

Unit 1

Introduction to Database Management System

Introduction

- Managing Database.
- Data : Raw Facts
 Unorganized
 Unprocessed Information

Eg. Marks/ Results of complete class

- Information:
- Processing of Data
- Organized fashion
- Eg. Average Marks or Passed percentage

- Database is collection of Data or information that is organized in such way that it can be easily accessible or managed and updated.
- Objectives:
 - Mass Storage-Large amount of data
 - Remove Duplicates
 - Multi-User- Access can be done by multiple users.
 - Protection: Security for data access.
- DBMS: It is collection of Interrelated data and set of programs to access those data

Purpose of Database System

- **Data Redundancy and inconsistency:** Duplication of data , all copies may not be updated.
- **Difficulty in accessing data:** new programme needed for each task
- **Data isolation:** Data in different format.
- **Integrity problem:** Accuracy cannot be maintained.
- **Atomicity of updates:** Failures of data may result inconsistent.
- **Concurrent access by multiple users:** Multiple users can access data.
- **Security Problem:** Restricted access
- **Support for multiple views of data:** Different view of data for many users

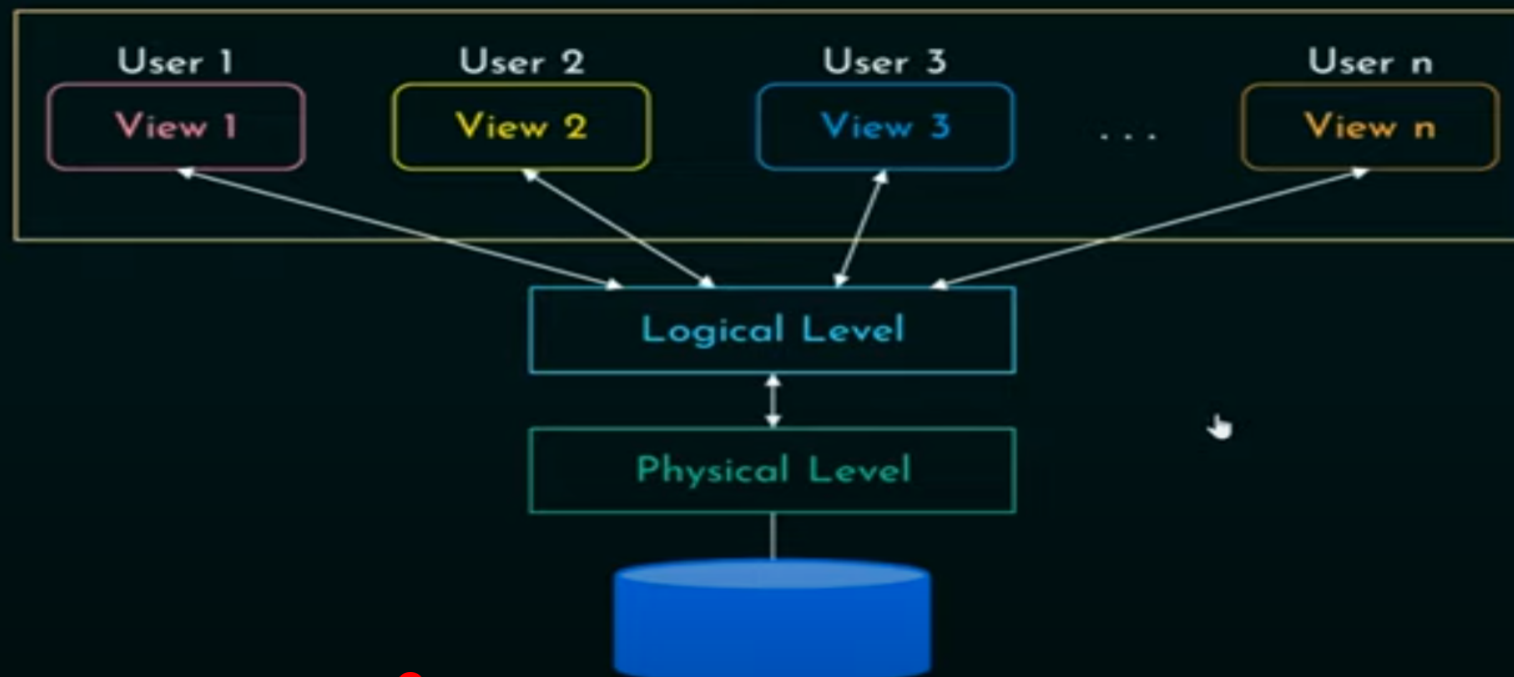
Database System Applications

- Airlines
- Business
- E-commerce
- Universities
- Banking
- Sales:
- Fianance

View of Data

- Data Abstraction :(Different Levels of Abstraction)
- Primary goal of database: Store and Retrieve
- Convenient and efficient.
- Complex Data Structure
- Hiding the Complexity
- Several Levels of Abstraction:
- Three Levels:
 - Physical Level
 - Logical Level
 - View Level

Levels of Data Abstraction



Physical Level

- Lowest Level of abstraction
- Deals with how data are stored.
- Complex low level data structures.
- Database level.
- Storage level.

Logical Level

- Deals with what and relationship.
- Entire database with simple data structures.
- Physical Data Independence.
- DBA.

View Level

- Highest Level
- Users and Access.
- Interaction with the system.
- Applications programs.
- Several views and Security.
- E.g. ATM and Bank

Database Languages

- A **database** is a collection of organized information or data that is stored on a computer.
- Database Language:
- **Database language** refers to the specific types of commands or instructions used to communicate with a database.
- It helps users or applications to create, manage, update, and retrieve data from a database.
- Types of Database Languages in DBMS
 - **DDL**
 - **DML**
 - **DCL**
 - **TCL**

DBMS Languages



```
graph TD; A[DBMS Languages] --> B((DDL)); A --> C((DCL)); A --> D((DML)); A --> E((TCL));
```

DDL

DCL

DML

TCL

- **DDL(Data Definition Language)**

- The DDL stands for [Data Definition Language](#), Which is used to define the database's internal structure and Pattern of the Database.
- The DDL is used for creating tables, indexes, constraints, and schema in the Database.
- By using DDL statements we can able to create the architecture of the required database.
- **Create**
- **Drop**
- **Alter**
- **Truncate**
- **Rename**
- **Comment**

- **DML:(Data Manipulation Language)**

- The [Data Manipulation Language](#) is used to Manipulate the data in the database by using different commands.
- **Select**
- **Insert**
- **Update**
- **Delete**
- **Merge**

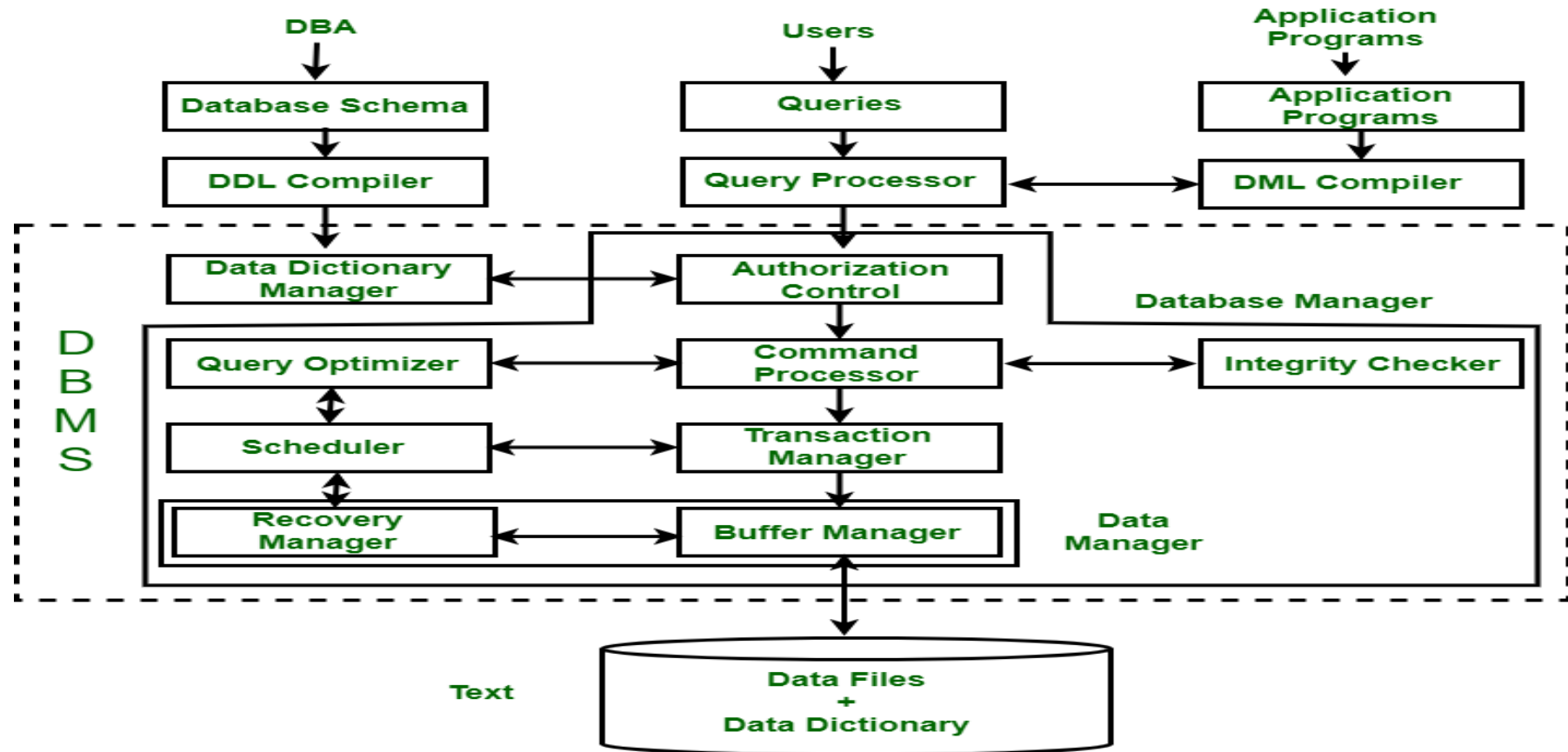
- **DCL (Data Control Language)**

- The DCL stands for Data Control Language means these commands are used to retrieve the saved data from the database.
- **Grant**
- **Revoke**

- **TCL (Transaction Control Language)**

- The TCL full form is [Transaction Control Language](#) commands are used to run the changes made by the DML commands one more thing is TCL can be grouped into a logical transaction.
- **Commit**
- **Rollback**

Database System Structure



- Structure of Database Management System is also referred to as Overall System Structure or Database Architecture but it is different from the tier architecture of Database.
- **Components of a Database System**
- Query Processor, Storage Manager, and Disk Storage. These are explained as following below.
- **1. Query Processor:**
- It interprets the requests (queries) received from end user via an application program into instructions. It also executes the user request which is received from the DML compiler.
- Query Processor contains the following components –
- **DML Compiler:** It processes the DML statements into low level instruction (machine language), so that they can be executed.
- **DDL Interpreter:** It processes the DDL statements into a set of table containing meta data (data about data).
- **Embedded DML Pre-compiler:** It processes DML statements embedded in an application program into procedural calls.
- **Query Optimizer:** It executes the instruction generated by DML Compiler.

- **2. Storage Manager:**

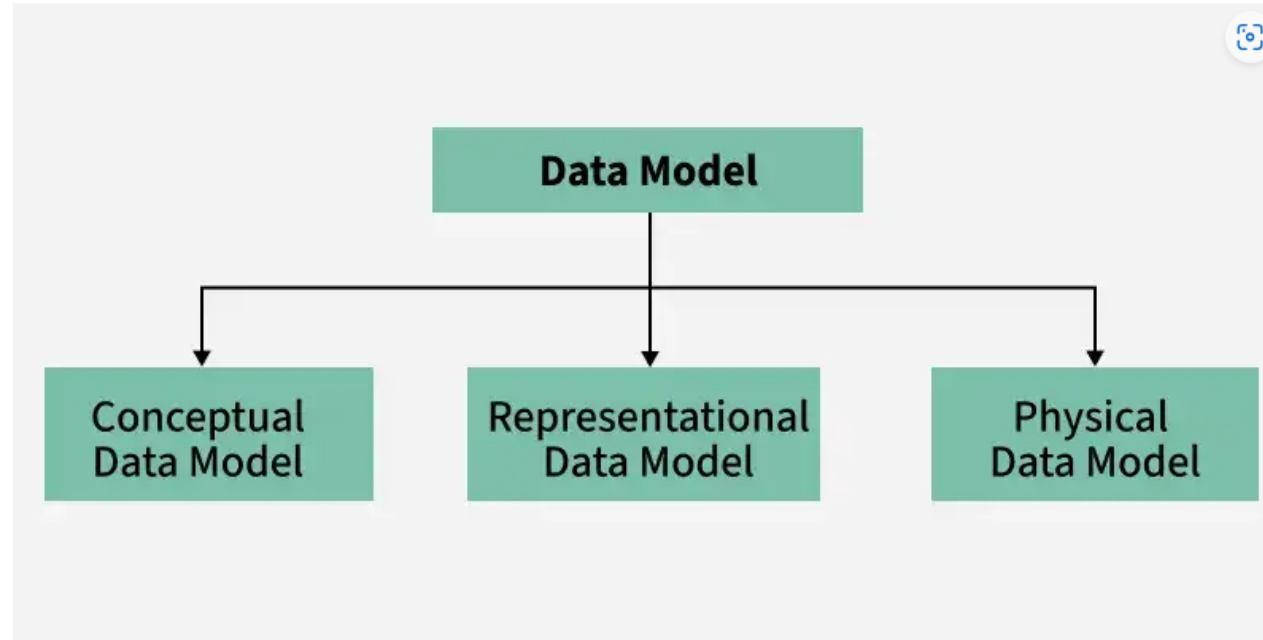
- Storage Manager is a program that provides an interface between the data stored in the database and the queries received.
- It is also known as Database Control System.
- It maintains the consistency and integrity of the database by applying the constraints and executing the DCL statements.
- It is responsible for updating, storing, deleting, and retrieving data in the database.
- It contains the following components –
- **Authorization Manager:** It ensures role-based access control, i.e., checks whether the particular person is privileged to perform the requested operation or not.
- **Integrity Manager:** It checks the integrity constraints when the database is modified.
- **Transaction Manager:** It controls concurrent access by performing the operations in a scheduled way that it receives the transaction. Thus, it ensures that the database remains in the consistent state before and after the execution of a transaction.

- **File Manager:** It manages the file space and the data structure used to represent information in the database.
- **Buffer Manager:** It is responsible for cache memory and the transfer of data between the secondary storage and main memory.
- **3. Disk Storage:**
 - It contains the following components:
 - **Data Files:** It stores the data.
 - **Data Dictionary:** It contains the information about the structure of any database object. It is the repository of information that governs the metadata.
 - **Indices:** It provides faster retrieval of data item.

Data Models

- A Data Model in Database Management System (DBMS) is the concept of tools that are developed to summarize the description of the database.
- Data Models provide us with a transparent picture of data which helps us in creating an actual database.
- It shows us from the design of the data to its proper implementation of data.

- Types of Data Models:



- **Conceptual Data Model:**

- The conceptual data model describes the database at a very high level and is useful to understand the needs or requirements of the database.
- It is used in the requirement-gathering process i.e. before the Database Designers start making a particular database.
- One such popular model is the entity/relationship model (ER model).
- The E/R model specializes in entities, relationships, and even attributes that are used by database designers.
- Discussion can be made with non-technical person or users and stakeholders and their requirements can be understood.

- **Characteristics of a conceptual data model:**

- Offers Organization-wide coverage of the business concepts.
- This type of Data Models are designed and developed for a business audience.
- The conceptual model is developed independently of hardware specifications like data storage capacity, location or software specifications like DBMS vendor and technology. The focus is to represent data as a user will see it in the “real world.”

- **2. Representational Data Model:**

- This type of data model is used to represent only the logical part of the database and does not represent the physical structure of the database.
- The representational data model allows us to focus primarily, on the design part of the database.
- A popular representational model is a Relational model.
- The relational Model consists of Relational Algebra and Relational Calculus.
- In the Relational Model, we basically use tables to represent our data and the relationships between them.
- It is a theoretical concept whose practical implementation is done in Physical Data Model.
- **Characteristics of Representational Data Model:**
- Represents the logical structure of the database.
- Relational models like Relational Algebra and Relational Calculus are commonly used.
- Uses tables to represent data and relationships.
- Provides a foundation for building the physical data model.

- **3. Physical Data Model:**

- The physical Data Model is used to practically implement Relational Data Model.
- All data in a database is stored physically on a secondary storage device such as discs and tapes.
- This is stored in the form of files, records, and certain other data structures.
- It has all the information on the format in which the files are present and the structure of the databases, the presence of external data structures, and their relation to each other.
- In order to come up with a good physical model, we have to work on the relational model in a better way.
- Structured Query Language (SQL) is used to practically implement Relational Algebra.
- This Data Model describes HOW the system will be implemented using a specific DBMS system.
- This model is typically created by DBA and developers.
- The purpose is actual implementation of the database.

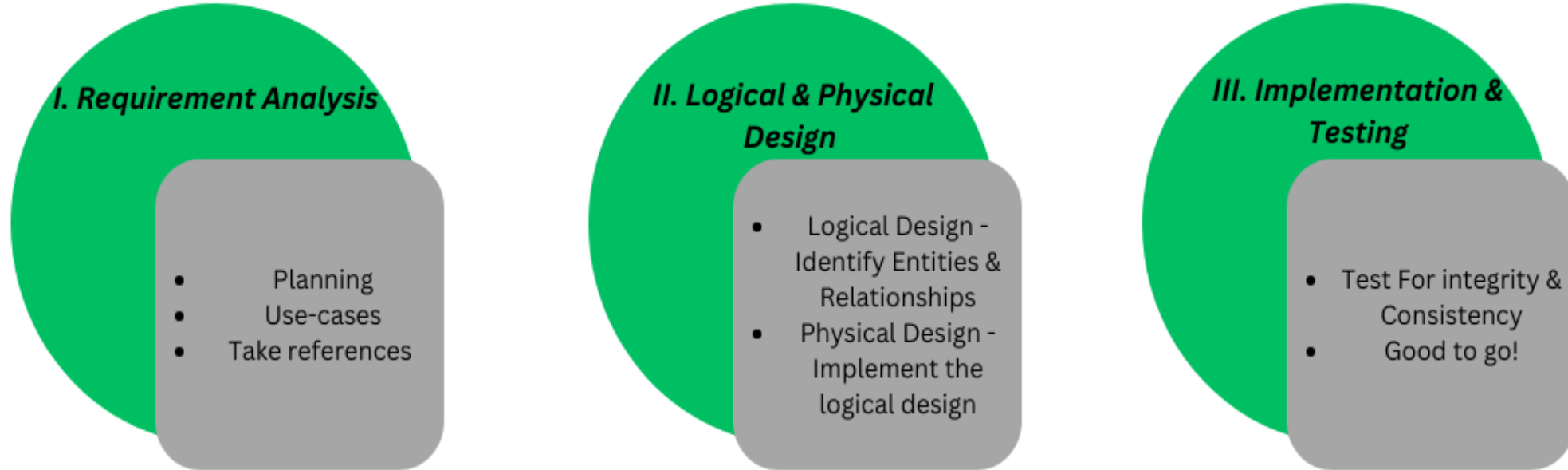
- **Characteristics of a physical data model:**

- The physical data model describes data need for a single project or application though it maybe integrated with other physical data models based on project scope.
- Data Model contains relationships between tables that which addresses cardinality and nullability of the relationships.
- Developed for a specific version of a DBMS, location, data storage or technology to be used in the project.
- Columns should have exact datatypes, lengths assigned and default values.
- Primary and Foreign keys, views, indexes, access profiles, and authorizations, etc. are defined

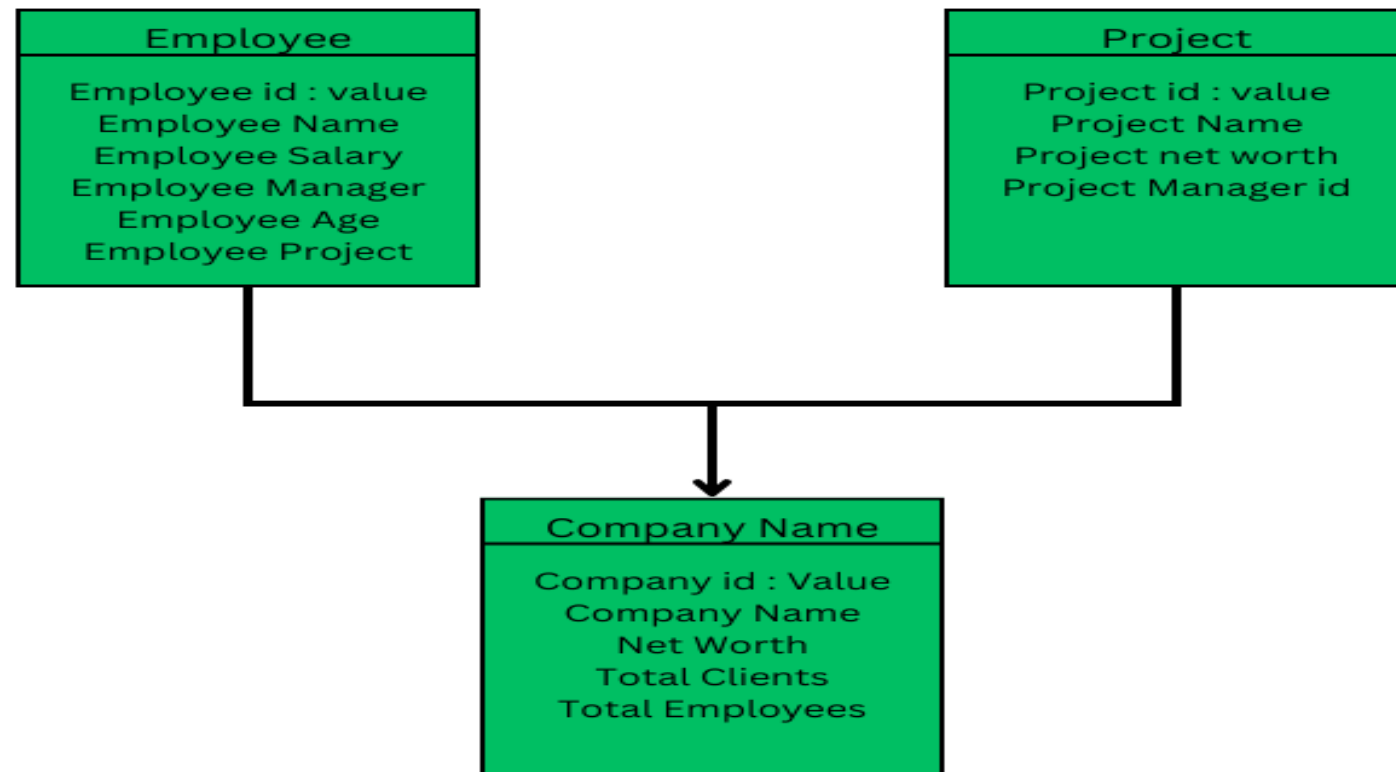
Database Design and ER Model:

- Database Design can be defined as a set of procedures or collection of tasks involving various steps taken to implement a database.
- A good database design is important.
- It helps you get the right information when you need it.
- Following are some critical points to keep in mind to achieve a good database design:
- **Data consistency and integrity must be maintained.**
- **Low Redundancy**
- **Faster searching through indices**
- **Security measures should be taken by enforcing various integrity constraints.**
- **Data should be stored in fragmented bits of information in the most atomic format possible.**

• Database Design Lifecycle:



- **An Example of Logical Design**



Logical Design

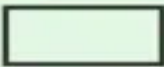




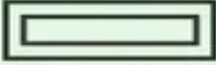
- The Entity Relationship Model is a model for identifying entities (like student, car or company) to be represented in the database and representation of how those entities are related.
- The ER data model specifies enterprise schema that represents the overall logical structure of a database graphically.

