DIGITAL NOTES ON

INTRODUCTION TO DBMS

(R20A0551)

B.TECH - ECE II YEAR - II SEM

(2022-23)

Prepared

By

Dr. M. Arunkumar (Assoc. Professor)

Dr. P. Vanitha (Assoc. Professor)

Mrs. K. Vijaya Bharathi (Asst. Professor)

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

II Year B.Tech. ECE- II Sem **L/T/P/C**
3/-/-/3

OPEN ELECTIVE - I
(R20A0551) INTRODUCTION TO DBMS

COURSE OBJECTIVES:

1) To understand the basic concepts and the applications of database systems
2) To Master the basics of SQL and construct queries using SQL
3) To understand the relational database design principles
4) To become familiar with the basic issues of transaction processing and concurrency control
5) To become familiar with database storage structures and access techniques

UNIT I:

INTRODUCTION

Database: Purpose of Database Systems, File Processing System Vs DBMS, History, Characteristic- Three schema Architecture of a database, Functional components of a DBMS. DBMS Languages- Database users and DBA.

UNIT II:

DATABASE DESIGN

ER Model: Objects, Attributes and its Type. Entity set and Relationship set-Design Issues of ER model-Constraints. Keys-primary key, Super key, candidate keys. Introduction to relational model-Tabular, Representation of Various ER Schemas. ER Diagram Notations- Goals of ER Diagram- Weak Entity Set- Views.

UNIT III:

STRUCTURED QUERY LANGUAGE

SQL: Overview, The Form of Basic SQL Query -UNION, INTERSECT, and EXCEPT– join operations: equi join and non equi join-Nested queries - correlated and uncorrelated- Aggregate Functions-Null values. Views, Triggers.

UNIT IV :

DEPENDENCIES AND NORMAL FORMS

Importance of a good schema design,:- Problems encountered with bad schema designs, Motivation for normal forms- functional dependencies, -Armstrong's axioms for FD's- Closure of a set of FD's,- Minimal covers-Definitions of 1NF, 2NF, 3NF and BCNF- Decompositions and desirable properties.

UNIT V:

Transactions: Transaction concept, transaction state, System log, Commit point, Desirable Properties of a Transaction, concurrent executions, serializability, recoverability, implementation of isolation, transaction definition in SQL, Testing for serializability**,** Serializability by Locks-Locking Systems with Several Lock Modes- Concurrency Control by Timestamps, validation.

TEXT BOOKS:

1) Abraham Silberschatz, Henry F. Korth, S. Sudarshan,‖ Database System Concepts‖, McGraw- Hill, 6th Edition , 2010.

2) Fundamental of Database Systems, by Elmasri, Navathe, Somayajulu, and Gupta, Pearson Education.

REFERENCE BOOKS:

1) Raghu Ramakrishnan, Johannes Gehrke, -Database Management System**||**, McGraw Hill., 3rd Edition 2007.
2) Elmasri&Navathe,‖Fundamentals of Database System,‖ Addison-Wesley Publishing, 5th Edition, 2008.
3) Date.C.J, -An Introduction to Database**||**, Addison-Wesley Pub Co, 8th Edition, 2006.
4) Peterrob, Carlos Coronel, -Database Systems – Design, Implementation, and Management‖, 9th Edition, Thomson Learning, 2009.

COURSE OUTCOMES:

1) Understand the basic concepts and the applications of database systems
2) Master the basics of SQL and construct queries using SQL
3) Understand the relational database design principles
4) Familiarize with the basic issues of transaction processing and concurrency control
5) Familiarize with database storage structures and access techniques

UNIT 1

INTRODUCTION

Data:
The un processed facts that can be recorded and which have implicit meaning known as"Data".
Example:
Customer -----
1. cname.
2. cno.
3. ccity.
Information: Processed data**.**
Database:
A database is an organized collection of data that can be modified, retrieved, or updated.
Data, DBMS, and applications associated with them together form the database concept.
A database can be of any size and varying complexity. A database may be generated and manipulated manually or it may be computerized.
It organizes the data in the form of tables, views, schemas, reports etc. For Example, university database organizes the data about students, faculty, and admin staff etc. which helps in efficient retrieval, insertion and deletion of data from it.

Meta Data: It is Database definition (or) complete description of Database.

Database Management System:
A **database management system (DBMS)** is a collection of interrelated data and a set of programs to access those data.
It is a software application that is used to create, access, maintain, and manage databases. DBMS accepts the incoming data either from an application or from a user who is manually entering it.

Purpose of Database Systems:

- The primary goal of a Database Systems is to provide a way to store and retrievedatabase information that is both *convenient* and *efficient*.
- Database systems are designed to manage large bodies of information.

- Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information.
- In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access.
- If data are to be shared among several users, the system must avoid possible anomalous results.

File Processing System:

- Before the introduction of DBMS, data was stored in a computer in operating system files.
- To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including programs
- A typical **file-processing system** is supported by a conventional operating system.
- The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files.

Disadvantages of File Processing System:

- **Data redundancy and inconsistency**:
  - Since different programmers create the files and application programs over a long period, the various files are likely to have different structures and the programs may be written in several programming languages.
  - There is always possibility of duplication of data.
  - Some data can be stored in two different files. This redundancy leads to higher storage and access cost. In addition, it may lead to **data inconsistency**; that is, the various copies of the same data may no longer agree.
- **Difficulty in accessing data:**
  - To carry out any new task a new program has to be written.
  - For example, an application program may be available to generate a list of all students, but if we are required to generate a list of students who stay in a particular location, we need to write a new program as this is not anticipated.

- **Data isolation:** As data is scattered in various files, and stored in different file formats, writing new application programs to retrieve the appropriate data is difficult.

- **Integrity problems**:
  - The data values stored in the database must satisfy certain types of consistency constraints.
  - Example, bank account balance should never fall below zero.
  - Developers enforce these constraints in the system by adding appropriate code in the various application programs.
  - However, when new constraints are added, it is difficult to change the programs to enforce them.
- **Atomicity problems:**
  - Atomicity means that operations must complete or fail as a whole unit.
  - There should not be any partial complete, especially intransactions.
  - If a failure occurs data has to be restored to the consistence state that existed prior to the failure. Example: online transactions or reservations.
  - It is difficult to ensure atomicity in a conventional file-processing system.
- **Concurrent-access anomalies**.
  - For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously.
  - But if data is getting updates by two users at the same time, there is possibility of inconsistency of data.
  - To guard against this possibility, the system must maintain some form of supervision.
  - In file processing system supervision is difficult to provide because data may be accessed by many different application programs that are coordinated.
- **Security problems**.
  - Access to the database must be restricted to authorized users only.
  - Not every user of the database system should be able to access all the data.
  - Since application programs are added to the file-processing system in an ad hoc manner, enforcing such security constraints is difficult.

Advantages of DBMS:

- **Controlling of Redundancy:**

- In a database system, by having a centralized database and centralized control of data by the DBA the unnecessary duplication of data is avoided.
- It also eliminates the extra time for processing the large volume of data.
- It results in saving the storage space.

- **Improved Data Sharing:** DBMS allows a user to share the data in any number of application programs.
- **Data Integrity:**
  - Centralized control of the data helps in permitting the administrator to define integrity constraints to the data in thedatabase.
  - New constraints can be enforced easily.
- **Security:**
  - Complete authority over the operational data is provided todatabase administrators (DBA).
  - The DBA can define authorization checks to be carried outwhenever access to sensitive data is attempted.
- **Data Consistency :**
  - By eliminating data redundancy, we greatly reduce theopportunities for inconsistency.
  - Also updating data values is greatly simplified when each value is stored in one place only.
  - Finally, one can avoid the wasted storage that results fromredundant data storage.
- **Efficient Data Access :** In a database system, the data is managed by the DBMS and all access to the data is through the DBMS providing a key to effective data processing.
- **Data Independence** :
  - The dbms provides the interface between the application programs and the data.
  - Any changes in data representation will not change the way data is provided to the application programs.
  - The DBMS handles the task of transformation of data wherever necessary.
- **Reduced Application Development and Maintenance Time :** DBMS supports many important functions that are common to many applications, accessing data stored in the DBMS, which facilitates the quick development of application

Disadvantages of DBMS

- It is bit complex. Since it supports multiple functionality to give the userthe best, the underlying software has become complex. The designers and developers should have thorough knowledge about the software toget the most out of it.
- Because of its complexity and functionality, it uses large amount of memory. It also needs large memory to run efficiently.
- DBMS system works on the centralized system, i.e.; all the users from all over the world access this database. Hence any failure of the DBMS, will impact all the users.
- DBMS is generalized software, i.e.; it is written work on the entire systems rather specific one. Hence some of the application will run slow.

Difference between File System and DBMS:

Basis	File System	DBMS
Structure	The file system is software that manages and organizes the files in a storage medium within a computer.	DBMS is software for managing the database.
Data Redundancy	Redundant data can be present in afile system.	In DBMS there is no redundant data.
Backup andRecovery	It doesn't provide backup andrecovery of data if it is lost.	It provides backup and recovery of data even if it islost.
Query processing	There is no efficient query processingin the file system.	Efficient query processing isthere in DBMS.
Consistency	There is less data consistency in thefile system.	There is more data consistency because of theprocess of normalization.

Basis	File System	DBMS
Complexity	It is less complex as compared toDBMS.	It has more complexity in handling as compared to thefile system.
Security Constraints	File systems provide less security incomparison to DBMS.	DBMS has more security mechanisms as compared tofile systems.
Cost	It is less expensive than DBMS.	It has a comparatively highercost than a file system.
Data Independence	There is no data independence.	In DBMS data independenceexists.
User Access	Only one user can access data at atime.	Multiple users can accessdata at a time.
Meaning	The user has to write procedures formanaging databases	The user not required to write procedures.
Sharing	Data is distributed in many files. So,not easy to share data	Due to centralized naturesharing is easy
Data Abstraction	It give details of storage andrepresentation of data	It hides the internal details ofDatabase
Integrity Constraints	Integrity Constraints are difficult to implement	Integrity constraints are easyto implement

Example	Cobol, C++	Oracle, SQL Server

Data Abstraction:

Data Abstraction is a process of hiding unwanted or irrelevant details from the end user. It provides a different view and helps in achieving data independence which is used to enhance the security of data.

Levels of data abstraction:

There are mainly three levels of data abstraction:

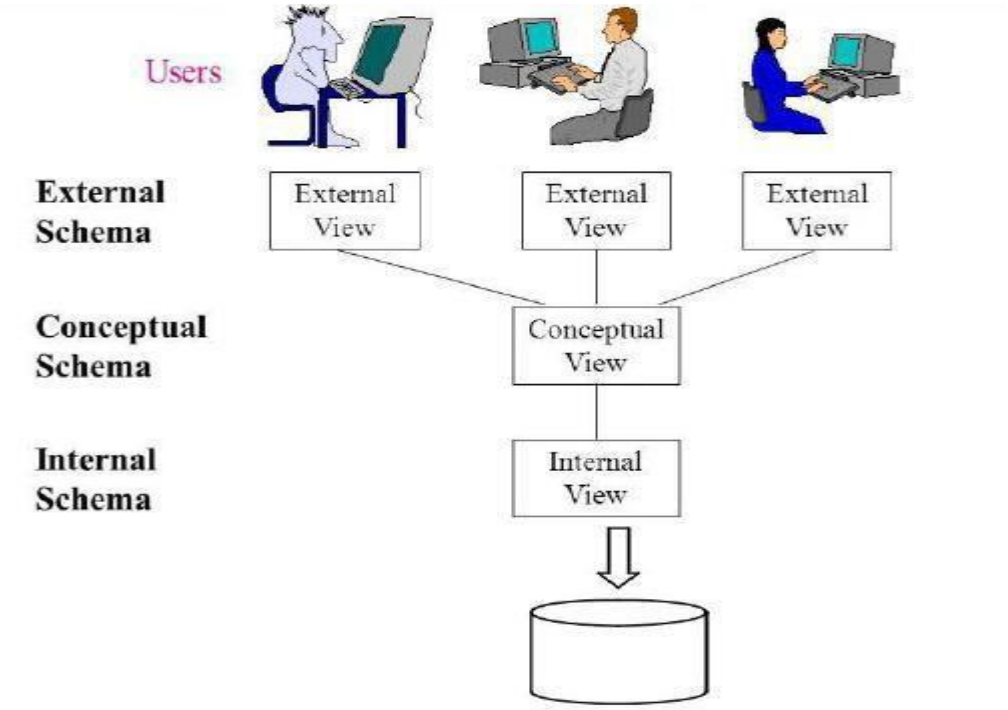


Fig: three levels of data abstraction

- **Physical level/Internal Level:** The lowest level of abstraction describes *how* the data are actually stored. It describes complex low-level data structures in detail. The Database Administrators(DBA) decide how the data has to be fragmented, where it has to be stored etc. It totally depends on the DBA, how he/she manages the database at the physical level.
- **Conceptual Level /Logical Level:** This is the intermediate level of abstraction describes *what* data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. This level is maintained by the database administrators.
- **External Level/View Level:** This is the topmost level where application programs try to view the data. Only the data needed is shown and rest of the details are hidden from this view. Different users will have different

view according to the authorization they have.

Example: If we take college database, at physical level student data, faculty data, department data etc are stored in the database using data structures. At logical level the interrelationship among different data is defined and the data-type of data stored is also defined. At view level several views of the database are defined and a database user sees some or all of these views.

Instances and Schemas:

- ✓ The collection of information stored in the database at a particular moment is called an **instance** of the database.
- ✓ The overall design of the database is called the database **Schema**. Schemas don't change frequently.

Based on the levels of abstraction we have **physical schema** and **logical schema**:

- The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level.
- Physical schema tells how data is physically organized in database and what type of data is stored.
- Logical schema specifies the actual data to be stored based on the datatype of the fields and relationship among different records and fields.
- A database may also have several schemas at the view level, sometimes called **subschemas** that describe different views of the database.
- Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema.
- Any change in logical schema affects the view of the application.
- The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs.
- Application programs exhibit **physical data independence** and thus need not be rewritten if the physical schema changes.

Data Models:

- **Data model** is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.
- A data model provides a way to describe the design of a database at the physical, logical, and view levels.

The data models can be classified into four different categories:

Relational Model:

- The relational model uses a collection of tables to represent both data and the relationships among those data.
- Each table has multiple columns, and each column has a unique name.
- Tables are also known as **relations**.
- The relational model is an example of a record-based model.
- Each table contains records of a particular type.
- This is the most widely used commercial data model.

Entity-Relationship Model:

- The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and *relationships* among these objects.
- An entity is a "thing" or "object" in the real world that is distinguishable from other objects.
- It is the logical representation of data as objects.
- A set of attributes describe the entity.
  Example: student_id,student_name describe the student entity.

Object-Based Data Model:

- Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology.
- This led to the development of an object-oriented data model that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity.
- Here objects are data carrying its properties.
- The object-relational data model combines features of the object-oriented data model and relational data model.

Semistructured Data Model:

- The semistructured data model permits the specification of data where individual data items of the same type may have different sets of attributes.
- This is in contrast to the data models mentioned earlier, where everydata item of a particular type must have the same set of attributes.
- The **Extensible Markup Language (XML)** is widely used to represent semistructured data.
- This data model is useful for exchange of data between different systems. Two different DMBS can be converted to XML and data can beexchanged in this format. Later from XML data is imported to database.

History of DBMS:

- The development of database technology can be divided into three eras based on data model or structure: navigational, SQL/relational, and post-relational.
- The two main early navigational data models were the hierarchical model and the CODASYL model (network model). These were characterized by the use of pointers to follow relationships from onerecord to another.
- In 1960 Charles Bachman designed the Integrated Data Store (IDS) whichis the first DBMS based on network model.
- In late 1960's IBM developed Information Management System (IMS) based on hierarchical model.
- Later Edgar F. Codd who worked for IBM was unhappy with the lack of search engine inthese models.
- He insisted that application should search for data by content ratherthan by following links.
- In 1970 he developed relational data model. Relational systems dominated in all large-scale data processing applications till 1990's.
- Even till 2018 some of them were dominant like oracle, MySQl, SQLserver.
- In 1980's object-oriented model was developed. Post relational erastarted.
- In late 2000's XML based NoSQl and NEW SQL databases weredeveloped.
- NoSQL database provide mechanism for storage and reteival of data in means other than tables. It accommodates data as key-value, document, columnar and graph formats. Eg: mongoDB.

- New SQL database is a modern relational database developed as a combination of relational model with advancement in scalability and flexibility with types of data. Eg: voltDB, clustrixDB.

Database Architecture:

- Database architecture is a representation of DBMS design.
- A DBMS design depends on its architecture and architecture depends on how users are connected to database to get their request done.
- DBMS architecture allows dividing the system into individual components that can be independently modified.

Types of DBMS Architecture:
There are mainly three types of DBMS architecture:

- **1- Tier Architecture (Single Tier Architecture)**
- **2- Tier Architecture**
- **3 - Tier Architecture**

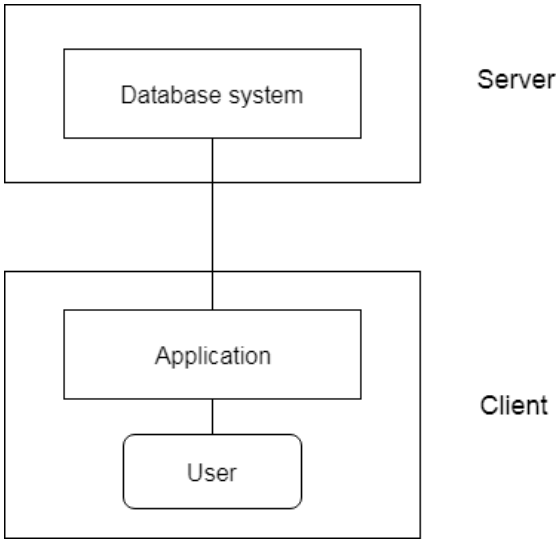
1-Tier Architecture:

- In 1- Tier Architecture the database is directly available to the user.
- The client, server, and Database all reside on the same machine.
- But such architecture is rarely used in production. It is used for local application development.

