rule 7: High-level insert, update and delete rule**- Relational Model should support insert, delete, update, etc. operations at each level of relations. Also, set operations like Union, Intersection, and minus should be supported.

# Codd's Twelve Rules of Relational Database

- **Rule 8: Physical data independence-** Any modification in the physical location of a table should not enforce modification at the application level.
- **Rule 9: Logical data independence-** Any modification in the logical or conceptual schema of a table should not enforce modification at the application level. For example, merging two tables into one should not affect the application accessing it which is difficult to achieve.
- **Rule 10: Integrity Independence-** Integrity constraints modified at the database level should not enforce modification at the application level.
- **Rule 11: Distribution Independence-** Distribution of data over various locations should not be visible to end-users.
- **Rule 12: Non-Subversion Rule- Low-level** access to data should not be able to bypass the integrity rule to change data.

# Advantages of the Relational Model

- **Simple model:** Relational Model is simple and easy to use in comparison to other languages.
- **Flexible:** Relational Model is more flexible than any other relational model present.
- **Secure:** Relational Model is more secure than any other relational model.
- **Data Accuracy:** Data is more accurate in the relational data model.
- **Data Integrity:** The integrity of the data is maintained in the relational model.
- **Operations can be Applied Easily:** It is better to perform operations in the relational model.



# Disadvantages of the Relational Model

- Relational Database Model is not very good for large databases.
- Sometimes, it becomes difficult to find the relation between tables.
- Because of the complex structure, the response time for queries is high.

# Characteristics of the Relational Model

- Data is represented in rows and columns called relations.
- Data is stored in tables having relationships between them called the Relational model.
- The relational model supports the operations like Data definition, Data manipulation, and Transaction management.
- Each column has a distinct name and they are representing attributes.
- Each row represents a single entity.

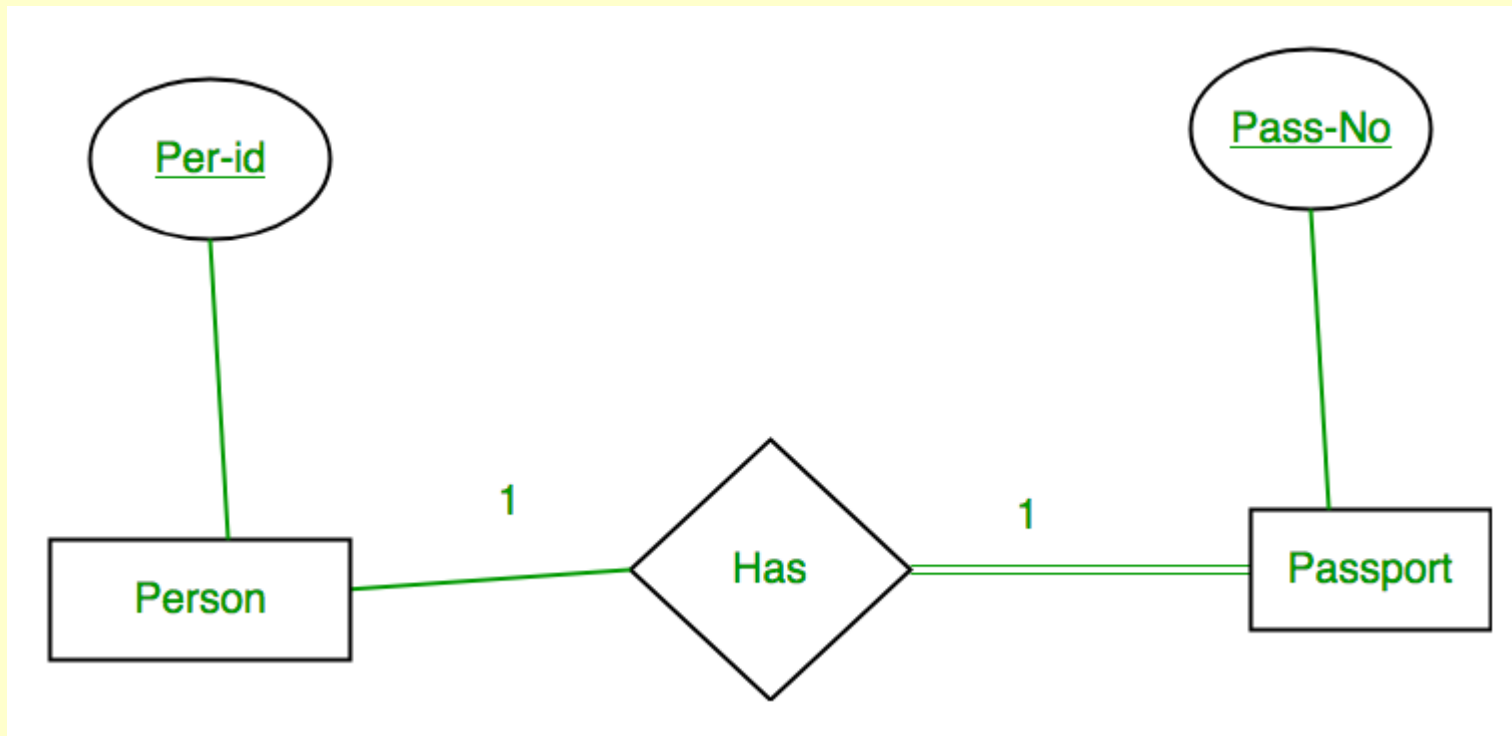
# Mapping from ER Model to Relational Model

- After designing the ER diagram of system, we need to convert it to Relational models which can directly be implemented by any RDBMS like Oracle, MySQL etc. In this article we will discuss how to convert ER diagram to Relational Model for different scenarios.

# Mapping from ER Model to Relational Model

- **Case 1: Binary Relationship with 1:1 cardinality with total participation of an entity**

E.g. ; A person has 0 or 1 passport number and Passport is always owned by 1 person. So it is 1:1 cardinality with full participation constraint from Passport.



# Mapping from ER Model to Relational Model

- **First Convert each entity and relationship to tables.** Person table corresponds to Person Entity with key as Per-Id. Similarly Passport table corresponds to Passport Entity with key as Pass-No. Has Table represents relationship between Person and Passport (Which person has which passport). So it will take attribute Per-Id from Person and Pass-No from Passport.

Person			Has			Passport	
<u>Per-Id</u>	Other Person Attribute		<u>Per-Id</u>	Pass-No		<u>Pass-No</u>	Other PassportAttribute
PR1	—		PR1	PS1		PS1	—
PR2	—		PR2	PS2		PS2	—
PR3	—						

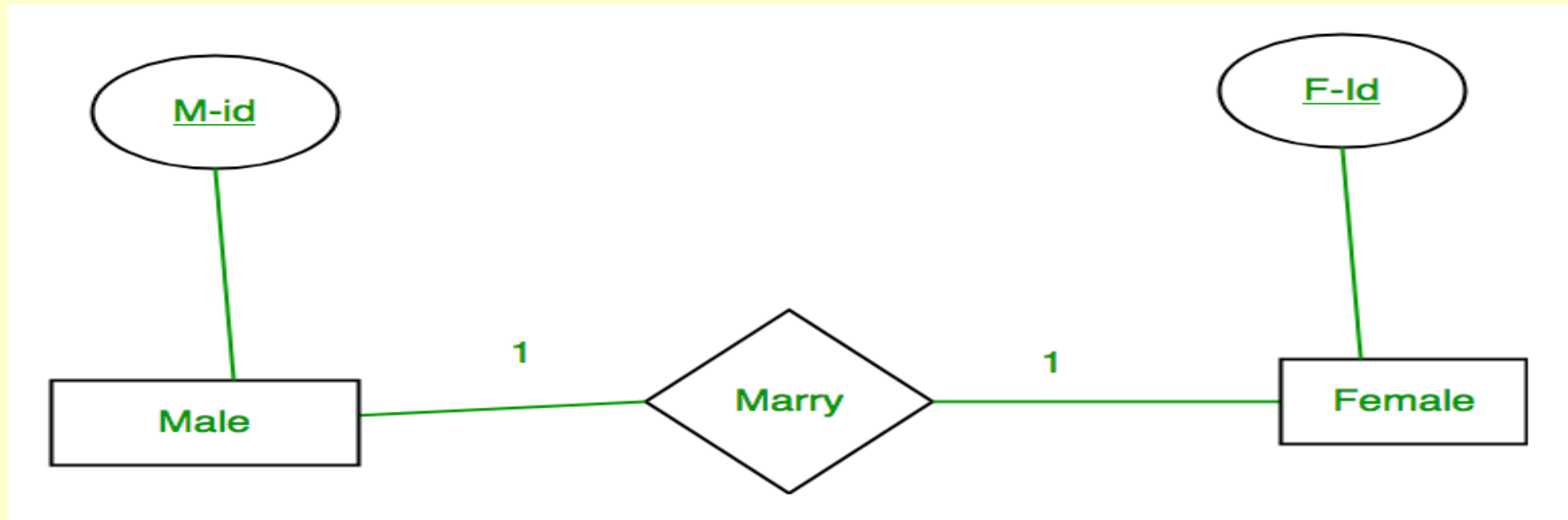
# Mapping from ER Model to Relational Model

- As we can see from Table, each Per-Id and Pass-No has only one entry in Has Table. So we can merge all three tables into 1 with attributes shown in Table. Each Per-Id will be unique and not null. So it will be the key. Pass-No can't be key because for some person, it can be NULL.

<u>Per-Id</u>	Other Person Attribute	Pass-No	Other Passport Attribute
---------------	---------------------------	---------	-----------------------------

# Mapping from ER Model to Relational Model

- **Case 2: Binary Relationship with 1:1 cardinality and partial participation of both entities**
- E.g. A male marries 0 or 1 female and vice versa as well. So it is 1:1 cardinality with partial participation constraint from both. First Convert each entity and relationship to tables. Male table corresponds to Male Entity with key as M-Id. Similarly Female table corresponds to Female Entity with key as F-Id. Marry Table represents relationship between Male and Female (Which Male marries which female). So it will take attribute M-Id from Male and F-Id from Female.



# Mapping from ER Model to Relational Model

Male			Marry			Female	
<u>M-Id</u>	Other Male Attribute		<u>M-Id</u>	F-Id		<u>F-Id</u>	Other Female Attribute
M1	—		M1	F2		F1	—
M2	—		M2	F1		F2	—
M3	—					F3	—

# Mapping from ER Model to Relational Model

- As we can see from Table, some males and some females do not marry. If we merge 3 tables into 1, for some M-Id, F-Id will be NULL. So there is no attribute which is always not NULL. So we can't merge all three tables into 1. We can convert into 2 tables. In table, M-Id who are married will have F-Id associated. For others, it will be NULL. Another Table will have information of all females. Primary Keys have been underlined.

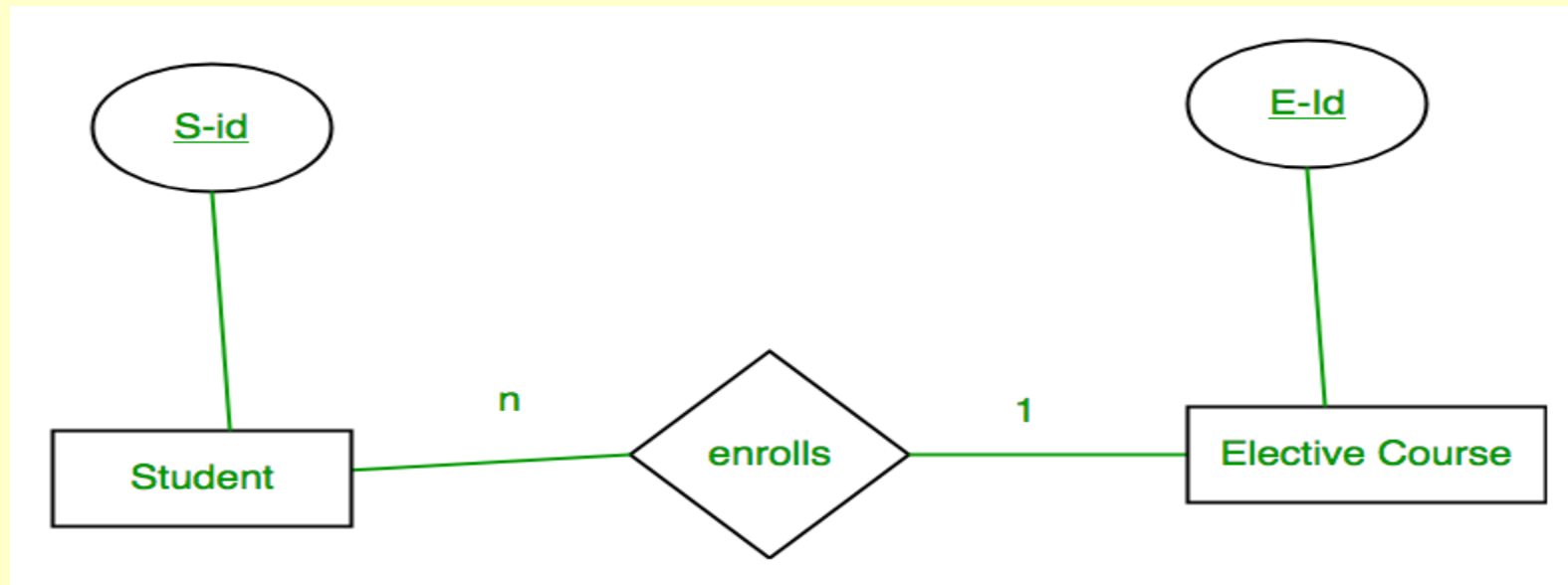
<u>M-Id</u>	Other Male Attribute	F-Id
<u>F-Id</u>	Other Female Attribute	

**Note:** Binary relationship with 1:1 cardinality will have 2 table if partial participation of both entities in the relationship. If at least 1 entity has total participation, number of tables required will be 1.