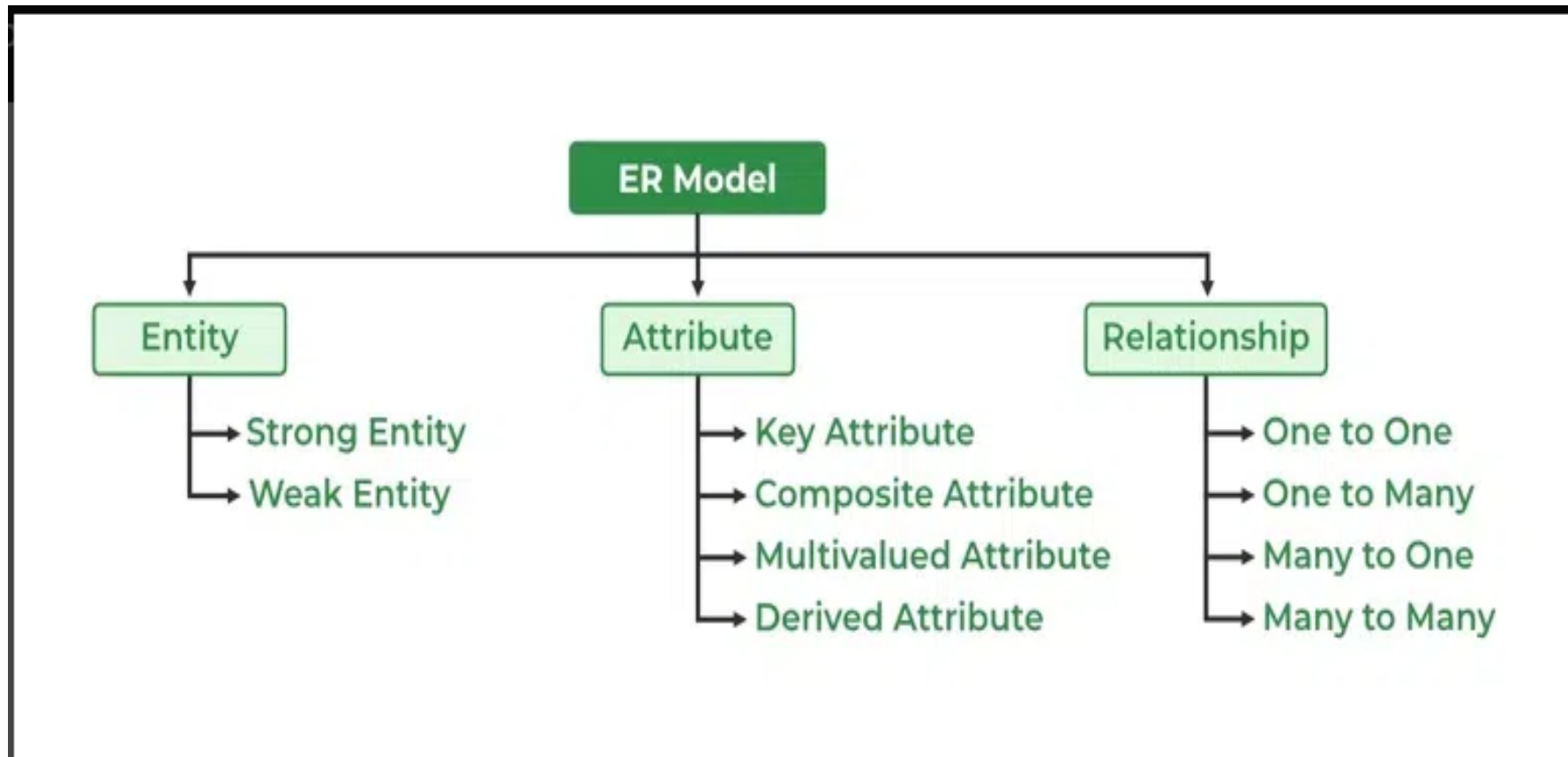
- **Why Use ER Diagrams In DBMS?**

- ER diagrams represent the E-R model in a database, making 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 of the underlying DBMS used.
- 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.

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



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:

The diagram consists of a large rectangular frame. Inside this frame, at the top, is a smaller rectangle labeled "Student". Below this rectangle is the text "Entity Type". Further down is an oval labeled "Entity Set" which contains three vertically stacked labels: "E1", "E2", and "E3".

Student

Entity Type

E1

E2

E3

Entity Set

- **Types of Entity:**

- 1. **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.

- 2. **Weak Entity:**

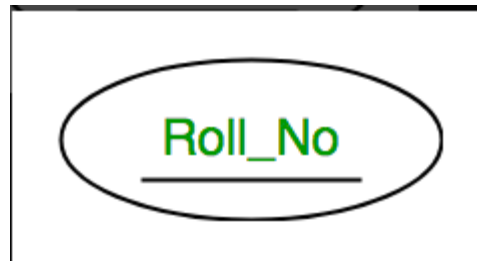
- A weak entity type is represented by a Double Rectangle. The participation of weak entity types is always total. The relationship between the weak entity type and its identifying strong entity type is called identifying relationship and it is represented by a double diamond.

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.

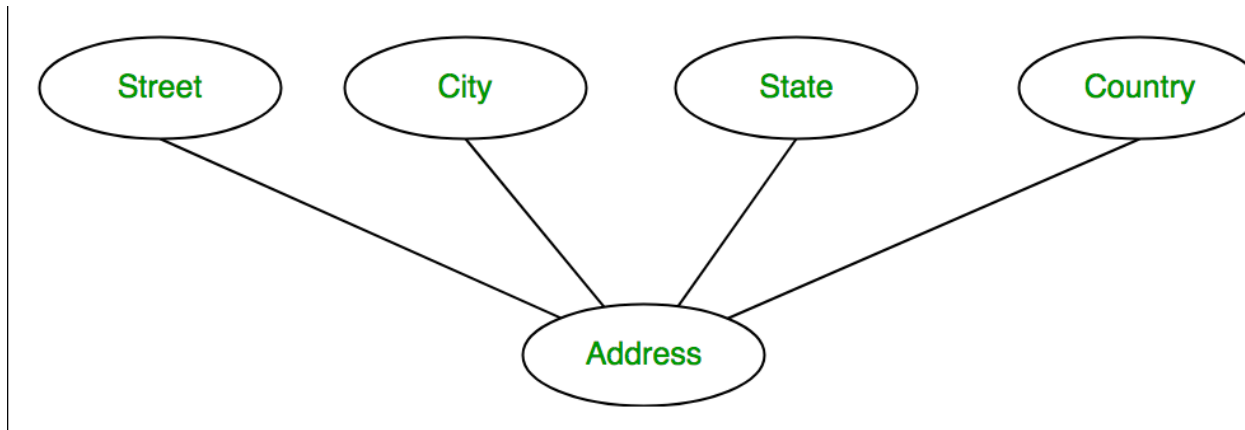
Types of Attributes

- **1. 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.



- **2. 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.



- **3. 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.



Relationships

Keys

- Keys are one of the basic requirements of a relational database model. It is widely used to identify the tuples(rows) uniquely in the table. We also use keys to set up relations amongst various columns and tables of a relational database.
- **Need of Keys in DBMS:**
- We require keys in a DBMS to ensure that data is organized, accurate, and easily accessible.
- Keys help to uniquely identify records in a table, which prevents duplication and ensures data integrity.
- Keys also establish relationships between different tables, allowing for efficient querying and management of data.
- Without keys, it would be difficult to retrieve or update specific records, and the database could become inconsistent or unreliable.

- **Different Types of Database Keys:**
- **Super Key**
- **Candidate Key**
- **Primary Key**
- **Alternate Key**
- **Foreign Key**

- **Super Key:**

- The set of one or more attributes (columns) that can uniquely identify a tuple (record) is known as Super Key. For Example, STUD_NO, (STUD_NO, STUD_NAME), etc.
- A super key is a group of single or multiple keys that uniquely identifies rows in a table. It supports NULL values in rows.

Example:

Table STUDENT

STUD_NO	SNAME	ADDRESS	PHONE
1	Shyam	Delhi	123456789
2	Rakesh	Kolkata	223365796
3	Suraj	Delhi	175468965

- **Candidate Key:**

- The minimal set of attributes that can uniquely identify a tuple is known as a candidate key. For Example, STUD_NO in STUDENT relation.
- A candidate key is a minimal super key, meaning it can uniquely identify a record but contains no extra attributes.
- It is a super key with no repeated data is called a candidate key.
- The minimal set of attributes that can uniquely identify a record.
- A candidate key must contain unique values, ensuring that no two rows have the same value in the candidate key's columns.
- Every table must have at least a single candidate key.
- A table can have multiple candidate keys but only one primary key.

Table STUDENT_COURSE

STUD_NO	TEACHER_NO	COURSE_NO
1	001	C001
2	056	C005

Primary Key

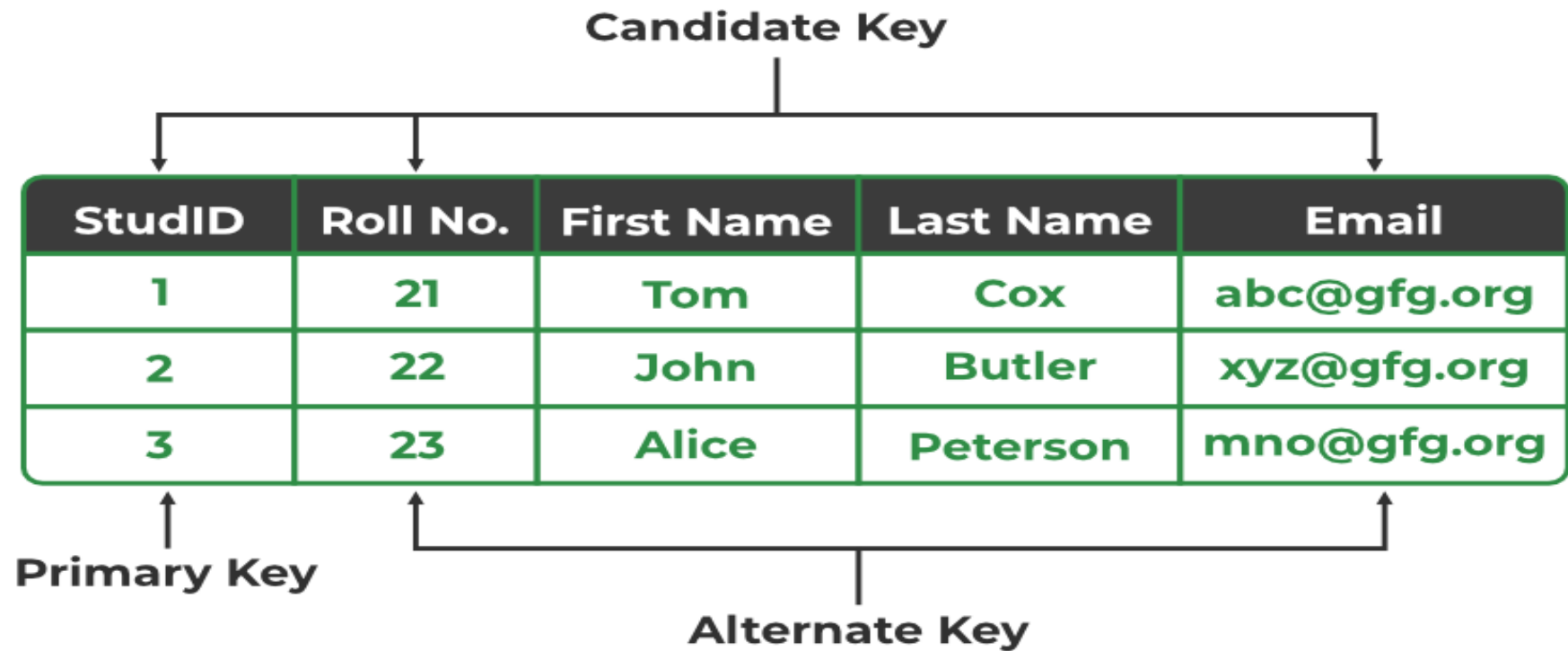
- There can be more than one candidate key in relation out of which one can be chosen as the primary key. For Example, STUD_NO, as well as STUD_PHONE, are candidate keys for relation STUDENT but STUD_NO can be chosen as the primary key (only one out of many candidate keys).
- A primary key is a unique key, meaning it can uniquely identify each record (tuple) in a table.
- It must have unique values and cannot contain any duplicate values.
- A primary key cannot be NULL, as it needs to provide a valid, unique identifier for every record.
- A primary key does not have to consist of a single column. In some cases, a composite primary key (made of multiple columns) can be used to uniquely identify records in a table.
- Databases typically store rows ordered in memory according to primary key for fast access of records using primary key.

Table STUDENT

STUD_NO	SNAME	ADDRESS	PHONE
1	Shyam	Delhi	123456789
2	Rakesh	Kolkata	223365796
3	Suraj	Delhi	175468965

- **Alternate Key:**

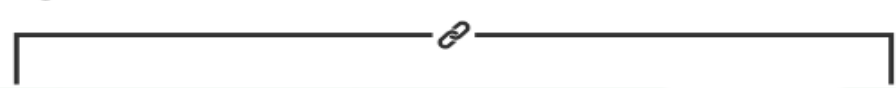
- An alternate key is any candidate key in a table that is **not** chosen as the **primary key**. In other words, all the keys that are not selected as the primary key are considered alternate keys.
- An alternate key is also referred to as a **secondary key** because it can uniquely identify records in a table, just like the primary key.
- An alternate key can consist of **one or more columns** (fields) that can uniquely identify a record, but it is not the primary key
- Eg:- SNAME, and ADDRESS is Alternate keys



- **Foreign Key:**

- A foreign key is an attribute in one table that refers to the primary key in another table. The table that contains the foreign key is called the referencing table, and the table that is referenced is called the referenced table.
- A foreign key in one table points to the primary key in another table, establishing a relationship between them.
- It helps connect two or more tables, enabling you to create relationships between them. This is essential for maintaining data integrity and preventing data redundancy.
- They act as a cross-reference between the tables.
- For example, DNO is a primary key in the DEPT table and a non-key in EMP

Primary Key



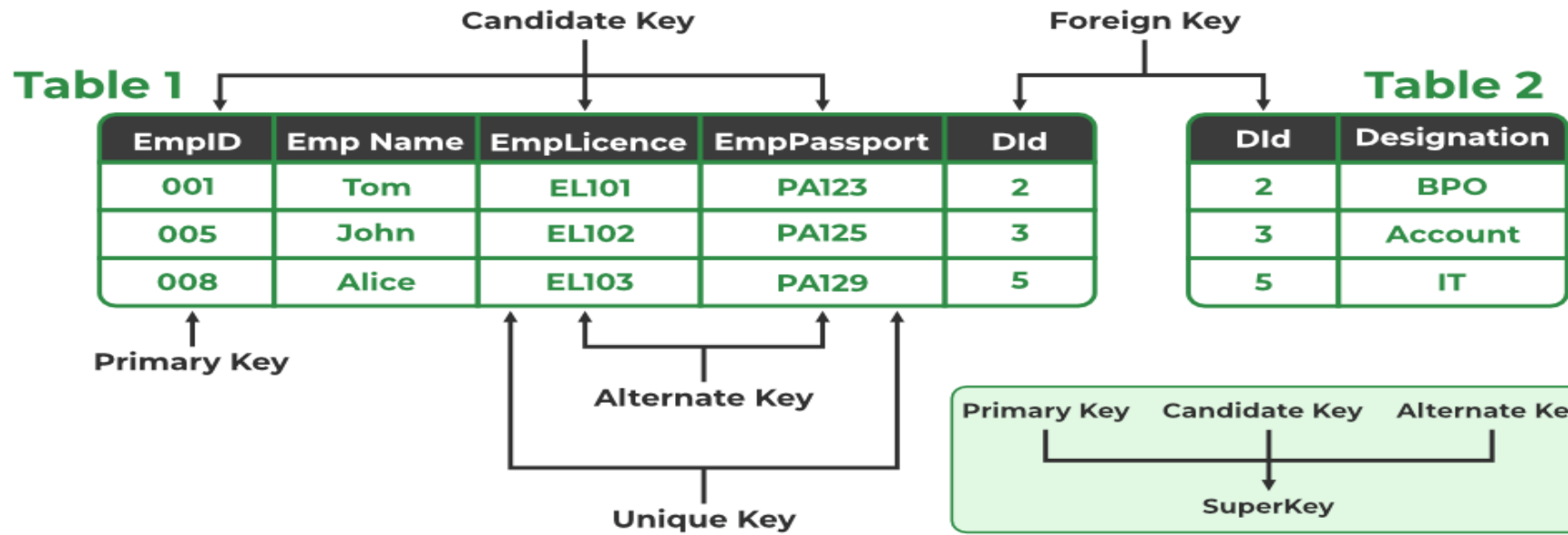
ID	Name	Course
2041	Tom	Java
2204	John	C++
2043	Alice	Python
2032	Bob	Oracle

Student Details

Foreign Key

ID	Marks
2041	65
2204	55
2043	73
2032	62

Student Marks

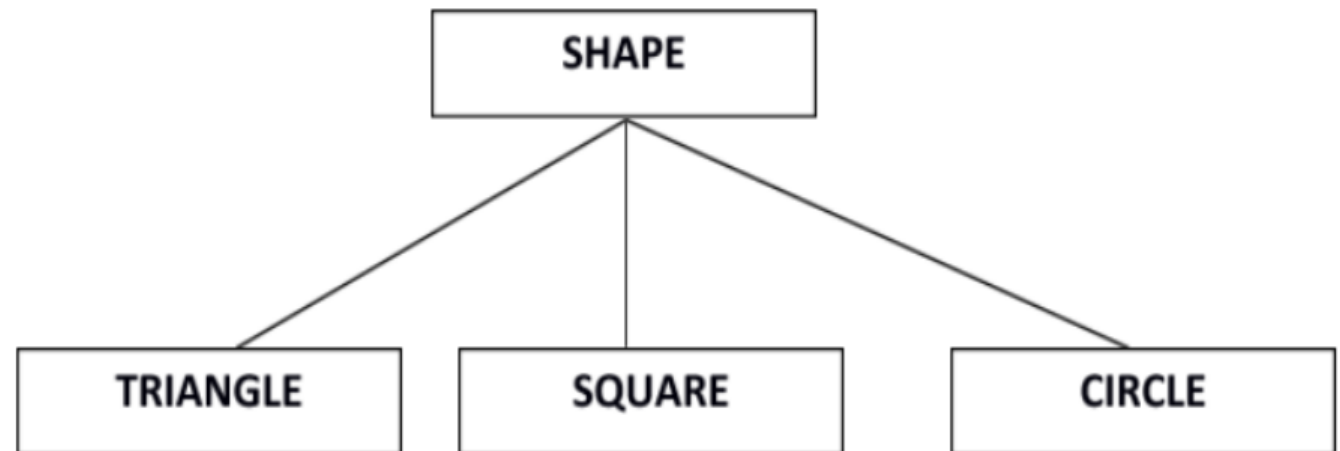


Extended-ER Diagram(EER)

- EER is a high-level data model that incorporates the extensions to the original ER model.
- Enhanced ERD are high level models that represent the requirements and complexities of complex database
- In addition to ER model concepts EE-R includes –.
- **Subclasses and Super classes.**
- **Specialization and Generalization.**
- **Category or union type.**
- **Aggregation.**

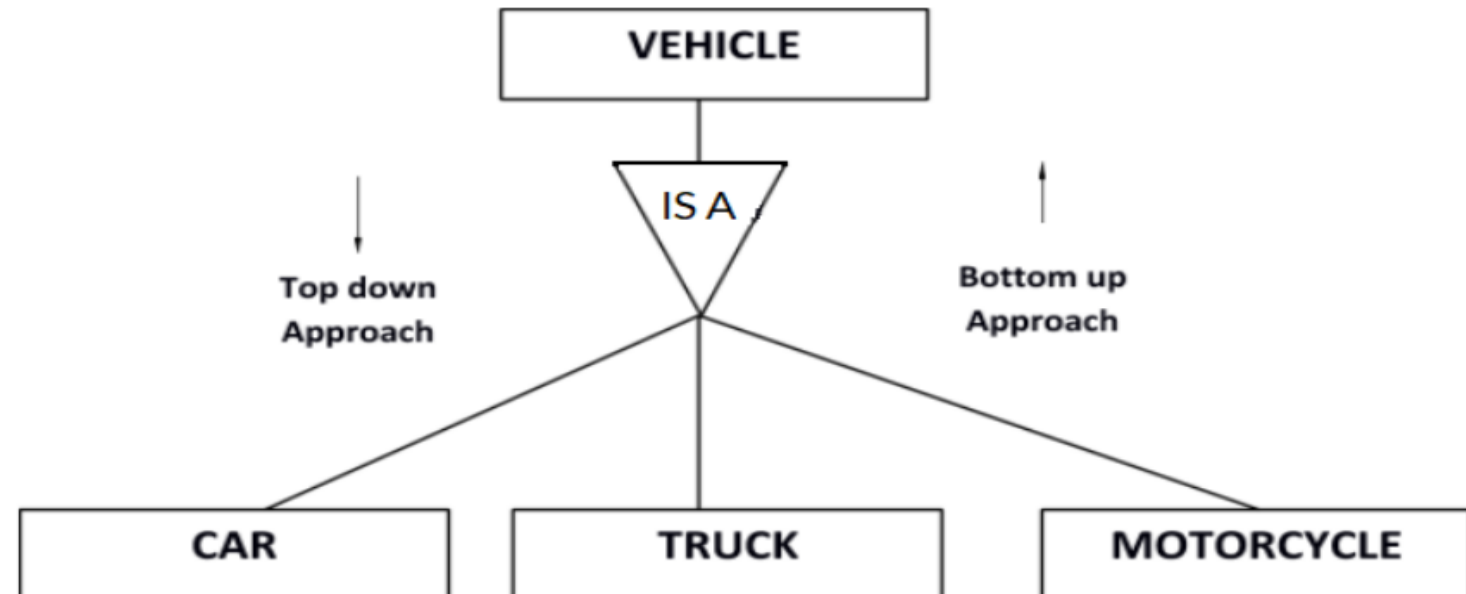
- **Subclasses and Super class**

- Super that can be divided into further subtype.
- Sub classes class is an entity are the group of entities with some unique attributes. Sub class inherits the properties and attributes from super class.