Single Tier Architecture

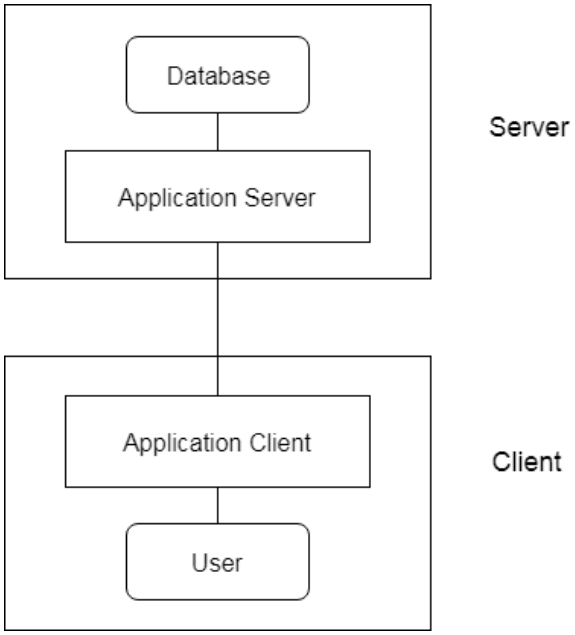
2-Tier Architecture:

- Basic client-server architecture where the application programs and user interface run on the client-side and data resides on the server side.
- We can have multiple clients connected to a single server.
- The client-side application establishes a connection with the server side communicates directly with the database at the server side.



3-Tier Architecture:

- 3-Tier Architecture in DBMS is the most popular client server architecture in DBMS in which the development and maintenance of functional processes, logic, data access, data storage, and user interface is done independently as separate modules.
- 3-Tier architecture contains a presentation layer, an application layer, and a database server.
- The client does not directly communicate with the server.
- The application on the client-side interacts with an application server which further communicates with the database system.
- This intermediate layer acts as a medium for the exchange of partially processed data between server and client.
- This type of architecture is used in the case of large web applications.



Three schema Architecture of Database:

- The three schema architecture is also called ANSI/SPARC architecture or three-level architecture.
- The three-schema architecture divides the database into three-level used to create a separation between the physical database and the user application.
- This architecture hides the details of physical storage from the user.
- The framework of this type of architecture includes an external schema, conceptual schema, internal schema and database itself.
- Mapping is used to transform the request and response between various database levels of architecture.
- In External / Conceptual mapping, the request is transformed from external level to conceptual schema.
- In Conceptual / Internal mapping, the request is transformed from the conceptual to internal level.

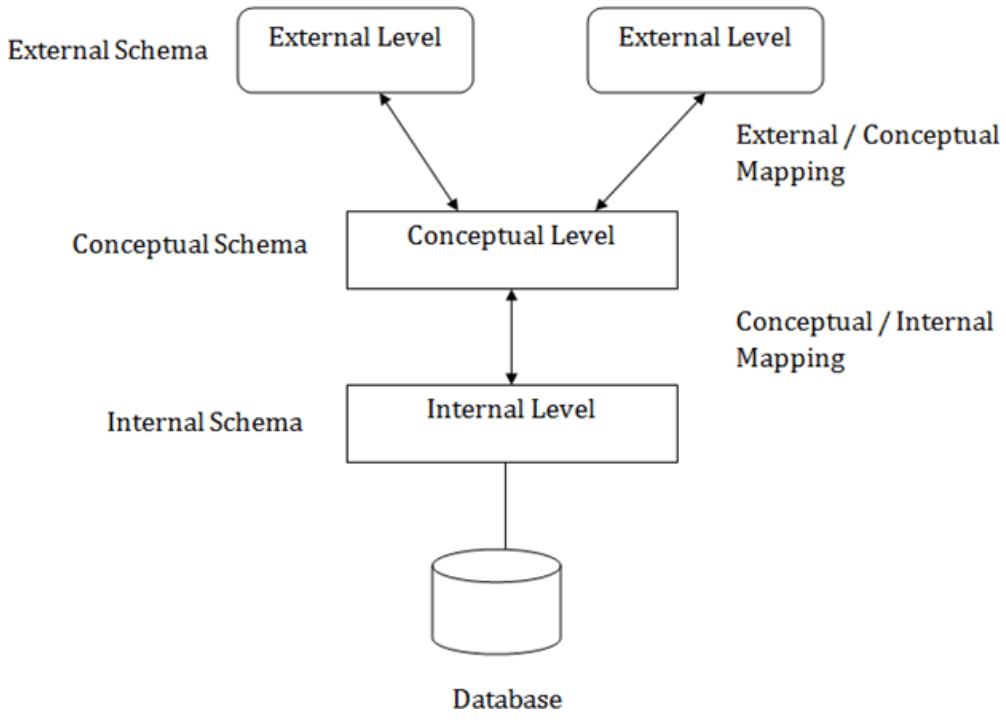


Fig: Three schema Architecture

Internal schema:

- The internal schema is also known as a physical schema.
- It uses the physical data model to describe complex low-level data structures in detail.

Activities of this field are:

- Storage space allocations.

    - For Example: B-Trees, Hashing etc.
  - Access paths.
    - For Example: Specification of primary and secondary keys, indexes,pointers and sequencing.
  - Data compression and encryption techniques.
  - Optimization of internal structures.
  - Representation of stored fields.

Conceptual schema:

- Conceptual schema is also known as logical schema.
- This schema describes the data datatypes and the relationship among the data stored in database.
- Internal details such as an implementation of the data structure are hidden from this schema.
- There is only one conceptual schema per database.

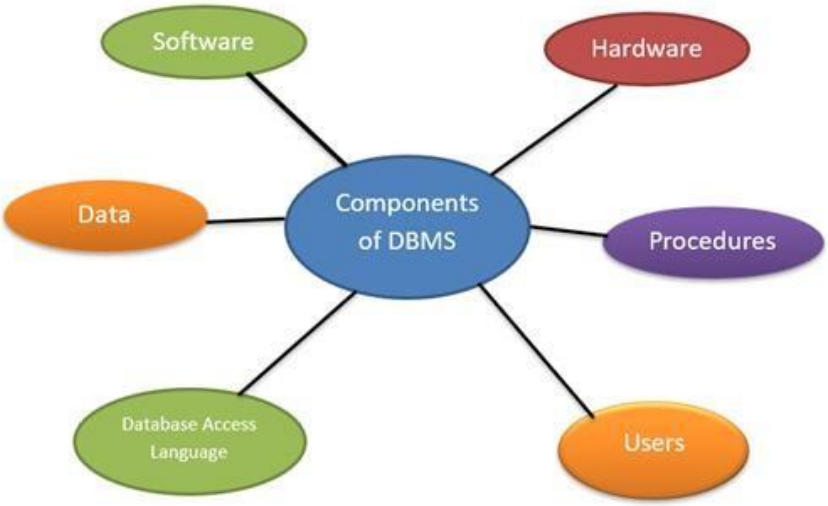
External schema:

- An external schema is also known as view schema.
- At the external level, a database contains several schemas that sometimes called as **subschema**.
- The subschema is used to describe the different view of the database.
- Each view schema describes the end user interaction with database systems.

Functional Components of Database:

A database consist of following functional components:

- Hardware
- Software
- Data
- Procedures
- Database Access Language
- Users



Hardware

- The hardware is the actual computer system used for keeping and accessing the database.
- Databases run on the range of machines from micro computers to mainframes.

Software

- It is the main component of DBMS.
- Software is a set of programs used to manage and control the database
- It includes the database software, operating system, network software used to share the data with other users, and the applications used to access the data.

Procedures:

- These are general instructions that are used for managing the DBMS and its applications.
- Procedures are generally used to take back up of the database and to change the structure of database etc.

Data:

- It is also the most important component of the database management system.
- The main task of DBMS is to process the data.
- Here, databases are defined, constructed, and then data is stored, retrieved, and updated to and from the databases.

Users:

- The users are the people who control and manage the databases and perform different types of operations on the databases in the *database management system*.

There are three types of user who play different roles in DBMS:

- Application Programmers
- Database Administrators
- End-Users

Application Programmers:

The users who write the application programs in programming languages to interact with databases are called Application Programmers.

Database Administrators (DBA):

A person who manages the overall DBMS is called a database administrator or simply DBA.

End-Users:

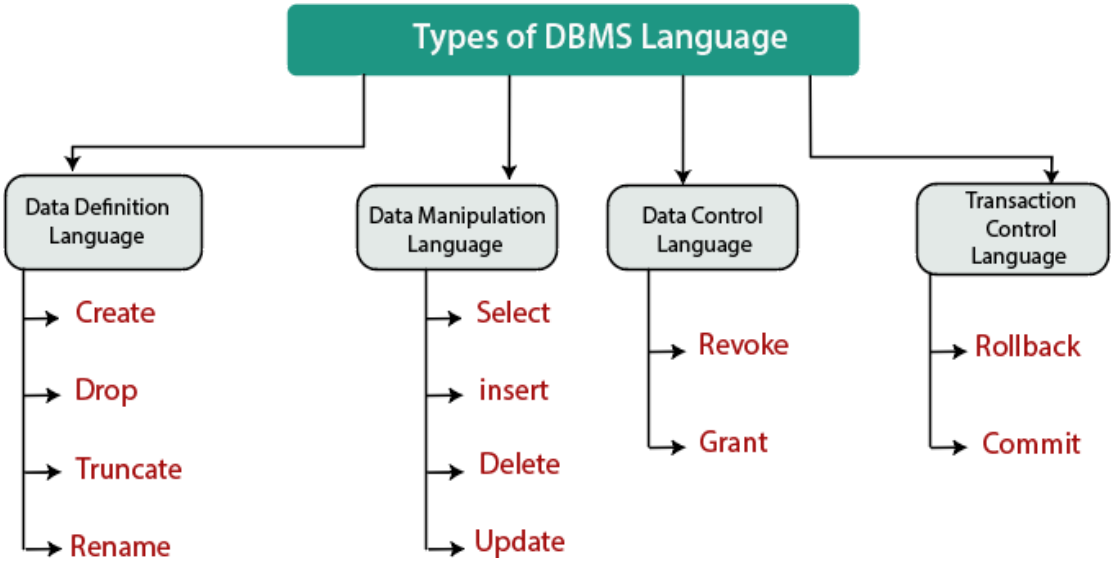
End-users are those who use the application program to interact with DBMS.

Database Languages:

User can access, update, delete, and store data or information in the database using **database languages**.

Following are different types of database languages:

1. Data Definition Language
2. Data Manipulation Language
3. Data Control Language
4. Transaction Control Language



Data Definition Language (DDL)

- Data Definition Language is used for defining the structure or schema of the database.
- It is also used for creating tables, indexes, applying constraints, etc. in the database.

Here are some tasks that come under DDL:

- **Create:** It is used to create objects in the database.
- **Alter:** It is used to alter the structure of the database.
- **Drop:** It is used to delete objects from the database.
- **Truncate:** It is used to remove all records from a table.
- **Rename:** It is used to rename an object.
- **Comment:** It is used to comment on the data dictionary.

Data Manipulation Language (DML)

- **D**ata **M**anipulation **L**anguage is used for accessing and manipulating datain a database.
- It handles user requests.

Here are some tasks that come under DML:

- **Select:** It is used to retrieve data from a database.
- **Insert:** It is used to insert data into a table.
- **Update:** It is used to update existing data within a table.
- **Delete:** It is used to delete all records from a table.
- **Merge:** It performs UPSERT operation, i.e., insert or update operations.

Data Control Language (DCL)

- Data Control Language is used to control privilege in Databases.
- To perform any operation in the database, such as for creating tables,sequences, or views, we need privileges.
- Privileges can be set for the system or for an object.

Tasks that come under DCL are:

- **Grant:** It is used to give user access privileges to a database.
- **Revoke:** It is used to take back permissions from the user.

Transaction Control Language (TCL)

- Transaction Control Language is used to run the changes made by theDML statement.
- It allows statements to be grouped into logical transactions.

Tasks that come under TCL are:

- **Commit:** It is used to save the transaction on the database.
- **Rollback:** It is used to restore the database to original since the last Commit.
- **Savepoint:** It used to identify a point in transaction to which one can laterrollback.

Database Users:

- Database users are people who interact with the database usingapplication and interfaces provided by the DBMS.
- Database users are divided into different types based on the way theyinteract with the database.

Types of users are:

Native/Naive Users:

These are the end users who use the existing application to interact with the database. For example, users logging into gmail using login id and password to access mails.

Application Programmer:

These are the software professionals who write application programs and user interface. They use tools like Rapid Application Development (RAD) for creatingforms, reports and UI with minimal efforts.

Sophisticated Users:

These are analysts who interact with the database using query languagelike SQL. They submit each query to a query processor whose function is to breakdown DML statements into instructions to the database.

Specialized Users:

- Users who write complex programs and specialized database applications that do not fit into the traditional data processing framework.
- Example, expert system, knowledge based system etc.

Database Administrators (DBA)

Database Administrator is a person or a group of person who are responsible for managing all the activities related to database system. DBA has central control over the DBMS.

UNIT 2

DATABASE DESIGN

Entity Relationship Model (ER-Model):

Objects: The ER model describes data objects as *entities*, *relationships*, and *attributes*. The ERmodel is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. The ER-Model also has an associated diagrammatic representation, the E-R diagram.

Entity:

- The basic concept that the ER model represents is an **entity**, which is a *thing* or *object* in the real world with an independent existence.
- For example each person in a university is an entity. An entity has a set of properties, and the values for some set of properties may uniquely identify an entity. For instance, a person may have a *person id* property whose value uniquely identifies that person.
- An **entity set** is a set of entities of the same type that share the same properties, or attributes. The set of all people who are instructors at a given university, for example, can be defined as the entity set *instructor*. Similarly, the entity set *student* might represent the set of all students in the university.

Attributes:

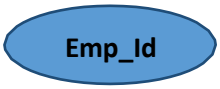
- An entity is represented by a set of **attributes**. Attributes are descriptive properties possessed by each member of an entity set.
- Each entity has a **value** for each of its attributes. For instance, a particular Employee entity may have the value 12121 for *ID*, the value Kamalfor *name*, the value Finance for *dept name*, and the value 90000 for *salary*.

There are five such types of attributes: Simple, Composite, Single-valued,Multi-valued, and Derived attribute.

1. Simple attribute :

An attribute that cannot be further subdivided into components is a simple attribute.

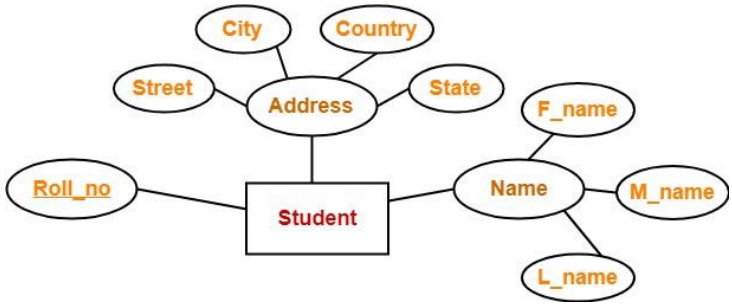
Example: The roll number of a student, the id number of an employee.



2. Composite attribute :

An attribute that can be split into components is a composite attribute.

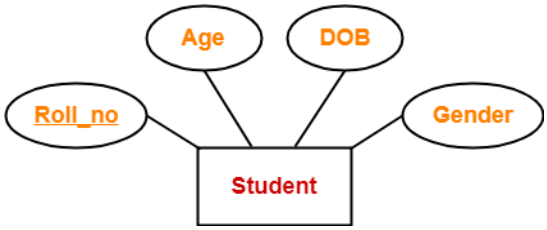
Example: The address can be further split into house number, street number, city, state, country, and pin code, the name can also be split into first name middle name, and last name.



3. Single-valued attribute :

The attribute which takes up only a single value for each entity instance is a single-valued attribute.

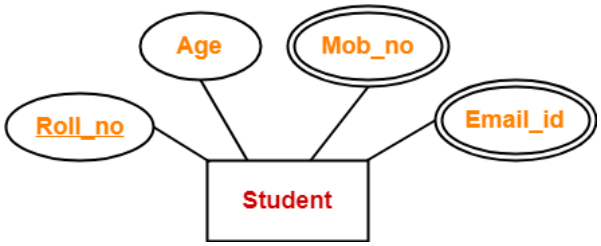
Example: The gender of a student.



4. Multi-valued attribute :

The attribute which takes up more than a single value for each entity instance is a multi-valued attribute.

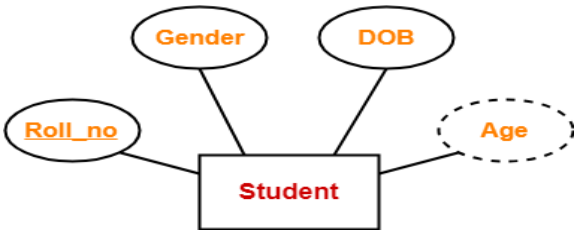
Example: Phone number of a student: Landline and mobile, email id.



5. Derived attribute :

An attribute that can be derived from other attributes is derived attributes.

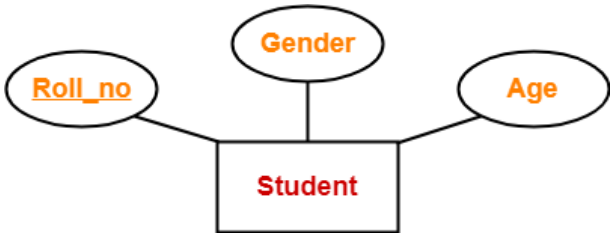
Example: Total and average marks of a student, age can be derived from DOB.



Key Attribute:

It is the attribute which can identify an entity uniquely in an entity set.

Example: "Roll_no" is a key attribute as it can identify any student uniquely.



Domain of Attributes

The set of possible values that an attribute can take is called the domain of the attribute.

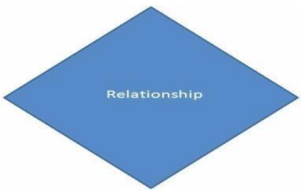
For example, age attribute can take only positive integer values, the attribute day may take any value from the set {Monday, Tuesday ... Friday}. Hence this set canbe termed as the domain of the attribute day.

RELATIONSHIPS:

Associations between entities are called relationships

Example: An employee works for an organization. Here "works for" is a relation between the entities employee and organization.

In ER modeling, notation for relationship is given below.



A **relationship set** is a set of relationships of the same type. We define the relationship set "*works for*" to denote the association between employee and

organization.

- The association between entity sets is referred to as participation; that is, the entity sets *E*1*, E*2*, . . . , En* **participate** in relationship set *R*. A **relationship instance** in an E-R schema represents an association between the named entities in the real-world enterprise that is being modeled.
- The function that an entity plays in a relationship is called that entity's **role**. Since entity sets participating in a relationship set are generally distinct, roles may be different if the same entity set participates in a relationship set more than once.

Degree of a Relationship:

The **degree** of a relationship type is the number of participating entity types. The n-ary relationship is the general form for degree n. Special cases are unary, binary, and ternary, where the degree is 1, 2, and 3, respectively.