# Mapping from ER Model to Relational Model

- **Case 3: Binary Relationship with n: 1 cardinality**
- E.g. ; In this scenario, every student can enroll only in one elective course but for an elective course there can be more than one student. First Convert each entity and relationship to tables. Student table corresponds to Student Entity with key as S-Id. Similarly Elective\_Course table corresponds to Elective\_Course Entity with key as E-Id. Enrolls Table represents relationship between Student and Elective\_Course (Which student enrolls in which course). So it will take attribute S-Id from Student and E-Id from Elective\_Course.



# Mapping from ER Model to Relational Model

Student			Enrolls			Elective_Course	
<u>S-Id</u>	Other Student Attribute		<u>S-Id</u>	E-Id		<u>E-Id</u>	Other Elective CourseAttribute
S1	—		S1	E1		E1	—
S2	—		S2	E2		E2	—
S3	—		S3	E1		E3	—
S4	—		S4	E1			

# Mapping from ER Model to Relational Model

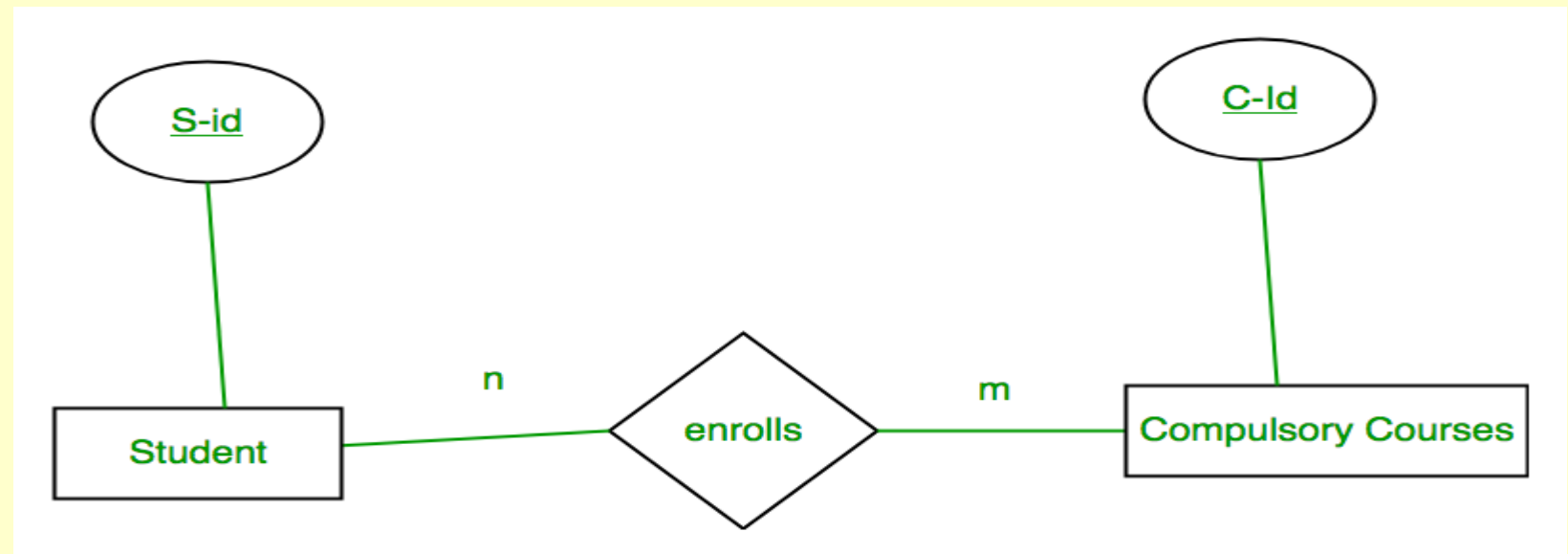
- As we can see from Table, S-Id is not repeating in Enrolls Table. So it can be considered as a key of Enrolls table. Both Student and Enrolls Table's key is same; we can merge it as a single table. The resultant tables are shown. Primary Keys have been underlined.

<u>S-Id</u>	Other Student Attribute	E-Id
-------------	----------------------------	------

<u>E-Id</u>	Other Elective CourseAttribute
-------------	--------------------------------

# Mapping from ER Model to Relational Model

- **Case 4: Binary Relationship with m: n cardinality**
- E.g. ; In this scenario, every student can enroll in more than 1 compulsory course and for a compulsory course there can be more than 1 student. First Convert each entity and relationship to tables. Student table corresponds to Student Entity with key as S-Id. Similarly Compulsory\_Courses table corresponds to Compulsory Courses Entity with key as C-Id. Enrolls Table represents relationship between Student and Compulsory\_Courses (Which student enrolls in which course). So it will take attribute S-Id from Person and C-Id from Compulsory\_Courses.



# Mapping from ER Model to Relational Model

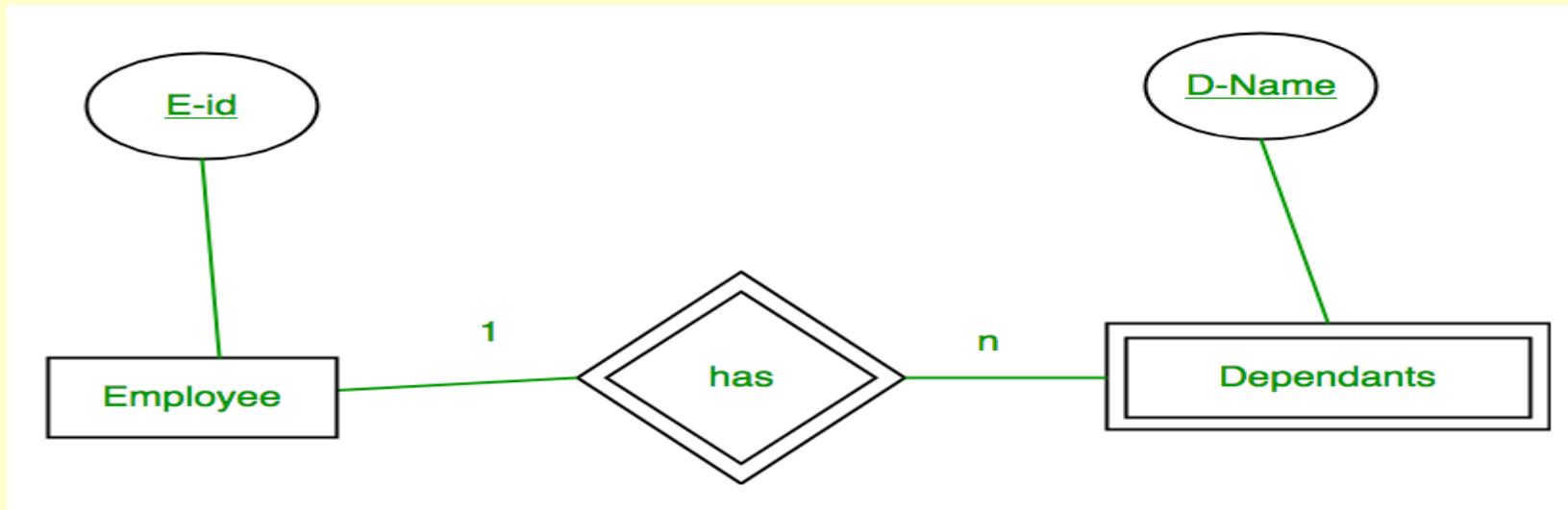
Student			Enrolls			Compulsory_Courses	
<u>S-Id</u>	Other Student Attribute		<u>S-Id</u>	<u>C-Id</u>		<u>C-Id</u>	Other Compulsory CourseAttribute
S1	—		S1	C1		C1	—
S2	—		S1	C2		C2	—
S3	—		S3	C1		C3	—
S4	—		S4	C3		C4	—
			S4	C2			
			S3	C3			

# Mapping from ER Model to Relational Model

- As we can see from Table, S-Id and C-Id both are repeating in Enrolls Table. But its combination is unique; so it can be considered as a key of Enrolls table. All tables' keys are different, these can't be merged. Primary Keys of all tables have been underlined.

# Mapping from ER Model to Relational Model

- **Case 5: Binary Relationship with weak entity**
- In this scenario, an employee can have many dependents and one dependent can depend on one employee. A dependent does not have any existence without an employee (e.g; you as a child can be dependent of your father in his company). So it will be a weak entity and its participation will always be total. Weak Entity does not have key of its own. So its key will be combination of key of its identifying entity (E-Id of Employee in this case) and its partial key (D-Name).
- First Convert each entity and relationship to tables. Employee table corresponds to Employee Entity with key as E-Id. Similarly Dependents table corresponds to Dependent Entity with key as D-Name and E-Id. Has Table represents relationship between Employee and Dependents (Which employee has which dependents). So it will take attribute E-Id from Employee and D-Name from Dependents.



# Mapping from ER Model to Relational Model

Employee			Has			Dependents		
<u>E-Id</u>	Other Employee Attribute		<u>E-Id</u>	<u>D-Name</u>		<u>D-Name</u>	<u>E-Id</u>	Other Dependent sAttribute
E1	–		E1	ARIF		ARIF	E1	–
E2	–		E1	SUDIP		SUDIP	E1	–
E3	–		E2	ARIF		ARIF	E2	–
			E3	SAZIA		SAZIA	E3	–



# Mapping from ER Model to Relational Model

- As we can see from Table, E-Id, D-Name is key for **Has** as well as Dependents Table. So we can merge these two into 1. So the resultant tables are shown. Primary Keys of all tables have been underlined.

<u>E-Id</u>	Other Employee Attribute
-------------	--------------------------

<u>D-Name</u>	<u>E-Id</u>	Other DependentsAttribute
---------------	-------------	---------------------------

Thank You

The background is a solid light yellow color. It is decorated with several small, solid-colored squares in teal, orange, and pink, scattered across the page. There are also thin, vertical pink lines of varying lengths positioned on the left, right, and center of the slide.

# Relational Algebra

---

# Preview

- Relational Algebra
- SELECT( $\sigma$ )
- Projection( $\pi$ )
- Rename ( $\rho$ )
- Union operation ( $\cup$ )
- Set Difference ( $-$ )
- Intersection
- Cartesian product( $\times$ )
- Join Operations
- Inner Join
- Theta Join
- EQUI join

# Relational Algebra

- **RELATIONAL ALGEBRA** is a widely used **procedural query language**. It **collects instances of relations as input and gives occurrences of relations as output**. It uses various operations to perform this action. SQL Relational algebra query operations are performed recursively on a relation. The output of these operations is a new relation, which might be formed from one or more input relations.

# Relational Operations

- **Unary Relational Operations**

- SELECT (symbol:  $\sigma$ )
- PROJECT (symbol:  $\pi$ )
- RENAME (symbol:  $\rho$ )

- **Relational Algebra Operations From Set Theory**

- UNION ( $\cup$ )
- INTERSECTION ( $\cap$ ),
- DIFFERENCE ( $-$ )
- CARTESIAN PRODUCT ( $\times$ )

# Relational Operations

- **Binary Relational Operations**
  - JOIN
  - DIVISION

# Select ( $\sigma$ )

The SELECT operation is used for selecting a subset of the tuples according to a given selection condition. **Sigma( $\sigma$ )** Symbol denotes it. It is used as an expression to choose tuples which meet the selection condition. Select operator **selects tuples that satisfy a given predicate.**

**$\sigma_p(r)$**

$\sigma$  is the predicate

$r$  stands for relation which is the name of the table

$p$  is propositional logic



## Select ( $\sigma$ )

**$\sigma$  topic = "Database" (Tutorials)**

Selects tuples from Tutorials where topic = 'Database'.

**$\sigma$  topic = "Database" and author = "Chaplin" (Tutorials)**

Selects tuples from Tutorials where the topic is 'Database' and 'author' is Chaplin

**$\sigma$  sales > 50000 (Customers)**

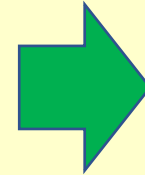
Selects tuples from Customers where sales is greater than 50000

# Projection( $\pi$ )

- The projection eliminates all attributes of the input relation but those mentioned in the projection list. The projection method defines a relation that contains a vertical subset of Relation.
- This helps to extract the values of specified attributes. **( $\pi$ ) symbol is used to choose attributes from a relation.** This operator helps you to keep specific columns from a relation and discards the other columns.
- Also, **all the duplicate tuples are removed from the relation** in the resulting relation. This is called as Duplicate elimination.

# Projection( $\pi$ ) - Example

<u>Customers</u>		
CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive
4	Alibaba	Active



<u>Output</u>	
CustomerName	Status
Google	Active
Amazon	Active
Apple	Inactive
Alibaba	Active

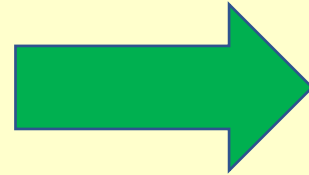
$\Pi$  CustomerName, Status (**Customers**)

# Projection( $\pi$ ) - Example

## Faculty

Class	Dept	Position
-------	------	----------

5	CSE	Assistant Professor
5	CSE	Assistant Professor
6	EE	Assistant Professor
6	EE	Assistant Professor



## Output

Class	Dept
-------	------

5	CSE
6	EE

$\Pi$  Class, Dept (**Faculty**)

# Rename ( $\rho$ )

- The RENAME operation is used to rename the relation, attribute or both.
- Sometimes it is simple and suitable to break a complicated sequence of operations and rename it as a relation with different names. Reasons to rename a relation can be many, like –
  - We may want to save the result of a relational algebra expression as a relation so that we can use it later.
  - We may want to join a relation with itself, in that case, it becomes too confusing to specify which one of the tables we are talking about, in that case, we rename one of the tables and perform join operations on them.
  - **$\rho$  X (R)** - Where the symbol ' $\rho$ ' is used to denote the RENAME operator and R is the result of the sequence of operation or expression which is saved with the name X.