- **Specialization and Generalization:**

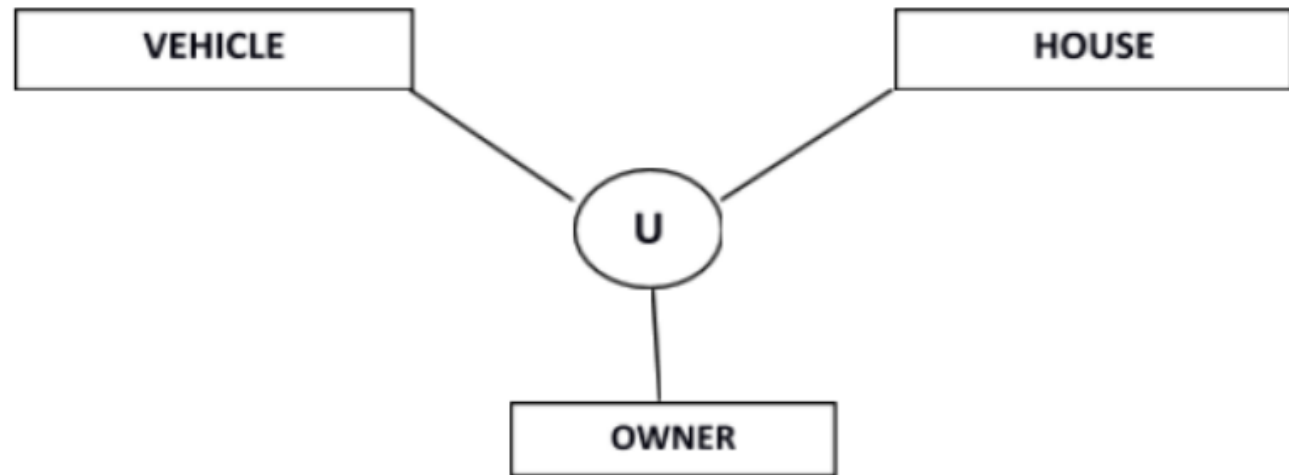
- Generalization is a process of generalizing an entity which contains generalized attributes or properties of generalized entities.



- It is a Bottom up process i.e. consider we have 3 sub entities Car, Truck and Motorcycle. Now these three entities can be generalized into one super class named as Vehicle.
- Specialization is a process of identifying subsets of an entity that share some different characteristic.
- It is a top down approach in which one entity is broken down into low level entity.
- In above example Vehicle entity can be a Car, Truck or Motorcycle.

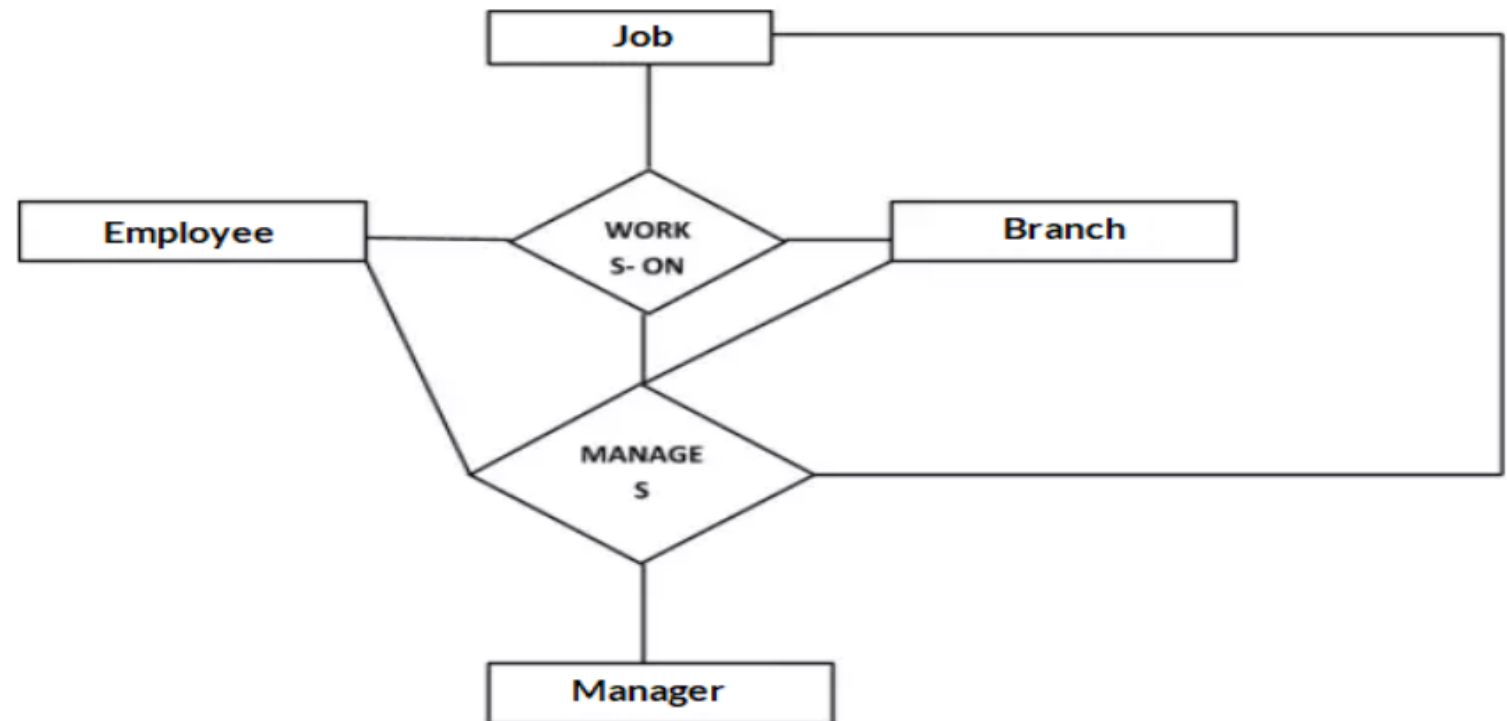
- **Category or Union**

- Relationship of one super or sub class with more than one super class.



- **Aggregation:**

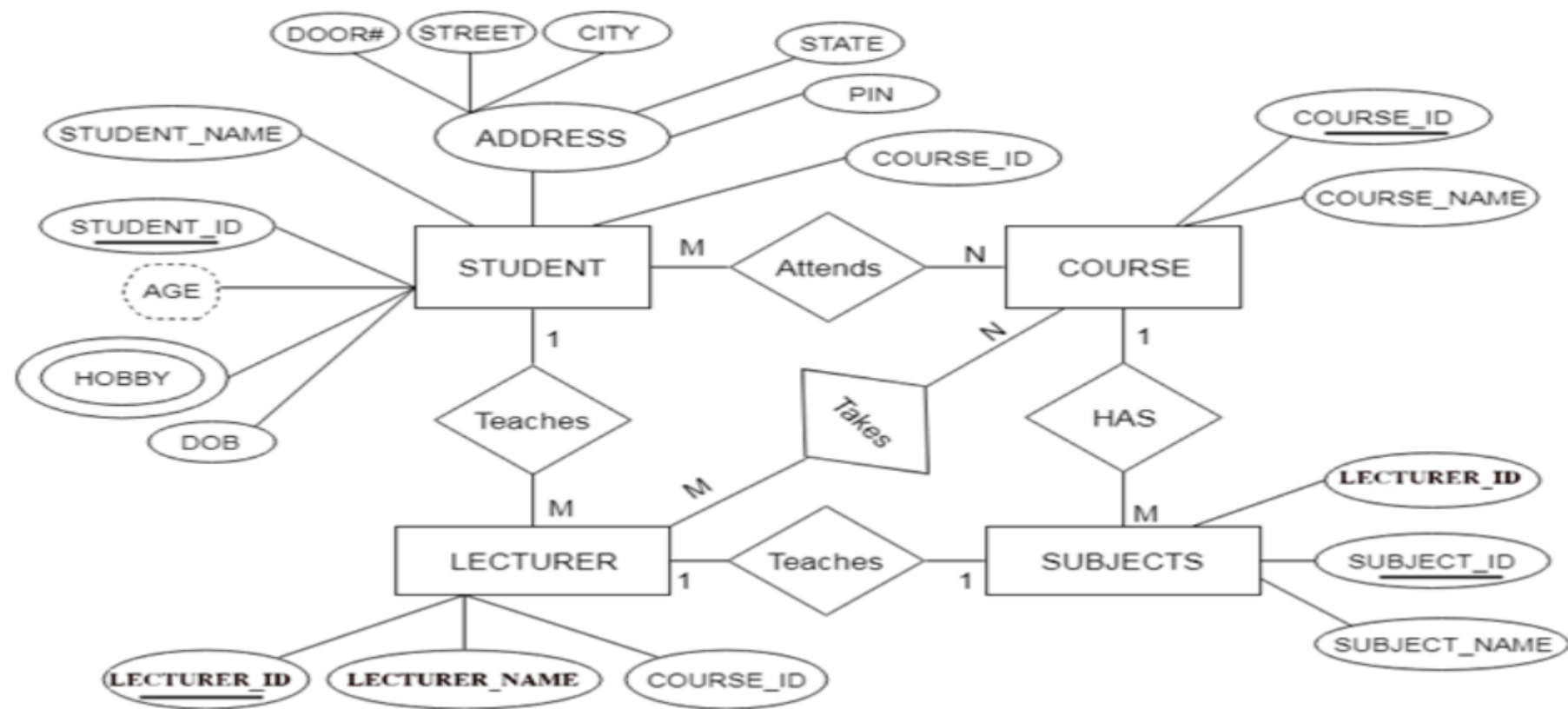
- Represents relationship between a whole object and its component.



- Consider a ternary relationship Works_On between Employee, Branch and Manager.
- The best way to model this situation is to use aggregation, So, the relationship-set, Works_On is a higher level entity-set.
- Such an entity-set is treated in the same manner as any other entity-set.
- We can create a binary relationship, Manager, between Works_On and Manager to represent who manages what tasks.

Converting ER Diagrams into tables

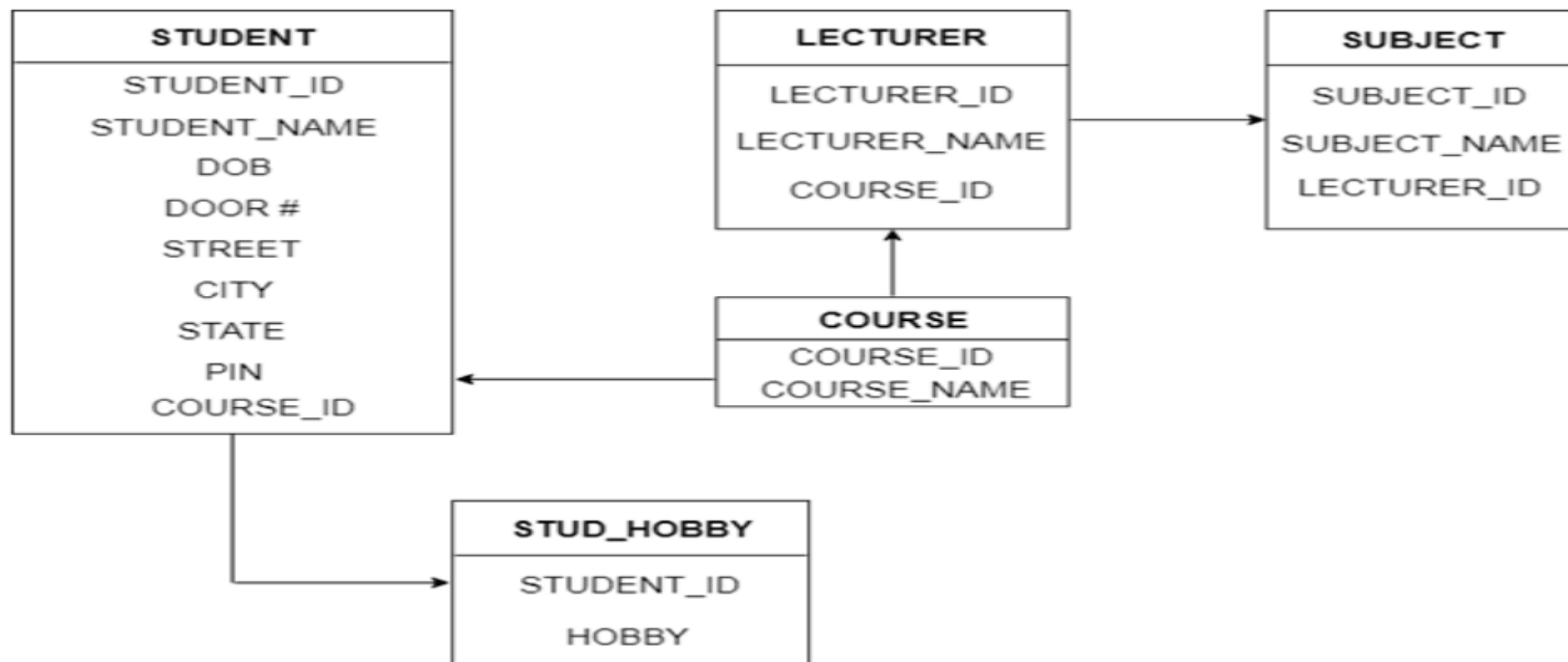
- The database can be represented using the notations, and these notations can be reduced to a collection of tables.
- In the database, every entity set or relationship set can be represented in tabular form.



- There are some points for converting the ER diagram to the table:
- **Entity type becomes a table.**
- In the given ER diagram, LECTURE, STUDENT, SUBJECT and COURSE forms individual tables.
- **All single-valued attribute becomes a column for the table.**
- In the STUDENT entity, STUDENT_NAME and STUDENT_ID form the column of STUDENT table. Similarly, COURSE_NAME and COURSE_ID form the column of COURSE table and so on.
- **A key attribute of the entity type represented by the primary key.**
- In the given ER diagram, COURSE_ID, STUDENT_ID, SUBJECT_ID, and LECTURE_ID are the key attribute of the entity.

- **The multivalued attribute is represented by a separate table.**
- In the student table, a hobby is a multivalued attribute. So it is not possible to represent multiple values in a single column of STUDENT table. Hence we create a table STUD_HOBBY with column name STUDENT_ID and HOBBY. Using both the column, we create a composite key.
- **Composite attribute represented by components.**
- In the given ER diagram, student address is a composite attribute. It contains CITY, PIN, DOOR#, STREET, and STATE. In the STUDENT table, these attributes can merge as an individual column.
- **Derived attributes are not considered in the table.**
- In the STUDENT table, Age is the derived attribute. It can be calculated at any point of time by calculating the difference between current date and Date of Birth.

- Using these rules, you can convert the ER diagram to tables and columns and assign the mapping between the tables. Table structure for the given ER diagram is as below:



Unit 2

SQL and PL/SQL

SQL: Characteristics and Advantages

- SQL stands for Structured Query Language.
- It was first developed in the 1970s by IBM researchers, and has since become a standard language for managing and querying databases across various platforms and industries.
- It is used for storing and managing data in relational database management system (RDMS).
- In RDBMS data stored in the form of the tables.
- It is a standard language for Relational Database System.
- It enables a user to create, read, update and delete relational databases and tables.
- SQL allows users to query the database in a number of ways, using English-like statements.

- **Need of SQL :**

- It is widely used in the Business Intelligence tool.
- Data Manipulation and data testing are done through SQL.
- Data Science tools depend highly on SQL. Big data tools such as Spark, Impala are dependent on SQL.
- It is one of the demanding industrial skills.

- **Characteristics of SQL:**

- SQL is easy to learn.
- SQL is used to access data from relational database management systems.
- SQL can execute queries against the database.
- SQL is used to describe the data.
- SQL is used to define the data in the database and manipulate it when needed.
- SQL is used to create and drop the database and table.
- SQL is used to create a view, stored procedure, function in a database.
- SQL allows users to set permissions on tables, procedures, and views.

- **Advantages of SQL :**

- **Faster Query Processing:** Large amount of data is retrieved quickly and efficiently. Operations like Insertion, deletion, manipulation of data is also done in almost no time.
- **No Coding Skills:** For data retrieval, large number of lines of code is not required. All basic keywords such as SELECT, INSERT INTO, UPDATE, etc are used and also the syntactical rules are not complex in SQL, which makes it a user-friendly language.
- **Standardized Language:** Due to documentation and long establishment over years, it provides a uniform platform worldwide to all its users.
- **Portable:** It can be used in programs in PCs, server, laptops independent of any platform (Operating System, etc). Also, it can be embedded with other applications as per need/requirement/use.

- **Interactive Language** : Easy to learn and understand, answers to complex queries can be received in seconds.
- **Multiple data views** : One of the advantages of SQL is its ability to provide **multiple data views** . This means that SQL allows users to create different views or perspectives of the data stored in a database, depending on their needs and permissions.
- **Scalability** : SQL databases can handle large volumes of data and can be scaled up or down as per the requirements of the application.
- **Security** : SQL databases have built-in security features that help protect data from unauthorized access, such as user authentication, encryption, and access control.

- **Data Integrity** : SQL databases enforce data integrity by enforcing constraints such as unique keys, primary keys, and foreign keys, which help prevent data duplication and maintain data accuracy.
- **Backup and Recovery** : SQL databases have built-in backup and recovery tools that help recover data in case of system failures, crashes, or other disasters.
- **Data Consistency**: SQL databases ensure consistency of data across multiple tables through the use of transactions, which ensure that changes made to one table are reflected in all related tables.

SQL Data Types and Literals

- An [SQL](#) developer must know what data type will be stored inside each column while creating a table.
- The data type guideline for SQL is to understand what type of data is expected inside each column and it also identifies how SQL will interact with the stored data.
- **Numeric Datatypes**
- **Character and String Data Types**
- **Date and Time Data Types**
- **Binary Data Types**
- **Boolean Data Types**
- **Special Data Types**

- **Numeric Data Types:**

- Numeric data types are fundamental to database design and are used to store numbers, whether they are integers, decimals, or floating-point numbers.
- These data types allow for mathematical operations like addition, subtraction, multiplication, and division, which makes them essential for managing financial, scientific, and analytical data.

- **Exact Numeric Datatype:**

- Exact numeric types are used when precise numeric values are needed, such as for financial data, quantities, and counts.

Data Type	From	To
BigInt	-2^{63} (-9,223,372,036,854,775,808)	$2^{63} - 1$ (9,223,372,036,854,775,807)
Int	-2^{31} (-2,147,483,648)	$2^{31} - 1$ (2,147,483,647)
smallint	-2^{15} (-32,768)	$2^{15} - 1$ (32,767)
tinyint	0	$2^8 - 1$ (255)
bit	0	1
decimal	$-10^{38} + 1$	$10^{38} - 1$
numeric	$-10^{38} + 1$	$10^{38} - 1$
money	-922,337,203,685,477.5808	922,337,203,685,477.5807

Approximate Numeric Datatype

These types are used to store approximate values, such as scientific measurements or large ranges of data that don't need exact precision.

Data Type	From	To
Float	-1.79E+308	1.79E+308
Real	-3.40E+38	3.40E+38

String Data Types

Character data types are used to store text or character-based data.

Character String Datatype

The subtypes are given in below table

Data Type	Description
char	The maximum length of 8000 characters.(Fixed-Length non-Unicode Characters)
varchar	The maximum length of 8000 characters.(Variable-Length non-Unicode Characters)
varchar(max)	The maximum length of 231 characters(SQL Server 2005 only).(Variable Length non-Unicode data)
text	The maximum length of 2,127,483,647 characters(Variable Length non-Unicode data)

Date and Time Data Type in SQL

SQL provides several data types for storing date and time information. These are given in the below table.

Data Type	Description
DATE	A data type is used to store the data of date in a record
TIME	A data type is used to store the data of time in a record
DATETIME	A data type is used to store both the data,date, and time in the record.

Boolean Data Type in SQL

Boolean data types are used to store logical values, typically TRUE or FALSE.

Data Type	Description
BOOLEAN	Stores a logical value (TRUE/FALSE).

DDL, DML, DCL, TCL

- **SQL CREATE TABLE Statement**

- To create a new table in the database, use the SQL CREATE TABLE statement. A table's structure, including column names, data types, and constraints like NOT NULL, PRIMARY KEY, and CHECK, are defined when it is created in SQL.
- The CREATE TABLE command is a crucial tool for database administration because of these limitations, which aid in ensuring data integrity.

Syntax:

To create a table in SQL, use this **CREATE TABLE** syntax:

```
CREATE table table_name  
(  
  Column1 datatype (size),  
  column2 datatype (size),  
  .  
  .  
  columnN datatype(size)  
);
```

- **table_name**: The name you assign to the new table.
- **column1, column2, ...** : The names of the columns in the table.
- **datatype(size)**: Defines the data type and size of each column.

- To add data to the table, we use **INSERT INTO** command, the syntax is as shown below:

Syntax:

- ***INSERT INTO*** *table_name* (*column1, column2, ...*) ***VALUES*** (*value1, value2, ...*);

- **Create Table From Another Table**

Syntax:

- ***CREATE TABLE*** *new_table_name* ***AS***
 SELECT *column1, column2, ...*
 FROM *existing_table_name*
 WHERE;

Syntax:

DROP object object_name ;

DROP and TRUNCATE in SQL

- The DROP and **TRUNCATE** commands in SQL are used to remove data from a table, but they work differently
- **DROP:**
- In SQL, the DROP command is used to **permanently remove** an object from a database, such as a table, **database**, index, or **view**.
- When you DROP a table, the table structure and its data are permanently deleted, leaving no trace of the object.
- **Syntax:**
- ***DROP object object_name ;***

- **DROP Table**

- **Syntax:**

- *DROP TABLE table_name;*

- **TRUNCATE**

- The TRUNCATE command is a Data Definition Language ([DDL](#)) action that removes all rows from a table but preserves the structure of the table for future use.
- Although [TRUNCATE](#) is similar to the DELETE command (without the WHERE clause), it is much faster because it bypasses certain integrity constraints and locks. It was officially introduced in the SQL:2008 standard.

- **Syntax:**
- *TRUNCATE TABLE table_name;*
- **Where,**
- **table_name:** Name of the table to be truncated.
- **DATABASE name:** student_data

- **SQL ALTER TABLE**

- The SQL ALTER TABLE statement is a powerful tool that allows you to modify the structure of an existing table in a database.
- Whether you're adding new columns, modifying existing ones, deleting columns, or renaming them, the ALTER TABLE statement enables you to make changes without losing the data stored in the table.