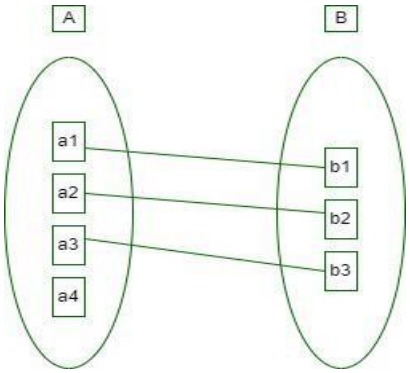
Example for unary relationship: An employee is a manager of another employee
Example for binary relationship: An employee works-for department.
Example for ternary relationship: customer purchase item from a shop keeper

Mapping cardinalities: express the number of entities to which another entity can be associated via a relationship set.

Mapping cardinalities are most useful in describing binary relationship sets, although they can contribute to the description of relationship sets that involve more than two entity sets.

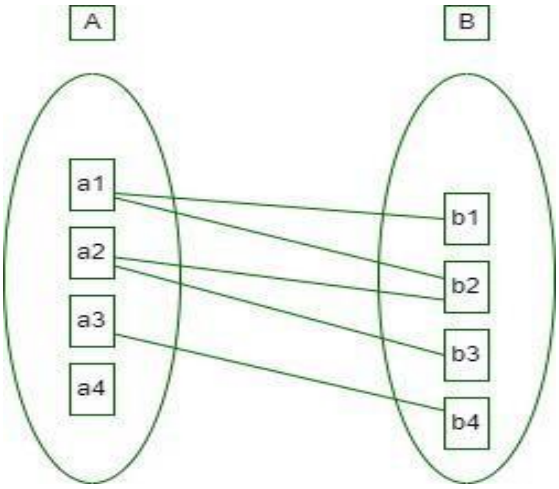
For a binary relationship set *R* between entity sets *A* and *B*, the mapping cardinality must be one of the following:

• **One-to-one**: An entity in *A* is associated with *at most* one entity in *B*, and an entity in *B* is associated with *at most* one entity in *A*.



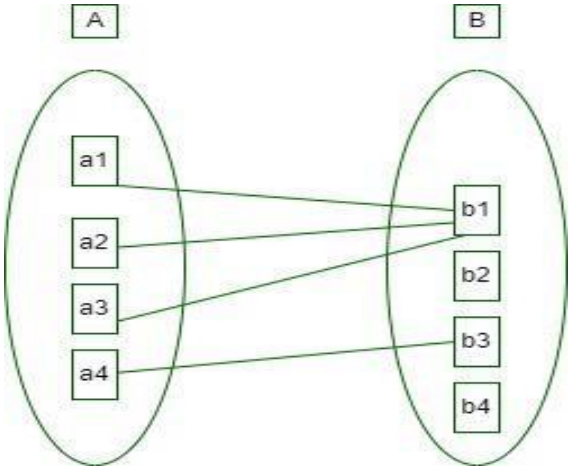
Example:
In a particular hospital, the surgeon department has one head of department.They both serve one-to-one relationships.

• **One-to-many**: An entity in *A* is associated with any number (zero or more) of entities in *B*. An entity in *B*, however, can be associated with *at most* one entity in *A*.



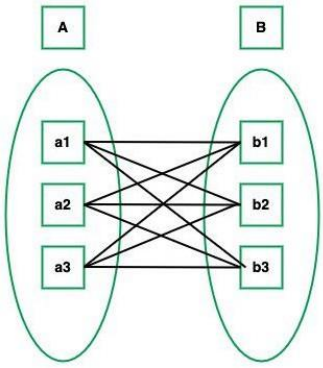
Example:
In a particular hospital, the surgeon department has multiple doctors. Theyserve one-to-many relationships

• **Many-to-one**: An entity in *A* is associated with *at most* one entity in *B*. An entity in *B*, however, can be associated with any number (zero or more) of entities in *A*.



Example:
In a particular hospital, multiple surgeries are done by a single surgeon. Such a type of relationship is known as a many-to-many relationship.

• **Many-to-many**: An entity in *A* is associated with any number (zero or more) of entities in *B*, and an entity in *B* is associated with any number (zero or more) of entities in *A*.



Example:
In a particular company, multiple people work on multiple projects. They serve many-to-many relationships.


Empoyees
M
Works on
M
Multiple projects

KEYS:

- Keys are a set of attributes whose values can be used to uniquely identify an individual entity in an entity set.
- Key is an important constraint on an entity.
- Eg: ID, Aadhar no, PAN card no etc.

Types of Keys:
Super key: is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation.

For example, the *ID* attribute of the relation *instructor* is sufficient to distinguish one instructor tuple from another. The combination of *ID* and *name* is a superkey for the relation *instructor*.
{ID, name}, {ID, name, address} examples of super key.

Candidate Key: The minimal attribute super keys are candidate keys.

- These are keys which cannot be further subdivided into keys to uniquely identify an entity or a record in a relation.
- If any one of the attribute is removed from this set then we cannot uniquelyidentify a record.
- Eg: {name, address}, {name, contact no}, {Id} these are candidate keys whose values cannot be same for two different records at any time.
-

Primary Key: It is that candidate key whose value alone can be used to uniquely identify an entity.

- These are simple attributes which do not allow duplicate values.
- They cannot take null values.
- Eg: Id
- We use the convention that the attributes that form the primary key of a relation are underlined.
-

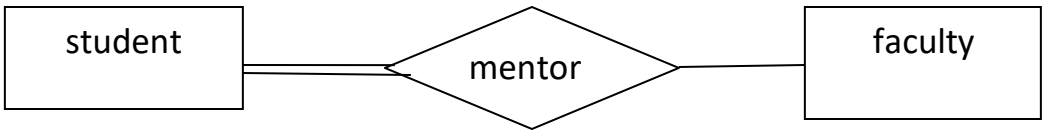
Secondary/Alternate key: These are the candidate keys that are not chosen as primary key. These are used for accessing records.
Eg: {name, address}, {name, contact no}
Foreign Key: It is the attribute of an entity which is a primary key in the related entity.

- It is used to establish mapping between two or more entities.
- Eg: In an **employee** relation **deptno** refers to the department an employee works. But **deptno** is a **primary key** in **department** relation. Hence **deptno** is **Foreign key** in **employee** relation.

Design Constraints in ER Model

Participation Constraints:

- The participation of an entity set *E* in a relationship set *R* is said to be **total** if every entity in *E* participates in at least one relationship in *R*.
- If only some entities in *E* participate in relationships in *R*, the participation of entity set *E* in relationship *R* is said to be **partial**.
- In a one to one relationship, participation of both the sets is total whereas in other relationships it is partial from one of the sets.
- For example, we expect every ***student*** entity to be related to at least one **faculty** through the ***mentor*** relationship. Therefore the participation of ***student*** in the relationship set ***mentor*** is **total**. In contrast, a faculty need not mentor any student. Hence, it is possible that only some of the ***faculty*** entities are related to the ***student*** entity set through the ***mentor*** relationship, and the participation of ***faculty*** in the ***mentor*** relationship set is therefore **partial**.

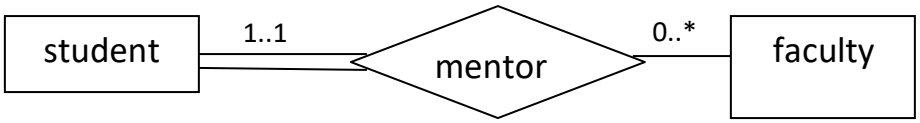


Total Participation is indicated by double line.

Cardinality Constraint:

- We can show minimum and maximum cardinality while showing the relationship between two entities.
- It indicates the minimum and maximum numbers of entities from an entity set that are associated in a relationship.

- Eg: In mentor relation, a student must have exactly one mentor, so here min. and max. is 1 whereas a faculty can be a mentor to 0 or more students.
- Different notations are available for representing this.



Key Constraints:

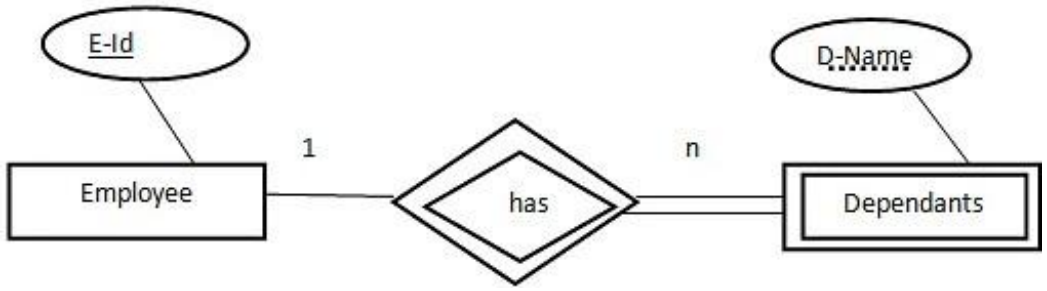
- The constraint for keys is that no two entities can have same values for a key.
- Primary key cannot be null.

Strong Entity: Entities which have sufficient key attributes are called Strong entities. Strong entities have primary keys of their own.

Weak Entity: Entities which don't have sufficient key attributes are called weak entities.

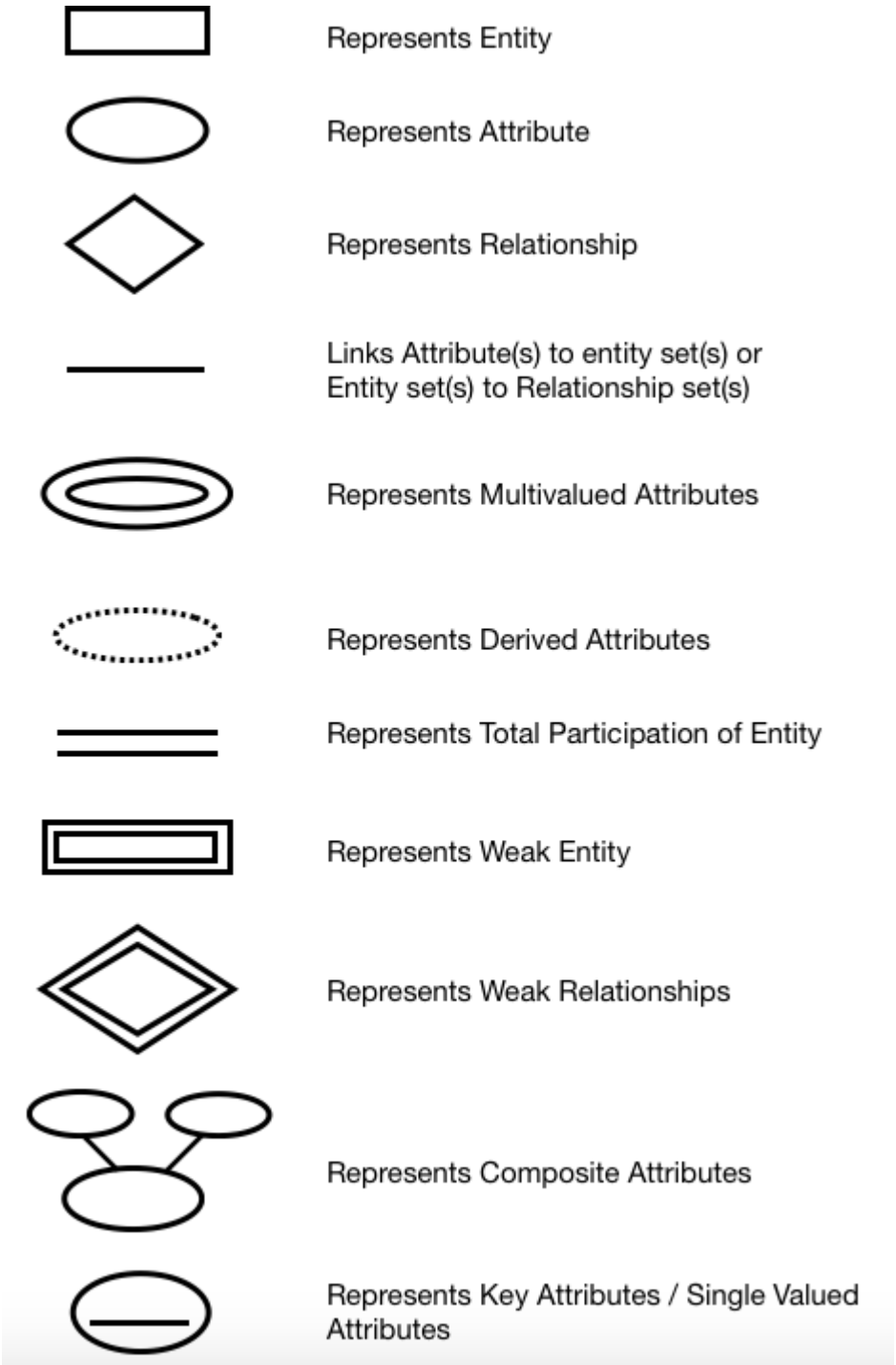
- Entities belonging to a weak entity type are identified by the **identifying** or **owner entity type** with which it is associated.
- The relationship associating a weak entity type to its owner is called the **identifying relationship.**
- A weak entity type always has a *total participation constraint* (existence dependency) with respect to its identifying relationship because a weak entity cannot be identified without an owner entity.
- The identifying relationship is many-to-one from the weak entity set to the identifying entity set, and the participation of the weak entity set in the relationship is total.
- A weak entity type normally has a **partial key**, also known a discriminator, which is the attribute that can uniquely identify weak entities that are *related to the same owner entity*.
- Eg: The entity type DEPENDENT, related to EMPLOYEE is a weak entity. {dependent_name, dependent_address} of the weak entity can be taken as partial key.
- The primary key of a weak entity set is formed by the primary key of the identifying entity set, plus the weak entity set's discriminator.

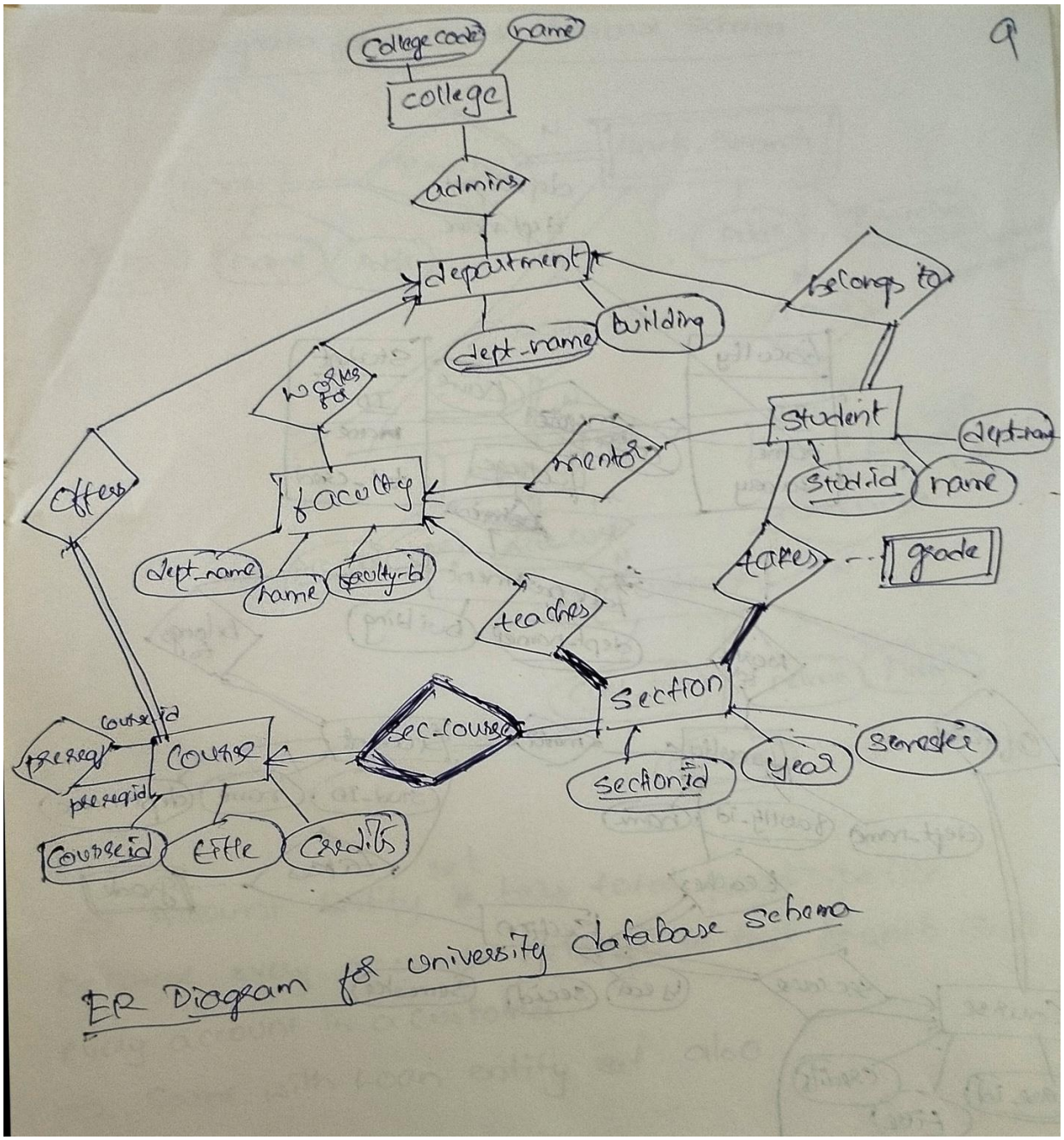
- In the above case {emp_id, dependent_name, dependent_address} is the primary key of Dependent entity.