# Rename ( $\rho$ ) - Examples

- **Rename a relation**

- Suppose we have a relation named ***Students*** and we want to change it to ***FinalYrStudents***, the rename operation works as follows:

$$\rho_{FinalYrStudents}(\mathbf{Students})$$

- **Rename an attribute**

- Suppose we have a relation named **Students** and we want to change its attributes ***StudentID***, ***StudentName*** to ***SID*** and ***SName***, the rename operation works as follows:

$$\rho_{(SID,SName)}(\mathbf{Students})$$

The attributes will be ordered as same as the tables and must be renamed in the same order. If only selective attributes are to be changed, rewrite the original attribute names of those that are supposed to be unchanged.

# Rename ( $\rho$ ) - Examples

- **Rename both**

- Next, we'll change both the relation name and attributes of the *Students* class:

$\rho_{FinalYrStudents(SID, SName)}(\mathbf{Students})$

Once relations and attributes are renamed, they can be used in the **projection** and **selection** operations to access the relations.

# Union Operation (u)

- UNION is symbolized by U symbol. It includes all tuples that are in tables A or in B. It also eliminates duplicate tuples. So, set A UNION set B would be expressed as:

$$A \cup B$$

- For a union operation to be valid, the following conditions must hold –
  - R and S must be the same number of attributes.
  - Attribute domains need to be compatible.
  - Duplicate tuples should be automatically removed.



# Union Operation ( $\cup$ ) - Examples

Consider the following tables.

Table A

column 1	column 2
1	1
1	2

Table B

column 1	column 2
1	1
1	3

Table A  $\cup$  B

column 1	column 2
1	1
1	2
1	3

# Set Difference (-)

– Symbol denotes it. The result of A – B, is a relation which includes **all tuples that are in A but not in B**.

- The attribute name of A has to match with the attribute name in B.
- The two-operand relations A and B should be either compatible or Union compatible.
- It should be defined relation consisting of the tuples that are in relation A, but not in B.

<u>Table A – B</u>	
column 1	column 2
1	2

# Intersection

An intersection is defined by the symbol  $\cap$

$$A \cap B$$

Defines a relation consisting of a set of all tuple that are in both A and B.  
However, A and B must be union-compatible.

Table  $A \cap B$

column 1	column 2
----------	----------

1	1
---	---

# Cartesian Product(X)

Cartesian Product in DBMS is an operation used to merge columns from two relations. Generally, a cartesian product is never a meaningful operation when it performs alone. However, it becomes meaningful when it is followed by other operations. It is also called Cross Product or Cross Join.

Example – Cartesian product  
 $\sigma_{\text{column 2} = '1'} (A \times B)$

Table A

column 1	column 2
1	1
1	2

Table B

column 3	column 4
1	1
1	3

# Cartesian Product(X)

(A X B)			
column 1	column 2	column 3	column 4
1	1	1	1
1	1	1	3
1	2	1	1
1	2	1	3

Output – Below is shown all rows from relation A and B whose column 2 has value 1

<u>σ column 2 = '1' (A X B)</u>			
column 1	column 2	column 3	column 4
1	1	1	1
1	1	1	3

# Cartesian Product(X)

Generally, a Cartesian product is never a meaningful operation when it performs alone.

Name	Age	Sex
Ram	14	M
Sona	15	F
Kim	20	M

ID	Course
1	DS
2	DBMS

Name	Age	Sex	ID	Course
R	14	M	1	DS
R	14	M	2	DBMS
S	15	F	1	DS
S	15	F	2	DBMS
K	20	M	1	DS
K	20	M	2	DBMS

# Derived Operators

**1. Natural Join( $\bowtie$ ):** Natural join is a binary operator. Natural join between two or more relations will result in a set of all combinations of tuples where they have an equal common attribute.

Name	ID	Dept_Name
A	120	IT
B	125	HR
C	110	Sales
D	111	IT

EMP

Dept_Name	Manager
Sales	Y
Production	Z
IT	A

DEPT

Name	ID	Dept_Name	Manager
A	120	IT	A
C	110	Sales	Y
D	111	IT	A

EMP  $\bowtie$  DEPT

# Derived Operators

**2. Conditional Join (Theta Join):** Conditional join works similarly to natural join. In natural join, by default condition is equal between common attributes while in conditional join we can specify any condition such as greater than, less than, or not equal.

ID	Sex	Marks	ID	Sex	Marks
1	F	45	10	M	20
2	F	55	11	M	22
3	F	60	12	M	59

R

S

$R \bowtie R.Marks \geq S.Marks (B)$

R.ID	R.Sex	R.Marks	S.ID	S.Sex	S.Marks
1	F	45	10	M	20
1	F	45	11	M	22
2	F	55	10	M	20
2	F	55	11	M	22
3	F	60	10	M	20
3	F	60	11	M	22
3	F	60	12	M	59



# Derived Operators

**3. EQUI join** - When a theta join uses only equivalence condition, it becomes a equi join.

$$R \bowtie R.Marks = S.Marks (B)$$

## Inner Join

- All the above three types of joins are called Inner Joins. In an inner join, only those tuples that satisfy the matching criteria are included, while the rest are excluded.

# Derived Operators

## OUTER JOIN

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

- Left Outer Join( $A \bowtie B$ )
- In the left outer join, operation allows keeping all tuple in the left relation. However, if there is no matching tuple is found in right relation, then the attributes of right relation in the join result are filled with null values.

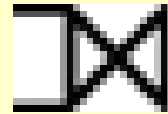


# Derived Operators

Consider the following 2 Tables

A	
Num	Square
2	4
3	9
4	16

B	
Num	Cube
2	8
3	18
5	75



A Left Outer Join B

Num	Square	Cube
2	4	8
3	9	18
4	16	—

# Derived Operators

- **Right Outer Join (  $A \bowtie_r B$  )**
- In the right outer join, operation allows keeping all tuple in the right relation. However, if there is no matching tuple is found in the left relation, then the attributes of the left relation in the join result are filled with null values.

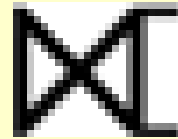


# Derived Operators

Consider the following 2 Tables

A	
Num	Square
2	4
3	9
4	16

B	
Num	Cube
2	8
3	18
5	75



A Right Outer Join B

Num	Cube	Square
2	8	4
3	18	9
5	75	—

# Derived Operators

## Full Outer Join ( $A \bowtie B$ )

In a full outer join, all tuples from both relations are included in the result, irrespective of the matching condition.

Consider the following 2 Tables

A	
Num	Square
2	4
3	9
4	16

B	
Num	Cube
2	8
3	18
5	75



A Full Outer Join B

Num	Square	Cube
2	4	8
3	9	18
4	16	–
5	–	75

**Q) Consider the following schema:**

**Suppliers** (sid : integer, sname : string, address : string)

**Parts** (pid : integer, pname : string, color : string)

**Catalog** (sid : integer, pid : integer, cost : real)

The key fields are underlined and domain of each field is listed after the field name. **Note - CONVENTION USED : \$ AS NATURAL JOIN**

1) Find the name of suppliers who supply some red parts

We first find the pids of parts that are red in color and then we compute the natural join of this with catalog from this we project sid which gives ids of the supplier who supply some red part, then we take the natural join of this with supplier and project names which gives us the names of suppliers who supply some red part

Step 1 :  $R1 = \pi_{pid} (\sigma_{color = 'red'} (\mathbf{parts}))$

Step 2 :  $R2 = \pi_{sid} (R1 \Join Catalog)$

Step 3 :  $R3 = \pi_{name} (R2 \Join Suppliers)$

Required answer is R3

## 2) Find the sids of suppliers who supply some red or green parts

Step1:  $R1 = \pi_{pid} (\sigma_{color = 'red' \vee 'green'} (parts))$

Step 2 :  $R2 = \pi_{sid} (R1 \bowtie Catalog)$

Same as above one but here we have to choose red or green parts and we have to have sids of suppliers so we can stop after step 2 after choosing parts either in red color or green color.

## 3) Find the sids of suppliers who supply some red part or are at 221 packer Ave

- Sids of suppliers who supply some red part

Step 1 :  $R1 = \pi_{pid} (\sigma_{color = 'red'} (parts))$

Step 2 :  $R2 = \pi_{sid} (R1 \bowtie Catalog)$

- Sids of suppliers who are at 221 packer Ave

Step 1 :  $R3 = \pi_{sid} (\sigma_{address = '221 packer Ave'} (Suppliers))$

Therefore sids of suppliers who supply some red part or are at 221 packer Ave  
Is  $R2 \cup R3$



**4) Find the sids of suppliers who supply some red part and some green part**

$R1 = \pi_{sid}(\pi_{pid}(\sigma_{color = 'red'}(parts)) \bowtie Catalog)$

$R2 = \pi_{sid}(\pi_{pid}(\sigma_{color = 'green'}(parts)) \bowtie Catalog)$

From question one we get the sids of suppliers who supply some red part (R1)

Similarly R2 is the sids of suppliers who supply some green part

Required list of sids who supply some red and some green part is R1

Intersection R2

- **Question:**
- Consider the following relational database schema consisting of the four relation schemas:
- **passenger** ( pid, pname, pgender, pcity)
- **agency** ( aid, aname, acity)
- **flight** ( fid, fdate, time, src, dest)
- **booking** ( pid, aid, fid, fdate)
- Answer the following questions using relational algebra queries;

**a) Get the complete details of all flights to New Delhi.**

- $\sigma_{destination = "New Delhi"}(flight)$

**b) Get the details about all flights from Chennai to New Delhi.**

- $\sigma_{src = "Chennai" \wedge dest = "New Delhi"}(flight)$

c) Find only the flight numbers for passenger with pid 123 for flights to Chennai before 06/11/2020.

- $\Pi_{fid} (\sigma_{pid = 123} (\text{booking}) \bowtie \sigma_{dest = \text{"Chennai"} \wedge fdate < 06/11/2020} (\text{flight}))$
- ***Hint: Given conditions are pid, dest, and fdate. To get the flight id for a passenger given a pid, we have two tables flight and booking to be joined with necessary conditions. From the result, the flight id can be projected***

d) Find the passenger names for passengers who have bookings on at least one flight.

- $\Pi_{pname} (\text{passenger} \bowtie \text{booking})$

e) Find the passenger names for those who do not have any bookings in any flights.

- $\Pi_{pname} ((\Pi_{pid} (\text{passenger}) - \Pi_{pid} (\text{booking})) \bowtie \text{passenger})$
- ***Hint: here applied a set difference operation. The set difference operation returns only pids that have no booking. The result is joined with passenger table to get the passenger names.***

**f) Find the agency names for agencies that located in the same city as passenger with passenger id 123**

- $\Pi_{aname} (\text{agency} \bowtie_{acity = pcity} (\sigma_{pid = 123} (\text{passenger})))$
- **[Hint: we performed a theta join on equality conditions (equi join) here. This is done between details of passenger 123 and the agency table to get the valid records where the city values are same. From the results, aname is projected.]**

**g) Get the details of flights that are scheduled on both dates 01/12/2020 and 02/12/2020 at 16:00 hours.**

- $(\sigma_{fdate = 01/12/2020 \wedge time = 16:00} (\text{flight})) \cap (\sigma_{fdate = 02/12/2020 \wedge time = 16:00} (\text{flight}))$

**[Hint: the requirement is for flight details for both dates in common. Hence, set intersection is used between the temporary relations generated from application of various conditions.]**

Thank You

Structured Query Language

**SQL**

# What is SQL?

- SQL is a short-form of the structured query language, and it is pronounced as S-Q-L or sometimes as See-Quell.
- This database language is mainly designed for maintaining the data in relational database management systems. It is a special tool used by data professionals for handling structured data (data which is stored in the form of tables).
- You can easily create and manipulate the database, access and modify the table rows and columns, etc. This query language became the standard of ANSI in the year of 1986 and ISO in the year of 1987.

# Why SQL?

- The basic use of SQL for data professionals and SQL users is to insert, update, and delete the data from the relational database.
- SQL allows the data professionals and users to retrieve the data from the relational database management systems.
- It also helps them to describe the structured data.
- It allows SQL users to create, drop, and manipulate the database and its tables.
- It also helps in creating the view, stored procedure, and functions in the relational database.
- It allows you to define the data and modify that stored data in the relational database.
- It also allows SQL users to set the permissions or constraints on table columns, views, and stored procedures.



# Applications of SQL

Execute different database queries against a database.

Define the data in a database and manipulate that data.

Create data in a relational database management system.

Access data from the relational database management system.

Create and drop databases and tables.

Create and maintain database users.

Create view, stored procedure, functions in a database.

Set permissions on tables, procedures and views.