- **SQL ALTER TABLE STATEMENT**

- The ALTER TABLE statement in [SQL](#) is used to add, remove, or modify columns in an existing table. The ALTER TABLE statement is also used to add and remove various constraints on existing tables.
- It allows for structural changes like adding new columns, modifying existing ones, deleting columns, and renaming columns within a table.

- **Syntax:**
- To alter/modify the table use the ALTER TABLE syntax:
- Alter table table_name
 clause[column _name] [datatype];

Common Use Cases for SQL ALTER TABLE

- **1. ADD – To add a new column to the table**
- The ADD clause is used to add a new column to an existing table. You must specify the name of the new column and its data type.
- **Query:**
- ALTER TABLE table_name
ADD column_name datatype;

- **2. MODIFY/ALTER – To change the data type of an existing column**
- The MODIFY (or ALTER COLUMN in some databases like SQL Server) clause is used to modify the definition of an existing column, such as changing its data type or size.
- **Query:**
- ALTER TABLE table_name
 Modify Column_name datatype;

- **3. DROP – To delete an existing column from the table**
- The DROP clause allows you to remove a column from a table. Be cautious when using this command as it will permanently remove the column and its data.
- ALTER TABLE table_name
DROP Column Column_name;

- **4. RENAME COLUMN – To rename an existing column**
- You can rename an existing column using the RENAME COLUMN clause. This allows you to change the name of a column while preserving its data type and content.
- **Query:**
- **Alter table table_name**
Rename to new_table_name;

Data Manipulation Language (DML)

- The **SQL commands** that deal with the **manipulation of data** present in the database belong to **DML** or Data Manipulation Language and this includes most of the **SQL statements**.
- **SQL SELECT Query:**
 - The select query in SQL is one of the most commonly used **SQL** commands to retrieve data from a **database**.
 - The **SELECT statement** in **SQL** is used to fetch or retrieve data from a database. It allows users to access the data and retrieve specific data based on specific conditions.

- **Syntax:**

- The syntax for the SELECT statement is:
- *SELECT column1,column2.... FROM table_name ;*
-

2. SQL INSERT INTO Statement:

- The INSERT INTO statement is a fundamental SQL command used to add new rows of data into a table in a database. It is one of the most commonly used SQL statements for manipulating data and plays a key role in database management.

- There are two primary syntaxes of **INSERT INTO** statements depending on the requirements. The two syntaxes are:
- **Syntax 1. Only Values**
- The first method is to specify only the value of data to be inserted without the column names.
- *INSERT INTO table_name
VALUES (value1, value2, value);*
- **Parameters:**
- **table_name:** name of the table.
- **value1, value2:** value of first column, second column,... for the new record

- **Syntax 2. Column Names And Values Both**
- In the second method we will specify both the columns which we want to fill and their corresponding values as shown below:
- *INSERT INTO table_name (column1, column2, column3)
VALUES (value1, value2, value);*
- **Parameters:**
- **table_name:** name of the table.
- **column1, column2..:** name of first column, second column.
- **value1, value2, value..:** the values for each specified column of the new record.

- **SQL UPDATE Statement**

- The UPDATE statement in [SQL](#) is used to update the data of an existing table in the database. We can update single columns as well as multiple columns using the UPDATE statement as per our requirement.
- **Syntax:**
- The syntax for SQL UPDATE Statement is :
- *UPDATE table_name SET column1 = value1, column2 = value2,...
WHERE condition;*
- Where,
- **table_name**: name of the table
- **column1**: name of first, second, third column....
- **value1**: new value for first, second, third column....
- **condition**: condition to select the rows for which the

- **SQL DELETE Statement**

- The **SQL DELETE** statement is one of the most commonly used **commands** in SQL (Structured Query Language).
- It allows you to remove one or more rows from the table depending on the situation.
- Unlike the **DROP statement**, which removes the entire table, the **DELETE statement** removes data (rows) from the table retaining only the table structure, constraints, and schema
- **Syntax:**
- *DELETE FROM table_name*
WHERE some_condition;

SQL Operators

- The SQL reserved words and characters are called operators, which are used with a WHERE clause in a SQL query.
- In SQL, an operator can either be a unary or binary operator.
- The unary operator uses only one operand for performing the unary operation, whereas the binary operator uses two operands for performing the binary operation.
- **Precedence of SQL Operator**
 - The precedence of SQL operators is the sequence in which the SQL evaluates the different operators in the same expression.
 - Structured Query Language evaluates those operators first, which have high precedence.

- In the following table, the operators at the top have high precedence, and the operators that appear at the bottom have low precedence.

SQL Operator Symbols	Operators
**	Exponentiation operator
+, -	Identity operator, Negation operator
*, /	Multiplication operator, Division operator
+, -,	Addition (plus) operator, subtraction (minus) operator, String Concatenation operator
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparison Operators
NOT	Logical negation operator
&& or AND	Conjunction operator
OR	Inclusion operator

- For Example,

UPDATE employee

SET salary = 20 - 3 * 5 WHERE Emp_Id = 5;

Types of Operator

- SQL operators are categorized in the following categories:
- SQL Arithmetic Operators
- SQL Comparison Operators
- SQL Logical Operators
- SQL Set Operators
- SQL Bit-wise Operators
- SQL Unary Operators

- **SQL Arithmetic Operators:**

- The **Arithmetic Operators** perform the mathematical operation on the numerical data of the SQL tables. These operators perform addition, subtraction, multiplication, and division operations on the numerical operands.
- **Following are the various arithmetic operators performed on the SQL data:**
 - SQL Addition Operator (+)
 - SQL Subtraction Operator (-)
 - SQL Multiplication Operator (*)
 - SQL Division Operator (/)
 - SQL Modulus Operator (%)

- **SQL Comparison Operators:**

- The **Comparison Operators** in SQL compare two different data of SQL table and check whether they are the same, greater, and lesser. The SQL comparison operators are used with the WHERE clause in the SQL queries
- SQL Equal Operator (=)
- SQL Not Equal Operator (!=)
- SQL Greater Than Operator (>)
- SQL Greater Than Equals to Operator (>=)
- SQL Less Than Operator (<)\
- SQL Less Than Equals to Operator (<=)

- **SQL Logical Operators:**

- The **Logical Operators** in SQL perform the Boolean operations, which give two results **True** and **False**. These operators provide **True** value if both operands match the logical condition.
- **Following are the various logical operators which are performed on the data stored in the SQL database tables:**
 - SQL ALL operator
 - SQL AND operator
 - SQL OR operator
 - SQL BETWEEN operator
 - SQL IN operator
 - SQL NOT operator
 - SQL ANY operator
 - SQL LIKE operator

- **SQL ALL Operator**

- The ALL operator in SQL compares the specified value to all the values of a column from the sub-query in the SQL database.
- This operator is always used with the following statement:
- SELECT,
- HAVING, and
- WHERE.
- **Syntax of ALL operator:**
- SELECT column_Name1, ..., column_NameN FROM table_Name WHERE column Comparison_operator ALL (SELECT column FROM tablename2)

- If we want to access the employee id and employee names of those employees from the table whose salaries are greater than the salary of employees who lives in Jaipur city, then we have to type the following query in SQL.

This example consists of an **Employee_details** table, which has three columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Gurgaon
202	Ankit	45000	Delhi
203	Bheem	30000	Jaipur
204	Ram	29000	Mumbai
205	Sumit	40000	Kolkata

- `SELECT Emp_Id, Emp_Name FROM Employee_details WHERE Emp_Salary > ALL (SELECT Emp_Salary FROM Employee_details WHERE Emp_City = Jaipur)`
- **SQL AND Operator**
- The **AND operator** in SQL would show the record from the database table if all the conditions separated by the AND operator evaluated to True. It is also known as the conjunctive operator and is used with the WHERE clause.
- **Syntax of AND operator:**
- `SELECT column1, ..., columnN FROM table_Name WHERE condition1 AND condition2 AND condition3 AND AND conditionN;`
- Example.
- **Suppose, we want to access all the records of those employees from the Employee_details table whose salary is 25000 and the city is Delhi.**

- **SQL OR Operator:**
- The OR operator in SQL shows the record from the table if any of the conditions separated by the OR operator evaluates to True.
- It is also known as the conjunctive operator and is used with the WHERE clause.
- **Syntax of OR operator:**
- SELECT column1, ..., columnN FROM table_Name WHERE condition1 OR condition2 OR condition3 OR OR conditionN;
- **Example**
- If we want to access all the records of those employees from the **Employee_details** table whose salary is 25000 or the city is Delhi.
- SELECT * FROM Employee_details WHERE Emp_Salary = 25000 OR Emp_City = 'Delhi' ;

- **SQL BETWEEN Operator:**
- The **BETWEEN operator** in SQL shows the record within the range mentioned in the SQL query. This operator operates on the numbers, characters, and date/time operands.
- If there is no value in the given range, then this operator shows NULL value.
- **Syntax of BETWEEN operator:**
- `SELECT column_Name1, column_Name2, column_NameN FROM table_Name WHERE column_name BETWEEN value1 and value2;`
- **E.g.**
- **Suppose, we want to access all the information of those employees from the Employee_details table who is having salaries between 20000 and 40000.**
- `SELECT * FROM Employee_details WHERE Emp_Salary BETWEEN 30000 AND 45000;`

- **SQL IN Operator**

- The **IN operator** in SQL allows database users to specify two or more values in a WHERE clause. This logical operator minimizes the requirement of multiple OR conditions.
- This operator makes the query easier to learn and understand. This operator returns those rows whose values match with any value of the given list.
- **Syntax of IN operator:**
- SELECT column_Name1, column_Name2, column_NameN FROM table_Name WHERE column_name IN (list_of_values);
- Suppose, we want to show all the information of those employees from the **Employee_details** table whose **Employee Id** is 202, 204, and 205.
- SELECT * FROM Employee_details WHERE Emp_Id IN (202, 204, 205);

- Suppose, we want to show all the information of those employees from the **Employee_details** table whose **Employee Id** is not equal to 202 and 205.
- `SELECT * FROM Employee_details WHERE Emp_Id NOT IN (202,205);`
- **SQL NOT Operator:**
 - The **NOT operator** in SQL shows the record from the table if the condition evaluates to false. It is always used with the WHERE clause.
 - **Syntax of NOT operator:**
 - `SELECT column1, column2, columnN FROM table_Name WHERE NOT condition;`
 - Suppose, we want to show all the information of those employees from the **Employee_details** table whose City is not Delhi.
 - `SELECT * FROM Employee_details WHERE NOT Emp_City = 'Delhi' ;`

- Suppose, we want to show all the information of those employees from the **Employee_details** table whose City is not Delhi and Chandigarh.
- **SQL ANY Operator:**
- The **ANY operator** in SQL shows the records when any of the values returned by the sub-query meet the condition.
- The ANY logical operator must match at least one record in the inner query and must be preceded by any SQL comparison operator.
- **Syntax of ANY operator:**
- `SELECT column1, column2, columnN FROM table_Name WHERE column_name comparison_operator ANY (SELECT column_name FROM table_name WHERE condition(s))`

- **SQL LIKE Operator:**

- The **LIKE operator** in SQL shows those records from the table which match with the given pattern specified in the sub-query.
- The percentage (%) sign is a wildcard which is used in conjunction with this logical operator.
- This operator is used in the WHERE clause with the following three statements:
 - SELECT statement
 - UPDATE statement
 - DELETE statement
- **Syntax of LIKE operator:**
 - SELECT column_Name1, column_Name2, column_NameN FROM table_Name WHERE column_name LIKE pattern;

Let's understand the below example which explains how to execute LIKE logical operator in SQL query:

This example consists of an **Employee_details** table, which has four columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Chandigarh
203	Saket	30000	Delhi
204	Abhay	25000	Delhi
205	Sumit	40000	Kolkata

- If we want to show all the information of those employees from the Employee_details whose name starts with "s".
 - For this, we have to write the following query in SQL:
 - **SELECT * FROM Employee_details WHERE Emp_Name LIKE 's%' ;**
 - In this example, we used the SQL LIKE operator with Emp_Name column because we want to access the record of those employees whose name starts with s.
-
- If we want to show all the information of those employees from the **Employee_details** whose name ends with "y". For this, we have to write the following query in SQL:
 - **SELECT * FROM Employee_details WHERE Emp_Name LIKE '%y' ;**

- If we want to show all the information of those employees from the Employee_details whose name starts with "S" and ends with "y". For this, we have to write the following query in SQL:
- **SELECT * FROM Employee_details WHERE Emp_Name LIKE 'S%y' ;**

SQL Set Operators

- The **Set Operators** in SQL combine a similar type of data from two or more SQL database tables. It mixes the result, which is extracted from two or more SQL queries, into a single result.
- Set operators combine more than one select statement in a single query and return a specific result set.
- **Following are the various set operators which are performed on the similar data stored in the two SQL database tables:**
 - SQL Union Operator
 - SQL Union ALL Operator
 - SQL Intersect Operator
 - SQL Minus Operator

- **SQL Union Operator**

- The SQL Union Operator combines the result of two or more SELECT statements and provides the single output.
- The data type and the number of columns must be the same for each SELECT statement used with the UNION operator. This operator does not show the duplicate records in the output table.

- **Syntax of UNION Set operator:**

- SELECT column1, column2, columnN FROM table_Name1 [WHERE conditions]
- UNION
- SELECT column1, column2, columnN FROM table_Name2 [WHERE conditions];

- In this example, **we used two tables**. Both tables have four columns **Emp_Id**, **Emp_Name**, **Emp_Salary**, and **Emp_City**.

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi
203	Saket	30000	Aligarh

Table: Employee_details1

Emp Id	Emp Name	Emp Salary	Emp City
203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala
201	Sanjay	25000	Delhi

- Suppose, we want to see the employee name and employee id of each employee from both tables in a single output. For this, we have to write the following query in SQL:
- `SELECT Emp_ID, Emp_Name FROM Employee_details1
UNION
SELECT Emp_ID, Emp_Name FROM Employee_details2 ;`
- **SQL Union ALL Operator**
- **Syntax of UNION ALL Set operator:**
`SELECT column1, column2, columnN FROM table_Name1 [WHERE conditions]
UNION ALL
SELECT column1, column2, columnN FROM table_Name2 [WHERE conditions];`

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi
203	Saket	30000	Aligarh

Table: Employee_details1

Emp Id	Emp Name	Emp Salary	Emp City
203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala
201	Sanjay	25000	Delhi

- If we want to see the employee name of each employee of both tables in a single output. For this, we have to write the following query in SQL:
- `SELECT Emp_Name FROM Employee_details1`
- `UNION ALL`
- `SELECT Emp_Name FROM Employee_details2 ;`

- **SQL Intersect Operator**
- The SQL Intersect Operator shows the common record from two or more SELECT statements.
- The data type and the number of columns must be the same for each SELECT statement used with the INTERSECT operator.
- **Syntax of INTERSECT Set operator:**
- SELECT column1, column2, columnN FROM table_Name1 [WHERE conditions]
- INTERSECT
- SELECT column1, column2, columnN FROM table_Name2 [WHERE conditions];

- **SQL Minus Operator**

- The SQL Minus Operator combines the result of two or more SELECT statements and shows only the results from the first data set.
- **Syntax of MINUS operator:**
- SELECT column1, column2, columnN FROM First_tablename [WHERE conditions]
- MINUS
- SELECT column1, column2, columnN FROM Second_tablename [WHERE conditions]
;

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi
203	Saket	30000	Aligarh

Table: Employee_details1

Emp Id	Emp Name	Emp Salary	Emp City
203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala
201	Sanjay	25000	Delhi

Tables: Creating, Modifying, Deleting, Updating

SQL DML Queries: SELECT Query and clauses

- **SQL CLAUSES:**
- SQL clause helps us to retrieve a set or bundles of records from the table.
- SQL clause helps us to specify a condition on the columns or the records of a table.
- **Different clauses available in the Structured Query Language are as follows:**
- WHERE CLAUSE
- GROUP BY CLAUSE
- HAVING CLAUSE
- ORDER BY CLAUSE

- **WHERE CLAUSE:**

- A WHERE clause in SQL is used with the SELECT query, which is one of the data manipulation language commands.
- WHERE clauses can be used to limit the number of rows to be displayed in the result set, it generally helps in filtering the records.
- It returns only those queries which fulfill the specific conditions of the WHERE clause.
- WHERE clause is used in SELECT, UPDATE, DELETE statement, etc.
- WHERE clause with SELECT Query
- Asterisk symbol is used with a WHERE clause in a SELECT query to retrieve all the column values for every record from a table.

- **Syntax of where clause with a select query to retrieve all the column values for every record from a table:**
- **SELECT * FROM** TABLENAME **WHERE** CONDITION;
- **If according to the requirement, we only want to retrieve selective columns, then we will use below syntax:**
- **SELECT** COLUMNNAME1, COLUMNNAME2 **FROM** TABLENAME **WHERE** CONDITION ;
- Example 1:
- Write a query to retrieve all those records of an employee where employee salary is greater than 50000.
- **SELECT * FROM** employees **WHERE** Salary > 50000;

- Example 2:
- Write a query to update the employee's record and set the updated name as 'Harshada Sharma' where the employee's city name is Jaipur.
- **UPDATE employees SET Name = "Harshada Sharma" WHERE City = "Jaipur";**
- Example 3:
- Write a query to delete an employee's record where the employee's joining date is "2013-12-12".
- **DELETE FROM employees WHERE Date_of_Joining = "2013-12-12";**

2. GROUP BY CLAUSE

- The Group By clause is used to arrange similar kinds of records into the groups in the Structured Query Language.
- The Group by clause in the Structured Query Language is used with Select Statement.
- Group by clause is placed after the where clause in the SQL statement.
- The Group By clause is specially used with the aggregate function, i.e., max (), min (), avg (), sum (), count () to group the result based on one or more than one column.
- **The syntax of Group By clause:**
 - **SELECT * FROM TABLENAME GROUP BY COLUMNNAME;**
- **The syntax of Group By clause with Aggregate Functions:**
 - **SELECT COLUMNNAME1, Aggregate_FUNCTION (COLUMNNAME) FROM TABLENAME GROUP BY COLUMNNAME;**

- Example 1:
- Write a query to display all the records of the employees table but group the results based on the age column.
- **SELECT * FROM employees GROUP BY Age;**
- Example 2:
- Write a query to display all the records of the employees table grouped by the designation and salary.
- **SELECT * FROM employees GROUP BY Salary, Designation;**

- Examples of Group By clause using aggregate functions
- **Example 1:**
- Write a query to list the number of employees working on a particular designation and group the results by designation of the employee.
- **SELECT COUNT (E_ID) AS Number_of_Employees, Designation FROM employees GROUP BY Designation;**
- Write a query to display the sum of an employee's salary as per the city grouped by an employee's age.
- **SELECT SUM (Salary) AS Salary, City FROM employees GROUP BY City;**

- **3. HAVING CLAUSE:**

- When we need to place any conditions on the table's column, we use the WHERE clause in SQL. But if we want to use any condition on a column in Group By clause at that time, we will use the HAVING clause with the Group By clause for column conditions.
- Syntax:
- **TABLENAME GROUP BY COLUMNNAME HAVING CONDITION;**
- Example 1:
- Write a query to display the name of employees, salary, and city where the employee's maximum salary is greater than 40000 and group the results by designation.
- **SELECT Name, City, MAX (Salary) AS Salary FROM employees GROUP BY Designation HAVING MAX (Salary) > 40000;**

- Example 2:
- Write a query to display the name of employees and designation where the sum of an employee's salary is greater than 45000 and group the results by city.
- **SELECT Name, Designation, SUM (Salary) AS Salary FROM employees GROUP BY City HAVING SUM (Salary) > 45000;**

- **ORDER BY CLAUSE:**

- Whenever we want to sort anything in SQL, we use the ORDER BY clause.
- The ORDER BY clause in SQL will help us to sort the data based on the specific column of a table.
- This means that all the data stored in the specific column on which we are executing the ORDER BY clause will be sorted.
- The corresponding column values will be displayed in the sequence in which we have obtained the values in the earlier step.
- Syntax of ORDER BY clause without asc and desc keyword:
- **SELECT** COLUMN_NAME1, COLUMN_NAME2 **FROM** TABLE_NAME **ORDER BY** COLUMNNAME;

- **Syntax of ORDER BY clause to sort in ascending order:**
- **SELECT COLUMN_NAME1, COLUMN_NAME2 FROM TABLE_NAME ORDER BY COLUMN_NAME ASC;**
- **Syntax of ORDER BY clause to sort in descending order:**
- **SELECT COLUMN_NAME1, COLUMN_NAME2 FROM TABLE_NAME ORDER BY COLUMN_NAME DESC;**
- **Example 1:**
- **Write a query to sort the records in the ascending order of the employee designation from the employees table.**
- **SELECT * FROM employees ORDER BY Designation;**

- **Example 2:**
- **Write a query to display employee name and salary in the ascending order of the employee's salary from the employees table.**
- **SELECT Name, Salary FROM employees ORDER BY Salary ASC;**
- **Example 3:**
- **Write a query to sort the data in descending order of the employee name stored in the employees table.**
- **SELECT * FROM employees ORDER BY Name DESC;**

E_ID	Name	Salary	City	Designation	Date_of_Joining	Age
1	Sakshi Kumari	50000	Mumbai	Project Manager	2021-06-20	24
2	Tejaswini Naik	75000	Delhi	System Engineer	2019-12-24	23
3	Anuja Sharma	40000	Jaipur	Manager	2021-08-15	26
4	Anushka Tripathi	90000	Mumbai	Software Tester	2021-06-13	24
5	Rucha Jagtap	45000	Bangalore	Project Manager	2020-08-09	23
6	Rutuja Deshmukh	60000	Bangalore	Manager	2019-07-17	26
7	Swara Baviskar	55000	Jaipur	System Engineer	2021-10-10	24
8	Sana	45000	Pune	Software	2020-09-	26

E_ID	Name	Salary	City	Designation	Date_of_Joining	Age
1	Sakshi Kumari	50000	Mumbai	Project Manager	2021-06-20	24
2	Tejaswini Naik	75000	Delhi	System Engineer	2019-12-24	23
3	Anuja Sharma	40000	Jaipur	Manager	2021-08-15	26
4	Anushka Tripathi	90000	Mumbai	Software Tester	2021-06-13	24
5	Rucha Jagtap	45000	Bangalore	Project Manager	2020-08-09	23
6	Rutuja Deshmukh	60000	Bangalore	Manager	2019-07-17	26
7	Swara Ravickar	55000	Jaipur	System Engineer	2021-10-10	24

Sequence in SQL

- SQL sequences are an essential feature of relational database management systems (RDBMS) used to generate unique numeric values in a sequential order.
- These values are widely used for generating primary keys, unique keys, and other numeric identifiers in databases.
- SQL sequences offer flexibility, performance, and ease of use, making them indispensable in managing and organizing data in large-scale applications.
- SQL sequences are user-defined database objects designed to generate a series of numeric values.
- Unlike identity columns, which are tightly bound to specific tables, sequences are independent objects and can be used across multiple tables.
- They allow applications to retrieve the next number in a sequence whenever needed, offering a simple and efficient way to generate unique numbers on demand.