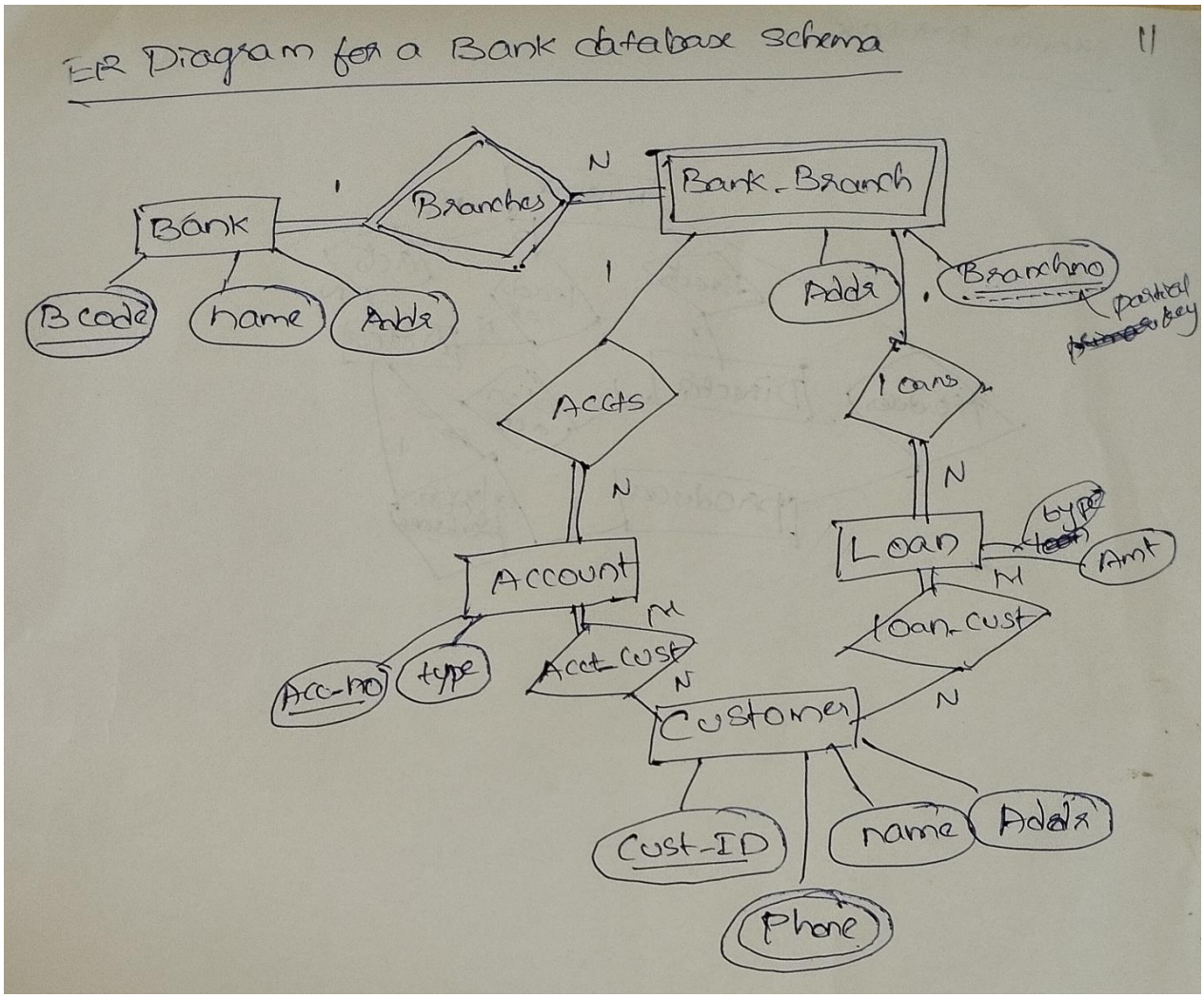
	Strong Entity	Weak Entity
1	Strong entity has a primary key.	Weak entity has a partial key.
2	Strong entity is independent	Weak entity is dependent on a strongentity
3	Strong entity indicated by a singlerectangle.	Strong entity indicated by a doublerectangle.
4	Two strong entity's relationship isindicated by a single diamond.	One strong and one weak entity isindicated by a double diamond.
5	Strong entity may be or may notparticipate in relationships.	Weak entity always participates inrelationships.
6	In strong entity connecting line is asingle line	In weak entity connecting line is a doubleline

ER Diagram Notations:





ER Diagram for university database schema



Relational Model:

- It is the primary data model for commercial data processing applications compared to earlier models.
- It is simple and easy to use.
- The relational model in DBMS is an abstract model used to organize and manage the data stored in a database.
- It was designed in 1969 by scientist Edgar F. Codd.
- Relational database stores data in the form of relations.
- Relations are represented as a collection of data in the form of tables with rows and columns.

Relational Model Concepts:

- **Relation :** Two-dimensional table used to store a collection of data elements.
- **Tuple :** Row of the relation, depicting a real-world entity.
- **Attribute/Field :** Column of the relation, depicting properties that define the relation.
- **Attribute Domain :** Set of pre-defined atomic values that an attribute can take i.e., it describes the legal values that an attribute can take.
- **Degree :** It is the total number of attributes present in the relation.
- **Cardinality :** It specifies the number of entities involved in the relation i.e., it is the total number of rows present in the relation.
- **Relational Schema :** It is the logical blueprint of the relation i.e., it describes the design and the structure of the relation. It contains the table name, its attributes, and their types:
  TABLE_NAME(ATTRIBUTE_1 TYPE_1, ATTRIBUTE_2 TYPE_2, ...)

For our Student relation example, the relational schema will be:
STUDENT(ROLL_NUMBER INTEGER, NAME VARCHAR(20),…)

- **Relational Instance :** It is the collection of records present in the relation at a given time.
- **Relation Key :** It is an attribute or a group of attributes that can be used to uniquely identify an entity in a table or to determine the relationship between two tables.

Constraints in Relational Model:

Relational models make use of some rules to ensure the accuracy and accessibility of the data. These rules or constraints are known as **Relational Integrity Constraints**. These constraints are checked before performing any operation like insertion, deletion, or updation on the data present in a relational database.
These constraints include:

- **Domain Constraint :** It specifies that every attribute is bound to have a value that lies inside a specific range of values. It is implemented with the help of the Attribute Domain concept.
  - If we set a constraint on an attribute like age>0 then its age attribute should not accept negative values.

- **Key Constraint :** It states that every relation must contain an attribute or a set of attributes (Primary Key) that can uniquely identify a tuple in that relation. This key can never be NULL or contain the same value for two different tuples.
- **Referential Integrity Constraint :** It is defined between two inter-related tables. It works on **foreign key** concept. It states that if a given relation refers to a key attribute of a different or same table, then that key must exist in the given relation.

Advantages of using the relational model:

The advantages and reasons due to which the relational model in DBMS is widely accepted as a standard are:

- **Simple and Easy To Use -** Storing data in tables is much easier to understand and implement as compared to other storage techniques.
- **Manageability -** Because of the independent nature of each relation in a relational database, it is easy to manipulate and manage. This improves the performance of the database.
- **Query capability -** With the introduction of relational algebra, relational databases provide easy access to data via high-level query language like SQL.
- **Data integrity -** With the introduction and implementation of relational constraints, the relational model can maintain data integrity in the database.

Disadvantages of using the relational model

The main disadvantages of relational model in DBMS occur while dealing with a huge amount of data as:

- The performance of the relational model depends upon the number of relations present in the database.
- Hence, as the number of tables increases, the requirement of physical memory increases.
- The structure becomes complex and there is a decrease in the response time for the queries.
- Because of all these factors, the cost of implementing a relational database increase.

UNIT 3

STRUCTURED QUERY LANGUAGE

What is SQL?

- SQL is Structured Query Language, which is a database language designed for the retrieval and management of data in a relational database.
- All the RDBMS systems like MySQL, MS Access, Oracle, Sybase, Postgres, and SQL Server use SQL as their standard database language.

Why to Use SQL?

SQL provides an interface to a relational database.

Here, are important reasons for using SQL

- It helps users to access data in the RDBMS system.
- It helps us to describe the data.
- It allows us to define the data in a database and manipulate that specific data.
- With the help of SQL commands in DBMS, we can create and drop databases and tables.
- SQL offers us to use the function in a database, create a view, and stored procedure.
- We can set permissions on tables, procedures, and views.

History of SQL

"A Relational Model of Data for Large Shared Data Banks" was a paper which was published by the great computer scientist "E.F. Codd" in 1970.

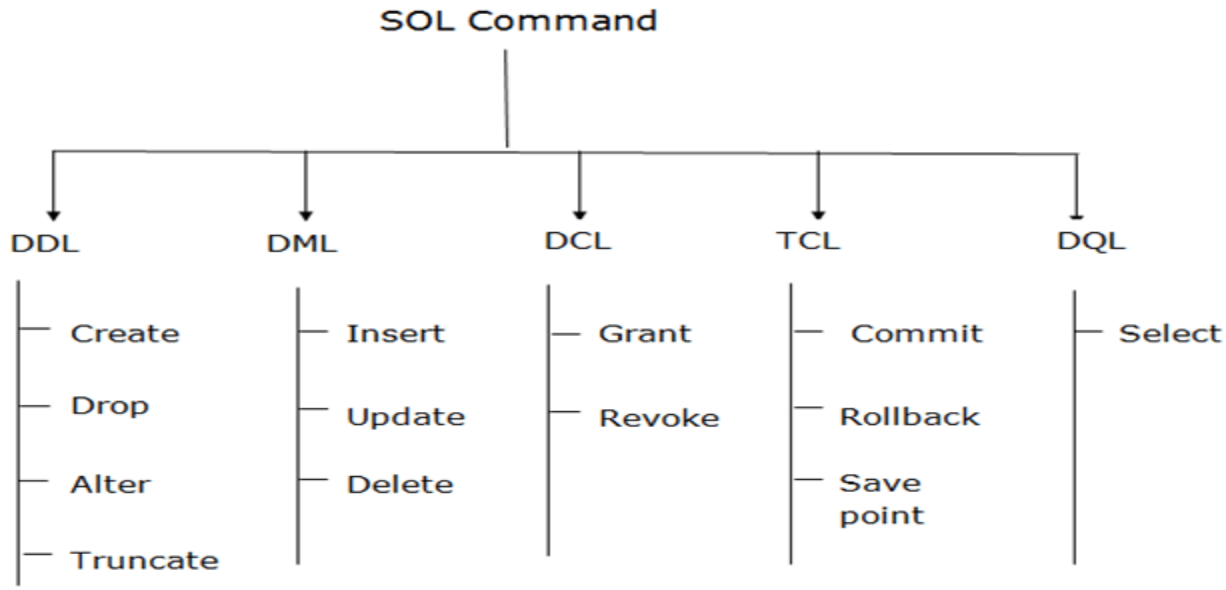
The IBM researchers Raymond Boyce and Donald Chamberlin originally developed the SEQUEL (Structured English Query Language) after learning from the paper given by E.F. Codd. They both developed the SQL at the San Jose Research laboratory of IBM Corporation in 1970. In 1979, Relational Software, Inc. (now Oracle) introduced the first commercially available implementation of SQL.

SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987.[11] Since then, the standard has been revised to include a larger set of features. Despite the existence of standards, most SQL code requires at least some changes before being ported to different database systems. New versions of the standard were published and most recently, 2016.

Types of SQL

Here are five types of widely used SQL queries.

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



All operations performed on the information in a database are run using SQL **statements**. A SQL statement consists of identifiers, parameters, variables, names, data types, and SQL **reserved words**.

What is DDL?

Definition: The Language used to define the database structure or schema is called "Data Definition Language".

- The Commands (or) statements used to define the structure of database are:

1. CREATE
2. ALTER
3. DROP
4. TRUNCATE
5. RENAME

1. CREATE

Create command can be used to create

(i) Databases

(ii) Tables and

(iii) Views.

(i) Creating Database

Syntax:

```
create  database database name;
```

Ex: create database MRCET_ITA;

(ii) Creating Table

Syntax:

```
Create table tablename(Columnname1 Datatype,
        Columnname2 Datatype,……., Columnnamen
        Datatype);
```

Ex: create table Student(SRno integer(5),

Snamevarchar(20),

Address varchar(15));

2. ALTER Command

- The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.
- The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

1. ALTER TABLE - ADD Column

To add a column in a table, use the following syntax:

```
ALTER TABLE table_name
ADD column_namedatatype;
```

Ex: The following SQL adds an "Email" column to the "Customers" table:

```
ALTERTABLE Customers
        ADD Email varchar(255);
```

ALTER TABLE - DROP COLUMN

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

Syntax:

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

Ex: The following SQL deletes the "Email" column from the "Customers" table:

```
ALTERTABLE Customers
    DROP COLUMN Email;
```

ALTER TABLE - ALTER/MODIFY COLUMN

To change the data type of a column in a table, use the following syntax:

SQL Server / MS Access:

```
ALTER TABLE table_name
ALTER COLUMN column_namedatatype;
```

```
Ex: ALTER TABLE supplier
    ALTER COLUMN supplier_nameVARCHAR(100) NOT NULL;
```

My SQL / Oracle (prior version 10G):

```
ALTER TABLE table_name
MODIFY COLUMN column_namedatatype;
```

Example 1: Modifying single Column

```
ALTER TABLE supplier
 MODIFY supplier_namechar(100) NOT NULL;
```

Example 2: Modifying Multiple Columns

```
ALTER TABLE supplier
 MODIFY supplier_nameVARCHAR(100) NOT NULL,
 MODIFY city VARCHAR(75);
```

Oracle 10G and later:

```
ALTER TABLE table_name
MODIFY column_namedatatype;
```

3.Drop Command

Syntax

To drop a column in an existing table, the SQL ALTER TABLE syntax is:

```
ALTER TABLE table_name
 DROP COLUMN column_name;
```

Example

Let's look at an example that drops (ie: deletes) a column from a table.

For example:

```
ALTER TABLE supplier
 DROP COLUMN supplier_name;
```

This SQL ALTER TABLE example will drop the column called ***supplier_name*** from the table called ***supplier***.

TRUNCATE:

This command used to delete all the rows from the table and free the space containing the table.

Syntax:

```
TRUNCATE TABLE table_name;
```

Example:

```
TRUNCATE table students;
```

What is Data Manipulation Language?

Data Manipulation Language (DML) allows user to modify the database instance by inserting, modifying, and deleting its data. It is responsible for performing all types data modification in a database.

There are three basic constructs which allow database program and user to enter data and information are:

Here are some important DML commands in SQL:

- INSERT
- UPDATE
- DELETE

INSERT: This statement is a SQL query. This command is used to insert data into the row of a table.

Syntax:

```
INSERT INTO TABLE_NAME  (col1, col2, col3,.... col N)
VALUES (value1, value2, value3,.... valueN);
Or
INSERT INTO TABLE_NAME
VALUES (value1, value2, value3,.... valueN);
```

For example:

```
INSERT INTO students (RollNo, FIrstName, LastName) VALUES ('60', 'Tom', Erichsen');
```

UPDATE:

This command is used to update or modify the value of a column in the table.

Syntax:

```
UPDATE table_name SET [column_name1= value1,...column_nameN = valueN] [WHERE CONDITION]
```

For example:

```
UPDATE students
SET FirstName = 'Jhon', LastName= 'Wick'
WHERE StudID = 3;
```

DELETE:

This command is used to remove one or more rows from a table.

Syntax:

```
DELETE FROM table_name [WHERE condition];
```

For example:

```
DELETE FROM students
WHERE FirstName = 'Jhon';
```

What is DCL?

DCL (Data Control Language) includes commands like GRANT and REVOKE, which are useful to give "rights & permissions." Other permission controls parameters of the database system.

Examples of DCL commands:

Commands that come under DCL:

- Grant
- Revoke

Grant:

This command is use to give user access privileges to a database.

Syntax:

```
GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;
```

For example:

```
GRANT SELECT ON Users TO'Tom'@'localhost;
```

Revoke:

It is useful to back permissions from the user.

Syntax:

```
REVOKE privilege_nameONobject_nameFROM {user_name |PUBLIC |role_name}
```

For example:

```
REVOKE SELECT, UPDATE ON student FROM BCA, MCA;
```

What is TCL?

Transaction control language or TCL commands deal with the transaction within the database.

Commit: This command is used to save all the transactions to the database.

Syntax:

```
Commit;
```

For example:

```
DELETE FROM Students
WHERE RollNo =25;
COMMIT;
```

Rollback

Rollback command allows you to undo transactions that have not already been saved to the database.

Syntax:

```
ROLLBACK;
```

Example:

```
DELETE FROM Students
WHERE RollNo =25;
```

SAVEPOINT

This command helps you to sets a savepoint within a transaction.

Syntax:

```
SAVEPOINT SAVEPOINT_NAME;
```

Example:

```
SAVEPOINT RollNo;
```

What is DQL?

Data Query Language (DQL) is used to fetch the data from the database. It uses only one command:

SELECT:

This command helps you to select the attribute based on the condition described by the WHERE clause.

Syntax:

```
SELECT expressions
FROM TABLES
WHERE conditions;
```

For example:

```
SELECT FirstName
FROM Student
```

```
WHERE RollNo> 15;
```

TCL Commands

TCL Commands in SQL- Transaction Control Language Examples: Transaction Control Language can be defined as the portion of a database language used for maintaining consistency of the database and managing transactions in database. A set of SQL statements that are co-related logically and executed on the data stored in the table is known as transaction. In this tutorial, you will learn different TCL Commands in SQL with examples and differences between them.

1. Commit Command
2. Rollback Command
3. Savepoint Command

TCL Commands in SQL- Transaction Control Language Examples

The modifications made by the DML commands are managed by using TCL commands. Additionally, it makes the statements to grouped together into logical transactions.

TCL Commands

There are three commands that come under the TCL:

1. **Commit**

The main use of Commit command is to make the transaction permanent. If there is a need for any transaction to be done in the database that transaction permanent through commit command.