# Data types in SQL

A data type is an attribute that specifies the type of data that the object can hold:

integer data,  
character data,  
monetary data,  
date and time data,  
binary strings, and so on

# Types of SQL Statements

**Here we have attached five types of widely used SQL queries.**

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Transaction Control Language (TCL)
- Data Control Language (DCL)
- Data Query Language (DQL)

# Types of SQL Statements

- **What is DDL?** DDL→ Data definition language
- Helps to define the database schema
- Deals with a description of the database
- Also, have the power to deal with creating and modifying the structure of database object
- **Types of DDL commands**
- **CREATE** – This command is used to create the database. It also has some objects like table, index, function, views, store procedure, and triggers
- **DROP** – DROP is mainly used to delete objects from the database
- **ALTER** –ALTER is used to alter the structure of the database
- **TRUNCATE** – TRUNCATE command is used to remove all records from a table which includes all memory allocated for the records are removed.
- **COMMENT** – It is used to add comments to the data dictionary
- **RENAME**– This is used to rename an object existing in the database.

# Types of SQL Statements

- **What is a DML command?**
- DML→ Data Manipulation Language
- This DML command handles all the data manipulation part
- It includes the most important parts of the SQL
- **Types of DML commands**
- INSERT – INSERT is used to insert the data in the table
- UPDATE – Helps to update existing data within a table
- DELETE – To delete records from a database table

# Types of SQL Statements

- **What is a DQL command? DQL→ Data Query Language**
- DQL is used to make queries on the data within schema objects
- The main focus of the DQL Command is to get some schema relation based on the query passed into it
- **SELECT→**This command is used to retrieve all the data from the table
- **What is a DCL command? DCL→ Data Control Language**
- Deals with the rights and permission of the database
- Works for the controlling part of the data
- **GRANT→** provide user's access privileges to the database.
- **REVOKE→**Helps to withdraw the user's access privileges given by using the GRANT command.

# Types of SQL Statements

- **What is a TCL command and give an example of TCL?**
- TCL→ Transaction Control Language
- COMMIT– commits a Transaction
- ROLLBACK – rollbacks a transaction for any error that occurs
- SAVEPOINT– use a savepoint within a transaction
- SET TRANSACTION–specify the characteristics for the transaction

# Various Syntax in SQL

 id	book_title	book_type	publish_year	copies_sold
1	Wake Up Sid!	original	2018	140
2	Trip to Hawaii	translated	2020	120
3	Eat, Pray, Love	original	2019	230
4	Life of my Dream	original	2018	190
5	Orange is the new Black	translated	2019	170
6	My Last Words	original	2019	150



# Various Syntax in SQL

## SQL SELECT Statement

```
SELECT column1, column2....columnN  
FROM table_name;
```

Eg 1.

```
1 SELECT * FROM Books
```


The \* (asterisk) symbol signifies “everything, all columns”.

```
1 SELECT book_title, publish_year, copies_sold FROM Books
```

This time, **SELECT book\_title, publish\_year, copies\_sold** means “select the specified columns”. The entire SQL query will show all the book titles alongside the corresponding year they were published and the number of copies that were sold from the table **Books**.

# Various Syntax in SQL

When I run this query in our database, as a result, we'll get something like this:

 book_title	publish_year	copies_sold
Wake Up Sid!	2018	140
Trip to Hawaii	2020	120
Eat, Pray, Love	2019	230
Life of my Dream	2018	190
Orange is the new Black	2019	170
My Last Words	2019	150

# Various Syntax in SQL

- **How to Filter SQL SELECT statement?**
- If you have a table with a large number of rows, you may want to limit the number of rows returned by the query. Let me take the above **Books** table to explain the filter functions like **WHERE**, **LIKE**, and **TOP** with our **SELECT** statement.
- Before diving into the details, let's take a glance at the syntax of the commands using WHERE, LIKE and TOP functions.
- **WHERE-**
  - *SELECT column\_name1,..., column\_name(n) FROM table\_name WHERE condition;*
- **LIKE-**
  - *SELECT column\_name1,..., column\_name(n) FROM table\_name WHERE condition LIKE pattern;*
- **TOP-**
  - *SELECT TOP (n) FROM table\_name;*

# Various Syntax in SQL

Now let's assume we want the list of books that were translated. So, I will write the below query to get the result-

```
1 SELECT id, book_title, book_type FROM Books WHERE book_type = 'translated'
```

Here is the resulting table-

!	id	book_title	book_type
2		Trip to Hawaii	translated
5		Orange is the new Black	translated

# Various Syntax in SQL

**Limitation of Where Clause** - It can't be used with an aggregate function directly. Because where clauses are evaluated row-by-row, where an aggregate function works on multiple rows to return a single result.

Employee ID	Name	Department	CTC(in Lacs)
1001	Ajay	Engineering	25
1002	Babloo	Engineering	23
1003	Chhavi	HR	15
1005	Evina	Marketing	16

```
SELECT Employee ID, Name,  
Department, CTC  
FROM Employee  
WHERE CTC > AVG (CTC); - WRONG
```

```
SELECT Employee ID, Name,  
Department, CTC  
FROM Employee  
WHERE CTC >= (SELECT AVG (CTC)  
FROM Employee); - RIGHT
```

# Various Syntax in SQL

Operators	Description
=	Equal to
< > , !=	Not Equal to
>	Greater than
>=	Greater than or Equal
<	Less than
<=	Less than or Equal

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulo

# Various Syntax in SQL

Now, for example, we want to retrieve the book whose title starts with the letter 'E'. Here I will write a query using the LIKE clause to get the desired output-

```
1 SELECT id, book_title FROM Books WHERE book_title LIKE 'E%'
```

Here is the resulting table-

⋮	id	book_title
3		Eat, Pray, Love

# Various Syntax in SQL

## Wildcard Characters used with LIKE

<u>Wildcard Character</u>	<u>Description</u>
LIKE 'n%'	Find the values that start with n
LIKE '%g'	Find the values that end with g
LIKE '%learn%'	Find the value that has to learn in any position
LIKE '_n%'	Find the letter that has n at the second position
LIKE 'n_%'	Find the word that starts with n but has at least two characters in length
LIKE 'n%g'	Find the word that starts with n and ends with g



# Various Syntax in SQL

Next, we can use the following query to obtain first two entries from the table.

```
1 SELECT TOP (2) * FROM Books
```

Here is the resulting table-

!	id	book_title	book_type	publish_year	copies_sold
1		Wake Up Sid!	original	2018	140
2		Trip to Hawaii	translated	2020	120

# Various Syntax in SQL

At the same time, we can limit the output of the SQL SELECT statement with a percent value. Such as, the following query returns only 40 percent of the total data set.

```
1 SELECT TOP(40) PERCENT * FROM Books
```

Here is the resulting table-

id	book_title	book_type	publish_year	copies_sold
1	Wake Up Sid!	original	2018	140
2	Trip to Hawaii	translated	2020	120
3	Eat, Pray, Love	original	2019	230

# Various Syntax in SQL


- Moving ahead, let's check out the usage of the DISTINCT function with the SELECT statement.
- **How to use DISTINCT with SELECT statement?**
- A column in a table often has multiple duplicate values, and you may only wish to list the different (distinct) values. Only distinct (different) data is returned with the DISTINCT command.
- Let's see the syntax of such statements—
- *SELECT DISTINCT column\_name1,...,  
column\_name(n) FROM table\_name;*

# Various Syntax in SQL

For instance, I want to select the DISTINCT year from our table-

```
1 SELECT DISTINCT publish_year FROM Books
```

We will get the below table as the result-

 publish_year
2018
2019
2020

# Various Syntax in SQL

- You might want to display the rows in a different order than SQL Server does when it produces the results. The SQL **ORDER BY** clause can be used to do this.
- Following is the syntax to write a query using **ORDER BY** clause-
  - *SELECT column\_name1,..., column\_name(n)*
  - *FROM table\_name*
  - *ORDER BY column\_name1,..., column\_name(n) ASC/DESC;*

# Various Syntax in SQL

- Now let us take an example and return rows in ascending order of titles of the books.

```
1 SELECT id, book_title, publish_year
2 FROM Books
3 ORDER BY book_title ASC;
```

- So, the following is the output that we will get from the above query

	id	book_title	publish_year
3		Eat, Pray, Love	2019
4		Life of my Dream	2018
6		My Last Words	2019
5		Orange is the new Black	2019
2		Trip to Hawaii	2020
1		Wake Up Sid!	2018

# Various Syntax in SQL

- Taking the previous example, I will now try to return the rows in the descending order of the titles.

```
SELECT id, book_title, publish_year  
FROM Books  
ORDER BY book_title DESC;
```

- So, the following is the output that we will get from the above query

id	book_title	publish_year
1	Wake Up Sid!	2018
2	Trip to Hawaii	2020
5	Orange is the new Black	2019
6	My Last Words	2019
4	Life of my Dream	2018
3	Eat, Pray, Love	2019

# Various Syntax in SQL

- You have to keep the below points in mind when using the **ORDER BY** clause-
- You can sort on one or more columns and in either ascending (**ASC**) or descending order (**DESC**).
- By default, the **ORDER BY** clause sorts the records in the column(s) in ascending order.
- To sort the records in decreasing order, use the **DESC** keyword.



# Various Syntax in SQL

- Moving on in this article, let us understand how to use SQL SELECT with the GROUP BY clause.
- **How to use GROUP BY with SELECT statement?**
- Instead of individual detail rows from a database, the **GROUP BY** clause can be used to provide aggregated results.
- *SELECT column\_name1,..., <aggregate function>column\_name (n)*
- *FROM table\_name*
- *GROUP BY column\_name1,..., column\_name (n) ;*
- Note that to group the result set by one or more columns, the GROUP BY statement is generally used with aggregate functions like **MAX ()**, **MIN ()**, **COUNT ()**, **SUM ()**, **AVG ()**, etc.

# Various Syntax in SQL

Now suppose I want the list of the number of books published each year.

```
1 SELECT COUNT(book_title) AS row_count, publish_year
2 FROM Books
3 GROUP BY publish_year;
```

You can check out the output table below-

i	row_count	publish_year
2		2018
3		2019
1		2020

You can see that a new name – row\_count – has been assigned to the result using the AS keyword. “Assigning an alias” is the term for this.

# Various Syntax in SQL

Note that to group the result set by one or more columns, the **GROUP BY** statement is generally used with aggregate functions like **MAX ()**, **MIN ()**, **COUNT ()**, **SUM ()**, **AVG ()**, etc.

Note that for every column that isn't a constant and isn't used inside an aggregation function, you should put it in the **GROUP BY** clause. If I add book type to the **SELECT** but not to the **GROUP BY** in the prior query, for example, I will get error.

# Various Syntax in SQL

- The **WHERE** clause can be used to filter individual rows. *But what if you wish to filter on the output of an aggregated function?* This is not possible in the **WHERE** clause because such results do not exist in the original table. We can utilize the **HAVING** clause to accomplish this.
- *SELECT column\_name1,..., column\_name(n)*
- *FROM table*
- *WHERE condition*
- *GROUP BY column\_name1,..., column\_name (n)*
- *HAVING condition*
- *ORDER BY column\_name1,..., column\_name(n) ASC/DESC;*

# Various Syntax in SQL

- For instance, I want the list of the number of books published each year where the number of books is  $> 1$  and are sorted in descending order.

```
1 SELECT COUNT(book_title) AS row_count, publish_year
2 FROM Books
3 GROUP BY publish_year
4 HAVING COUNT(book_title) > 1
5 ORDER BY COUNT(book_title) DESC;
```

Here is the output table-

	row_count	publish_year
3		2019
2		2018

# Various Syntax in SQL

- **SQL IN Operator**

```
SELECT column_names  
FROM table_names  
WHERE column_name IN (value_1, value_2, value_3,.....,value_n);
```

```
SELECT column_names  
FROM table_name  
WHERE condition IN (SELECT query);
```

# Various Syntax in SQL

<u>Employee ID</u>	<u>Name</u>	<u>Gender</u>	<u>DepartmentID</u>	<u>Education</u>	<u>Month of Joining</u>	<u>CTC(in Lacs)</u>
1001	Ajay	M	1	Doctoral	January	25
1002	Babloo	M	1	UG	February	23
1003	Chhavi	F	2	P	March	15
1004	Dheeraj	M	2	UG	January	12
1005	Evina	F	3	UG	March	16
1006	Fredy	M	4	UG	December	10
1007	Garima	F	4	PG	March	10
1008	Hans	M	5	PG	November	8
1009	Ivanka	F	5	Intermediate	June	7
1010	Jai	M	6		December	4

<u>DepartmentID</u>	<u>Department Name</u>
1	Engineering
2	HR
3	Marketing
4	Sales
5	Admin
6	Peon

# Various Syntax in SQL

## • SQL IN Operator

```
SELECT EmployeeID, Name, DepartmentID  
FROM Employee  
WHERE CTC IN (15, 12, 10);
```

EmployeeID	Name	DepartmentID
1003	Chhavi	2
1004	Dheeraj	2
1006	Fredy	4
1007	Garima	4

```
SELECT EmployeeID, Name, Month of Joining  
FROM Employee  
WHERE Month of Joining  
IN ('January', 'February', 'November');
```

EmployeeID	Name	Month of Joining
1001	Ajay	January
1002	Babloo	February
1004	Dheeraj	January
1008	Hans	November



# Various Syntax in SQL

- **SQL IN Operator**

SELECT Name, Education, CTC

FROM Employee

WHERE DepartmentID NOT IN (2, 4);

Name	Education	CTC
Ajay	Doctoral	25
Babloo	UG	23
Evina	UG	16
Hans	PG	8
Ivanka	Intermediate	7
Jai		4

# Various Syntax in SQL

- **SQL IN Operator**

SELECT EmployeeID, Name, Gender

FROM Employee

WHERE DepartmentID IN (SELECT Department Name FROM Department WHERE DepartmentID > 3);

EmployeeID	Name	Gender	Department Name
1006	Fredy	M	Sales
1007	Garima	F	Sales
1008	Hans	M	Admin
1009	Ivanka	F	Admin
1010	Jai	M	Peon

# Various Syntax in SQL

## SQL DROP TABLE Statement

```
DROP TABLE table_name;
```

## SQL DESC Statement

```
DESC table_name;
```

## SQL INSERT INTO Statement

```
INSERT INTO table_name( column1, column2....columnN)  
VALUES ( value1, value2....valueN);
```

# Some Important References

# Data Types in SQL – Date & Time

DATE	Stores date in the format YYYY-MM-DD
TIME	Stores time in the format HH:MI:SS
DATETIME	Stores date and time information in the format YYYY-MM-DD HH:MI:SS
TIMESTAMP	Stores number of seconds passed since the Unix epoch ('1970-01-01 00:00:00' UTC)
YEAR	Stores year in a 2-digit or 4-digit format. Range 1901 to 2155 in 4-digit format. Range 70 to 69, representing 1970 to 2069.