- The values in a sequence can be configured to be generated in ascending or descending order, and the sequence can be set to restart (cycle) once the maximum value is exceeded.
- This makes SQL sequences particularly useful in scenarios where there is a need for continuous, unique values, such as generating primary keys or serial numbers.
- **Key Features of SQL Sequences:**
 - **Automatic Primary Key Generation:** Sequences automatically generate unique values that can be used for primary or unique keys in database tables.
 - **Ascending or Descending Order:** Sequences can be configured to generate numbers in either ascending or descending order.
 - **Multiple Table Usage:** A single sequence can be used to generate values for multiple tables, making it flexible and reusable.
 - **Independent of Tables:** Unlike identity columns, sequences are independent and can be used across different tables.
 - **Efficient:** Sequences reduce the complexity and overhead of manually generating unique values, which saves time and reduces application code.

- **How SQL Sequences Work:**
- When creating a sequence, we specify the **starting point**, the **increment** (how much the sequence increases with each step), and optionally the minimum and maximum values.
- Sequences can be set to cycle, which means they restart from the beginning when they reach the **maximum value**.

- **Syntax:**

CREATE SEQUENCE sequence_name

START WITH initial_value

INCREMENT BY increment_value

MINVALUE minimum value

MAXVALUE maximum value

CYCLE|NOCYCLE ;

- **Example 1:** Creating a Sequence in Ascending Order
- `CREATE SEQUENCE sequence_1`
start with 1
increment by 1
minvalue 0
maxvalue 100
cycle;
- **Explanation:**
 - The above query will create a sequence named sequence_1. The sequence will start from 1 and will be incremented by 1 having maximum value of 100.
 - The sequence will repeat itself from the start value after exceeding 100.

- **Example 2: Creating a Sequence in Descending Order**

- `CREATE SEQUENCE sequence_2`
start with 100
increment by -1
min value 1
max value 100
cycle;

- **Explanation:**

- The above query will create a sequence named sequence_2. The sequence will start from 100 and should be less than or equal to a maximum value and will be incremented by -1 having a minimum value of 1.

- **Using SQL Sequences in Database Operations**

- Once a sequence is created, it can be used across multiple tables to generate unique values, such as primary key identifiers or serial numbers.
- This allows for consistent, efficient value generation, reducing the need for manual input and ensuring uniqueness across different rows and tables.

- **Example: Using a Sequence to Insert Values**

- Let's create a students table and use the sequence to automatically generate unique student IDs.
- In this example, the NEXTVAL function is used to retrieve the next value in the sequence (sequence_1) and insert it into the ID column for each student.

- CREATE TABLE students
(
ID number(10),
NAME char(20)
);

```
INSERT into students VALUES  
(sequence_1.nextval,'Shubham');  
INSERT into students VALUES  
(sequence_1.nextval,'Aman');
```

- **Practical Use Cases for SQL Sequences:**

- **1.Primary Key Generation**

- Sequences are commonly used to generate **unique primary key** values for database tables. This is especially useful in applications where a large volume of records needs to be inserted into a table, and each record requires a unique identifier.

- **2. Serial Numbers and Order IDs**

- SQL sequences can be used to generate serial numbers for products or order IDs in e-commerce systems, ensuring that each order has a unique identifier.

- **3. Auditing and Tracking**

- Sequences are also useful for tracking events or transactions that require unique identifiers, such as logging system activities or generating unique reference numbers for transactions.

Views:

- Views in SQL are a type of virtual table that simplifies how users interact with data across one or more tables.
- Unlike traditional tables, a view in SQL does not store data on disk; instead, it dynamically retrieves data based on a pre-defined query each time it's accessed.
- SQL views are particularly useful for managing complex queries, enhancing security, and presenting data in a simplified format.
- A view in SQL is a saved SQL query that acts as a virtual table. It can fetch data from one or more tables and present it in a customized format, allowing developers to:
- **Simplify Complex Queries:** Encapsulate complex joins and conditions into a single object.
- **Enhance Security:** Restrict access to specific columns or rows.
- **Present Data Flexibly:** Provide tailored data views for different users

- **CREATE VIEWS in SQL**

- We can create a view using **CREATE VIEW** statement. A View can be created from a single table or multiple tables.

- **Syntax:**

- *CREATE VIEW view_name AS*
- *SELECT column1, column2.....*
- *FROM table_name*
- *WHERE condition;*

- **Example 1: Creating View From a Single Table**

Create view Detailsview As

Select Name , Address

From Studentdetails

Where S_ID<5;

- **Example 2: Create View From Table**

CREATE VIEW StudentNames AS

SELECT S_ID, NAME

FROM StudentDetails

ORDER BY NAME;

- **Example 3: Creating View From Multiple Tables**

```
CREATE VIEW MarksView AS  
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS  
FROM StudentDetails, StudentMarks  
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

- **Listing all Views in a Database**

```
USE "database_name";  
SHOW FULL TABLES  
WHERE table_type LIKE "%VIEW";
```


- **Delete View in SQL**

- **DROP VIEW** view_name;

- **Update View in SQL**

- **UPDATE** view_name **SET** column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];

- **Rules to Update Views in SQL:**

- Certain conditions need to be satisfied to update a view. If any of these conditions are **not** met, the view can not be updated.
- The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
- The SELECT statement should not have the DISTINCT keyword.
- The View should have all NOT NULL values.
- The view should not be created using nested queries or complex queries.
- The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

```
CREATE OR REPLACE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS,
StudentMarks.AGE
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

- **Uses of a View**

- A good database should contain views for the given reasons:
- **Restricting data access** – Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.
- **Hiding data complexity** – A view can hide the complexity that exists in multiple joined tables.
- **Simplify commands for the user** – Views allow the user to select information from multiple tables without requiring the users to actually know how to perform a join.
- **Store complex queries** – Views can be used to store complex queries.
- **Rename Columns** – Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of columns specified in a select statement. Thus, renaming helps to hide the names of the columns of the base tables.
- **Multiple view facility** – Different views can be created on the same table for different users.

Predicate

- A predicate is simply an expression that evaluates to TRUE, FALSE, or UNKNOWN.
- Predicates are typically employed in the search condition of WHERE and HAVING clauses, the join conditions of FROM clauses, as well as any other part of a query where a boolean value is required.

JOINS:

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

SQL joins are the foundation of database management systems, enabling the combination of data from multiple tables based on relationships between columns.

Joins allow efficient data retrieval, which is essential for generating meaningful observations and solving complex business queries.

- [SQL JOIN](#) clause is used to query and access data from multiple tables by establishing logical relationships between them.
- It can access data from multiple tables simultaneously using common key values shared across different tables.
- We can use SQL JOIN with multiple tables. It can also be paired with other clauses, the most popular use will be using JOIN with [WHERE clause](#) to filter data retrieval.

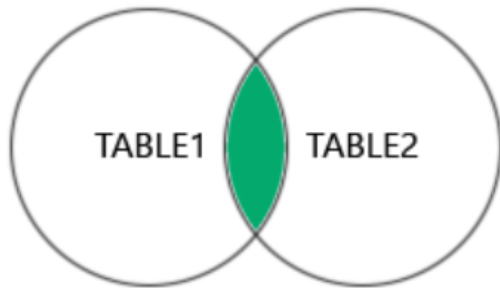
- **Example of SQL JOINS**

- [SQL JOIN](#) clause is used to query and access data from multiple tables by establishing logical relationships between them.
- It can access data from multiple tables simultaneously using common key values shared across different tables.
- We can use SQL JOIN with multiple tables. It can also be paired with other clauses, the most popular use will be using JOIN with [WHERE clause](#) to filter data retrieval.

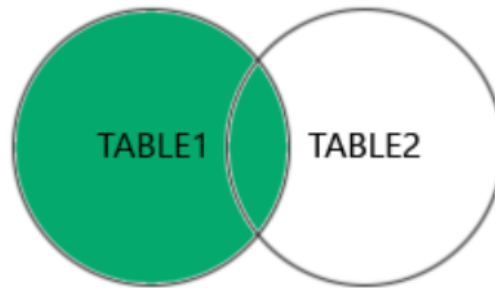
- **Different Types of SQL JOINS**

- **(INNER) JOIN:** Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN:** Returns all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN:** Returns all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN:** Returns all records when there is a match in either left or right table

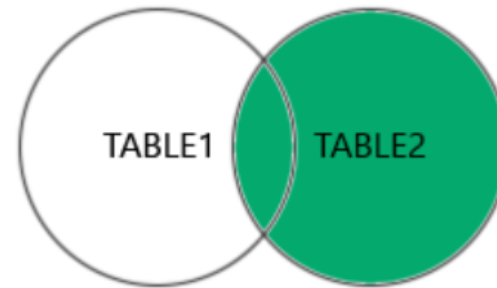
INNER JOIN



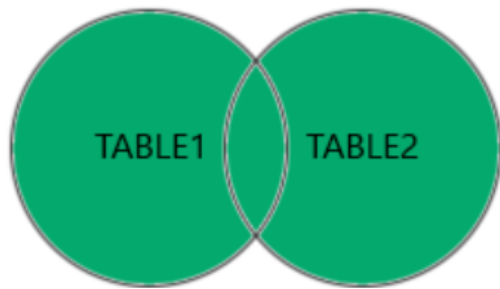
LEFT JOIN



RIGHT JOIN



FULL OUTER JOIN



- **Types of JOIN in SQL**

- There are many types of Joins in [SQL](#). Depending on the use case, we can use different type of SQL JOIN clause. Below, we explain the most commonly used join types with syntax and examples:
- **1. SQL INNER JOIN**
- The [INNER JOIN](#) keyword selects all rows from both the tables as long as the condition is satisfied.
- This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.

- SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
INNER JOIN table2
ON table1.matching_column = table2.matching_column;

- **INNER JOIN Example**

- SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM Student
INNER JOIN StudentCourse
ON Student.ROLL_NO = StudentCourse.ROLL_NO;

professor table

ID	Name	Salary
1	Rohan	57000
2	Aryan	45000
3	Arpit	60000
4	Harsh	50000
5	Tara	55000

teacher Table

course_id	prof_id	course_name
1	1	English
1	3	Physics
2	4	Chemistry
2	5	Mathematics

Output

course_id	prof_id	Name	Salary
1	1	Rohan	57000
1	3	Arpit	60000
2	4	Harsh	50000
2	5	Tara	55000

- **2. SQL LEFT JOIN**

- **LEFT JOIN** returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join.
- For the rows for which there is **no matching row** on the right side, the result-set will contain **null**. LEFT JOIN is also known as **LEFT OUTER JOIN**.
- **Syntax:**
- ```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
LEFT JOIN table2
ON table1.matching_column = table2.matching_column;
```

- **LEFT JOIN Example**

- ```
SELECT Student.NAME, StudentCourse.COURSE_ID
FROM Student
LEFT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Student Table

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	XXXXXXXXXX	18
2	PRATIK	BIHAR	XXXXXXXXXX	19
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19

StudentCourse Table

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

Output

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	<i>NULL</i>
ROHIT	<i>NULL</i>
NIRAJ	<i>NULL</i>

- **3. SQL RIGHT JOIN**

- [RIGHT JOIN](#) returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join.
- It is very similar to LEFT JOIN for the rows for which there is no matching row on the left side, the result-set will contain null.
- RIGHT JOIN is also known as RIGHT OUTER JOIN.
- **Syntax:**
- ```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;
```



## Output

| NAME        | COURSE_ID |
|-------------|-----------|
| HARSH       | 1         |
| PRATIK      | 2         |
| RIYANKA     | 2         |
| DEEP        | 3         |
| SAPTARHI    | 1         |
| <i>NULL</i> | 4         |
| <i>NULL</i> | 5         |
| <i>NULL</i> | 4         |

- **4. SQL FULL JOIN**

- [FULL JOIN](#) creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN.
- The result-set will contain all the rows from both tables. For the rows for which there is no matching, the result-set will contain NULL values.
- ```
SELECT table1.column1,table1.column2,table2.column1,....  
FROM table1  
FULL JOIN table2  
ON table1.matching_column = table2.matching_column;
```
-

PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	NULL
ROHIT	NULL
NIRAJ	NULL
NULL	4

- **5. SQL Natural Join**

- [Natural join](#) can join tables based on the **common columns** in the tables being joined.
- A natural join returns all rows by matching values in common columns having same name and **data type** of columns and that column should be present in both tables.
- Both table must have at least one common column with same column name and same data type.
- The two table are joined using **Cross join**.
- DBMS will look for a common column with same name and data type. Tuples having exactly same values in common columns are kept in result.

Employee		
Emp_id	Emp_name	Dept_id
1	Ram	10
2	Jon	30
3	Bob	50

Department	
Dept_id	Dept_name
10	IT
30	HR
40	TIS

Emp_id	Emp_name	Dept_id	Dept_id	Dept_name
1	Ram	10	10	IT
2	Jon	30	30	HR
Employee data			Department data	

SQL Functions

- **SQL Functions** are built-in programs that are used to perform different operations on the database.
- There are two types of functions in SQL:
- **Aggregate Functions**
- **Scalar Functions**

SQL Aggregate Functions

- SQL Aggregate Functions operate on a data group and return a singular output. They are mostly used with the GROUP BY clause to summarize data.

AVG()	Calculates the average value	SELECT AVG(column_name) FROM table_name;
COUNT()	Counts the number of rows	SELECT COUNT(column_name) FROM table_name
FIRST()	Returns the first value in an ordered set of values	SELECT FIRST(column_name) FROM table_name;
LAST()	Returns the last value in an ordered set of values	SELECT LAST(column_name) FROM table_name;
MAX()	Retrieves the maximum value from a column	SELECT MAX(column_name) FROM table_name;
MIN()	Retrieves the minimum value from a column	SELECT MIN(column_name) FROM table_name;
SUM()	Calculates the total sum of	SELECT SUM(column_name)

SQL Scalar functions

- **SQL Scalar Functions** are built-in functions that operate on a single value and return a single value.

Scalar function	Description	Syntax
UCASE()	Converts a string to uppercase	SELECT UCASE(column_name) FROM table_name;
LCASE()	Converts a string to lowercase	SELECT LCASE(column_name) FROM table_name;
MID()	Extracts a substring from a string	SELECT MID(column_name, start, length) FROM table_name;
LEN()	Returns the length of a string	SELECT LEN(column_name) FROM table_name;
ROUND()	Rounds a number to a specified number of decimals	SELECT ROUND(column_name, decimals) FROM table_name;

MID()	Extracts a substring from a string	SELECT MID(column_name, start, length) FROM table_name;
LEN()	Returns the length of a string	SELECT LEN(column_name) FROM table_name;
ROUND()	Rounds a number to a specified number of decimals	SELECT ROUND(column_name, decimals) FROM table_name;
NOW()	Returns the current date and time	SELECT NOW();
FORMAT()	Formats a value with the specified format	SELECT FORMAT(column_name, format) FROM table_name;

Nested Queries in SQL

- **Nested queries in SQL** are a powerful tool for retrieving data from **databases** in a structured and efficient manner.
- They allow us to execute a query within another query, making it easier to handle complex data operations.
- A nested query (also called a subquery) is a query embedded within another SQL query.
- The result of the inner query is used by the outer query to perform further operations.
- Nested queries are commonly used for filtering data, performing calculations, or joining datasets indirectly.

- **Key Characteristics:**

- The inner query runs before the outer query.
- The result of the inner query can be used by the outer query for comparison or as input data.
- Nested queries are particularly useful for breaking down complex problems into smaller, manageable parts, making it easier to retrieve specific results from large datasets.

1. STUDENT Table

The **STUDENT** table stores information about students, including their unique ID, name, address, phone number, and age.

S_ID	S_NAME	S_ADDRESS	S_PHONE	S_AGE
S1	RAM	DELHI	9455123451	18
S2	RAMESH	GURGAON	9652431543	18
S3	SUJIT	ROHTAK	9156253131	20
S4	SURESH	DELHI	9156768971	18

STUDENT Table

2. COURSE Table

The **STUDENT_COURSE** table maps students to the courses they have enrolled in. It uses the student and course IDs as foreign keys.

C_ID	C_NAME
C1	DSA
C2	Programming
C3	DBMS

COURSE Table

3. STUDENT_COURSE Table

This table maps students to the courses they have enrolled in, with columns for student ID (**S_ID**) and course ID (**C_ID**):

S_ID	C_ID
S1	C1
S1	C3
S2	C1
S3	C2
S4	C2
S4	C3

Student_Course Table

- **Types of Nested Queries in SQL**
- **1. Independent Nested Queries:**
- In **independent nested queries**, the execution of the inner query is not dependent on the outer query.
- The result of the inner query is used directly by the outer query. Operators like **IN**, **NOT IN**, **ANY**, and **ALL** are commonly used with this type of nested query.
- Example 1: Using IN
- Find the S_IDs of students who are enrolled in the courses 'DSA' or 'DBMS'.
- Step 1: Find the C_IDs of the courses:
- This query retrieves the IDs of the courses named 'DSA' or 'DBMS' from the

- `SELECT C_ID FROM COURSE WHERE Output IN ('DSA', 'DBMS');`

C_ID
C1
C3

- **Step 2: Use the result of Step 1 to find the corresponding S_IDs:**
- The inner query finds the course IDs, and the outer query retrieves the student IDs associated with those courses from the **STUDENT_COURSE** table

- `SELECT S_ID FROM STUDENT_COURSE WHERE C_ID IN (SELECT C_ID FROM COURSE WHERE C_NAME IN ('DSA', 'DBMS'))` **Output**

S_ID
S1
S2
S4

- **2. Correlated Nested Queries**

- In correlated **nested queries**, the inner query depends on the outer query for its execution.
- For each row processed by the outer query, the inner query is executed. The **EXISTS** keyword is often used with correlated queries.
- Example 2: Using EXISTS
- Find the names of students who are enrolled in the course with C_ID = 'C1':

- ```
SELECT S_NAME FROM STUDENT S
WHERE EXISTS (
 SELECT 1 FROM STUDENT_COURSE SC
 WHERE S.S_ID = SC.S_ID AND SC.C_ID = 'C1'
);
```

Output

| S_NAME |
|--------|
| RAM    |
| RAMESH |

- **SQL Operators for Nested Queries**
- **1. IN Operator**
- **2. NOT IN Operator**
- **3.EXISTS**
- **4.ANY and ALL**

# Advantages of Nested Queries

- **Simplifies complex queries:** Nested queries allow us to divide complicated SQL tasks into smaller, more manageable parts. This modular approach makes queries easier to write, debug, and maintain.
- **Enhances flexibility:** By enabling the use of results from one query within another, nested queries allow dynamic filtering and indirect joins, which can simplify query logic.
- **Supports advanced analysis:** Nested queries empower developers to perform operations like conditional aggregation, subsetting, and customized calculations, making them ideal for sophisticated data analysis tasks.
- **Improves readability:** When properly written, nested queries can make complex operations more intuitive by encapsulating logic within inner queries.

# PL/SQL:

- PL/SQL (Procedural Language/Structured Query Language) is a **block-structured language** developed by **Oracle** that allows developers to combine the power of **SQL** with procedural programming constructs.
- The PL/SQL language enables efficient data manipulation and control-flow logic, all within the **Oracle Database**.

- **Basics of PL/SQL**

- PL/SQL stands for Procedural Language extensions to the Structured Query Language (SQL).
- PL/SQL is a combination of SQL along with the procedural features of programming languages.
- Oracle uses a PL/SQL engine to process the PL/SQL statements.
- PL/SQL includes procedural language elements like conditions and loops. It allows declaration of constants and variables, procedures and functions, types and variable of those types and triggers.



## • **Features of PL/SQL**

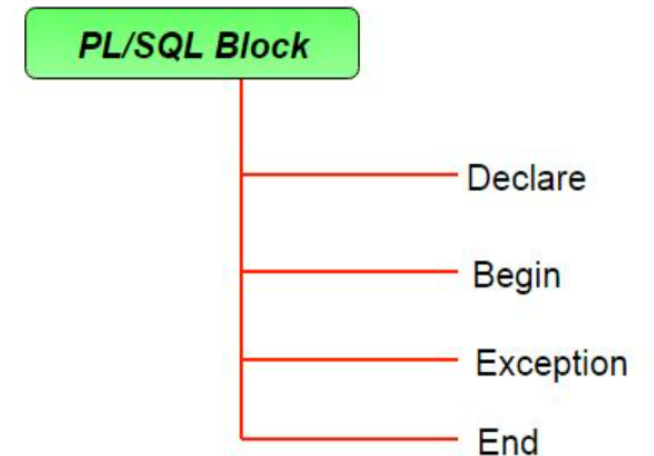
1. PL/SQL is basically a procedural language, which provides the functionality of decision-making, iteration, and many more features of procedural programming languages.
2. PL/SQL can execute a number of queries in one block using single command.
3. One can create a PL/SQL unit such as procedures, functions, packages, triggers, and types, which are stored in the database for reuse by applications.
4. PL/SQL provides a feature to handle the exception which occurs in PL/SQL block known as exception handling block.
5. Applications written in PL/SQL are portable to computer hardware or operating system where Oracle is operational.
6. PL/SQL Offers extensive error checking.

# Difference Between SQL & PL/SQL

| SQL                                                                                            | PL/SQL                                                                                             |
|------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| SQL is a single query that is used to perform DML and DDL operations.                          | PL/SQL is a block of codes that used to write the entire program blocks/ procedure/ function, etc. |
| It is declarative, that defines what needs to be done, rather than how things need to be done. | PL/SQL is procedural that defines how the things needs to be done.                                 |
| Execute as a single statement.                                                                 | Execute as a whole block.                                                                          |
| Mainly used to manipulate data.                                                                | Mainly used to create an application.                                                              |
| Cannot contain PL/SQL code in it.                                                              | It is an extension of SQL, so it can contain SQL inside it.                                        |

# Structure of PL/SQL Block

- PL/SQL extends [SQL](#) by adding constructs found in **procedural languages**, resulting in a structural language that is more powerful than SQL.
- The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other.



- Typically, each block performs a logical action in the program. A block has the following structure:

```
DECLARE
 declaration statements;

BEGIN
 executable statements

EXCEPTIONS
 exception handling statements

END;
```

- Declare section starts with **DECLARE** keyword in which variables, constants, records as cursors can be declared which stores data temporarily. It basically consists definition of PL/SQL identifiers. This part of the code is optional.
- Execution section starts with **BEGIN** and ends with **END** keyword. This is a mandatory section and here the program logic is written to perform any task like loops and conditional statements. It supports all DML commands, DDL commands and SQL\*PLUS built-in functions as well.
- Exception section starts with **EXCEPTION** keyword. This section is optional which contains statements that are executed when a run-time error occurs. Any exceptions can be handled in this section

- **PL/SQL Identifiers**

- There are several PL/SQL identifiers such as variables, constants, procedures, cursors, triggers etc.
- **Variables**: Like several other programming languages, variables in PL/SQL must be declared prior to its use. They should have a valid name and data type as well. Syntax for declaration of variables:
- **variable\_name datatype [NOT NULL := value ];**

## Example to show how to declare variables in PL/SQL :

```
SQL> SET SERVEROUTPUT ON;
```

```
SQL> DECLARE
```

```
 var1 INTEGER;
```

```
 var2 REAL;
```

```
 var3 varchar2(20) ;
```

```
BEGIN
```

```
 null;
```

```
END;
```

```
/
```

- **SET SERVEROUTPUT ON**: It is used to display the buffer used by the dbms\_output.
- **var1 INTEGER** : It is the declaration of variable, named ***var1*** which is of integer type. There are many other data types that can be used like float, int, real, smallint, long etc. It also supports variables used in SQL as well like NUMBER(prec, scale), varchar, varchar2 etc.
- **PL/SQL procedure successfully completed.**: It is displayed when the code is compiled and executed successfully.
- **Slash (/) after END;**: The slash (/) tells the SQL\*Plus to execute the block.
- **Assignment operator (:=)** : It is used to assign a value to a variable.



## Displaying Output:

- The outputs are displayed by using DBMS\_OUTPUT which is a built-in package that enables the user to display output, debugging information, and send messages from PL/SQL blocks, subprograms, packages, and triggers.
- SQL> SET SERVEROUTPUT ON;
- SQL> DECLARE
- var varchar2(40) := 'I love BVDU' ;
- BEGIN
- dbms\_output.put\_line(var);
- END;
- /

*dbms\_output.put\_line* : This command is used to direct the PL/SQL output to a screen.

**2. Using Comments**: Like in many other programming languages, in PL/SQL also, comments can be put within the code which has no effect in the code. There are two syntaxes to create comments in PL/SQL :

**2. Single Line Comment**: To create a single line comment , the symbol – – is used.

**3. Multi Line Comment**: To create comments that span over several lines, the symbol /\* and \*/ is used.

- **Taking input from user**: Just like in other programming languages, in PL/SQL also, we can take input from the user and store it in a variable.