Syntax:

COMMIT;

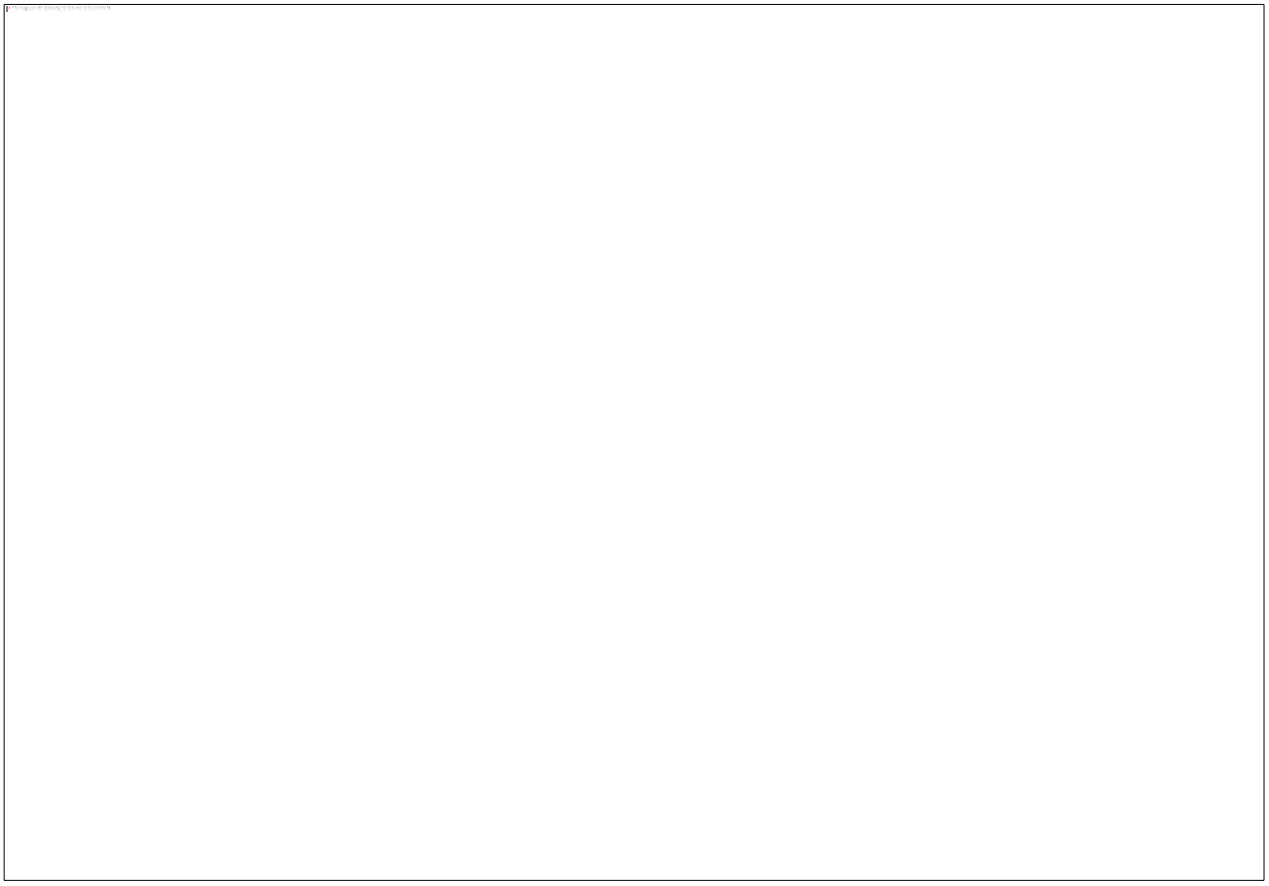
For Example

UPDATE STUDENT SET STUDENT_NAME = 'Maria' WHERE STUDENT_NAME = 'Meena';

COMMIT;

- By using the above set of instructions, you can update the wrong student name by the correct one and save it permanently in the database. The update transaction gets completed when commit is used. If commit is not used, then there will be lock on 'Meena' record till the rollback or commit is issued.

- Now have a look at the below diagram where 'Meena' is updated and there is a lock on her record. The updated value is permanently saved in the database after the use of commit and lock is released.



2. Rollback

- Using this command, the database can be restored to the last committed state.
- Additionally, it is also used with savepoint command for jumping to a savepoint in a transaction.

Syntax:

Rollback to savepoint-name;

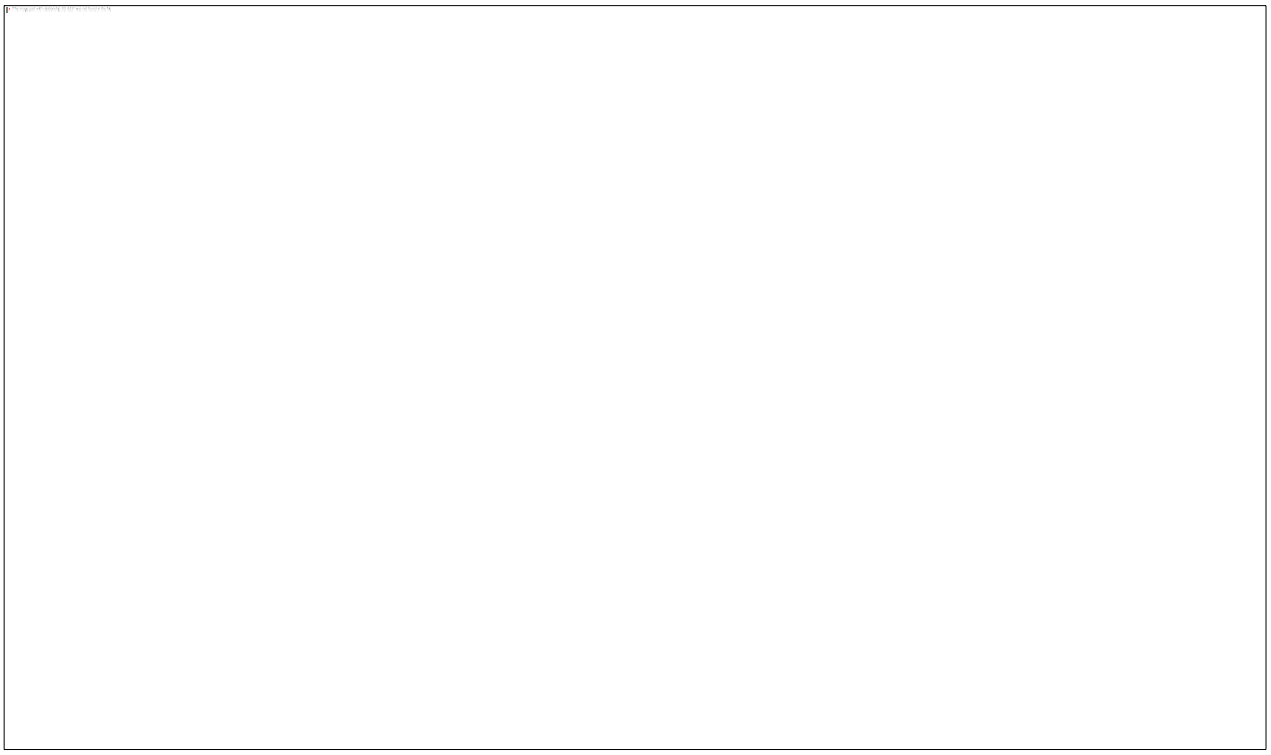
For example

UPDATE STUDENT SET STUDENT_NAME = 'Manish' WHERE STUDENT_NAME = 'Meena'; ROLLBACK;

- This command is used when the user realizes that he/she has updated the wrong information after the student name and wants to undo this update.

- The users can issues ROLLBACK command and then undo the update.

Have a look at the below tables to know better about the implementation of this command.



3. **Savepoint**

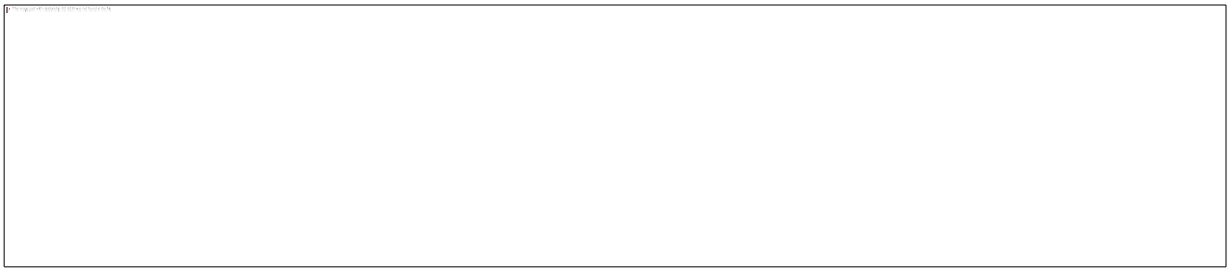
The main use of the Savepoint command is to save a transaction temporarily. This way users can rollback to the point whenever it is needed.

The general syntax for the savepoint command is mentioned below:

savepointsavepoint-name;

For Example

Following is the table of a school class



Use some SQL queries on the above table and then watch the results

INSERT into CLASS VALUES (101, ‘Rahul);

Commit;

UPDATE CLASS SET NAME= ‘Tyler’ where id= 101;

SAVEPOINT A;

INSERT INTO CLASS VALUES (102, ‘Zack’);

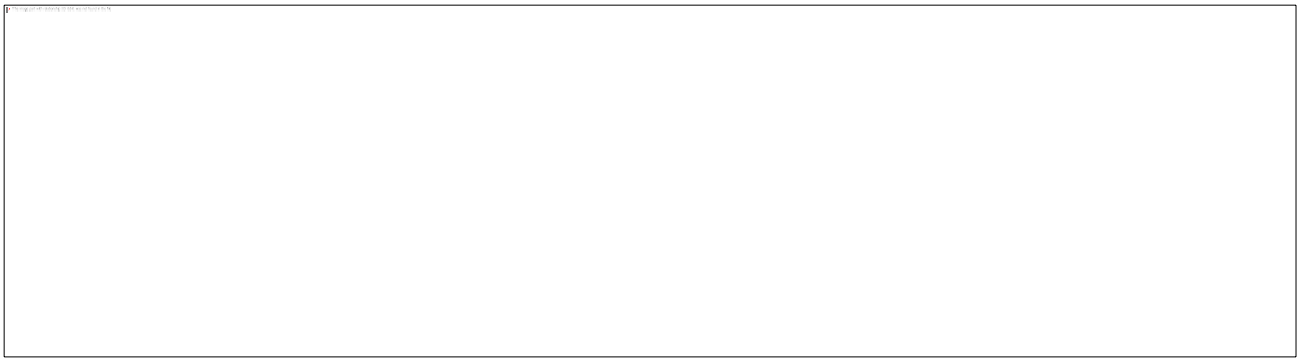
Savepoint B;

INSERT INTO CLASS VALUES (103, ‘Bruno’)

Savepoint C;

Select * from Class;

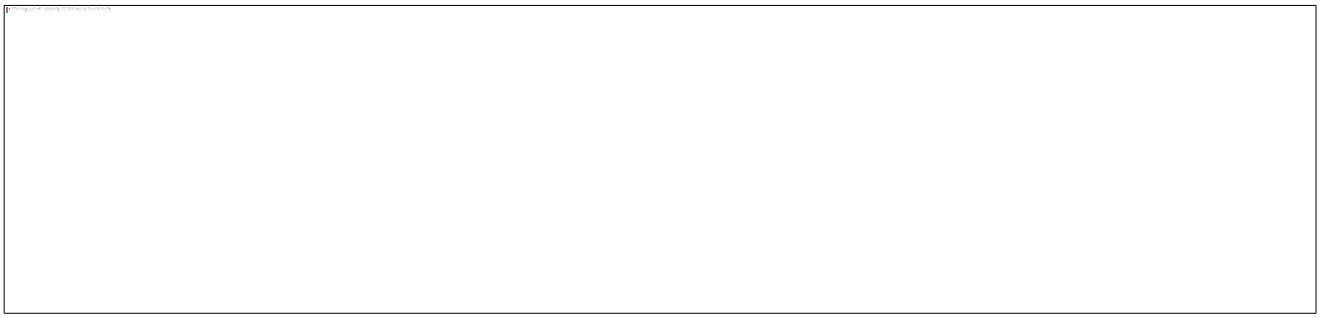
The result will look like



Now rollback to savepoint B

Rollback to B;

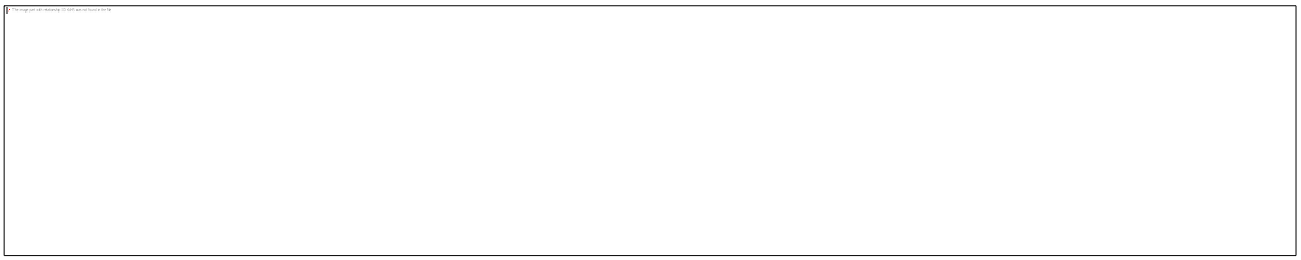
SELECT * from Class;



Now rollback to savepoint A

rollback to A;

SELECT * from class;



Difference between rollback, commit and savepointtcl commands in SQL.

	Rollback	Commit	Savepoint

1.	Database can be restored to the last committed state	Saves modification made by DML Commands and it permanently saves the transaction.	It saves the transaction temporarily.
2.	Syntax- ROLLBACK [To SAVEPOINT_NAME];	Syntax- COMMIT;	Syntax- SAVEPOINT [savepoint_name;]
3.	Example- ROLLBACK Insert3;	Example- SQL> COMMIT;	Example- SAVEPOINT table_create;

```
START TRANSACTION;
savepoint a;
update t1 set n1=18 where n1=13;
rollbackto a;
```

In relational database the data is stored as well as retrieved in the form of relations (tables). Table 1 shows the relational database with only one relation called **STUDENT** which stores **ROLL_NO**, **NAME**, **ADDRESS**, **PHONE** and **AGE** of students.

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20
4	SURESH	DELHI	9156768971	18

These are some important terminologies that are used in terms of relation.

Attribute: Attributes are the properties that define a relation. e.g.; **ROLL_NO, NAME** etc.

Tuple: Each row in the relation is known as tuple. The above relation contains 4 tuples, one of which is shown as:

1 RAM DELHI 9455123451 18

Degree: The number of attributes in the relation is known as 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: Column represents the set of values for a particular attribute. The column ROLL_NO isextracted from relation STUDENT.

ROLL_NO

1

2

3

4

SQL Set Operations

Set operations allow the results of multiple queries to be combined into a single result set.

The **Set Operators** 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.

Set operators include `UNION`, `INTERSECT`, and `EXCEPT`.

UNION

In SQL the UNION clause combines the results of two SQL queries into a single table of all matching rows. The two queries must result in the same number of columns and compatible data types in order to unite. Any duplicate records are automatically removed unless UNION ALL is used.

Syntax of UNION:

SELECT column1, column2.... , columnN FROM table_Name1 [WHERE conditions]

UNION

SELECT column1, column2.... , columnN FROM table_Name2 [WHERE conditions];

A simple example would be a database having tables sales2005 and sales2006 that have identical structures but are separated because of performance considerations. A UNION query could combine results from both tables.

Note that UNION ALL does not guarantee the order of rows. Rows from the second operand may appear before, after, or mixed with rows from the first operand. In situations where a specific order is desired, ORDER BY must be used.

Note that UNION ALL may be much faster than plain UNION.

sales2005

person	amount
Joe	1000
Alex	2000
Bob	5000

sales2006

person	amount
Joe	2000
Alex	2000
Zach	35000

Executing this statement:

SELECT * FROM sales2005UNIONSELECT * FROM sales2006;

yields this result set, though the order of the rows can vary because no ORDER BY clause was supplied:

person	amount
Joe	1000
Alex	2000
Bob	5000
Joe	2000
Zach	35000

UNION ALL gives different results, because it will not eliminate duplicates. Executing this statement:

SELECT * FROM sales2005UNION ALLSELECT * FROM sales2006;

would give these results, again allowing variance for the lack of an ORDER BY statement:

person	amount
Joe	1000
Joe	2000
Alex	2000
Alex	2000
Bob	5000
Zach	35000

INTERSECT

The SQL INTERSECT operator takes the results of two queries and returns only rows that appear in both result sets. For purposes of duplicate removal the INTERSECT operator does not distinguish between NULLs.

The INTERSECT operator removes duplicate rows from the final result set. The INTERSECT ALL operator does not remove duplicate rows from the final result set, but if a row appears X times in the first query and Y times in the second, it will appear min(X, Y) times in the result set.

The data type and the number of columns must be the same for each SELECT statement used with the INTERSECT operator.

Syntax of INTERSECT

SELECT column1, column2.... , columnN FROM table_Name1 [WHERE conditions]

INTERSECT

SELECT column1, column2.... , columnN FROM table_Name2 [WHERE conditions];

Let's understand the below example which explains how to execute INTERSECT operator in Structured Query Language:

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

Employee_details1:

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

Employee_details2:

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 a common record of the employee from both the tables in a single output. For this, we have to write the following query in SQL:

```
SELECT Emp_Name FROM Employee_details1

INTERSECT

SELECT Emp_Name FROM Employee_details2 ;
```

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

EXCEPT

The SQL EXCEPT operator takes the distinct rows of one query and returns the rows that do not appear in a second result set. For purposes of row elimination and duplicate removal, the EXCEPT operator does not distinguish between NULLs. The EXCEPT ALL operator does not remove duplicates, but if a row appears X times in the first query and Y times in the second, it will appear max(X - Y, 0) times in the result set.

Notably, the Oracle platform provides a MINUS operator which is functionally equivalent to the SQL standard EXCEPT DISTINCT operator.

The following example EXCEPT query returns all rows from the Orders table where Quantity is between 1 and 49, and those with a Quantity between 76 and 100.

Worded another way; the query returns all rows where the Quantity is between 1 and 100, apart from rows where the quantity is between 50 and 75.

```
SELECT *FROM Orders WHERE Quantity BETWEEN 1 AND 100

EXCEPT

SELECT *FROM  Orders WHERE Quantity BETWEEN 50 AND 75;
```

Joins

A join is a query that combines rows from two or more tables, views, based on a common field between them.

Consider the following two tables −

Table 1 − CUSTOMERS Table

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

Table 2 − ORDERS Table

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
|102|2009-10-0800:00:00|3|3000|
```

```
|100|2009-10-0800:00:00|3|1500|
|101|2009-11-2000:00:00|2|1560|
|103|2008-05-2000:00:00|4|2060|
+---+-------------------+----+----+
```

Now, let us join these two tables in our SELECT statement as shown below.