# Data Types in SQL – Character & String

<u>Data Type</u>	<u>Description</u>
CHAR	Fixed length with a maximum length of 8,000 characters
VARCHAR	Variable-length storage with a maximum length of 8,000 characters
VARCHAR(max)	Variable-length storage with provided max characters, not supported in MySQL. (SQL Server 2005 only).
TEXT	Variable-length storage with a maximum size of 2GB data

# Data Types in SQL – Unicode Character & String

<u>Data Type</u>	<u>Description</u>
NCHAR	Fixed length with a maximum length of 4,000 characters
NVARCHAR	Variable-length storage with a maximum length of 4,000 characters
NVARCHAR(max)	Variable-length storage with provided max characters
NTEXT	Variable-length storage with a maximum size of 1GB data

# Data Types in SQL – Binary Data Types

<u>Data Type</u>	<u>Description</u>
BINARY	Fixed length with a maximum length of 8,000 bytes
VARBINARY	Variable-length storage with a maximum length of 8,000 bytes
VARBINARY(max)	Variable-length storage with provided max bytes
IMAGE	Variable-length storage with a maximum size of 2 GB binary data



# SQL Arithmetic Operators

Operator	Description (Assume a = 10 and b = 20)	Example
+ (Addition)	Adds values on either side of the operator.	a + b will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand.	a - b will give -10
* (Multiplication)	Multiplies values on either side of the operator.	a * b will give 200
/ (Division)	Divides left hand operand by right hand operand.	b / a will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder.	b % a will give 0

# SQL Comparison Operators

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.	(a !< b) is false.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	(a !> b) is true.

# SQL Logical Operators

**ALL** - The ALL operator is used to compare a value to all values in another value set.

**AND** - The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.

**ANY** - The ANY operator is used to compare a value to any applicable value in the list as per the condition.

**BETWEEN** - The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.

**EXISTS** - The EXISTS operator is used to search for the presence of a row in a specified table that meets a certain criterion.

**IN** - The IN operator is used to compare a value to a list of literal values that have been specified.

**LIKE** - The LIKE operator is used to compare a value to similar values using wildcard operators.

# SQL Logical Operators

## NOT

The NOT operator reverses the meaning of the logical operator with which it is used.

Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.**

## OR

The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.

## IS NULL

The NULL operator is used to compare a value with a NULL value.

## UNIQUE

The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

# Some examples

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

# Some examples

```
SELECT AGE FROM CUSTOMERS
WHERE EXISTS (SELECT AGE FROM
CUSTOMERS WHERE SALARY > 6500);
```

AGE
32
25
23
25
27
22
24

```
SELECT * FROM CUSTOMERS
WHERE AGE > ALL (SELECT AGE
FROM CUSTOMERS WHERE
SALARY > 6500);
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00

# Some examples

```
SELECT * FROM CUSTOMERS
```

```
WHERE AGE > ANY (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00

**Thank you**



# Advanced SQL

# Constraints

- Constraints are the rules enforced on the data columns of a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.
- Constraints could be either on a column level or a table level. The column level constraints are applied only to one column, whereas the table level constraints are applied to the whole table.

# Commonly used Constraints available in SQL

- **NOT NULL** Constraint – Ensures that a column cannot have a null value

```
CREATE TABLE CUSTOMERS (  
ID INT NOT NULL,  
NAME VARCHAR (20) NOT NULL,  
AGE INT NOT NULL,  
ADDRESS CHAR (25) ,  
SALARY DECIMAL (18, 2),  
PRIMARY KEY (ID)  
);
```

# Commonly used Constraints available in SQL

- If CUSTOMERS table has already been created, then to add a NOT NULL constraint to the SALARY column in Oracle and MySQL, you would write a query like the one that is shown in the following code block.

```
ALTER TABLE CUSTOMERS
```

```
    MODIFY SALARY DECIMAL (18, 2) NOT NULL;
```

# Commonly used Constraints available in SQL

- **DEFAULT** Constraint - Provides a default value for a column when none is specified.

```
CREATE TABLE CUSTOMERS(  
  ID INT NOT NULL,  
  NAME VARCHAR (20) NOT NULL,  
  AGE INT NOT NULL,  
  ADDRESS CHAR (25) ,  
  SALARY DECIMAL (18, 2) DEFAULT 5000.00,  
  PRIMARY KEY (ID)  
);
```

# Commonly used Constraints available in SQL

- If the CUSTOMERS table has already been created, then to add a DEFAULT constraint to the SALARY column, you would write a query like the one which is shown in the code block below.

```
MODIFY SALARY DECIMAL (18, 2) DEFAULT 5000.00;
```

To drop a DEFAULT constraint, use the following SQL query.

```
ALTER TABLE CUSTOMERS
```

```
    ALTER COLUMN SALARY DROP DEFAULT;
```

# Commonly used Constraints available in SQL

- **UNIQUE** Constraint – to apply a unique constraint to age the syntax would be

```
CREATE TABLE CUSTOMERS(  
  ID INT NOT NULL,  
  NAME VARCHAR (20) NOT NULL,  
  AGE INT NOT NULL UNIQUE,  
  ADDRESS CHAR (25) ,  
  SALARY DECIMAL (18, 2),  
  PRIMARY KEY (ID)  
);
```

# Commonly used constraints available in SQL

- If the CUSTOMERS table has already been created, then to add a UNIQUE constraint to the AGE column. You would write a statement like the query that is given in the code block below.

```
ALTER TABLE CUSTOMERS  
MODIFY AGE INT NOT NULL UNIQUE;
```

You can also use the following syntax, which supports naming the constraint in multiple columns as well.

```
ALTER TABLE CUSTOMERS  
ADD CONSTRAINT myUniqueConstraint UNIQUE(AGE, SALARY);
```

To drop a UNIQUE constraint, use the following SQL query.

```
ALTER TABLE CUSTOMERS  
DROP CONSTRAINT myUniqueConstraint;
```



# Commonly used constraints available in SQL

- **PRIMARY KEY** - Uniquely identifies each row/record in a database table.

```
CREATE TABLE CUSTOMERS(  
ID INT NOT NULL,  
NAME VARCHAR (20) NOT NULL,  
AGE INT NOT NULL UNIQUE,  
ADDRESS CHAR (25) ,  
SALARY DECIMAL (18, 2),  
PRIMARY KEY (ID)  
);
```

# Commonly used constraints available in SQL

- To create a PRIMARY KEY constraint on the "ID" column when the CUSTOMERS table already exists, use the following SQL syntax –

```
ALTER TABLE CUSTOMER ADD PRIMARY KEY (ID);
```

**NOTE** – If you use the ALTER TABLE statement to add a primary key, the primary key column(s) should have already been declared to not contain NULL values (when the table was first created)

## **Delete Primary Key**

```
ALTER TABLE CUSTOMERS DROP PRIMARY KEY ;
```

# SQL - Joins

- The SQL **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.
- There are different types of joins available in SQL –
- [INNER JOIN](#) – returns rows when there is a match in both tables.
- [LEFT JOIN](#) – returns all rows from the left table, even if there are no matches in the right table.
- [RIGHT JOIN](#) – returns all rows from the right table, even if there are no matches in the left table.
- [FULL JOIN](#) – returns rows when there is a match in one of the tables.
- [SELF JOIN](#) – is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- [CARTESIAN JOIN](#) – returns the Cartesian product of the sets of records from the two or more joined tables.

# Inner Join

- The most important and frequently used of the joins is the **INNER JOIN**.
- The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

```
SELECT table1.column1, table2.column2...  
FROM table1  
INNER JOIN table2  
ON table1.common_field = table2.common_field;
```

**Table 1** – CUSTOMERS Table is as follows.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

```
SQL> SELECT ID, NAME, AMOUNT, DATE
      FROM CUSTOMERS
      INNER JOIN ORDERS
      ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

**Table 2** – ORDERS Table is as follows.

OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

ID	NAME	AMOUNT	DATE
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00

# Left Join

- The SQL **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in the right table; the join will still return a row in the result, but with NULL in each column from the right table.
- This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

```
SELECT table1.column1, table2.column2...
```

```
FROM table1
```

```
LEFT JOIN table2
```

```
ON table1.common_field = table2.common_field;
```

**Table 1** – CUSTOMERS Table is as follows.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

```
SQL> SELECT ID, NAME, AMOUNT, DATE
      FROM CUSTOMERS
      LEFT JOIN ORDERS
      ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

**Table 2** – ORDERS Table is as follows.

OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20 00:00:00
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL

# Full Join

- The SQL **FULL JOIN** combines the results of both left and right outer joins.
- The joined table will contain all records from both the tables and fill in NULLs for missing matches on either side.

```
SELECT table1.column1, table2.column2...  
FROM table1  
FULL JOIN table2  
ON table1.common_field = table2.common_field;
```



**Table 1** – CUSTOMERS Table is as follows.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

**Table 2** – ORDERS Table is as follows.

OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

```
SQL> SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
FULL JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20 00:00:00
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL

# Self Join

- The SQL **SELF JOIN** is used to join a table to itself as if the table were two tables; temporarily renaming at least one table in the SQL statement.

```
SELECT a.column_name, b.column_name...  
FROM table1 a  
JOIN table1 b  
WHERE CONDITION;
```

Here, the WHERE clause could be any given expression based on your requirement.

Id	FullName	Salary	ManagerId
1	John Smith	10000	3
2	Jane Anderson	12000	3
3	Tom Lanon	15000	4
4	Anne Connor	20000	
5	Jeremy York	9000	1

Id	FullName	Salary	ManagerId
1	John Smith	10000	3
2	Jane Anderson	12000	3
3	Tom Lanon	15000	4
4	Anne Connor	20000	
5	Jeremy York	9000	1

```
SELECT
    employee.Id,
    employee.FullName,
    employee.ManagerId,
    manager.FullName as ManagerName
FROM Employees employee
JOIN Employees manager
ON employee.ManagerId = manager.Id
```

Id	FullName	ManagerId	ManagerName
1	John Smith	3	Tom Lanon
2	Jane Anderson	3	Tom Lanon
3	Tom Lanon	4	Anne Connor
5	Jeremy York	1	John Smith

# Cartesian Join

- The CARTESIAN JOIN or CROSS JOIN returns the Cartesian product of the sets of records from two or more joined tables.

```
SELECT ColumnName_1,  
ColumnName_2,  
ColumnName_N  
FROM [Table_1]  
CROSS JOIN [Table_2]
```

or

```
SELECT ColumnName_1,  
ColumnName_2,  
ColumnName_N  
FROM [Table_1],[Table_2]
```

# Union Operator

The SQL UNION clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.

To use this UNION clause, each SELECT statement must have

- The same number of columns selected
- The same number of column expressions
- The same data type and
- Have them in the same order

```
SELECT column1 [, column2 ]
```

```
FROM table1 [, table2 ]
```

```
[WHERE condition]
```

```
UNION
```

```
SELECT column1 [, column2 ]
```

```
FROM table1 [, table2 ]
```

```
[WHERE condition]
```

# Union All Operator

- The UNION ALL operator is used to combine the results of two SELECT statements including duplicate rows.
- The same rules that apply to the UNION clause will apply to the UNION ALL operator.

```
SELECT column1 [, column2 ]
```

```
FROM table1 [, table2 ]
```

```
[WHERE condition]
```

```
UNION ALL
```

```
SELECT column1 [, column2 ]
```

```
FROM table1 [, table2 ]
```

```
[WHERE condition]
```

# Intersect Operator

- The SQL **INTERSECT** clause/operator is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement. This means INTERSECT returns only common rows returned by the two SELECT statements.
- Just as with the UNION operator, the same rules apply when using the INTERSECT operator. MySQL does not support the INTERSECT operator.