```
SQL> SET SERVEROUTPUT ON;

SQL> DECLARE

 -- taking input for variable a
 a number := &a;

 -- taking input for variable b
 b varchar2(30) := &b;

BEGIN
 null;

END;
/
```

```
--PL/SQL code to print sum of two numbers taken from the user.
SQL> SET SERVEROUTPUT ON;

SQL> DECLARE

 -- taking input for variable a
 a integer := &a ;

 -- taking input for variable b
 b integer := &b ;
 c integer ;

BEGIN
 c := a + b ;
 dbms_output.put_line('Sum of '||a||' and '||b||' is = '||c);

END;
/
```

```
DECLARE
 -- declare variable a, b and c
 -- and these three variables datatype are integer
 a number;
 b number;
 c number;
BEGIN
 a:= 10;
 b:= 100;
 --find largest number
 --take it in c variable
 IF a > b THEN
 c:= a;
 ELSE
 c:= b;
 END IF;
 dbms_output.put_line(' Maximum number in 10 and 100: ' || c);
END;
/
-- Program End
```

---

# **Relational Database Design**

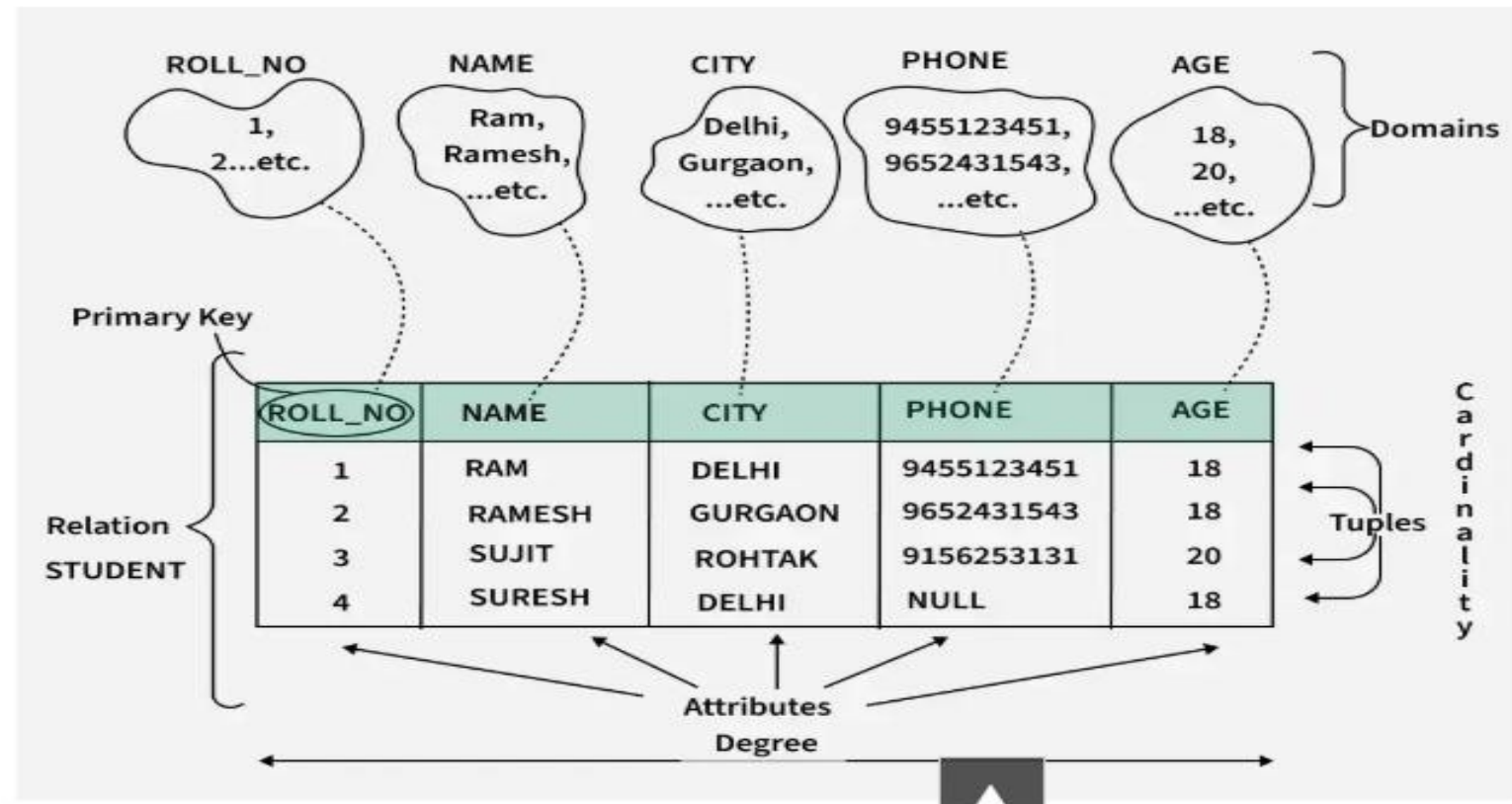
- The **Relational Model** represents data and their relationships through a collection of tables.
- Each table also known as a relation consists of rows and columns.
- Every column has a unique name and corresponds to a specific attribute, while each row contains a set of related data values representing a real-world entity or relationship.
- This model is part of the record-based models which structure data in fixed-format records each belonging to a particular type with a defined set of attributes.
- E.F. Codd introduced the Relational Model to organize data as relations or tables.
- After creating the conceptual design of a database using an ER diagram, this design must be transformed into a relational model which can then be implemented using relational database systems like Oracle SQL or MySQL.

# 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.



## Table STUDENT



- **Key Terms:**

- **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:

|   |     |       |            |    |
|---|-----|-------|------------|----|
| 1 | RAM | DELHI | 9455123451 | 18 |
|---|-----|-------|------------|----|

- **Relation Instance:** The set of tuples of a relation at a particular instance of time is called a relation instance. 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.
- **NULL Values:** The value which is not known or unavailable is called a NULL value. It is represented by NULL. 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](#)
  - [Candidate Key](#)
  - [Super Key](#)
  - [Foreign Key](#)
  - [Alternate Key](#)

- **Relational Model Notation:**

- Relation schema  $R$  of degree  $n$  is denoted by  $R(A_1, A_2, \dots, A_n)$ .
- Uppercase letters  $Q, R, S$  denote relation names.
- Lowercase letters  $q, r, s$  denote relation states.
- Letters  $t, u, v$  denote tuples.

- In general, the name of a relation schema such as STUDENT also indicates the current set of tuples in that relation.
- An attribute A can be qualified with the relation name R to which it belongs by using the dot notation R.A for example, STUDENT.Name or STUDENT.Age.
- An n-tuple t in a relation r(R) is represented as  $t = \langle v_1, v_2, \dots, v_n \rangle$  where  $v_i$  is the value corresponding to the attribute  $A_i$ . The value  $v_i$  for attribute  $A_i$  in tuple t can be accessed using  $t[A_i]$  or  $t.A_i$ .

# Characteristics of the Relational Model

- **Data Representation:** Data is organized in tables (relations), with rows (tuples) representing records and columns (attributes) representing data fields.
- **Atomic Values:** Each attribute in a table contains atomic values, meaning no multi-valued or nested data is allowed in a single cell.
- **Unique Keys:** Every table has a primary key to uniquely identify each record, ensuring no duplicate rows.
- **Attribute Domain:** Each attribute has a defined domain, specifying the valid data types and constraints for the



- **Tuples as Rows:** Rows in a table, called tuples, represent individual records or instances of real-world entities or relationships.
- **Relation Schema:** A table's structure is defined by its schema, which specifies the table name, attributes, and their domains.
- **Data Independence:** The model ensures logical and physical data independence, allowing changes in the database schema without affecting the application layer.

- **Integrity Constraints:** The model enforces rules like:
  - **Domain constraints:** Attribute values must match the specified domain.
  - **Entity integrity:** No primary key can have NULL values.
  - **Referential integrity:** Foreign keys must match primary keys in the referenced table or be NULL.

- **Relational Operations:** Supports operations like selection, projection, join, union, and intersection, enabling powerful data retrieval manipulation.
- **Data Consistency:** Ensures data consistency through constraints, reducing redundancy and anomalies.
- **Set-Based Representation:** Tables in the relational model are treated as sets, and operations follow mathematical set theory principles.

# 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**

- Domain Constraints ensure that the value of each attribute  $A$  in a tuple must be an atomic value derived from its specified domain,  $\text{dom}(A)$ .

Domains are defined by the data types associated with the attributes. Common data types include:

- **Numeric types:** Includes integers (short, regular, and long) for whole numbers and real numbers (float, double-precision) for decimal values, allowing precise calculations.
- **Character types:** Consists of fixed-length (CHAR) and variable-length (VARCHAR, TEXT) strings for storing text data of various sizes.

- **Boolean values:** Stores **true** or **false** values, often used for flags or conditional checks in databases.
- **Specialized types:** Includes types for **date** (DATE), **time** (TIME), **timestamp** (TIMESTAMP), and **money** (MONEY), used for precise handling of time-related and financial data.

- **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.

- **Referential Integrity Constraints**

- 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](#).



Table STUDENT

| ROLL_NO | NAME   | ADDRESS | PHONE      | AGE | BRANCH_CODE |
|---------|--------|---------|------------|-----|-------------|
| 1       | RAM    | DELHI   | 9455123451 | 18  | CS          |
| 2       | RAMESH | GURGAON | 9652431543 | 18  | CS          |
| 3       | SUJIT  | ROHTAK  | 9156253131 | 20  | ECE         |
| 4       | SURESH | DELHI   |            | 18  | IT          |

Table BRANCH

| BRANCH_CODE | BRANCH_NAME                                  |
|-------------|----------------------------------------------|
| CS          | COMPUTER SCIENCE                             |
| IT          | INFORMATION TECHNOLOGY                       |
| ECE         | ELECTRONICS AND<br>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).

- **Anomalies in the Relational Model**
- An anomaly is an irregularity or something which deviates from the expected or normal state.
- When designing databases, we identify three types of anomalies: **Insert**, **Update**, and **Delete**.

- **Insertion Anomaly in Referencing Relation**

- We can't insert a row in REFERENCING RELATION if referencing attribute's value is not present in the referenced attribute value. e.g.; Insertion of a student with BRANCH\_CODE 'ME' in STUDENT relation will result in an error because 'ME' is not present in BRANCH\_CODE of BRANCH.

- **Deletion/ Updation Anomaly in Referenced Relation:**
- We can't delete or update a row from REFERENCED RELATION if the value of REFERENCED ATTRIBUTE is used in the value of REFERENCING ATTRIBUTE. e.g. if we try to delete a tuple from BRANCH having BRANCH\_CODE 'CS', it will result in an error because 'CS' is referenced by BRANCH\_CODE of STUDENT, but if we try to delete the row from BRANCH with BRANCH\_CODE CV, it will be deleted as the value is not been used by referencing relation.

- It can be handled by the following method:
- **On Delete Cascade**
- It will delete the tuples from REFERENCING RELATION if the value used by REFERENCING ATTRIBUTE is deleted from REFERENCED RELATION. e.g.; if we delete a row from BRANCH with BRANCH\_CODE 'CS', the rows in STUDENT relation with BRANCH\_CODE CS (ROLL\_NO 1 and 2 in this case) will be deleted.

- **On Update Cascade**

- It will update the REFERENCING ATTRIBUTE in REFERENCING RELATION if the attribute value used by REFERENCING ATTRIBUTE is updated in REFERENCED RELATION. e.g., if we update a row from BRANCH with BRANCH\_CODE 'CS' to 'CSE', the rows in STUDENT relation with BRANCH\_CODE CS (ROLL\_NO 1 and 2 in this case) will be updated with BRANCH\_CODE 'CSE'.



# **Unit 4**

## **Database Transaction Management System**

# Introduction to Database Transaction

- Transactions are a set of operations used to perform a logical set of work.
- A transaction usually means that the data in the database has changed.
- One of the major uses of DBMS is to protect the user data from system failures.
- It is done by ensuring that all the data is restored to a consistent state when the computer is restarted after a crash.
- The transaction is any one execution of the user program in a DBMS.
- One of the important properties of the transaction is that it contains a finite number of steps.
- Executing the same program multiple times will generate multiple transactions.

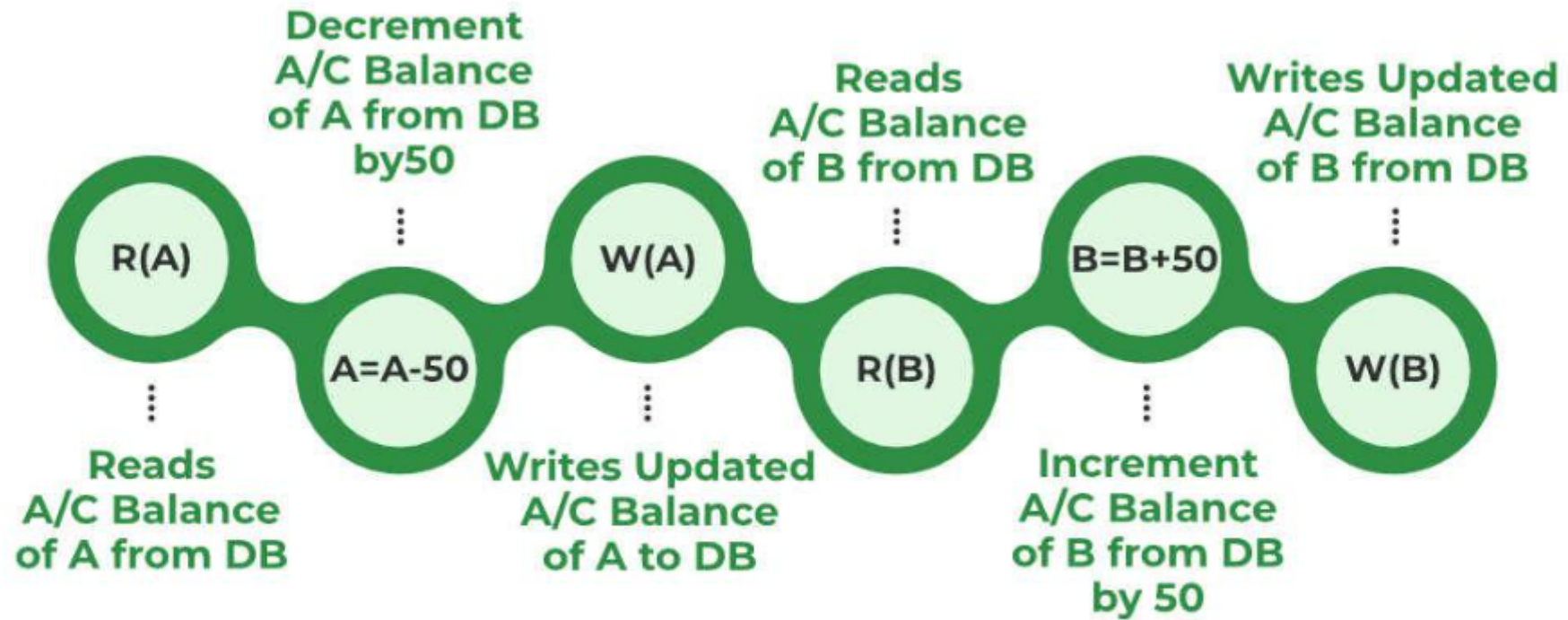
- **Example: Transaction operations to be performed to withdraw cash from an ATM**

**A transaction can include the following basic database access operation.**

- **Read/Access data (R):** Accessing the database item from disk (where the database stored data) to memory variable.
- **Write/Change data (W):** **Write** the data item from the memory variable to the disk.
- **Commit:** Commit is a transaction control language that is used to permanently save the changes done in a transaction

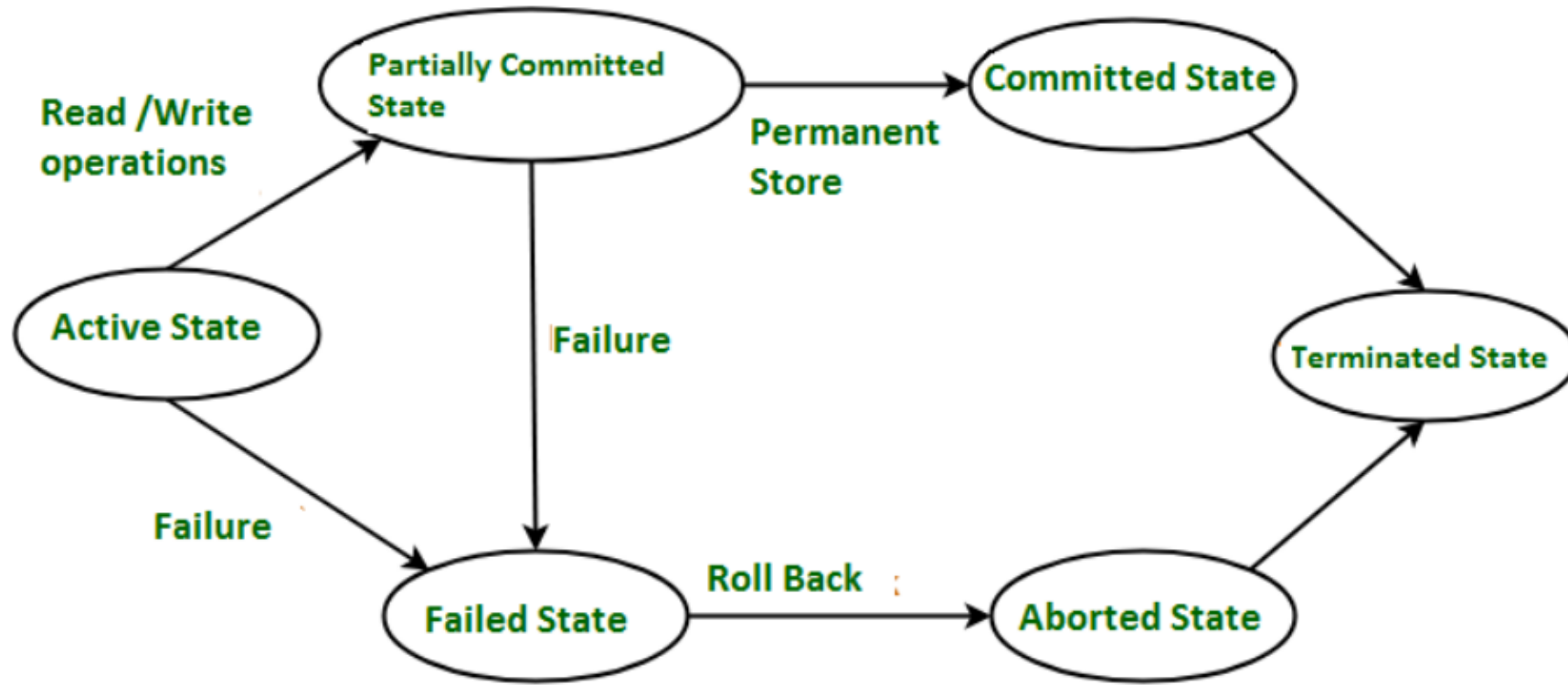
- **Example:** Transfer of 50₹ from Account A to Account B. Initially A= 500₹, B= 800₹. This data is brought to RAM from Hard Disk.