```
SELECT ID, NAME, AGE,AMOUNT FROM CUSTOMERS, ORDERS
WHERE  CUSTOMERS.ID= ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+---+----------+----+--------+
| ID | NAME     | AGE | AMOUNT |
+---+----------+----+--------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+---+----------+----+--------+
```

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal to symbol.

SQL JOINS: EQUI JOIN and NON EQUI JOIN

The are two types of SQL JOINS - EQUI JOIN and NON EQUI JOIN

1) SQL EQUI JOIN:

The SQL EQUI JOIN is a simple SQL join uses the equal sign(=) as the comparison operator for the condition. It has two types - SQL Outer join and SQL Inner join.

2) SQL NON EQUI JOIN :

The SQL NON EQUI JOIN is a join uses comparison operator other than the equal sign like >, <, >=, <= with the condition.

SQL EQUI JOIN : INNER JOIN and OUTER JOIN

The SQL EQUI JOIN can be classified into two types - **INNER JOIN** and **OUTER JOIN**

1. SQL INNER JOIN

This type of EQUI JOIN returns all rows from tables where the key record of one table is equal to the key records of another table.

2. SQL OUTER JOIN

This type of EQUI JOIN returns all rows from one table and only those rows from the secondary table where the joined condition is satisfying i.e. the columns are equal in both tables.

In order to perform a JOIN query, the required information we need are:

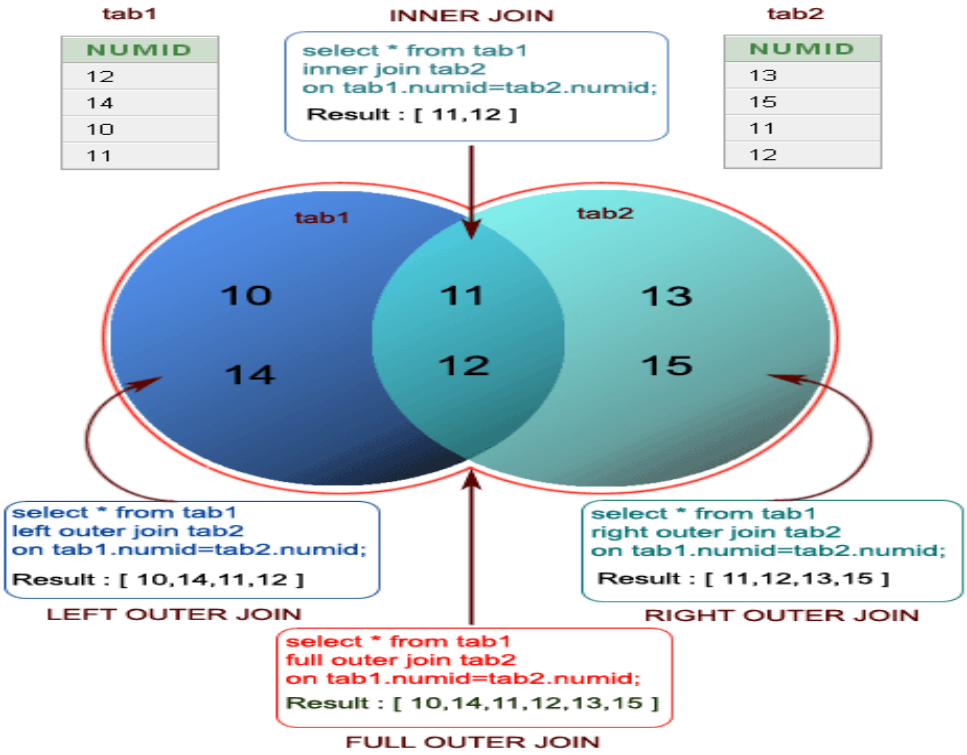
a) The name of the tables

b) Name of the columns of two or more tables, based on which a condition will perform.

Syntax:

FROM table1
join_type table2
[ON (join_condition)]
ON can be replaced with WHERE

Pictorial Presentation of SQL Joins:



Let's Consider the two tables given below.

Table name- Student:

id	Name	class	city
3	Hina	3	Delhi
4	Megha	2	Delhi
6	Gouri	2	Delhi

Table name — Record:

id	Class	City
9	3	Delhi
10	2	Delhi
12	2	Delhi

EQUI JOIN :

EQUI JOIN creates a JOIN for equality or matching column(s) values of the relative tables. EQUI JOIN also create JOIN by using JOIN with ON and then providing the names of the columns with their relative tables to check equality using equal sign (=).

Syntax :

```
SELECT column_list
FROM table1, table2....
WHERE table1.column_name =
table2.column_name;
```

Example –

```
SELECT student.name, student.id, record.class, record.city
FROM student, record
WHERE student.city = record.city;
```

Output :

name	Id	class	City
Hina	3	3	Delhi
Megha	4	3	Delhi
Gouri	6	3	Delhi
Hina	3	2	Delhi
Megha	4	2	Delhi
Gouri	6	2	Delhi
Hina	3	2	Delhi

name	Id	class	City
Megha	4	2	Delhi
Gouri	6	2	Delhi

2. NON EQUI JOIN :

NON EQUI JOIN performs a JOIN using comparison operator other than equal(=) sign like >, <, >=, <= with conditions.

Syntax:

SELECT *

FROM table_name1, table_name2

WHERE table_name1.column [>| < | >= | <=] table_name2.column;

Example –

SELECT student.name, record.id, record.city

FROM student, record

WHERE Student.id <Record.id ;

Output :

name	Id	city
Hina	9	Delhi
Megha	9	Delhi
Gouri	9	Delhi
Hina	10	Delhi
Megha	10	Delhi

name	Id	city
Gouri	10	Delhi
Hina	12	Delhi
Megha	12	Delhi
Gouri	12	Delhi

Nested Queries in SQL:

In nested queries, a query is written inside a query. The result of inner query is used in execution of outer query. Nested Queries are also called as**subqueries**.

Subqueries are useful when you must execute multiple queries to solve a single problem. Each query portion of a statement is called a query block. In the following query, the subquery in parentheses is the inner query block:

```
SELECT first_name, last_name FROM employees
WHERE  department_id
IN     ( SELECTdepartment_idFROM departments
         WHERE location_id = 1800 );
```

- The inner SELECT statement retrieves the IDs of departments with location ID 1800. These department IDs are needed by the outer query block, which retrieves names of employees in the departments whose IDs were supplied by the subquery.
- The structure of the SQL statement does not force the database to execute the inner query first. For example, the database could rewrite the entire query as a join of employees and departments, so that the subquery never executes by itself.

Subqueries can be **correlated** or **uncorrelated**.

Correlated subquery - In correlated subquery, inner query is dependent on the outer query. Outer query needs to be executed before inner query

Non-Correlated subquery - In non-correlated query inner query does not dependent on the outer query. They both can run separately.

Correlated Subqueries

A correlated subquery typically obtains values from its outer query before it executes. When the subquery returns, it passes its results to the outer query.

In the following example, the subquery needs values from the **addresses.state** column in the outer query:

```
=> SELECT name, street, city, state FROM addresses
    WHERE EXISTS (SELECT * FROM states WHERE states.state = addresses.state);
```

This query is executed as follows:

- The query extracts and evaluates each addresses.state value in the outer subquery records.
- Then the query—using the EXISTS predicate—checks the addresses in the inner (correlated) subquery.
- Because it uses the EXISTS predicate, the query stops processing when it finds the first match.

NoncorrelatedSubqueries

A noncorrelatedsubquery executes independently of the outer query. The subquery executes first, and then passes its results to the outer query, For example:

```
=> SELECT name, street, city, state FROM addresses WHERE state IN (SELECT state
FROM states);
```

This query is executed as follows:

- Executes the subquery SELECT state FROM states (in bold).
- Passes the subquery results to the outer query.

A query's WHERE and HAVING clauses can specify noncorrelatedsubqueries if the subquery resolves to a single row, as shown below:

In WHERE clause

=> SELECT COUNT(*) FROM SubQ1 WHERE SubQ1.a = (SELECT y from SubQ2);

In HAVING clause

=> SELECT COUNT(*) FROM SubQ1 GROUP BY SubQ1.a HAVING SubQ1.a = (SubQ1.a & (SELECT y from SubQ2)

Aggregate functions:

Aggregate functions operate on values across rows to perform mathematical calculations such as sum, average, counting, minimum/maximum values, standard deviation, and estimation, as well as some non-mathematical operations.

An aggregate function takes multiple rows (actually, zero, one, or more rows) as input and produces a single output.

Various Aggregate Functions:

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.
2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.
3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.
4. MAX (A): The maximum value in the A column.
5. MIN (A): The minimum value in the A column.

Let us consider a table that contains the following data:

```
selectx,yfromsimpleorderbyx,y;
+----+----+
|  X |  Y |
|----+----|
| 10 | 20 |
| 20 | 44 |
| 30 | 70 |
+----+----+
```

The aggregate function returns one output row for multiple input rows:

```
selectsum(x)
```

```
fromsimple;
+---------+
| SUM(X) |
|---------|
|    60 |
+---------+
```

Now let us understand each Aggregate function with a example:

Id	Name	Salary
1	A	80
2	B	40
3	C	60
4	D	70
5	E	60
6	F	Null

Count():

Count(*): Returns total number of records .i.e 6.

Count(salary): Return number of Non Null values over the column salary. i.e 5.

Count(Distinct Salary): Return number of distinct Non Null values over the column salary .i.e 4.

Sum():

sum(salary): Sum all Non Null values of Column salary i.e., 310

sum(Distinct salary): Sum of all distinct Non-Null values i.e., 250.

Avg():

Avg(salary) = Sum(salary) / count(salary) = 310/5

Avg(Distinct salary) = sum(Distinct salary) / Count(Distinct Salary) = 250/4

Min():

Min(salary): Minimum value in the salary column except NULL i.e., 40.

Max(salary): Maximum value in the salary i.e., 80.

Aggregate Functions and NULL Values

Some aggregate functions ignore NULL values. For example, AVG calculates the average of values 1, 5, and NULL to be 3, based on the following formula:

(1 + 5) / 2 = 3

If all of the values passed to the aggregate function are NULL, then the aggregate function returns NULL.

Some aggregate functions can be passed more than one column. For example:

select count(col1, col2) from table1;

In these instances, the aggregate function ignores a row if any individual column is NULL.

```
insertintot(x,y)values
(1,2),-- No NULLs.
(3,null),-- One but not all columns are NULL.
(null,6),-- One but not all columns are NULL.
(null,null);-- All columns are NULL.
```

Query the table:

```
selectcount(x,y)fromt;
+----------------+
| COUNT(X, Y) |
|----------------|
|          1 |
+----------------+
```

Similarly, if SUM is called with an expression that references two or more columns, and if one or more of those columns is NULL, then the expression evaluates to NULL, and the row is ignored:

```
selectsum(x+y)fromt;
+--------------+
```

```
| SUM(X + Y) |
|------------|
|          3 |
+------------+
```

SQL also provides a special comparison operator IS NULL to test whether a column value is null; for example the value of y IS NULL returns **true** when x is 3 and IS NOT NULL returns **false**.

INTRODUCTION TO VIEWS

A view is a table whose rows are not explicitly stored in the database but are computed as needed.

Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain condition.

Sample Tables:

StudentDetails

S_ID	NAME	ADDRESS
1	Harsh	Kolkata
2	Ashish	Durgapur
3	Pratik	Delhi
4	Dhanraj	Bihar
5	Ram	Rajasthan

StudentMarks

ID	NAME	MARKS	AGE
1	Harsh	90	19
2	Suresh	50	20
3	Pratik	80	19
4	Dhanraj	95	21
5	Ram	85	18

CREATING VIEWS

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

Syntax:

CREATE VIEW view_name ASSELECT column1, column2.....
FROM table_nameWHERE condition;

view_name: Name for the View
table_name: Name of the table
condition: Condition to select rows

Examples:

Creating View from a single table:
In this example we will create a View named DetailsView from the table StudentDetails.
Query:

CREATE VIEW DetailsView ASSELECT NAME, ADDRESS
FROM StudentDetailsWHERE S_ID < 5;

To see the data in the View, we can query the view in the same manner as we query a table.

SELECT * FROM DetailsView;

Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

Creating View from multiple tables: In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement.

Query:

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

To display data of View MarksView:

SELECT * FROM MarksView;

Output:

NAME	ADDRESS	MARKS
Harsh	Kolkata	90
Pratik	Delhi	80
Dhanraj	Bihar	95
Ram	Rajasthan	85

DELETING VIEWS

SQL allows us to delete an existing View. We can delete or drop a View using the DROP statement.

Syntax:

DROP VIEW view_name;

view_name: Name of the View which we want to delete.
For example, if we want to delete the View MarksView, we can do this as:

DROP VIEW MarksView;

UPDATING VIEWS

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is not met, then we will not be allowed to update the view.

1. The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
2. The SELECT statement should not have the DISTINCT keyword.
3. The View should have all NOT NULL values.
4. The view should not be created using nested queries or complex queries.

5. 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.

We can use the CREATE OR REPLACE VIEW statement to add or remove fields from a view.

Syntax:

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1,coulmn2,..
FROM table_name
WHERE condition;
```

For example, if we want to update the view MarksView and add the field AGE to this View from StudentMarks Table, we can do this as:

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

If we fetch all the data from MarksView now as:

```
SELECT * FROM MarksView;
```

Output:

NAME	ADDRESS	MARKS	AGE
Harsh	Kolkata	90	19
Pratik	Delhi	80	19
Dhanraj	Bihar	95	21
Ram	Rajasthan	85	18

Inserting a row in a view:

We can insert a row in a View in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.**Syntax**:

```
INSERT INTO view_name(column1, column2 , column3,..)
VALUES(value1, value2, value3..);
```

view_name: Name of the View

Example:

In the below example we will insert a new row in the View DetailsView which we have created above in the example of "creating views from a single table".

INSERT INTO DetailsView(NAME, ADDRESS)

VALUES("Suresh","Gurgaon");

If we fetch all the data from DetailsView now as,

SELECT * FROM DetailsView;

Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar
Suresh	Gurgaon

Deleting a row from a View:

Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view. Also deleting a row from a view first delete the row from the actual table and the change is then reflected in the view.**Syntax**:

DELETE FROM view_name

WHERE condition;

view_name:Name of view from where we want to delete rows

condition: Condition to select rows

Example:

In this example we will delete the last row from the view DetailsView which we just added in the above example of inserting rows.

DELETE FROM DetailsView

WHERE NAME="Suresh";

If we fetch all the data from DetailsView now as,

SELECT * FROM DetailsView;

Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

TRIGGERS

A trigger is a stored procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA. A database that has a set of associated triggers is called an active database. A trigger description contains three parts:

Event: A change to the database that activates the trigger.

Condition: A query or test that is run when the trigger is activated.

Action: A procedure that is executed when the trigger is activated and its con-dition is true.

A **trigger action** can examine the answers to the query in the condition part of the trigger, refer to old and new values of tuples modified by the statement activating the trigger, execute new queries, and make changes to the database.

Syntax:

create trigger [trigger_name]

[before | after]

{insert | update | delete}

on [table_name]

[for each row]

[trigger_body]

Explanation of syntax:

1. create trigger [trigger_name]: Creates or replaces an existing trigger with the trigger_name.
2. [before | after]: This specifies when the trigger will be executed.
3. {insert | update | delete}: This specifies the DML operation.
4. on [table_name]: This specifies the name of the table associated with the trigger.
5. [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
6. [trigger_body]: This provides the operation to be performed as trigger is fired

BEFORE and AFTER of Trigger:

BEFORE triggers run the trigger action before the triggering statement is run.

AFTER triggers run the trigger action after the triggering statement is run.

Examples of Triggers in SQL

The trigger called init count initializes a counter variable before every execution of an INSERT statement that adds tuples to the Students relation. The trigger called incr count increments the counter for each inserted tuple that satisfies the condition age < 18.

```
CREATE TRIGGER init count BEFORE INSERT ON Students /* Event */

DECLARE

count INTEGER;

BEGIN

count := 0;

END

/* Action */

CREATE TRIGGER incr count AFTER INSERT ON Students /* Event */

WHEN (new.age< 18) /* Condition; ‘new’ is just-inserted tuple */

FOR EACH ROW

BEGIN /* Action; a procedure in Oracle’s PL/SQL syntax */
```

count := count + 1;

END

(identifying the modified table, Students, and the kind of modifying statement, an INSERT), and the third field is the number of inserted Students tuples with age < 18. (The trigger in Figure 5.19 only computes the count; an additional trigger is required to insert the appropriate tuple into the statistics table.)

CREATE TRIGGER set count AFTER INSERT ON Students /* Event */

REFERENCING NEW TABLE AS InsertedTuples

FOR EACH STATEMENT

INSERT /* Action */

INTO StatisticsTable(ModifiedTable, ModificationType, Count) SELECT

'Students', 'Insert', COUNT * FROM InsertedTuples I WHERE I.age< 18

Unit 4

DEPENDENCE AND NORMAL FORMS

Importance of a good schema design:

What is a Database Schema?

A database schema is a blueprint that represents the tables and relations of a data set.

It is important to have a good database schema design. The reasons are:

- To avoid data redundancy which wastes memory and leads to data inconsistency.
- To have correctness and completeness of data.
- To maintain data accuracy and integrity.
- To write simple and easy queries.

Problems that arise with bad database schema is :

- Anomalies occur whenever data is inserted, modified or deleted in case of large database.
- This makes data integrity harder to maintain.
- Data inconsistency can occur.
- Difficulty to scale the database when future application functionality is added.
- Performance reduces.
- Maintenance also becomes difficult.

To prevent all these problems one has to normalize the database by efficiently organizing the data.

Normalization

- Normalization is a process of specifying and defining keys, columns, relationships in order to create an efficient database.

Objectives of Normalization

- Normalization reduces data redundancy there by reduces the amount of space used by database and ensures that data is stored efficiently.
- It divides large tables into many smaller tables and makes a relation between them.
- It reduces cause of anomalies when data is manipulated.

Normalization defines rules for the relational table in the form of **normal forms.**

Normal Form is a process that evaluates each relation against defined rules and criteria. It removes multi-valued primary keys, joins, functional dependencies etc., to improve the relational table integrity and efficiency.

Functional Dependency (FD):

- The functional dependency is a relationship that exists between two attributes.
- It is constraint where one attribute determines the value of another one.
- It plays a vital role to find the difference between good and bad database design.
- It typically exists between the primary key and non-key attribute within a table.

For any relation R, attribute Y is functionally dependent on attribute X (usually the PK), if for every valid instance of X, that value of X uniquely determines the value of Y. This relationship is indicated by the representation below :

X→Y

- The left side of FD is known as a determinant, the right side of the production is known as a dependent.

For example:
Assume we have an employee table with attributes: Emp_Id, Emp_Name, Emp_Address.

- Here Emp_Id attribute can uniquely identify the Emp_Name attribute of employee table because if we know the Emp_Id, we can tell that employee name associated with it.
- Functional dependency can be written as:

  Emp_Id → Emp_Name
- We can say that Emp_Name is functionally dependent on Emp_Id.

Types of Functional Dependencies:

There are mainly four types of Functional Dependency in DBMS:

- **Multivalued Dependency**
- **Trivial Functional Dependency**
- **Non-Trivial Functional Dependency**
- **Transitive Dependency**

Multivalued Functional Dependency

- Multivalued dependency occurs in the situation where there are multiple independent multivalued attributes in a single table.
- In Multivalued FD, entities of the dependent set are not dependent on each other.