```
SELECT column1 [, column2 ]  
FROM table1 [, table2 ]  
[WHERE condition]  
INTERSECT  
SELECT column1 [, column2 ]  
FROM table1 [, table2 ]  
[WHERE condition]
```

# Except Operator

- The SQL **EXCEPT** clause/operator is used to combine two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement. This means EXCEPT returns only rows, which are not available in the second SELECT statement.
- Just as with the UNION operator, the same rules apply when using the EXCEPT operator. MySQL does not support the EXCEPT operator.

SELECT column1 [, column2 ]

FROM table1 [, table2 ]

[WHERE condition]

EXCEPT

SELECT column1 [, column2 ]

FROM table1 [, table2 ]

[WHERE condition]



# Alter Table

- The SQL **ALTER TABLE** command is used to add, delete or modify columns in an existing table. You should also use the ALTER TABLE command to add and drop various constraints on an existing table.

ALTER TABLE table\_name ADD column\_name datatype; - to add column

ALTER TABLE table\_name DROP COLUMN column\_name; - drop column

ALTER TABLE table\_name MODIFY COLUMN column\_name datatype; - change datatype

ALTER TABLE table\_name MODIFY column\_name datatype NOT NULL; - add NOT NULL

ALTER TABLE table\_name

ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...); - add CONSTRAINT

ALTER TABLE table\_name

DROP CONSTRAINT MyUniqueConstraint – to drop constraint

Many other options are available with the ALTER COMMAND.

# Index

- Indexes are special lookup tables that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.
- For example, if you want to reference all pages in a book that discusses a certain topic, you first refer to the index, which lists all the topics alphabetically and are then referred to one or more specific page numbers.
- An index helps to speed up SELECT queries and WHERE clauses, but it slows down data input, with the UPDATE and the INSERT statements. Indexes can be created or dropped with no effect on the data.
- Creating an index involves the CREATE INDEX statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in an ascending or descending order.
- Indexes can also be unique, like the UNIQUE constraint, in that the index prevents duplicate entries in the column or combination of columns on which there is an index.

# Index

- **Single-Column Indexes** - A single-column index is created based on only one table column. The basic syntax is as follows.

```
CREATE INDEX index_name  
ON table_name (column_name);
```

- **Unique indexes** - are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows.

```
CREATE UNIQUE INDEX index_name  
on table_name (column_name);
```

- **A composite index** - is an index on two or more columns of a table. Its basic syntax is as follows.

```
CREATE INDEX index_name  
on table_name (column1, column2);
```

- **Implicit indexes** are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

# Index

- An index can be dropped using SQL DROP command. Care should be taken when dropping an index because the performance may either slow down or improve.

```
DROP INDEX index_name;
```

## When should indexes be avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided. The following guidelines indicate when the use of an index should be reconsidered.

- Indexes should not be used on small tables.
- Tables that have frequent, large batch updates or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

# Index

- Example - For example, the following SQL syntax creates a new table called CUSTOMERS and adds five columns in it.

```
CREATE TABLE CUSTOMERS(  
  ID INT NOT NULL,  
  NAME VARCHAR (20) NOT NULL,  
  AGE INT NOT NULL,  
  ADDRESS CHAR (25) ,  
  SALARY DECIMAL (18, 2),  
  PRIMARY KEY (ID));
```

- **Now, you can create an index on a single or multiple columns using the syntax given below.**

```
CREATE INDEX index_name  
ON table_name ( column1, column2.....);
```

- To create an INDEX on the AGE column, to optimize the search on customers for a specific age, you can use the follow SQL syntax which is given below –

```
CREATE INDEX idx_age  
ON CUSTOMERS ( AGE );
```

# Index

- DROP an INDEX Constraint - To drop an INDEX constraint, use the following SQL syntax.

```
ALTER TABLE CUSTOMERS  
DROP INDEX idx_age;
```

**Remember** - The INDEX is used to create and retrieve data from the database very quickly. An Index can be created by using a single or group of columns in a table. When the index is created, it is assigned a ROWID for each row before it sorts out the data.

Proper indexes are good for performance in large databases, but you need to be careful while creating an index. A Selection of fields depends on what you are using in your SQL queries.

# Truncate

- The SQL TRUNCATE TABLE command is used to delete all the records from an existing table by reinitializing the table's structure. This command instructs the database to deallocate the space for all records in a table and change this table's structure by resetting the table size. This is the reason why it is deemed to be a Data Definition Language (DDL) operation rather than Data Manipulation Language (DML), even though all the table data is removed.
- Logically, the TRUNCATE TABLE statement performs similarly to the DELETE TABLE statement but without the WHERE clause. However, TRUNCATE is much faster than DELETE and does not allow roll back once committed.
- You can also use DROP TABLE command to delete a table but it will remove the complete table structure from the database and you would need to re-create this table once again if you wish you store some data again.

**The basic syntax of a TRUNCATE TABLE command is as follows.**

```
TRUNCATE TABLE table_name;
```

# Truncate Vs Delete

DELETE	TRUNCATE
The DELETE command in SQL removes one or more rows from a table based on the conditions specified in a WHERE Clause.	SQL's TRUNCATE command is used to remove all of the rows from a table, regardless of whether or not any conditions are met.
It is a DML(Data Manipulation Language) command.	It is a DDL(Data Definition Language) command.
There is a need to make a manual COMMIT after making changes to the DELETE command, for the modifications to be committed.	When you use the TRUNCATE command, the modifications made to the table are committed automatically.
It deletes rows one at a time and applies some criteria to each deletion.	It removes all of the information in one go
When it comes to large databases, it is much slower.	It is faster.



# Views

- Creating a view is simply creating a virtual table using a query. A view is an SQL statement that is stored in the database with an associated name. It is actually a composition of a table in the form of a predefined SQL query.
- A view can contain rows from an existing table (all or selected). A view can be created from one or many tables which depends on the written SQL query to create a view. Unless indexed, a view does not exist in a database.
- Views, which are a type of virtual tables allow users to do the following –
  - Structure data in a way that users or classes of users find natural or intuitive.
  - Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
  - Summarize data from various tables which can be used to generate reports.

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

# Views

Example - Create view first\_view AS SELECT \* FROM customers;

- With Check Option.
- The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.
- If they do not satisfy the condition(s), the UPDATE or INSERT returns an error. The following code block has an example of creating same view CUSTOMERS\_VIEW with the WITH CHECK

```
CREATE VIEW CUSTOMERS_VIEW  
AS SELECT name, age FROM CUSTOMERS  
WHERE age IS NOT NULL  
WITH CHECK OPTION;
```

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

# Update Views – Only for simple view (One Table View)

- The SQL **UPDATE** Query is used to modify the existing records in a table or a view. It is a Data Manipulation Language Command as it only modifies the data of the database object.
- The UPDATE statement makes use of locks on each row while modifying them in a table or view, and once the row is modified, the lock is released. Therefore, it can either make changes to a single row or multiple rows with a single query.

```
UPDATE view_name
```

```
SET column1 = value1, column2 = value2...., columnN = valueN
```

```
WHERE [condition];
```

# Update Views – Example

Create view CUSTOMERS\_VIEW AS SELECT \* FROM CUSTOMERS;

- Now, through the view you created, you can update the age of Ramesh to 35 in the original CUSTOMERS table, using the following code block –

UPDATE CUSTOMERS\_VIEW SET AGE = 35 WHERE name = 'Ramesh';

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself.

Updating Multiple columns

```
UPDATE CUSTOMERS_VIEW  
SET NAME = 'Kaushik', AGE = 24  
WHERE ID = 3;
```

Multiple Rows

```
UPDATE CUSTOMERS_VIEW  
SET AGE = 24;
```

# Drop View

The SQL DROP View statement is used to delete an existing view, along with its definition and other information. Once the view is dropped, all the permissions for it will also be removed.

Suppose a table is dropped using the DROP TABLE command and it has a view associated to it, this view must also be dropped explicitly using the DROP VIEW command.

While trying to perform queries, the database engine checks all the objects referenced in that statement are valid and exist. So, if a view does not exist in the database, the DROP VIEW statement will throw an error.

**DROP VIEW view\_name;**

**DROP VIEW [IF EXISTS] view\_name;**

**(Does not throw error if view does not exist)**

# Stored Procedure

A stored procedure is a group of pre-compiled SQL statements (prepared SQL code) that can be reused again and again.

They can be used to perform a wide range of database operations such as inserting, updating, or deleting data, generating reports, and performing complex calculations. Stored procedures are very useful because they allow you to encapsulate (bundle) a set of SQL statements as a single unit and execute them repeatedly with different parameters, making it easy to manage and reuse code.

# Stored Procedure - Syntax

```
CREATE PROCEDURE procedure_name
```

```
    @parameter1 datatype,
```

```
    @parameter2 datatype
```

```
AS
```

```
BEGIN
```

```
    -- SQL statements to be executed
```

```
END
```

```
Where,
```

The CREATE PROCEDURE statement is used to create the procedure. After creating the procedure, we can define any input parameters that the procedure may require. These parameters are preceded by the '@' symbol and followed by their respective data types.

The AS keyword is used to begin the procedure definition. The SQL statements that make up the procedure are placed between the BEGIN and END keywords.

# Stored Procedure - Example

## Customer Table

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

```
CREATE PROCEDURE GetCustomerInfo
    @CutomerAge INT
AS
BEGIN
    SELECT * FROM CUSTOMERS
    WHERE AGE = @CutomerAge
END
```

```
EXEC GetCustomerInfo @CutomerAge = 25
```

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500.00
4	Chaitali	25	Mumbai	6500.00



# Procedure with IN Parameter

The IN parameter is the default parameter that will receive input value from the program.

We can pass the values as arguments when the stored procedure is being executed. These values are read-only, which means they cannot be modified by the stored procedure.

```
CREATE PROCEDURE GetCustomerSalary
```

```
    @CustomerID INT
```

```
AS
```

```
BEGIN
```

```
    SELECT SALARY FROM CUSTOMERS WHERE ID = @CustomerID
```

```
END
```

```
EXEC GetCustomerSalary @CustomerID = 6
```

# Procedure with OUT Parameter

The OUT parameter sends output value to the program. It allows us to return a value or set of values to the calling program.

Note that when using an OUT parameter, we must specify the keyword "OUT" after the parameter name when passing it to the stored procedure.

```
CREATE PROCEDURE GetCustomerSalary
```

```
    @CustomerID INT,
```

```
    @Salary DECIMAL(18,2) OUT
```

```
AS
```

```
BEGIN
```

```
    SELECT @Salary = SALARY FROM CUSTOMERS WHERE ID = @CustomerID
```

```
END
```

```
DECLARE @CustSalary DECIMAL(18, 2)
```

```
EXEC GetCustomerSalary @CustomerID = 4, @Salary = @CustSalary OUT
```

```
SELECT @CustSalary AS 'Customer Salary'
```

# Procedure – Other actions

## Rename

```
sp_rename 'old_procedure_name', 'new_procedure_name';
```

## Modify

```
ALTER PROCEDURE procedure_name  
AS  
BEGIN  
    -- New procedure code goes here  
END
```

## Drop

```
DROP PROCEDURE [IF EXISTS] procedure_name;
```

If the stored procedure is referenced by other objects such as views or other stored procedures, we need to update those references manually to reflect the new name of the stored procedure. Additionally, dropping a stored procedure may impact any scripts or applications that rely on the stored procedure.

# Advantages of Stored Procedures

- **Improved Performance** – Stored procedures are pre-compiled and stored on the server, so they can be executed more quickly than SQL statements that are sent from client applications.
- **Code Reuse** – Stored procedures can be called from different client applications, which means that the same code can be reused across different applications. This reduces development time and maintenance costs.
- **Reduced Network Traffic** – Because stored procedures are executed on the server, only the results are returned to the client, which reduces network traffic and improves application performance.
- **Better Security** – Stored procedures can be used to enforce security rules and prevent unauthorized access to sensitive data. They can also limit the actions that can be performed by users, making it easier to maintain data integrity and consistency.
- **Simplified Maintenance** – By storing SQL code in a single location, it becomes easier to maintain and update the code. This makes it easier to fix bugs, add new functionality, and optimize performance.

# Drawbacks of Stored Procedures

- **Increased Overhead** – Stored procedures can consume more server resources than simple SQL statements, particularly when they are used frequently or for complex operations.
- **Limited Portability** – Stored procedures are often specific to a particular database management system (DBMS), which means they may not be easily portable to other DBMSs.
- **Debugging Challenges** – Debugging stored procedures can be more challenging than debugging simple SQL statements, particularly when there are multiple layers of code involved.
- **Security Risks** – If stored procedures are not written correctly, they can pose a security risk, particularly if they are used to access sensitive data or to perform actions that could compromise the integrity of the database.

# Transactions

- A transaction is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.
- A transaction is the propagation of one or more changes to the database. For example, if you are creating a record or updating a record or deleting a record from the table, then you are performing a transaction on that table. It is important to control these transactions to ensure the data integrity and to handle database errors.
- Practically, you will club many SQL queries into a group and you will execute all of them together as a part of a transaction.

# Properties of Transactions

Transactions have the following four standard properties, usually referred to by the acronym ACID.

- **Atomicity** – ensures that all operations within the work unit are completed successfully. Otherwise, the transaction is aborted at the point of failure and all the previous operations are rolled back to their former state.
- **Consistency** – ensures that the database properly changes states upon a successfully committed transaction.
- **Isolation** – enables transactions to operate independently of and transparent to each other.
- **Durability** – ensures that the result or effect of a committed transaction persists in case of a system failure.

# Transaction Control

The following commands are commonly used to control transactions.

**COMMIT** – to save the changes.

**ROLLBACK** – to roll back the changes.

**SAVEPOINT** – creates points within the groups of transactions in which to ROLLBACK.

Transactional control commands are only used with the DML Commands such as - INSERT, UPDATE and DELETE only. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.



# The COMMIT Command

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for the COMMIT command is as follows.

**COMMIT;**

# The ROLLBACK Command

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for a ROLLBACK command is as follows –

**ROLLBACK;**

# The SAVEPOINT Command

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

The syntax for a SAVEPOINT command is as shown below.–

**SAVEPOINT SAVEPOINT\_NAME;**

This command serves only in the creation of a SAVEPOINT among all the transactional statements. The ROLLBACK command is used to undo a group of transactions.

The syntax for rolling back to a SAVEPOINT is as shown below.

**ROLLBACK TO SAVEPOINT\_NAME;**

# The SAVEPOINT Command - Example

**SQL> SAVEPOINT SP1;**

Savepoint created.

**SQL> DELETE FROM CUSTOMERS WHERE ID=1;**

1 row deleted.

**SQL> SAVEPOINT SP2;**

Savepoint created.

**SQL> DELETE FROM CUSTOMERS WHERE ID=2;**

1 row deleted.

**SQL> SAVEPOINT SP3;**

Savepoint created.

**SQL> DELETE FROM CUSTOMERS WHERE ID=3;**

1 row deleted.

**SQL> ROLLBACK TO SP2;  
Rollback complete.**

# Sequences

The sequences in SQL is a database object that generates a sequence of unique integer values. They are frequently used in databases because many applications require each row in a table to contain a unique value and sequences provide an easy way to generate them.

A sequence is created using the `CREATE SEQUENCE` statement in SQL. The statement specifies the name of the sequence, the starting value, the increment, and other properties of the sequence.

# Sequences

Following is the syntax to create a sequence in SQL –