```
R(A) -- 500 // Accessed from RAM.
A = A-50 // Deducting 50₹ from A.
W(A)--450 // Updated in RAM.
R(B) -- 800 // Accessed from RAM.
B=B+50 // 50₹ is added to B's Account.
W(B) --850 // Updated in RAM.
commit // The data in RAM is taken back to Hard Disk.
```



*Stages of Transaction*

# Transaction States



Transaction States in DBMS

**1. Active State** – This is the first stage of a transaction, when the transaction's instructions are being executed.

- It is the first stage of any transaction when it has begun to execute. The execution of the transaction takes place in this state.
- Operations such as insertion, deletion, or updation are performed during this state.
- During this state, the data records are under manipulation and they are not saved to the database, rather they remain somewhere in a buffer in the main memory.

## 2. Partially Committed –

- The transaction has finished its final operation, but the changes are still not saved to the database.
- After completing all read and write operations, the modifications are initially stored in main memory or a local buffer. If the changes are made permanent on the DataBase then the state will change to “committed state” and in case of failure it will go to the “failed state”.



- **3. Failed State –**

- If any of the transaction-related operations cause an error during the active or partially committed state, further execution of the transaction is stopped and it is brought into a failed state.
- Here, the database recovery system makes sure that the database is in a consistent state.

**5. Aborted State-** If a transaction reaches the failed state due to a failed check, the database recovery system will attempt to restore it to a consistent state. If recovery is not possible, the transaction is either rolled back or cancelled to ensure the database remains consistent.

In the aborted state, the DBMS recovery system performs one of two actions:

- **Kill the transaction:** The system terminates the transaction to prevent it from affecting other operations.
- **Restart the transaction:** After making necessary adjustments, the system reverts the transaction to an active state and attempts to continue its execution.

- **6. Committed-** This state of transaction is achieved when all the transaction-related operations have been executed successfully along with the Commit operation,
- i.e. data is saved into the database after the required manipulations in this state. This marks the successful completion of a transaction.
- **7. Terminated State** – If there isn't any roll-back or the transaction comes from the “committed state”, then the system is consistent and ready for new transaction and the old transaction is terminated.

# ACID Properties

Properties of Transaction:

- Atomicity
- Consistency
- Isolation
- Durability

## Atomicity

- States that all operations of the transaction take place at once if not, the transactions are aborted.
- There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.
- Atomicity involves the following two operations:
- **Abort:** If a transaction stops or fails, none of the changes it made will be saved or visible.
- **Commit:** If a transaction completes successfully, all the changes it made will be saved and visible.

## Consistency

- The rules (integrity constraint) that keep the database accurate and consistent are followed before and after a transaction.
- When a transaction is completed, it leaves the database either as it was before or in a new stable state.
- This property means every transaction works with a reliable and consistent version of the database.
- The transaction is used to transform the database from one consistent state to another consistent state. A transaction changes the database from one consistent state to another consistent state.

## Isolation

- It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.
- In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction T2 until the transaction T1 ends.
- The concurrency control subsystem of the DBMS enforced the isolation property

## Durability

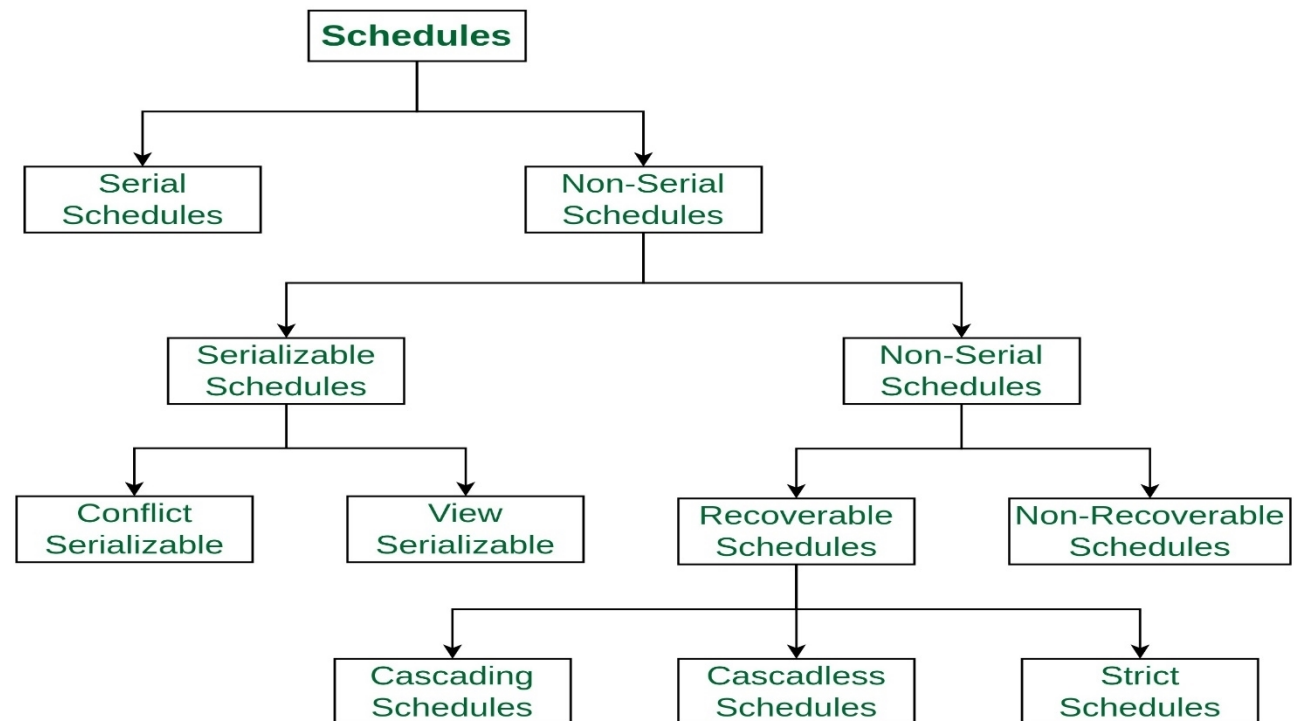
- The durability property is used to indicate the performance of the database's consistent state. It states that the transaction made the permanent changes.
- They cannot be lost by the erroneous operation of a faulty transaction or by the system failure. When a transaction is completed, then the database reaches a state known as the consistent state. That consistent state cannot be lost, even in the event of a system's failure.
- The recovery subsystem of the DBMS has the responsibility of Durability property.



# Concept of Schedule

- Sequence of operations from one transaction to the next is referred to as a schedule.
- Schedule, as the name suggests, is a process of lining the transactions and executing them one by one.
- When there are multiple transactions that are running in a concurrent manner and the order of operation is needed to be set so that the operations do not overlap each other, Scheduling is brought into play and the transactions are timed accordingly.

## Types of schedules in DBMS



- **Serial Schedules:**
- Schedules in which the transactions are executed non-interleaved, i.e., a serial schedule is one in which no transaction starts until a running transaction has ended are called serial schedules.

- Example: Consider the following schedule involving two transactions T1 and T2 .

| T <sub>1</sub> | T <sub>2</sub> |
|----------------|----------------|
| R(A)           |                |
| W(A)           |                |
| R(B)           |                |
|                | W(B)           |
|                | R(A)           |
|                | R(B)           |

- **Non-Serial Schedule:**
- This is a type of Scheduling where the operations of multiple transactions are interleaved.
- This might lead to a rise in the concurrency problem.
- The transactions are executed in a non-serial manner, keeping the end result correct and same as the serial schedule.
- Unlike the serial schedule where one transaction must wait for another to complete all its operation, in the non-serial schedule, the other transaction proceeds without waiting for the previous transaction to complete.

- This sort of schedule does not provide any benefit of the concurrent transaction.
- It can be of two types namely, Serializable and Non-Serializable Schedule.
- The Non-Serial Schedule can be divided further into Serializable and Non-Serializable.

- **Serializable:**
- This is used to maintain the consistency of the database. It is mainly used in the Non-Serial scheduling to verify whether the scheduling will lead to any inconsistency or not.
- Serial schedule does not need the serializability because it follows a transaction only when the previous transaction is complete.
- The non-serial schedule is said to be in a serializable schedule only when it is equivalent to the serial schedules, for an  $n$  number of transactions.

- Since concurrency is allowed in this case thus, multiple transactions can execute concurrently.
- A serializable schedule helps in improving both resource utilization and CPU throughput.



**1. Conflict Serializable:** A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations. Two operations are said to be conflicting if all conditions satisfy:

1. They belong to different transactions
2. They operate on the same data item
3. At Least one of them is a write operation

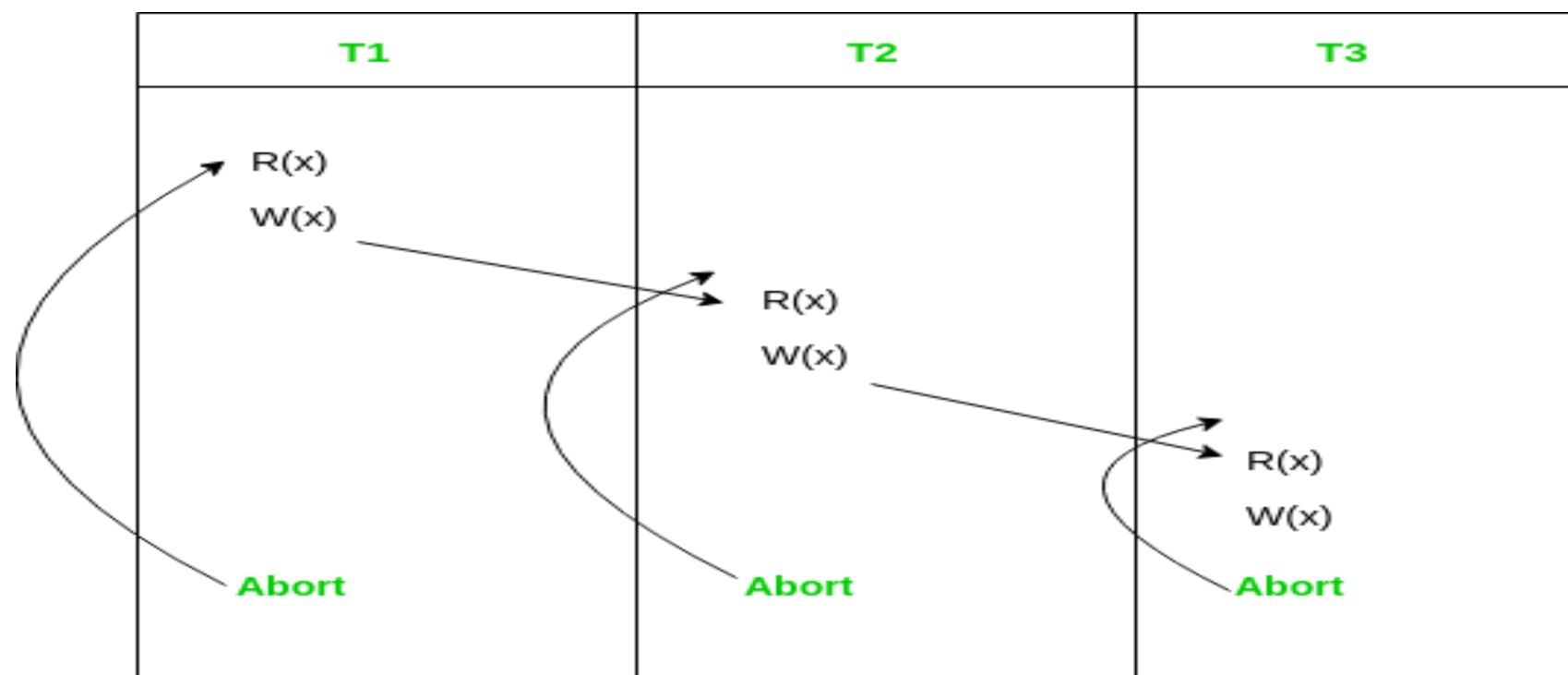
- **View Serializable:**

- A Schedule is called view serializable if it is view equal to a serial schedule (no overlapping transactions).
- A conflict schedule is a view serializable but if the serializability contains blind writes, then the view serializable does not conflict serializable.

- **Non-Serializable:** The non-serializable schedule is divided into two types,
- Recoverable and Non-recoverable Schedule.
- **Recoverable Schedule:**
- Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules.
- In other words, if some transaction  $T_j$  is reading value updated or written by some other transaction  $T_i$ , then the commit of  $T_j$  must occur after the commit of  $T_i$ .
- **Example** – Consider the following schedule involving two transactions  $T_1$  and  $T_2$ .

| $T_1$  | $T_2$  |
|--------|--------|
| R(A)   |        |
| W(A)   |        |
|        | W(A)   |
|        | R(A)   |
| commit |        |
|        | commit |

- This is a recoverable schedule since  $T_1$  commits before  $T_2$ , that makes the value read by  $T_2$  correct.
- There can be three types of recoverable schedule:
- **Cascading Schedule:**
- Also called Avoids cascading aborts/rollbacks (ACA). When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such scheduling is referred to as Cascading rollback or cascading abort.



**Figure** - Cascading Abort

- **Cascadeless Schedule:**

- Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules.
- Avoids that a single transaction abort leads to a series of transaction rollbacks.
- A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

- In other words, if some transaction  $T_j$  wants to read value updated or written by some other transaction  $T_i$ , then the commit of  $T_j$  must read it after the commit of  $T_i$ .
- **Example:** Consider the following schedule involving two transactions  $T_1$  and  $T_2$ .



| $T_1$  | $T_2$  |
|--------|--------|
| R(A)   |        |
| W(A)   |        |
|        | W(A)   |
| commit |        |
|        | R(A)   |
|        | commit |

- This schedule is cascadeless. Since the updated value of **A** is read by  $T_2$  only after the updating transaction i.e.  $T_1$  commits.
- **Example:** Consider the following schedule involving two transactions  $T_1$  and  $T_2$ .

| $T_1$ | $T_2$ |
|-------|-------|
| R(A)  |       |
| W(A)  |       |
|       | R(A)  |
|       | W(A)  |
| abort |       |
|       | abort |

- It is a recoverable schedule but it does not avoid cascading aborts.
- It can be seen that if  $T_1$  aborts,  $T_2$  will have to be aborted too in order to maintain the correctness of the schedule as  $T_2$  has already read the uncommitted value written by  $T_1$ .

- **Strict Schedule:** A schedule is strict if for any two transactions  $T_i, T_j$ , if a write operation of  $T_i$  precedes a conflicting operation of  $T_j$  (either read or write), then the commit or abort event of  $T_i$  also precedes that conflicting operation of  $T_j$ .
- In other words,  $T_j$  can read or write updated or written value of  $T_i$  only after  $T_i$  commits/aborts.
- **Example:** Consider the following schedule involving two transactions  $T_1$  and  $T_2$ .

| $T_1$  | $T_2$  |
|--------|--------|
| R(A)   |        |
|        | R(A)   |
| W(A)   |        |
| commit |        |
|        | W(A)   |
|        | R(A)   |
|        | commit |

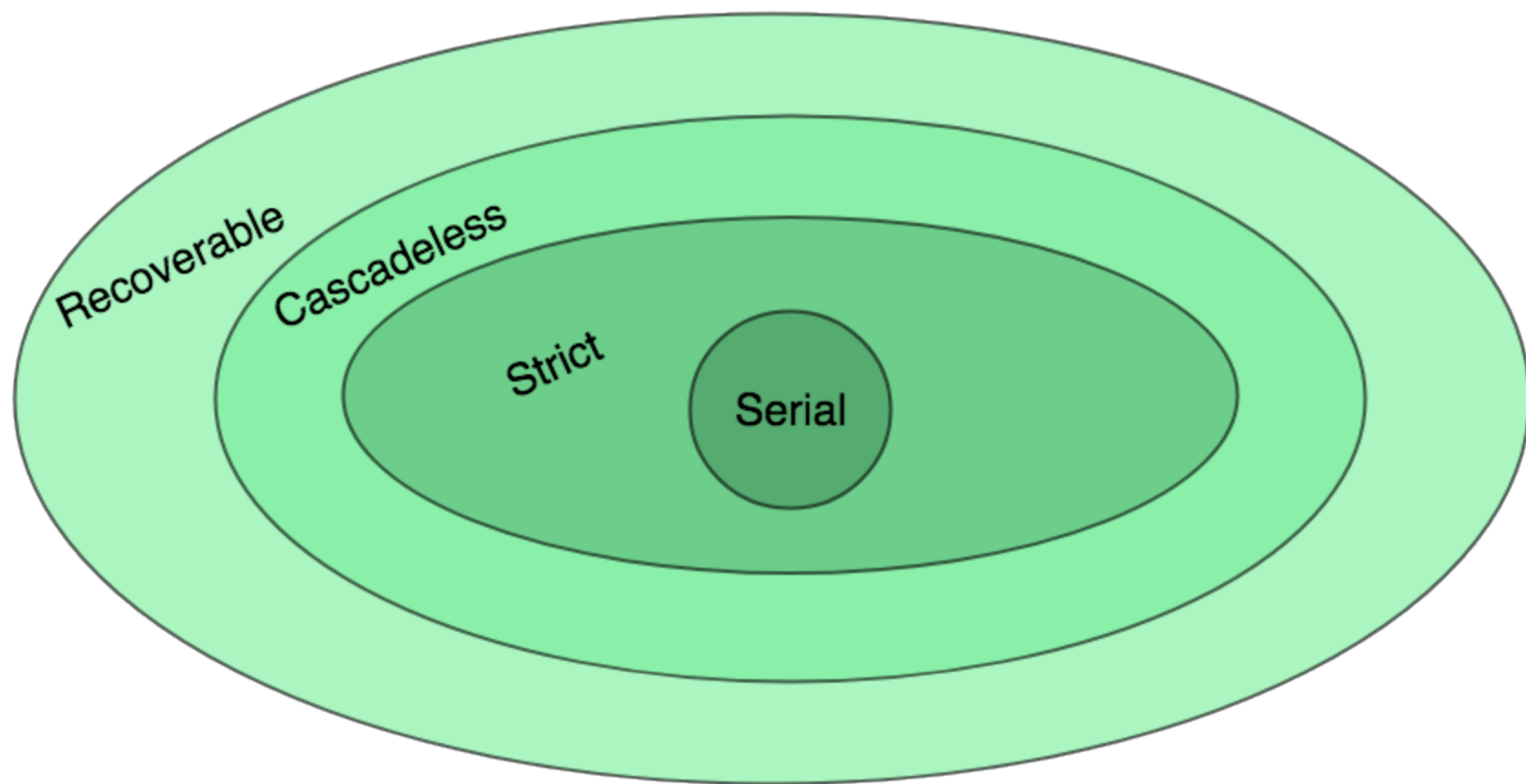
- This is a strict schedule since  $T_2$  reads and writes  $A$  which is written by  $T_1$  only after the commit of  $T_1$ .

| $T_1$ | $T_2$  |
|-------|--------|
| R(A)  |        |
| W(A)  |        |
|       | W(A)   |
|       | R(A)   |
|       | commit |
| abort |        |



- $T_2$  read the value of A written by  $T_1$ , and committed.  $T_1$  later aborted, therefore the value read by  $T_2$  is wrong, but since  $T_2$  committed, this schedule is **non-recoverable** .

1. Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable schedules.
2. Strict schedules are stricter than cascadeless schedules or are a subset of cascadeless schedules.
3. Serial schedules satisfy constraints of all recoverable, cascadeless and strict schedules and hence is a subset of strict schedules.



# Timestamp-based concurrency control

- Timestamp-based concurrency control is a method used in database systems to ensure that transactions are executed safely and consistently without conflicts, even when multiple transactions are being processed simultaneously.
- Approach relies on timestamps to manage and coordinate the execution order of transactions.
- Refer to the timestamp of a transaction  $T$  as  $TS(T)$ .

- The Timestamp Ordering Protocol is a method used in database systems to order transactions based on their timestamps
- A timestamp is a unique identifier assigned to each transaction, typically determined using the system clock or a logical counter.
- Transactions are executed in the ascending order of their timestamps, ensuring that older transactions get higher priority.
- For example:
  - If Transaction T1 enters the system first, it gets a timestamp  $TS(T1) = 007$  (assumption).
  - If Transaction T2 enters after T1, it gets a timestamp  $TS(T2) = 009$  (assumption).

- This means T1 is “older” than T2 and T1 should execute before T2 to maintain consistency.
- **Transaction Priority:**
- Older transactions (those with smaller timestamps) are given higher priority.
- For example, if transaction T1 has a timestamp of 007 times and transaction T2 has a timestamp of 009 times, T1 will execute first as it entered the system earlier

- **Early Conflict Management:**

- Unlike lock-based protocols, which manage conflicts during execution, timestamp-based protocols start managing conflicts as soon as a transaction is created.

- **Ensuring Serializability:**

- The protocol ensures that the schedule of transactions is serializable. This means the transactions can be executed in an order that is logically equivalent to their timestamp order.

# Deadlock Handling

- **Deadlock:**
- In DBMS a deadlock occurs when two or more transactions are unable to proceed because each transaction is waiting for the other to release locks on resources.
- This Situation Creates Cycle.
- It impacts the performance and reliability of a DBMS making it crucial to understand and manage them effectively.

**OR**

The Deadlock is a condition in a multi-user database environment where transactions are unable to complete because they are each waiting for the resources held by other transactions.



- **Characteristics of Deadlock:**

- Mutual Exclusion: Only one transaction can hold a particular resource at a time.
- Hold and Wait: The Transactions holding resources may request additional resources held by others.
- No Preemption: The Resources cannot be forcibly taken from the transaction holding them.
- Circular Wait: A cycle of transactions exists where each transaction is waiting for the resource held by the next transaction in the cycle.

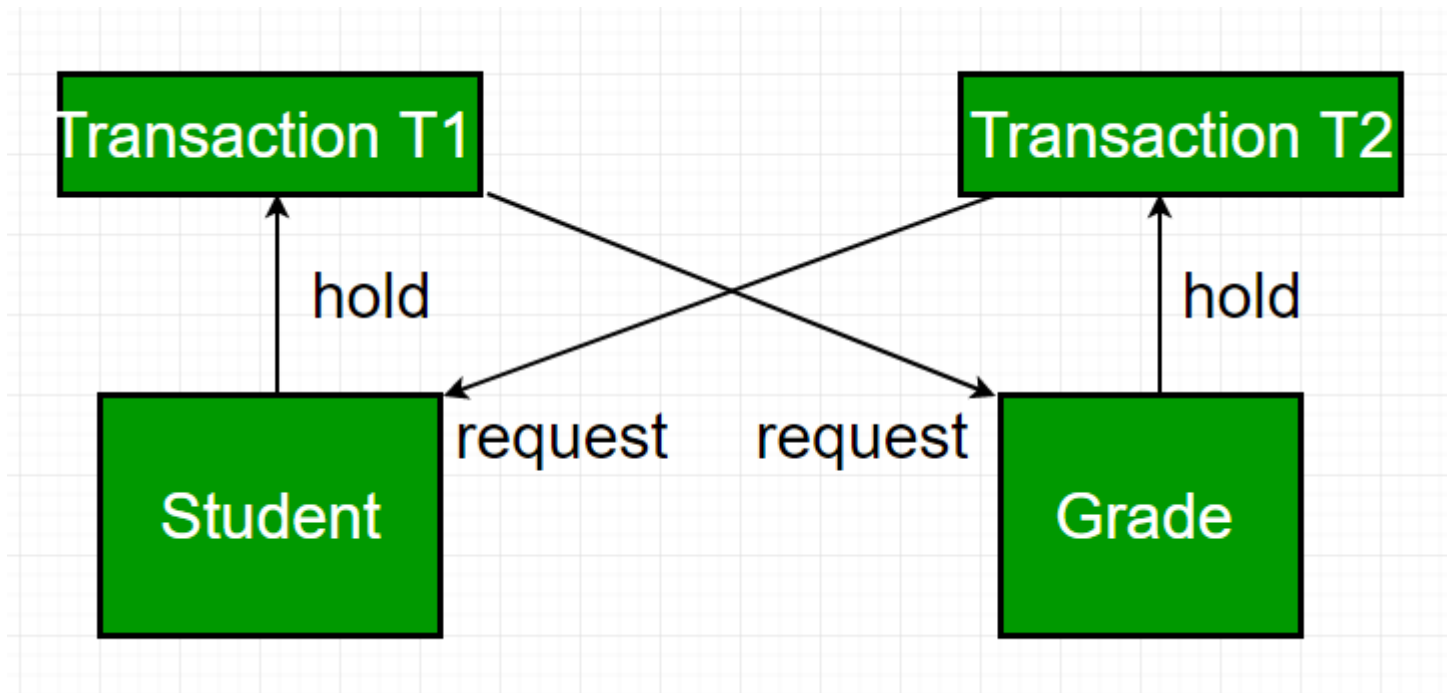
## Example –

Transaction T1 holds a lock on some rows in the Students table and **needs to update** some rows in the Grades table.

Simultaneously, Transaction **T2 holds** locks on those very rows (Which T1 needs to update) in the Grades table **but needs** to update the rows in the Student table **held by Transaction T1**.

- Now, the main problem arises.
- Transaction T1 will wait for transaction T2 to give up the lock, and similarly, transaction T2 will wait for transaction T1 to give up the lock.

- As a consequence, All activity comes to a halt and remains at a standstill forever unless the DBMS detects the deadlock and aborts one of the transactions.



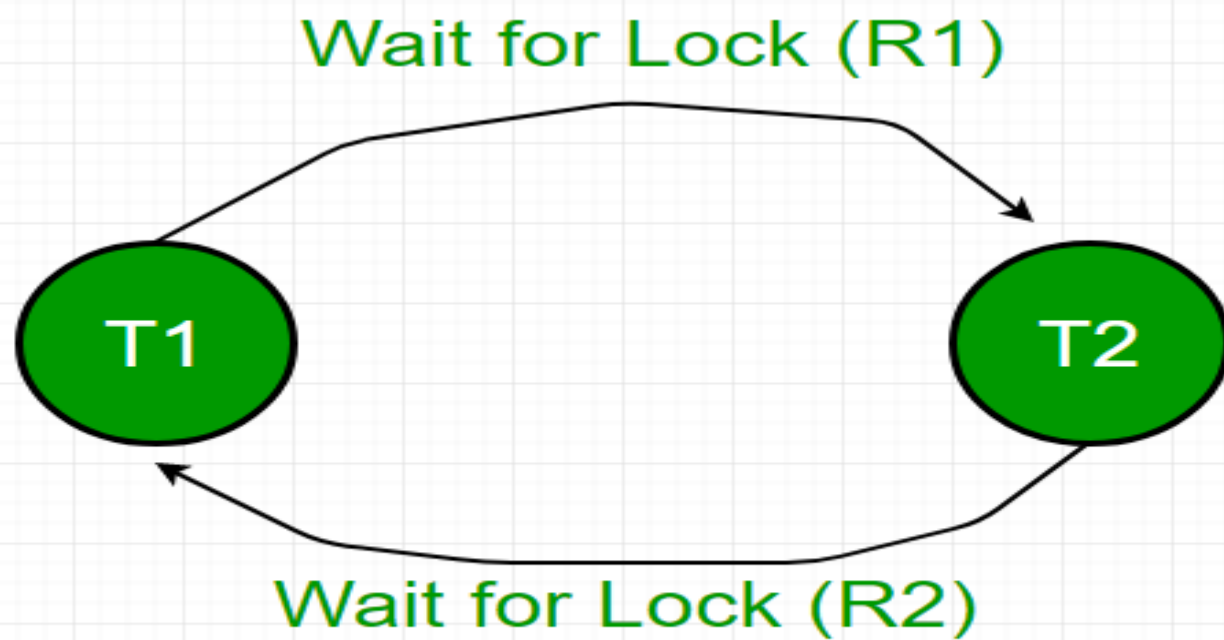
- **Deadlock Avoidance**

- When a database is stuck in a deadlock, It is always better to avoid the deadlock rather than restarting or aborting the database.
- One method of avoiding deadlock is using application-consistent logic.
- In the above-given example, Transactions that access Students and Grades should always access the tables in the same order.
- In this way, in the scenario described above, Transaction T1 simply waits for transaction T2 to release the lock on Grades before it begins. When transaction T2 releases the lock, Transaction T1 can proceed freely.

- Another method for avoiding deadlock is to apply both the row-level locking mechanism and the READ COMMITTED isolation level. However, It does not guarantee to remove deadlocks completely.

# Deadlock Detection

- When a transaction waits indefinitely to obtain a lock, The database management system should detect whether the transaction is involved in a deadlock or not.
- **Wait-for-graph** is one of the methods for detecting the deadlock situation.
- This method is suitable for smaller databases.
- In this method, a graph is drawn based on the transaction and its lock on the resource.
- If the graph created has a closed loop or a cycle, then there is a deadlock.



Deadlock Situation

# Deadlock Prevention

- For a large database, the deadlock prevention method is suitable. A deadlock can be prevented if the resources are allocated in such a way that a deadlock never occurs.
- The DBMS analyzes the operations whether they can create a deadlock situation or not, If they do, that transaction is never allowed to be executed.
- Two Methods for Deadlock Prevention:
- **Wait-Die Scheme:**
- In this scheme, If a transaction requests a resource that is locked by another transaction, then the DBMS simply checks the timestamp of both transactions and allows the older transaction to wait until the resource is available for execution.



- Suppose, there are two transactions T1 and T2, and Let the timestamp of any transaction T be  $TS(T)$ . Now, If there is a lock on T2 by some other transaction and T1 is requesting resources held by T2, then DBMS performs the following actions:
- Checks if  $TS(T1) < TS(T2)$  – if T1 is the older transaction and T2 has held some resource, then it allows T1 to wait until resource is available for execution.
- That means if a younger transaction has locked some resource and an older transaction is waiting for it, then an older transaction is allowed to wait for it till it is available.

- If T1 is an older transaction and has held some resource with it and if T2 is waiting for it, then T2 is killed and restarted later with random delay but with the same timestamp.
  - i.e. if the older transaction has held some resource and the younger transaction waits for the resource, then the younger transaction is killed and restarted with a very minute delay with the same timestamp.
- This scheme allows the older transaction to wait but kills the younger one.

- **Wound Wait Scheme:**

- In this scheme, if an older transaction requests for a resource held by a younger transaction, then an older transaction forces a younger transaction to kill the transaction and release the resource.
- The younger transaction is restarted with a minute delay but with the same timestamp.
- If the younger transaction is requesting a resource that is held by an older one, then the younger transaction is asked to wait till the older one releases it.

# Log based Recovery in DBMS

- Log-based recovery in DBMS ensures data can be maintained or restored in the event of a system failure.
- The DBMS records every transaction on stable storage, allowing for easy data recovery when a failure occurs.
- For each operation performed on the database, a log file is created.
- Transactions are logged and verified before being applied to the database, ensuring data integrity.

No  
Image

# Log in DBMS

- A log is a sequence of records that document the operations performed during database transactions.
- Logs are stored in a log file for each transaction, providing a mechanism to recover data in the event of a failure.
- For every operation executed on the database, a corresponding log record is created.
- It is critical to store these logs before the actual transaction operations are applied to the database, ensuring data integrity and consistency during recovery processes.

- For example, consider a transaction to modify a student's city. This transaction generates the following logs:
- **Start Log:** When the transaction begins, a log is created to indicate the start of the transaction.
- Format: <Tn, Start>
- Here, Tn represents the transaction identifier.
- Example: <T1, Start> indicates that Transaction 1 has started.

- **Operation Log:** When the city is updated, a log is recorded to capture the old and new values of the operation.
- Format: <Tn, Attribute, Old\_Value, New\_Value>
- Example: <T1, City, 'Gorakhpur', 'Noida'> shows that in Transaction 1, the value of the City attribute has changed from 'Gorakhpur' to 'Noida'.
- **Commit Log:** Once the transaction is successfully completed, a final log is created to indicate that the transaction has been completed and the changes are now permanent.



- Format: <T<sub>n</sub>, Commit>
- Example: <T<sub>1</sub>, Commit> signifies that Transaction 1 has been successfully completed.
- These logs play a crucial role in ensuring that the database can recover to a consistent state after a system crash.
- If a failure occurs, the DBMS can use these logs to either roll back incomplete transactions or redo committed transactions to maintain data consistency.

- **Key Operations in Log-Based Recovery**

## **Undo Operation**

The undo operation reverses the changes made by an uncommitted transaction, restoring the database to its previous state.

### **Example of Undo:**

Consider a transaction T1 that updates a bank account balance but fails before committing:

#### **Initial State:**

- Account balance = 500.

## **Transaction T1:**

- Update balance to 600.
- **Log entry:**
- <T1, Balance, 500, 600>

## **Failure:**

- T1 fails before committing.

## Undo Process:

- Use the old value from the log to revert the change.
- Set balance back to 500.
- Final log entry after undo:
- <T1, Abort>

## **Redo Operation**

The redo operation re-applies the changes made by a committed transaction to ensure consistency in the database.

### **Example of Redo:**

Consider a transaction T2 that updates an account balance but the database crashes before changes are permanently reflected:

#### **Initial State:**

- Account balance = 300.

#### **Transaction T2:**

- Update balance to 400.

- **Log entries:**

- <T2, Start><T2, Balance, 300, 400><T2, Commit>

**Crash:**

- Changes are not reflected in the database.

**Redo Process:**

- Use the new value from the log to reapply the committed change.
- Set balance to 400.

## Undo-Redo Example:

Assume two transactions:

- T1: Failed transaction (requires undo).
- T2: Committed transaction (requires redo).

## Log File:

- <T1, Start><T1, Balance, 500, 600><T2, Start><T2, Balance, 300, 400><T2, Commit><T1, Abort>

## **Recovery Steps:**

### **Identify Committed and Uncommitted Transactions:**

- T1: Not committed → Undo.
- T2: Committed → Redo.

### **Undo T1:**

- Revert balance from 600 to 500.

### **Redo T2:**

- Reapply balance change from 300 to 400.



- **Approaches to Modify the Database**

- In database systems, changes to the database can be made using two main methods: Immediate Modification and Deferred Modification.

- **1. Immediate Modification**

- In the Immediate Modification method, the database is updated as soon as a change is made during a transaction, even before the transaction is committed. Logs are written before making any changes to ensure recovery is possible in case of a system failure.

**No  
Image**

- **Transaction T0:**

- Start: Transaction T0 begins, and the log records  $\langle T\_0 \text{ start} \rangle$ .
- Change to A: A is updated from 500 to 450. The change is logged as  $\langle T\_0, A, 500, 450 \rangle$ , and A's new value is reflected in memory.
- Change to B: B is updated from 300 to 350. The change is logged as  $\langle T\_0, B, 300, 350 \rangle$ , and B's new value is reflected in memory.
- Commit: After both changes are completed, T0 is committed. The log records  $\langle T\_0 \text{ commit} \rangle$ , and the new values of A and B are permanently written to storage.

- **Transaction T1:**

- Start: Transaction T1 begins, and the log records <T\_1 start>.
- Change to C: C is updated from 200 to 180. The change is logged as <T\_1, C, 200, 180>, and C's new value is reflected in memory.
- Commit: After the change, T1 is committed. The log records <T\_1 commit>, and the new value of C is permanently written to storage.

- **Key Characteristics of Immediate Modification:**
- **Changes Are Applied Immediately:** Updates to the database are made as soon as a transaction executes an operation, even before the transaction commits.
- **Requires Undo and Redo for Recovery:** Uncommitted changes are reverted using undo, while committed changes are reapplied using redo during recovery.
- **Logs Are Written First:** All changes are logged before being applied to ensure recoverability and consistency in case of failure.

## 2. Deferred Modification

- In the Deferred Modification method, changes to the database are not applied immediately.
- Instead, they are logged and stored temporarily.
- The database is only updated after the transaction is fully committed.
- This method ensures that no partial changes are made to the database, reducing the risk of inconsistency.

No  
Image

- **Transaction T0:**

- Start: Transaction T0T\_0T0 begins, and the log records <T\_0 start>.
- Change to A: A is intended to be updated from 1000 to 950.
- The log entry <T\_0, A, 1000, 950> is recorded, but the change is not applied to the database yet. The value of A in the database remains 1000.
- Change to B: B is intended to be updated from 2000 to 2050.
- The log entry <T\_0, B, 2000, 2050> is recorded, but the change is not applied to the database yet. The value of B in the database remains 2000.
- Commit: When T0 commits, the changes to A and B are applied to the database: A=950 B=2050



- **Key Characteristics of Deferred Modification:**
- **Changes Are Logged First:** All updates are recorded in the log before any changes are applied to the database.
- **Changes Are Applied Only After Commit:** No updates are made to the database until the transaction commits. This prevents partial changes in case of a failure.
- **Simpler Recovery Process:** Since no changes are applied before commit, only redo operations are needed for recovery.

- **Recovery using Log records**

- Log-based recovery is a method used in database systems to restore the database to a consistent state after a crash or failure. The process uses a transaction log, which keeps a record of all operations performed on the database, including updates, inserts, deletes, and transaction states (start, commit, or abort).

- **How Log-Based Recovery Works ?**

- **Transaction Log:**

- The log stores all changes made by transactions, ensuring recoverability.
- Each transaction's start, changes (with old and new values), and its commit or abort state are recorded.

- **Recovery Process:**
- After a system crash, the database uses the log to determine:
- **Undo:** Transactions that started but didn't commit (incomplete transactions) are undone to reverse their changes.
- **Redo:** Transactions that committed before the crash are redone to ensure their changes are applied to the database.

## Checkpoints

- Checkpointing is a process used in DBMS to streamline the recovery procedure after a system crash by reducing the amount of log data that needs to be examined.
- It helps save the current state of the database and active transactions to make recovery faster and more efficient.
- For example: <checkpoint L> means the database state and the list of active transactions (L) were saved.

## Why Checkpointing is Important ?

- **Speeds Up Recovery:**
  - Searching through the entire log during recovery is time-consuming.
  - Checkpoints help limit the recovery process to only relevant transactions.
- **Avoids Unnecessary Redo:**
  - Transactions that have already saved their updates to the database don't need to be redone.
- **Recovery Using Checkpoints**
  - When a crash occurs, recovery involves the following steps:

- **Find the Most Recent Checkpoint:**
  - Scan the log backward to locate the last <checkpoint> record.
- **Identify Relevant Transactions:**
  - Continue scanning backward until the <Ti start> record of the oldest active transaction at the time of the checkpoint is found.

- Example:
- Consider the set of transactions  $\{T_0, T_1, \dots, T_{100}\}$ .
- Suppose that the most recent checkpoint took place during the execution of transaction  $T_{67}$  and  $T_{69}$ , while  $T_{68}$  and all transactions with subscripts lower than 67 completed before the checkpoint.
- Thus, only transactions  $T_{67}, T_{69}, \dots, T_{100}$  need to be considered during the recovery scheme.
- Each of them needs to be redone if it has completed (that is, either committed or aborted); otherwise, it was incomplete, and needs to be undone.

- Transactions that started before this can be ignored, as their updates are already saved.
- Perform Undo Operations:
  - For transactions without a  $\langle T_i \text{ commit} \rangle$  record, execute  $\text{undo}(T_i)$  to reverse incomplete changes. (Only needed in the immediate modification approach.)
- Perform Redo Operations:
  - For transactions with a  $\langle T_i \text{ commit} \rangle$  record, execute  $\text{redo}(T_i)$  to reapply their changes if needed.



## Shadow Paging

- Shadow paging is a fundamental recovery technique used in database management systems (DBMS) to ensure the reliability and consistency of data.
- It plays a crucial role in maintaining atomicity and durability which are the two core properties of transaction management.
- Unlike [log-based recovery](#) mechanisms that rely on recording detailed logs of changes, shadow paging offers a simpler, log-free approach by maintaining two versions of the database state: the shadow page table and the current page table. This technique is also known as Cut-of-Place updating.

- This technique ensures that a database can recover seamlessly from failures without losing data integrity.
- During a transaction, updates are made to a new version of the database pages tracked by the current page table, while the shadow page table preserves the pre-transaction state.
- This dual-table approach allows for efficient crash recovery and simplifies the commit and rollback processes
- **Page Table** : A [page table](#) is a data structure that maps logical pages (a logical division of data) to physical pages (actual storage on disk).
- Each entry in the page table corresponds to a physical page location on the disk.
- The database uses the page table to retrieve or modify data.

- **How Shadow Paging Works ?**

- Shadow paging is a recovery technique that views the database as a collection of fixed-sized logical storage units, known as pages, which are mapped to physical storage blocks using a structure called the page table.
- The page table enables the system to efficiently locate and manage database pages.

- **Start of Transaction:**

- The shadow page table is created by copying the current page table.
- The shadow page table represents the original, unmodified state of the database.
- This table is saved to disk and remains unchanged throughout the transaction.

| Logical Page | Shadow Page Table (Disk) | Current Page Table |
|--------------|--------------------------|--------------------|
| P1           | Address_1                | Address_1          |
| P2           | Address_2                | Address_2          |
| P3           | Address_3                | Address_3          |

- **Transaction Execution:**

- Updates are made to the database by creating new pages.
- The current page table reflects these changes, while the shadow page table remains unchanged.

- **Page Modification:**

- If a logical page (e.g. P2) needs to be updated:
- A new version of the page (P2') is created in memory and written to a new physical storage block.
- The current page table entry for P2 is updated to point to P2'.
- The shadow page table still points to the original page P2, ensuring it is unaffected by the changes.

| Logical Page | Shadow Page Table (Disk) | Current Page Table |
|--------------|--------------------------|--------------------|
| P1           | Address_1                | Address_1          |
| P2           | Address_2                | Address_4 (P2')    |
| P3           | Address_3                | Address_3          |

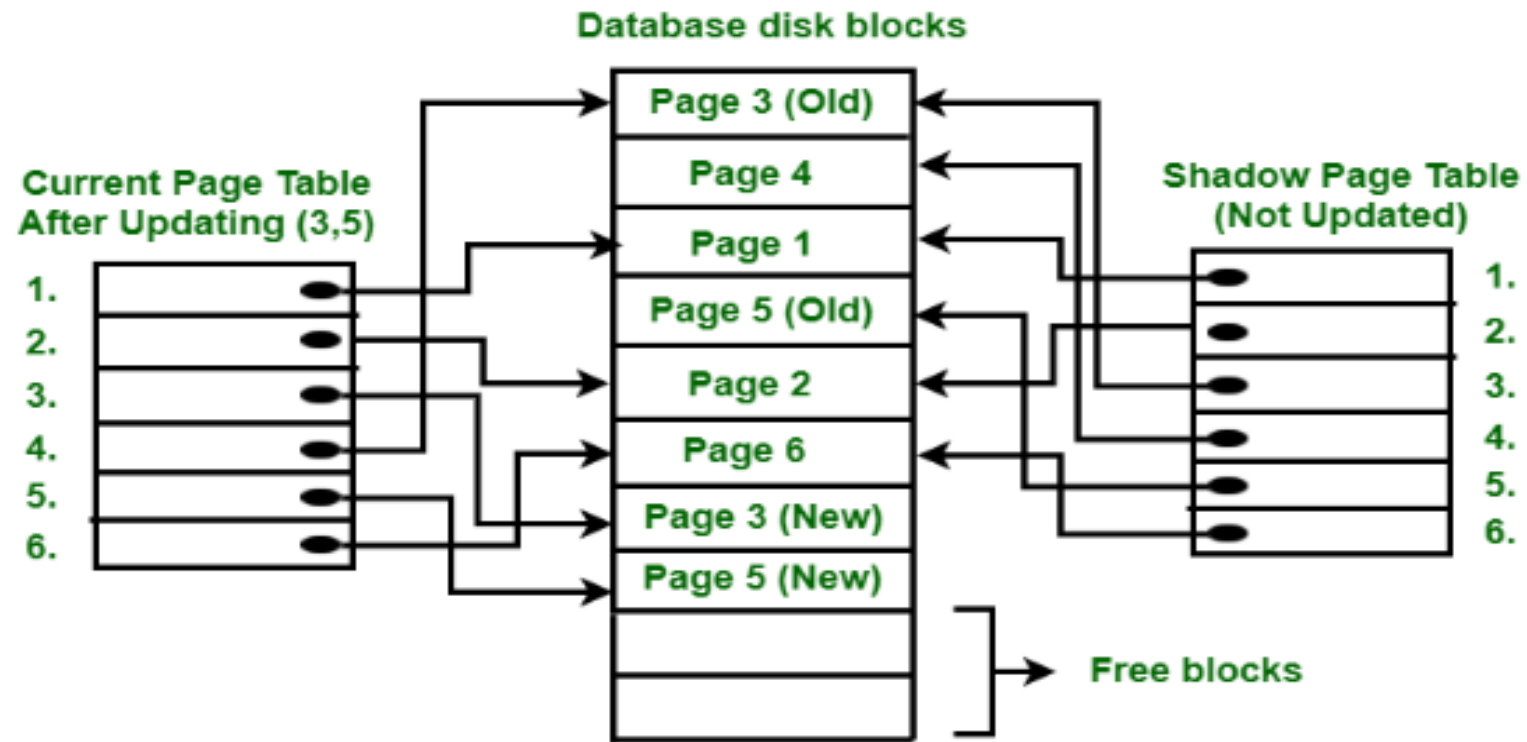
- **Commit:**
- If the transaction is successful, the shadow page table is replaced by the current page table.
- This replacement makes the changes permanent.



| Logical Page | Shadow Page Table (Disk) | Current Page Table |
|--------------|--------------------------|--------------------|
| P1           | Address_1                | Address_1          |
| P2           | Address_4 (P2')          | Address_4 (P2')    |
| P3           | Address_3                | Address_3          |

- **Abort:**

- If the transaction is aborted, the current page table is discarded, leaving the shadow page table intact.
- Since the shadow page table still points to the original pages, no changes are reflected in the database.



- To understand concept, consider above figure. In this 2 write operations are performed on page 3 and 5. Before start of write operation on page 3, current page table points to old page 3. When write operation starts following steps are performed :
- Firstly, search start for available free block in disk blocks.
- After finding free block, it copies page 3 to free block which is represented by Page 3 (New).
- Now current page table points to Page 3 (New) on disk but shadow page table points to old page 3 because it is not modified.
- The changes are now propagated to Page 3 (New) which is pointed by current page table.

- **COMMIT Operation** : To commit transaction following steps should be done :
- All the modifications which are done by transaction which are present in buffers are transferred to physical database.
- Output current page table to disk.
- Disk address of current page table output to fixed location which is in stable storage containing address of shadow page table. This operation overwrites address of old shadow page table. With this current page table becomes same as shadow page table and transaction is committed.

- **Failure :**
- If the system crashes during the execution of a transaction but before the commit operation, it is sufficient to free the modified database pages and discard the current page table.
- The database state can be restored to its pre-transaction condition by reinstalling the shadow page table.
- If the system crashes after the final write operation, the changes made by the transaction remain unaffected.
- These changes are preserved, and there is no need to perform a redo operation.