In an Emp table empname and salary attributes both depend on empId for identification. But both are independent to each other.

Emp_Id →{ Emp_Name,sal} is an example of multivalued FD.

Trivial Functional Dependency

In **Trivial Functional Dependency**, a dependent is always a subset of the determinant.

i.e. If **X → Y** and **Y is the subset of X**, then it is called trivial functional dependency.

In Emp table {Emp_Id,Emp_Name}→Emp_Name is a trivial FD as Emp_Name is a subset of {Emp_Id,Emp_Name}.

{Emp_Id,Emp_Name}→Emp_Id is also a trivial FD.

Non-Trivial Functional Dependency

In this FD, the dependent is strictly not a subset of the determinant.

If **X → Y** then **Y is not a subset of X**

{Emp_Id,Emp_Name} set can determine the value of Emp_Address or Salary. But Emp_Address or Salary doesn't belong to the set or not a subset of {Emp_Id,Emp_Name}

Hence, {Emp_Id,Emp_Name}→Sal is a non-trivial FD.
{Emp_Id,Emp_Name}→Emp_Address is also a non-trivial FD.

Transitive Functional Dependency

In transitive functional dependency, dependent is indirectly dependent on determinant. It is formed by two functional dependencies.

If A → **B** and B →**C** then A → **C**

If Emp_Id identifies Dept_No of an employee and Dept_No identifies Dept_Name then Emp_Id can indirectly identify Dept_Name according to transitive rule.

Emp_Id → Dept_No,
Dept_No → Dept_Name then Emp_Id → Dept_Name

Armstrong's Axioms for Functional Dependencies

- **Armstrong's Axioms** are a set of rules, developed by William W.Armstrong in 1974.
- It is used to infer all the functional dependencies on a relational database.
- For a set of functional dependencies F, if these rules are applied repeatedly, then they generate a closure of FDs denoted as, F^{+}.

Armstrong's Axioms has mainly two different sets of rules:

1. Primary Rule
2. Secondary Rule

Primary Rule:

1. Axiom of reflexivity –

If **A** is a set of attributes and **B** is subset of **A**, then **A** holds **B**.
If **B** ⊆ **A** then **A -> B** This property is trivial property.
e.g. {Emp_Id,Emp_Name}→Emp_Name

2. Axiom of augmentation –

If **A** holds **B** and **C** is an attribute set, then **AC** also holds **BC**. That is adding attributes in dependencies, does not change the basic dependencies.
The augmentation is also called as a partial dependency. In augmentation, if A determines B, then AC determines BC for any C.

if **A -> B** then **AC -> BC**
e.g. if Emp_Id→Emp_Name then
{Emp_Id,Emp_Address}→{Emp_Name,Emp_Adress}

3. Axiom of transitivity –

If **A** holds **B** and **B** holds **C**, then **A** also holds **C**. if A determines B and B determine C, then A must also determine C.

If A → **B** and B →**C** then A → **C**

e.g. if Emp_Id → Dept_No, Dept_No → Dept_Name then
Emp_Id → Dept_Name

Secondary Rule:

These are derived from above axioms.

1. **Union** –
   Union rule says, if A determines B and A determines C, then A must also determine B and C.
   If **A** holds **B** and **A** holds **C**, then **A** holds **BC**.
   If A → **B** and A →**C** then A → **BC**

2. **Decomposition –**
   Decomposition rule is also known as project rule. It is the reverse of union rule. This Rule says, if A determines B and C, then A determines B and A determines C separately.
   If **A** holds **BC** and **A** holds **B** then **A** holds **C**.
   If A → **BC** and A →**B** then A → **C**

3. **Pseudo Transitivity –**
   In Pseudo transitive Rule, if A determines B and BC determines D, then AC determines D.
   If **A** holds **B** and **BC** holds **D**, then **AC** holds **D**.
   If A → **B** and BC →**D** then AC → **D**
   Proof:
   if **A -> B** then **AC -> BC** **(Axiom of augmentation)**
   If AC → **BC** and BC →**D** then AC → **D** **(Axiom of Transitivity)**

Closure of Functional Dependencies

- The Closure Of Functional Dependencies means the complete set of all possible FDs that can be derived from given set of FDs using Armstrong's Rules.
- If **F** is a set of functional dependencies of relation R then a closure set of FDs implied by **F** is denoted by $\mathbf{F^+}$.

- Closure of a set of FDs can be achieved by finding closure of a set of attributes X.

Example 1

We are given the relation R(A, B, C, D, E). This means that the table R has five columns: A, B, C, D, and E. We are also given the set of functional dependencies: {A->B, B->C, C->D, D->E}.

What is $\{A\}^+$?

- First, we add A to $\{A\}^+$.
- What columns can be determined given A? We have A -> B, so we can determine B. Therefore, $\{A\}^+$ is now {A, B}.
- What columns can be determined given A and B? We have B -> C in the functional dependencies, so we can determine C. Therefore, $\{A\}^+$ is now {A, B, C}.
- Now, we have A, B, and C. What other columns can we determine? Well, we have C -> D, so we can add D to $\{A\}^+$.
- Now, we have A, B, C, and D. Can we add anything else to it? Yes, since D -> E, we can add E to $\{A\}^+$.
- We have used all of the columns in R and we have all used all functional dependencies. $\{A\}^+$ = {A, B, C, D, E}.

Example 2

We have a table Course Editions. The table contains information about editions of courses taught at a certain university.

Each year, each course can be taught by a different teacher. And each teacher has a date of birth. With the year and the date of birth, you can determine the age of the teacher at the time the course was taught.

Course Editions

course	year	teacher	date_of_birth	age
Databases	2019	Chris Cape	1974-10-12	45
Mathematics	2019	Daniel Parr	1985-05-17	34
Databases	2020	Jennifer Clock	1990-06-09	30

Here are the functional dependencies in this table:

- *course, year -> teacher*
  - Given the course and year, you can determine the teacher who taught the course that year.
- *teacher -> date_of_birth*
  - Given a teacher, you can determine the teacher's date of birth.
- *year, date_of_birth -> age*
  - Given the year and date of birth, you can determine the age of the teacher at the time the course was taught.

First, consider the closure of a set **{year},** denoted **{year}$^+$.** The first functional dependency *course, year -> teacher* requires the course in addition to the year, so it doesn't add anything to {year}$^+$. The functional dependency *year, date_of_birth -> age* requires the date of birth in addition to the year, so it doesn't add anything to {year}$^+$ either.

So, **{year}$^+$** contains only one column, year, that is **{year}$^+$ = {year}.**

Next, let's look at **{year, teacher}$^+$.** Given the year and teacher, what other columns can we determine?

If we know the teacher*, we* also know the date of birth because of the

teacher -> date_of_birth functional dependency. So, date_of_birth is also in **{year, teacher}$^+$,** and we know three columns: {*year*, *teacher*, *date_of_birth*}.

If we know the year and date of birth, we can also determine the age. Now, **{year, teacher}$^+$** has four columns {*year*, *teacher*, *date_of_birth, age*}.

We have used two of the three functional dependencies. we can't use the remaining dependency, *course, year -> teacher* because we don't know the course.

Now that we have used all of the dependencies I can, **{year, teacher}$^+$ = {*year*, *teacher*, *date_of_birth, age*}.**

Minimal Covers:

A minimal cover of a set of functional dependencies (FD) E is a minimal set of dependencies F that is equivalent to E.

The formal definition is: A set of FD F to be minimal if it satisfies the following conditions –

- Every dependency in F has a single attribute for its right-hand side.
- We cannot replace any dependency X->A in F with a dependency Y->A, where Y is a proper subset of X.
- We cannot remove any dependency from F.

Canonical cover is called minimal cover which is called the minimum set of FDs.
A set of FD FC is called canonical cover of F if each FD in FC is a –

- Simple FD.
- Left reduced FD.
- Non-redundant FD.

Simple FD – X->Y is a simple FD if Y is a single attribute.
Left reduced FD – X->Y is a left reduced FD if there are no extraneous attributes in X. {extraneous attributes: Let XA->Y then, A is a extraneous attribute if X_>Y}
Non-redundant FD – X->Y is a Non-redundant FD if it cannot be derived from F-{X->y}.

Example

Consider an example to find canonical cover of F.
The given functional dependencies are as follows –
A -> BC
B -> C
A -> B
AB -> C

- Minimal cover: The minimal cover is the set of FDs which are equivalent to the given FDs.
- Canonical cover: In canonical cover, the LHS (Left Hand Side) must be unique.

First of all, we will find the minimal cover and then the canonical cover.
First step – Convert RHS attribute into singleton attribute.
A -> B
A -> C
B -> C
A -> B
AB -> C
Second step – Remove the extra LHS attribute
Find the closure of A.
A+ = {A, B, C}

So, AB -> C can be converted into A -> C
A -> B
A -> C
B -> C
A -> B
A -> C
Third step – Remove the redundant FDs.
A -> B
A -> C
Now, we will convert the above set of FDs into canonical cover.
The canonical cover for the above set of FDs will be as follows –
A -> BC
B -> C

NORMAL FORMS

Given a relation schema, we need to decide whether it is a good design or whether we need to decompose it into smaller relations. Such a decision must be guided by an understanding of what problems, if any, arise from the current schema. To provide such guidance, several normal forms have been proposed. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise.
The normal forms based on FDs:

First Normal Form (1NF):

- First Normal Form is defined in the definition of relations (tables) itself. This rule defines that all the attributes in a relation must have atomic domains.
- In the *first normal form*, only single values are permitted at the intersection of each row and column; hence, there are no repeating groups.
- To normalize a relation that contains a repeating group, remove the repeating group and form two new relations.

Course	Content
Programming	Java, c++
Web	HTML, PHP, ASP

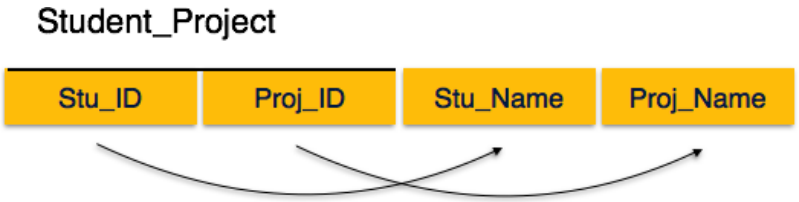
We re-arrange the relation (table) as below, to convert it to First Normal Form.

Course	Content
Programming	Java
Programming	c++
Web	HTML
Web	PHP
Web	ASP

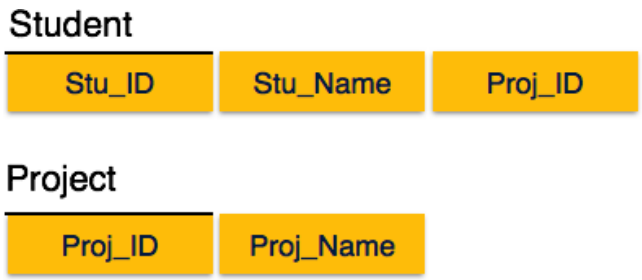
Second Normal Form (2NF):

Before we learn about the second normal form, we need to understand the following −

- **Prime Key attribute** − An attribute, which is a part of the candidate-key, is known as a prime attribute.
- **Non-prime attribute** − An attribute, which is not a part of the prime-key, is said to be a non-prime attribute.
- For the *second normal form*, the relation must first be in 1NF.
- The relation is automatically in 2NF if, and only if, the Prime Key comprises a single attribute.
- If the relation has a composite Prime Key, then each non-key attribute must be fully dependent on the entire PK and not on a subset of the PK.
- A relation is in 2NF if it has **No Partial Dependency.**
- **Partial Dependency –** If the proper subset of candidate key determines non-prime attribute, it is called partial dependency.



We see here in Student_Project relation that the prime key attributes are Stu_ID and Proj_ID. According to the rule, non-key attributes, i.e. Stu_Name and Proj_Name must be dependent upon both and not on any of the prime key attribute individually. But we find that Stu_Name can be identified by Stu_ID and Proj_Name can be identified by Proj_ID independently. This is called **partial dependency**, which is not allowed in Second Normal Form.



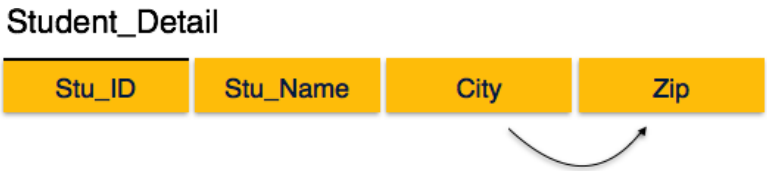
Third Normal Form (3NF):

To be in *third normal form*, the relation must be in second normal form. Also - all transitive dependencies must be removed; a non-key attribute may not be functionally dependent on another non-key attribute.

For any non-trivial functional dependency, X → A, then either –

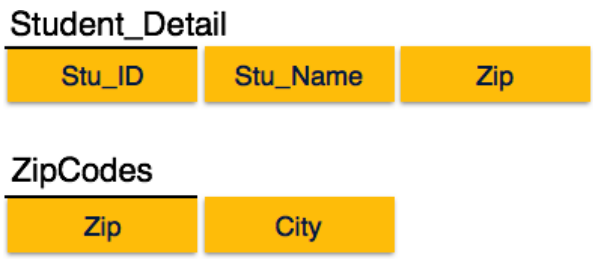
- X is a superkey or,
- A is prime attribute.

Transitive dependency – If A->B and B->C are two FDs then A->C is called transitive dependency.



We find that in the above Student_detail relation, Stu_ID is the key and only prime key attribute. We find that City can be identified by Stu_ID as well as Zip itself. Neither Zip is a superkey nor is City a prime attribute. Additionally, Stu_ID → Zip → City, so there exists **transitive dependency**.

To bring this relation into third normal form, we break the relation into two relations as follows –



Boyce-Codd Normal Form (BCNF):

Boyce-Codd Normal Form (BCNF) is an extension of Third Normal Form on strict terms.

A relation is in BCNF iff in every non-trivial functional dependency X –> Y, X is a super key.

In the above example, Stu_ID is the super-key in the relation Student_Detail and Zip is the super-key in the relation ZipCodes. So,

Stu_ID → Stu_Name, Zip

and

Zip → City

Which confirms that both the relations are in BCNF.

Example

Consider a relation R with attributes (student, subject, teacher).

Student	Teacher	Subject
Jhansi	P.Naresh	Database
Jhansi	K.Das	C
Subbu	P.Naresh	Database
Subbu	R.Prasad	C

F: { (student, Teacher) -> subject

(student, subject) -> Teacher

Teacher -> subject}

Candidate keys are (student, teacher) and (student, subject).

The above relation is in 3NF [since there is no transitive dependency]. A relation R is in BCNF if for every non-trivial FD X->Y, X must be a key.

The above relation is not in BCNF, because in the FD (teacher->subject), teacher is not a key.

So R is divided into two relations R1(Teacher, subject) and R2(student, Teacher).

R1

Teacher	Subject
P.Naresh	database
K.DAS	C
R.Prasad	C

R2

Student	Teacher
Jhansi	P.Naresh
Jhansi	K.Das
Subbu	P.Naresh
Subbu	R.Prasad

All the anomalies which were present in R, now removed in the above two relations.

DECOMPOSITIONS

A decomposition of a relation schema *R* consists of replacing the relation schema by two (or more) relation schemas that each contain a subset of the attributes of *R* and together include all attributes in *R*.

When a relation in the relational model is not appropriate normal form then the decomposition of a relation is required. In a database, breaking down the table into multiple tables termed as decomposition.

The properties of a relational decomposition are listed below :

1. **Attribute Preservation:** Using functional dependencies the algorithms decompose the universal relation schema R in a set of relation schemas D = { R1, R2, ….. Rn } relational database schema, where 'D' is called the Decomposition of R.

The attributes in R will appear in at least one relation schema Ri in the decomposition, i.e., no attribute is lost. This is called the *Attribute Preservation* condition of decomposition.

2. **Dependency Preservation:** If each functional dependency X->Y specified in F appears directly in one of the relation schemas Ri in the decomposition D or could be inferred from the dependencies that appear in some Ri. This is the *Dependency Preservation*.

If a relation R is decomposed into relation R1 and R2, then the dependencies of R either must be a part of R1 or R2 or must be derivable from the combination of functional dependencies of R1 and R2.

For example, suppose there is a relation R (A, B, C, D) with functional dependency set (A->BC). The relational R is decomposed into R1(ABC) and R2(AD) which is dependency preserving because FD A->BC is a part of relation R1(ABC).

3. **Lossless Join Decomposition:** Lossless join property is a feature of decomposition supported by normalization. It is the ability to ensure that any instance of the original relation can be identified from corresponding instances in the smaller relations.

For example: R : relation, F : set of functional dependencies on R, X, Y : decomposition of R, A decomposition {R1, R2, …, Rn} of a relation R is called a lossless decomposition for R if the natural join of R1, R2, …, Rn produces exactly the relation R.

The relation is said to be lossless decomposition if natural joins of all the decomposition give the original relation.

Decomposition is lossless if

1. The union of attributes of both the sub relations R1 and R2 must contain all the attributes of original relation R.

   R1 ⋈ R2 = R

2. The intersection of attributes of both the sub relations R1 and R2 must not be null, i.e., there should be some attributes that are present in both R1 and R2.

   R1 ∩ R2 ≠ ∅

3. The intersection of attributes of both the sub relations R1 and R2 must be the superkey of R1 or R2, or both R1 and R2.

   R1 ∩ R2 = Super key of R1 or R2

Let's see an example of a lossless join decomposition. Suppose we have the following relation EmployeeProjectDetail as:

<Employee Project Detail>

Employee_Code	Employee_Name	Employee_Email	Project_Name	Project_ID
101	John	john@demo.com	Project103	P03
101	John	john@demo.com	Project101	P01
102	Ryan	ryan@example.com	Project102	P02
103	Stephanie	stephanie@abc.com	Project102	P02

Now, we decompose this relation into EmployeeProject and ProjectDetail relations as:

<Employee Project>

Employee_Code	Project_ID	Employee_Name	Employee_Email
101	P03	John	john@demo.com
101	P01	John	john@demo.com
102	P04	Ryan	ryan@example.com
103	P02	Stephanie	stephanie@abc.com

The primary key of the above relation is {Employee_Code, Project_ID}.

<Project Detail>

Project_ID	Project_Name
P03	Project103
P01	Project101
P04	Project104
P02	Project102

The primary key of the above relation is {Project_ID}.