```
CREATE SEQUENCE Sequence_Name  
START WITH Initial_Value  
INCREMENT BY Increment_Value  
MINVALUE Minimum_Value  
MAXVALUE Maximum_Value  
CACHE | NOCACHE ----- Optional  
CYCLE | NOCYCLE;
```

# Sequences

**Sequence\_Name** – This specifies the name of the sequence.

**Initial\_Value** – This specifies the starting value from where the sequence should start.

**Increment\_Value** – This specifies the value by which the sequence will increment by itself. This can be valued positively or negatively.

**Minimum\_Value** – This specifies the minimum value of the sequence.

**Maximum\_Value** – This specifies the maximum value of the sequence.

**Cache** - Specify how many values of the sequence the database preallocates and keeps in memory for faster access. The maximum value allowed for CACHE must be less than the value determined by the following formula:  $(\text{CEIL}(\text{MAXVALUE} - \text{MINVALUE})) / \text{ABS}(\text{INCREMENT})$

**Nocache** -Specify NOCACHE to indicate that values of the sequence are not preallocated. If you omit both CACHE and NOCACHE, the database caches 20 sequence numbers by default.

**Cycle** – When the sequence reaches its Maximum\_Value, it starts again from the beginning.

**Nocycle** – An exception will be thrown if the sequence exceeds the Maximum\_Value.

# Sequences - Example

ID	NAME	AGE
NULL	Dhruv	20
NULL	Arjun	23
NULL	Dev	25
NULL	Riya	19
NULL	Aarohi	24
NULL	Lisa	20
NULL	Roy	24

```
SQL> CREATE SEQUENCE My_Sequence AS INT
      START WITH 1
      INCREMENT BY 1
      MINVALUE 1
      MAXVALUE 5
      CYCLE;
```

```
SQL> UPDATE STUDENTS SET ID = NEXT VALUE FOR My_Sequence;
```

ID	NAME	AGE
1	Dhruv	20
2	Arjun	23
3	Dev	25
4	Riya	19
5	Aarohi	24
1	Lisa	20
2	Roy	24



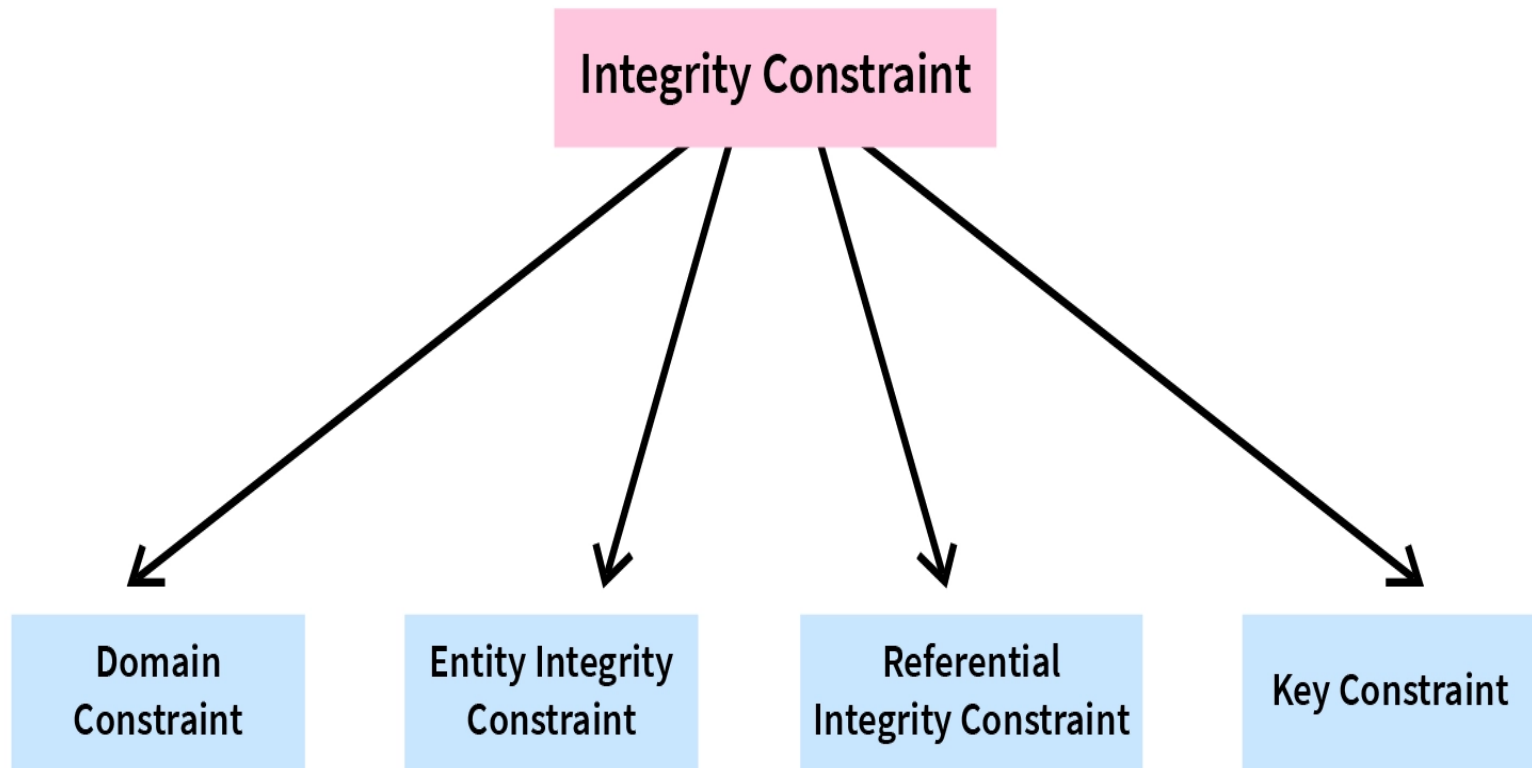
Thank You

# **Integrity Constraints**

# What are Integrity Constraints in DBMS?

In Database Management Systems, integrity constraints are pre-defined set of rules that are applied on the table fields(columns) or relations to ensure that the overall validity, integrity, and consistency of the data present in the database table is maintained. Evaluation of all the conditions or rules mentioned in the integrity constraint is done every time a table insert, update, delete, or alter operation is performed. The data can be inserted, updated, deleted, or altered only if the result of the constraint comes out to be True. Thus, integrity constraints are useful in preventing any accidental damage to the database by an authorized user.

# Types of Integrity Constraints



# Domain Constraint

Domain integrity constraint contains a certain set of rules or conditions to restrict the kind of attributes or values a column can hold in the database table. The data type of a domain can be string, integer, character, DateTime, currency, etc.

## **Example:**

Consider a Student's table having Roll No, Name, Age, Class of students.

In the above student's table, the value A in the last row last column violates the domain integrity constraint because the Class attribute contains only integer values while A is a character.

Roll No	Name	Age	Class
101	Adam	14	6
102	Steve	16	8
103	David	8	4
104	Bruce	18	12
105	Tim	6	A

# Entity Integrity Constraint

Entity Integrity Constraint is used to ensure that the primary key cannot be null. A primary key is used to identify individual records in a table and if the primary key has a null value, then we can't identify those records. There can be null values anywhere in the table except the primary key column.

## **Example:**

Consider Employees table having Id, Name, and salary of employees  
In the above employee's table, we can see that the ID column is the primary key and contains a null value in the last row which violates the entity

ID	Name	Salary
1101	Jackson	40000
1102	Harry	60000
1103	Steve	80000
1104	Ash	1800000
Null	James	36000

# Referential Integrity Constraint

Referential Integrity Constraint ensures that there must always exist a valid relationship between two relational database tables. This valid relationship between the two tables confirms that a foreign key exists in a table. It should always reference a corresponding value or attribute in the other table or be null.

**Referential Integrity Rule** in DBMS is constraint applied on **Primary key** in parent table which is **Foreign key** to the child table, which defines that a **Foreign key** value of child table should always have the same **Primary key** value in the parent table. So that, Reference from a parent table to child table is valid.

**Referential integrity** is a constraint applied on foreign, which requires the foreign key to have a matching primary key or provided primary key is not a null value. Therefore, a constraint is specified between two tables, which makes sure reference from a row in one table to another is valid.

Dept_ID	Dept_Name
1	Sales
2	HR
3	Technical

**Department Table**

ID	Name	Salary	Dept_ID
1101	Jackson	40000	3
1102	Harry	60000	2
1103	Steve	80000	4
1104	Ash	1800000	3
1105	James	36000	1

**Employees Table**

### Example:

Consider an Employee and a Department table where Dept\_ID acts as a foreign key between the two tables **Employees Table, Department Table**.

In the above example, Dept\_ID acts as a foreign key in the Employees table and a primary key in the Department table.

Row having DeptID=4 violates the referential integrity constraint since DeptID 4 is not defined as a primary key column in the Departments table.



# Key constraint

Keys are the set of entities that are used to identify an entity within its entity set uniquely. There could be multiple keys in a single entity set, but out of these multiple keys, only one key will be the primary key. A primary key can only contain unique and not null values in the relational database table.

## Example:

Consider a student's table. The last row of the student's table violates the key integrity constraint since Roll No 102 is repeated twice in the primary key column. A primary key must be unique and not null therefore duplicate values are not allowed in the Roll No column of the above student's table.

Roll No	Name	Age	Class
101	Adam	14	6
102	Steve	16	8
103	David	8	4
104	Bruce	18	12
102	Tim	6	2

# Conclusion

- Integrity Constraints in Database Management Systems are the set of pre-defined rules responsible for maintaining the quality and consistency of data in the database.
- Evaluation against the rules mentioned in the integrity constraint is done every time an insert, update, delete, or alter operation is performed on the table.
- Integrity Constraints in DBMS are of 4 types:
  - Domain Constraint
  - Entity Constraint
  - Referential Integrity Constraint
  - Key Constraint

Thank You

# Relational Database Design

# Components in a relational database model

**A relational database has these four basic components:**

- Tables that store data
- Columns in tables so that we can store different types of data.
- Primary key in a table to uniquely identify each row in a table.
- Relationships between tables. As we can't store all data in one table, we break them down and put into different tables and link them by common columns (known as foreign keys) so that we can have some sort of relationships to be used to pull out related data later from these tables.

# Database Normalization

Normalization is the process of breaking down our data and store them in different tables consistently.

There are mainly three levels of normalization (first, second, and third normal form). Further normal forms are available but not commonly used. You can ignore them completely.

Before we move onto the theories about normal forms, let's assume you are the company owner of “Northwind Trader” and you buy and sell goods. How would you store your company sales data in the old days?

By intuition, you would want to store data separately into different draws or folders. Products data in one draw and suppliers data in another draw. Naturally you want to relate products to suppliers to find out who supplied what. This type of thinking is the basis of relational database design.

# Database Normalization

## **First Normal Form (1st NF)**

In theory, this is the first thing you need to do to break down your data. In this step, we want to achieve the following three goals:

- Each row in a table must be uniquely identifiable.
- Each column must only contain one piece of information.
- No repeating columns - no two columns should contain the same data.

# Database Normalization

## Goal #1: Each row in a table must be uniquely identifiable

What are the consequences of unable to identify each row? We could either store the same information twice in the table, or update or delete wrong records as we are not able to distinguish between rows.

To uniquely identify a row, we define a single column or a group of columns as primary key. Below is customers table in Northwind database. Each customer is assigned a unique value for identification.

CustomerID	CompanyName	ContactName
ANTON	Antonio Moreno Taquería	Antonio Moreno
AROUT	Around the Horn	Thomas Hardy
BERGS	Berglunds snabbköp	Christina Berglund
BLAUS	Blauer See Delikatessen	Hanna Moos
BLONP	Blondel père et fils	Frédérique Citeaux
BOLID	Bólido Comidas preparadas	Martín Sommer



# Database Normalization

**Goal #2: Each column must only contain one piece of information.**

It's easier to explain this requirement by illustration. We have five address data as below:

- Obere Str. 57  
Berlin, 12209  
Germany
- 120 Hanover Sq.  
London, WA1 1DP  
UK
- 23 Tsawassen Blvd.  
Tsawassen, T2F 8M4  
Canada
- 54, rue Royale  
Nantes, 44000  
France
- Via Monte Bianco 34  
Torino, 10100  
Italy

# Database Normalization

- As we see above, if data is grouped together like this **in a single column**, it would be impossible to search postcode by country efficiently.
- To solve this problem, the First Normal Form requires that each of these items must be broken down and stored in its own column. This is known as storing **atomic data**. One column for street address, one column for City, one column for postcode, and one column for country.

Address	City	PostalCode	Country
Obere Str. 57	Berlin	12209	Germany
120 Hanover Sq.	London	WA1 1DP	UK
23 Tsawassen Blvd.	Tsawassen	T2F 8M4	Canada
54, rue Royale	Nantes	44000	France
Via Monte Bianco 34	Torino	10100	Italy

# Database Normalization

## Goal #3: No repeating columns - no two columns should contain the same data.

In other words, every column in a row should store data that is different from all other columns.

To illustrate this, think of the order\_details table that lists products purchased. We could structure a table shown as below:

OrderID	ProductID	UnitPrice	Quantity	ProductID2	UnitPrice2	Quantity2
10255	2	15.2	20	36	15.2	25
10255	16	13.9	35	59	44	30

# Database Normalization

- For the same order (OrderID 10255), the table above stores two products (ProductID = 2 and ProductID2 = 26) in the first row and another two products in the second row. The repeated columns are ProductID vs ProductID2, UnitPrice vs UniPrice2, and Quantity vs Quantity2. This structure presents two problems. First, it's difficult to search for information. Second, it creates redundant columns for the same information.
- To solve the problem, we split the columns and replace them with multiple rows.

OrderID	ProductID	UnitPrice	Quantity
10255	2	15.2	20
10255	16	13.9	35
10255	36	15.2	25
10255	59	44	30

# Functional Dependency

- We introduce functional dependency here because it's going to be used in the next two normal forms.
- Functional dependency describes the concept that all other columns in a table must depend completely on the primary key column.
- For example, the values in the CompanyName column and ContactName column depend completely on the value in the CustomerID column. In other words, if we know CustomerID, then we can tell the customer's company name and contact name. This reads like CustomerID determines CompanyName and ContactName.

CustomerID	CompanyName	ContactName
ANTON	Antonio Moreno Taquería	Antonio Moreno
AROUT	Around the Horn	Thomas Hardy
BERGS	Berglunds snabbköp	Christina Berglund
BLAUS	Blauer See Delikatessen	Hanna Moos
BLONP	Blondel père et fils	Frédérique Citeaux
BOLID	Bólido Comidas preparadas	Martín Sommer

**The effect of achieving functional dependency is that each related set of data is put into its own table. In other words, each table only represents one subject.**

It should be noted that functional dependencies are not limited to depending on a single column. We can have a combination of several columns determine other columns in the table.

# Normalization Contd...

## **Second Normal Form (2nd NF)**

- For a table to be in the Second Normal Form, it must already be in the First Normal Form.
- After 1st NF, every table has got a primary key. Sometimes the primary key consists of multiple columns (also known as composite primary key). When this happens, we should pay attention to other columns (non-key columns) in this table to make sure all these other non-key columns must fully depend on the whole primary key. If not, we will have to action on the following three steps:
  - Remove partial dependencies on the composite primary key.
  - Then build separate tables for each set of removed data.
  - Then build relationships between these tables.
- If we can't find partial dependencies of non-key columns on the composite primary key or the table uses a single column as primary key, we are already at Second Normal Form.
- The pre-requisite for 2nd Normal Form is that the primary key in the table must consist of multiple columns.

## Normalization Contd...

### 2nd Normal Form (partial dependency) example

- Table below has a composite primary key that consists of OrderID and ProductID column. The combination of OrderID and ProductID uniquely identifies each row in this table. Note that UnitPrice and Quantity are non-key column and are fully dependent on this composite primary key.
- Order\_details table:

OrderID	ProductID	UnitPrice	Quantity	OrderDate	RequiredDate
10248	11	14	12	1996-07-02	1996-08-01
10248	42	9.8	10	1996-07-02	1996-08-01
10248	72	34.8	5	1996-07-02	1996-08-01
10249	14	18.6	9	1996-07-05	1996-08-16
10249	51	42.4	40	1996-07-05	1996-08-16

## Normalization Contd...

Now, look at OrderDate and RequiredDate column. They are non-key column as well. We notice that:

- OrderDate 1996-07-02 was repeated three times and 1996-07-05 was repeated two times. The same is for RequiredDate.
- OrderDate and RequiredDate only depend on OrderID column and has nothing to do with ProductID column.

OrderDate and RequiredDate column are not fully dependent on the combination of OrderID and ProductID, which form the composite primary key. We just discovered a partial dependency. Therefore, this table is not in 2nd NF. Let's remove the partial dependency by splitting this table into the following two tables.

**Orders table**

OrderID	OrderDate	RequiredDate
10248	1996-07-02	1996-08-01
10249	1996-07-05	1996-08-16

**Order details table**

OrderID	ProductID	UnitPrice	Quantity
10248	11	14	12
10248	42	9.8	10
10248	72	34.8	5
10249	14	18.6	9
10249	51	42.4	40

Our new Orders table only stores data related to orders and we created a new table Order\_details.

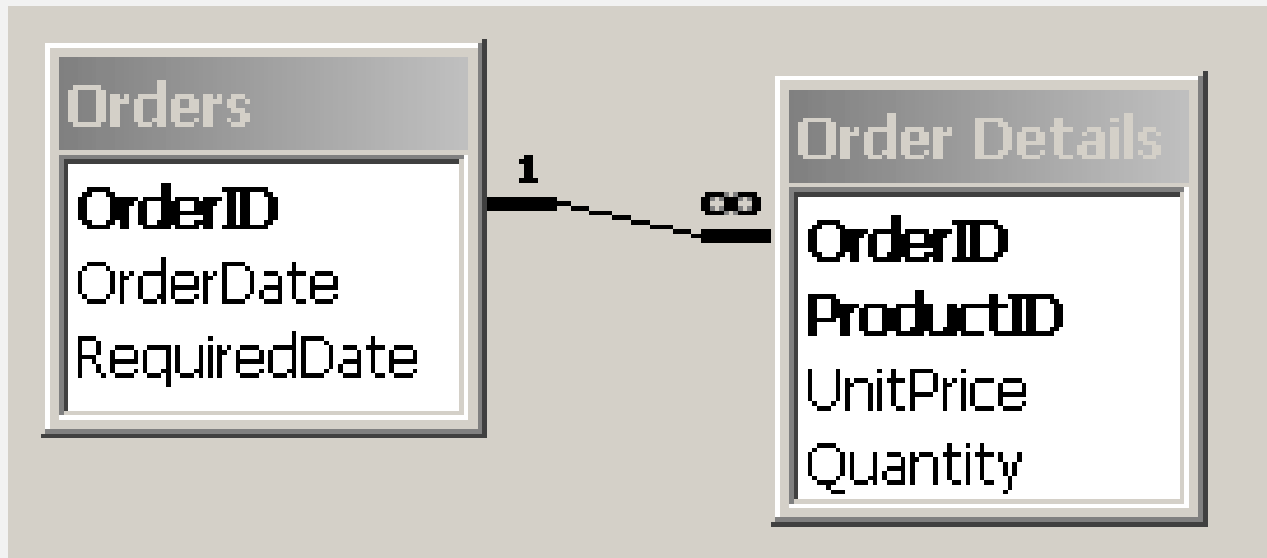
If we want to find out order date for order 10248 in Order\_details table, we use the relationship between the two tables to work out that information. This relationship is established by OrderID in both tables. This relationship is called foreign key relationship.



# Normalization and Relationship

## One-to-Many relationship:

- One OrderID 10248 in Orders table is mapped to three OrderIDs in Order\_details table.
- One OrderID 10249 in Orders table is mapped to two OrderIDs in Order\_details table.
- Foreign key relationship (one-to-many relationship)



# Normalization Contd...

## Third Normal Form (3rd NF)

The 3rd NF is also related to solving a dependency problem. But unlike 2nd NF, here the dependency is on **non-primary key** column(s). Third Normal Form requires that if any column that is not **DIRECTLY** dependent on primary key (either single column primary key or composite primary key), it should be removed and placed into a different table.

- Table below shows that ProductID is the primary key of the table. SupplierName is dependent on SupplierID column, which in turn depends on the primary key ProductID. This is called **transitive dependency**.
- The rule for the 3rd NF requires that each column be dependent on the primary key only, and not any other columns. Here SupplierName clearly violated this rule.

ProductID	ProductName	SupplierID	SupplierName
1	Chai	1	Exotic Liquids
2	Chang	1	Exotic Liquids
3	Aniseed Syrup	1	Exotic Liquids
9	Mishi Kobe Niku	4	Tokyo Traders
10	Ikura	4	Tokyo Traders
74	Longlife Tofu	4	Tokyo Traders

## Normalization Contd...

- To resolve the problem, we take out SupplierName and put it into a separate suppliers table. See below.

- Suppliers table:

SupplierID	SupplierName
1	Exotic Liquids
4	Tokyo Traders

- Now products table only contains SupplierID column which can be linked to suppliers table to get SupplierName.

- Products table:

ProductID	ProductName	SupplierID
1	Chai	1
2	Chang	1
3	Aniseed Syrup	1
9	Mishi Kobe Niku	4
10	Ikura	4
74	Longlife Tofu	4

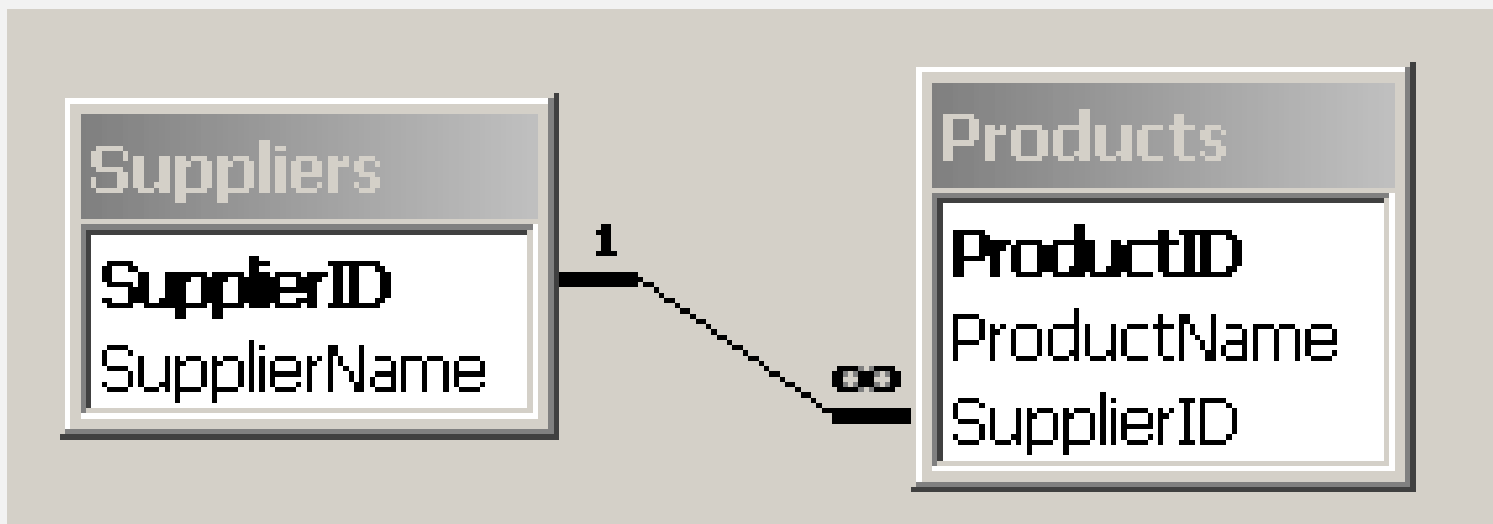
# Normalization and Relationship

- **One-to-Many relationship:**

One SupplierID 1 in Suppliers table is mapped to three SupplierIDs in Products table.

One SupplierID 4 in Suppliers table is mapped to three SupplierIDs in Products table.

- One-to-Many relationship:



# Benefits of Normalization

## The benefits of normalization

So far we have covered the three normal forms but we haven't explicitly listed why we need to normalize. Here are the two main benefits you get from normalization.

- **Benefit #1: Reduce data redundancy** - Unless absolutely necessary, storing redundant data is a waste of system resources. Nowadays hard disk space is cheap but is not free. More importantly, maintaining very large size of databases requires more work on database administrators and network engineers.
- **Benefit #2: Reduce data inconsistency** - The data redundancy issue can further cause data inconsistency issues. As we keep the same piece of data across different locations, we have to make sure that they are kept exactly the same all the time. For example, when one piece of data is updated, the same data in other locations has to be updated as well.

# Benefits of Normalization

The issues of data inconsistency are collectively called Data Anomaly. There are four types of possible errors they could cause. These four types of errors are the causes of data anomaly:

- **Select anomaly** (also known as join anomalies):  
This happens when we select the same piece of data from different tables but they could produce different results.
- **Update anomalies**  
This occurs when we update the same piece of data in more than one place. If care not taken, we could end up with updating the data in one place but forgot to update it in another place.
- **Insertion anomalies**  
This can happen when we are adding a new record for the data. If we forgot to insert it to other places for the same data, data inconsistency occurs.
- **Deletion anomalies**  
This happens when we delete data. If the deletion does not remove the same data from all places, data anomalies occur.

# Normalization - Summary

- 1st NF: Unique identifier, atomic data, no repeating columns.
- 2nd NF: Remove partial dependencies on composite primary key.
- 3rd NF: Remove transitive dependencies on non-primary key column(s).

Thank You