Let's first check the EmployeeProject ⊠ ProjectDetail:

<Employee Project ∪ Project Detail>

Employee_Code	Project_ID	Employee_Name	Employee_Email	Project_Name
101	P03	John	john@demo.com	Project103
101	P01	John	john@demo.com	Project101
102	P04	Ryan	ryan@example.com	Project104
103	P02	Stephanie	stephanie@abc.com	Project102

As we can see all the attributes of Employee Project and Project Detail are in Employee Project U Project Detail relation and it is the same as the original relation. So the first condition holds.

Now let's check the EmployeeProject ∩ ProjectDetail:

<EmployeeProject ∩ ProjectDetail>

Project_ID
P03
P01
P04
P02

As we can see this is not null, so the the second condition holds as well. Also the EmployeeProject ∩ ProjectDetail = Project_Id. This is the super key of the ProjectDetail relation, so the third condition holds as well.

Now, since **all three conditions hold** for our decomposition, this is a **lossless join decomposition.**

4. Lack of Data Redundancy

- Lack of Data Redundancy is also known as a **Repetition of Information.**
- The proper decomposition should not suffer from any data redundancy.
- The lack of data redundancy property may be achieved by Normalization process.

Unit 5
TRANSACTIONS

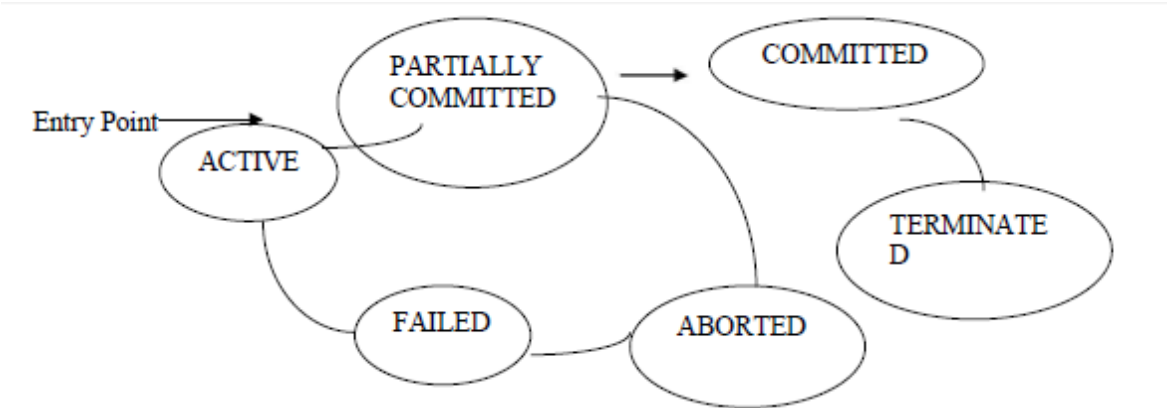
Transaction concept:

- *Transaction* refers to a collection of operations that form a single logical unit of work. For instance, transfer of money from one account to another is a transaction consisting of two updates, one to each account.
- A transaction includes one or more database access operations—these can include insertion, deletion, modification (update), or retrieval operations.
- One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program.
- If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction**; otherwise it is known as a **read-write transaction**.
- The transactions bring the database from an image which existed before the transaction occurred (called the **Before Image** or **BFIM**) to an image which exists after the transaction occurred (called the **After Image** or **AFIM**).

Transaction States:

There are the following six states in which a transaction may exist:

- **Active:** The initial state when the transaction has just started execution.
- **Partially Committed:** At any given point of time if the transaction is executing properly, then it is going towards it COMMIT POINT. The values generated during the execution are all stored in volatile storage.
- **Failed:** If the transaction fails for some reason. The temporary values are no longer
- **Aborted:** When the ROLLBACK operation is over, the database reaches the BFIM. The transaction is now said to have been aborted.
- **Committed:** If no failure occurs then the transaction reaches the COMMIT POINT. All the temporary values are written to the stable storage and the transaction is said to have been committed.
- **Terminated:** Either committed or aborted required, and the transaction is set to **ROLLBACK**.



System Log

- The system maintains a log to keep track of all transaction operationsthat affect the values of database items, as well as other transaction information that may be needed to permit recovery from failures.
- The log is a sequential, append-only file that is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.
- The log file from disk is periodically backed up to archival storage (tape)to guard against catastrophic failures.

Commit Point:

- A transaction *T* reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database have been recorded in the log.
- Beyond the commit point, the transaction is said to be **committed**, and its effect must be *permanently recorded* in the database.

Properties of Transactions

These properties are often called the **ACID properties**; the acronym is derived from the first letter of each of the four properties.

Atomicity: A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.

Consistency :A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another. Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

Isolation : A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

Durability : The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

Concurrent Execution

Schedule: A schedule for a set of transactions must consist of all instructions of those transactions, and must preserve the order in which the instructions appear in each individual transaction.
They represent the chronological order in which instructions areexecuted in the system. Depending upon how these transactions are arranged in within a schedule, aschedule can be of two types:

1.Serial: The transactions are executed one after another, in a non-preemptivemanner.

2.Concurrent: The transactions are executed in a preemptive, time shared method.

Serial:- Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.

- In Serial schedule, there is no question of sharing a single data item among many transactions, because not more than a single transaction is executing at any point of time.
- However, a serial schedule is inefficient in the sense that the transactions suffer for having a longer waiting time and response time, as well as low amount of resource utilization.

Concurrent: In concurrent schedule, CPU time is shared among two or more

transactions in order to run them concurrently.

- If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on.

Let us consider there are two transactions T1 and T2, whose instruction sets are given as following.

T1
Read A;
A = A – 100;
Write A;
Read B;
B = B + 100;
Write B;

T2
Read A;
Temp = A * 0.1;
Read C;
C = C + Temp;
Write C;

- T2 is a new transaction which deposits to account C 10% of the amount in account A.

If we prepare a serial schedule, then either T1 will completely finish before T2 can begin, or T2 will completely finish before T1 can begin.

- However, if we want to create a concurrent schedule, then some Context Switching need to be made, so that some portion of T1 will be executed, then some portion of T2 will be executed and so on.

For example say we have prepared the following concurrent schedule.

T1	T2
Read A;	
A = A – 100;	
Write A;	
	Read A;
	Temp = A * 0.1;
	Read C;
	C = C + Temp;
	Write C;
Read B;	
B = B + 100;	
Write B;	

- T1 first deducts Rs 100/- from A and writes the new value of Rs 900/- into A. T2 reads the value of A, calculates the value of Temp to be Rs 90/- and adds the value to C. The remaining part of T1 is executed and Rs 100/- is added to B.
- If control of concurrent execution is left entirely to the operating system, many possible schedules, they may leave the database in an inconsistent state.
- It is the job of the database system to ensure that any schedule that is executed will leave the database in a consistent state. The **concurrency-control** component of the database system carries out this task.
- We can ensure consistency of the database under concurrent execution by making sure that any schedule that is executed has the same effect as equivalent to a serial schedule. Such schedules are called **serializable** schedules.

Serializability:

When several concurrent transactions are trying to access the same data item, the instructions within these concurrent transactions must be ordered in some way so as there are no problem in accessing and releasing the shared data item. There are two aspects of serializability which are described here:

Conflict Serializability:

- Conflict Serializability deals with detecting whether the instructions are conflicting in any way, and specifying the order in which these two instructions will be executed in case there is any conflict.
- A **conflict** arises if at least one (or both) of the instructions is a write operation.

The following rules are important in Conflict Serializability:

1. If two instructions of the two concurrent transactions are both for read operation, then they are not in conflict, and can be allowed to take place in any order.
2. If one of the instructions wants to perform a read operation and the other instruction wants to perform a write operation, then they are in conflict, hence their ordering is important. If the read instruction is performed first, then it reads the old value of the data item and after the reading is over, the new value of the data item is written. It the write instruction is performed first, then updates the data item with the new value and the read instruction reads the newly updated value.
3. If both the transactions are for write operation, then they are in conflict but can be allowed to take place in any order, because the transaction

do not read the value updated by each other. However, the value that persists in the data item after the schedule is over is the one written by the instruction that performed the last write.

View Serializability:

- This is another type of serializability that can be derived by creating another schedule out of an existing schedule, involving the same set of transactions.
- The idea behind view serializability is that, as long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results. The read operations are hence said to *see the same view* in both schedules. The final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules.
- View serializability is not used in practice due to its high degree of computational complexity.

Testing for serializability

- A **precedence graph** (or **serialization graph**) is used to test a schedule for conflict serializability.
- It is a **directed graph** *G* = (V, *E*) that consists of a set of nodes /vertices V = {*T*1, *T*2, … , *Tn* } and a set of directed edges *E* = {*e*1, *e*2, … , *em* }. There is one node in the graph for each transaction *Ti* in the schedule. Each edge *ei* in the graph is of the form (*Tj* ➝*Tk* , where *Tj* is the **starting node** of *ei* and *Tk* is the **ending node** of e*i*.
- An edge e_i is constructed between nodes T_j to T_k if one of the operations in T_j appears in the schedule before some conflicting operation in T_k .

The Algorithm can be written as:

1. Create a node T in the graph for each participating transaction in the schedule.
2. If a Transaction T_j executes a read_item (X) after T_i executes a write_item (X), draw an edge from T_i to T_j in the graph.
3. If a Transaction T_j executes a write_item (X) after T_i executes a read_item (X), draw an edge from T_i to T_j in the graph.
4. If a Transaction T_j executes a write_item (X) after T_i executes a write_item (X), draw an edge from T_i to T_j in the graph.
5. **The Schedule S is serializable if there is no cycle in the precedence**

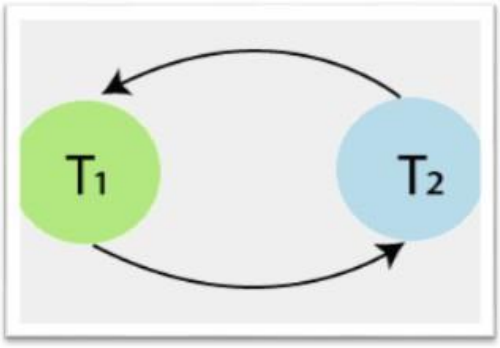
graph.

If there is no cycle in the precedence graph, it means we can construct a serial schedule S' which is conflict equivalent to schedule S.

Schedule S:

Time	Transaction T1	Transaction T2
t1	Read(A)	
t2		Read(A)
t3		Write(A)
t4	A=A+50	
t5	Write(A)	

Precedence graph of Schedule S

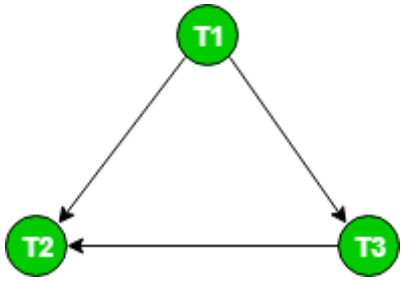


The precedence graph contains a cycle, that's why schedule S is non-serializable.

Schedule S2:

Time	Transaction T1	Transaction T2	Transaction T3
t1	Read(A)		
t2			Read(B)
t3	Write(A)		
t4		Write(B)	
t5			Read(A)
t6		Write(A)	

The graph for this schedule is :



Since the graph is acyclic, the schedule is conflict serializable.

- If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph.
- A serializability order of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph.

Recoverability

- For some schedules it is easy to recover from transaction and system failures, whereas for other schedules the recovery process can be quite involved. In some cases, it is even not possible to recover correctly after a failure. Hence, it is important to characterize the types of schedules for which *recovery is possible*, as well as those for which *recovery is relatively simple*.
- Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback.
- But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		
Failure Point				
Commit;				

- A schedule where a committed transaction may have to be rolled back during recovery is called **nonrecoverable schedule.**
- The above table 1 shows a schedule which has two transactions. T1 reads and writes the value of A and that value is read and written by T2. T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1. T2 should also be rollback because it reads the value written by T1, but T2 can't be rollback because it already committed. So this type of schedule is known as **irrecoverable schedule.**
- Schedules in which a transaction commits only after all transactions whose changes it reads commit are called **recoverable schedules.**
- The commit operation of the transaction performing read operation is delayed until the transactions performing write operations commit.

CASCADING ROLLBACKS

- A single transaction failure leads to a series of transaction rollbacks.
- Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable) If T10 fails, T11 and T12 must also be rolled back.
- Can lead to the undoing of a significant amount of work.

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

Cascadeless Schedules: When a transaction is not allowed to read data until the last transaction that has written is committed or aborted. Such schedules are called cascadeless schedules.

TRANSACTION DEFINITION IN SQL

A transaction can be executed implicitly or explicitly.

Following commands are used to control transactions:

1. BEGIN TRANSACTION: It indicates the start point of an explicit or local transaction.

Syntax:

```
BEGIN TRANSACTION transaction_name ;
```

2. SET TRANSACTION: Places a name on a transaction.
Syntax:

```
SET TRANSACTION [ READ WRITE | READ ONLY ];
```

3. COMMIT: If everything is in order with all statements within a single transaction, all changes are recorded together in the database is called **committed**. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

Syntax:

```
COMMIT;
```

4. ROLLBACK: If any error occurs with any of the SQL grouped statements, all changes need to be aborted. The process of reversing changes is called **rollback**. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

Syntax:

```
ROLLBACK;
```

5. SAVEPOINT: creates points within the groups of transactions in which to ROLLBACK.
A SAVEPOINT is a point in a transaction in which you can roll the transaction back to a certain point without rolling back the entire transaction.

Syntax for Savepoint command:

```
SAVEPOINT SAVEPOINT_NAME;
ROLLBACK TO SAVEPOINT_NAME;
```

6. RELEASE SAVEPOINT:- This command is used to remove a SAVEPOINT that you have created.

Syntax:

```
RELEASE SAVEPOINT SAVEPOINT_NAME
```

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the last SAVEPOINT.

Implementation of Isolation Levels

- Isolation Levels define the degree to which a transaction can be isolated from data modifications made by other transactions.

Transaction isolation levels are defined from following phenomena:

Dirty Read – A Dirty read is a situation when a transaction reads data that has not yet been committed.
Non Repeatable read – Non Repeatable read occurs when a transaction reads the same row twice and gets a different value each time. For example, suppose transaction T1 reads data. Due to concurrency, another transaction T2 updates the same data and commit, Now if transaction T1 rereads the same data, it will retrieve a different value.
Phantom Read – Phantom Read occurs when two same queries with same search criterion are executed, but the rows retrieved by the two, are different.

Based on these phenomena, The SQL standard defines four isolation levels :

1. **Read Uncommitted –** Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transactions, thereby allowing dirty reads.
2. **Read Committed –** This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allow dirty read. The transaction holds a read or write lock on the current row, and thus prevents other transactions from reading, updating, or deleting it.
3. **Repeatable Read –** This is the most restrictive isolation level. The transaction holds read and write locks on all rows it references. Since other transactions cannot read, update or delete these rows, consequently it avoids non-repeatable read.
4. **Serializable –** This is the highest isolation level. In this level concurrently executing transactions appears to be serially executing. This level avoids phantom reads by acquiring range locks on the search criterion apart from read and write locks.

Concurrency Control Protocols:

Different concurrency control protocols offer different benefits forachieving serializability and isolation of transactions. They are:

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols

Lock-based Protocols

Lock Based Protocols in DBMS is a mechanism in which a transaction cannot Read or Write the data until it acquires an appropriate lock. Lock based

protocols help to eliminate the concurrency problem in DBMS for simultaneous transactions by locking or isolating a particular transaction to a single user.

A data item can be locked in two modes:

1. Shared(S) lock: It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.

- It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

2. Exclusive(X) lock:

- In the exclusive lock, the data item can be both reads as well as written by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.

Consider the partial schedule

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

- Neither *T3* nor *T4* can make progress — executing **lock-S***(B)* causes *T4* to wait for *T3* to release its lock on *B*, while executing **lock-X***(A)* causes *T3* to wait for *T4* to release its lock on *A*. Such a situation is called a **deadlock**.
- To handle a deadlock one of *T3* or *T4* must be rolled back and its locks released.

THE TWO-PHASE LOCKING PROTOCOL

- This is a protocol which ensures conflict-serializable schedules.

This protocol requires that each transaction issue lock and unlock requests in two phases:

1. Growing phase. A transaction may obtain locks, but may not release any lock.

2. Shrinking phase. A transaction may release locks, but may not obtain any new locks.

- Initially, a transaction is in the growing phase. The transaction acquires locks as needed.
- Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.
- Transactions can be ordered according to their **lock points**—The point in the schedule where the transaction has obtained its final lock.
- Cascading rollback may occur under two-phase locking. can be avoided by a modification of two-phase locking called the **strict two-phase locking protocol**.
- This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits.
- This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.
- Another variant of two-phase locking is the **rigorous two-phase locking protocol**, which requires that all locks be held until the transaction commits.

TIMESTAMP-BASED PROTOCOLS

Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a ***timestamp-ordering*** scheme.

- Each transaction is issued a unique timestamp, TS(Ti) when it enters the system.
- This timestamp is assigned by the database system before the transaction *Ti* starts execution. If a transaction *Ti* has been assigned timestamp TS(*Ti*), and a new transaction *Tj* enters the system, then TS(*Ti*) < TS(*Tj*).

There are two simple methods for implementing this scheme:

1. Use the value of the **system clock** as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.

2. Use a **logical counter** that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

To implement this scheme, we associate with each data item *Q* two timestamp values:

• **W-timestamp**(*Q*) denotes the largest timestamp of any transaction that executed write(*Q*) successfully.
• **R-timestamp**(*Q*) denotes the largest timestamp of any transaction that executed read(*Q*) successfully.
These timestamps are updated whenever a new read(*Q*) or write(*Q*) instruction is executed.

The **timestamp-ordering protocol** ensures that any conflicting read and write operations are executed in timestamp order.
This protocol operates as follows:
1. Suppose that transaction *Ti* issues read(*Q*):
a. If TS(*Ti*) < W-timestamp(*Q*), then *Ti* needs to read a value of *Q* that was already overwritten. Hence, the read operation is rejected, and *Ti* is rolled back.
b. If TS(*Ti*) ≥ W-timestamp(*Q*), then the read operation is executed, and R-timestamp(*Q*) is set to the maximum of R-timestamp(*Q*) and TS(*Ti*).
2. Suppose that transaction *Ti* issues write(*Q*):
a. If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls *Ti* back.
b. If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*. Hence, the system rejects this write operation and rolls *Ti* back.
c. Otherwise, the system executes the write operation and sets W-timestamp(*Q*) to TS(*Ti*).

- If a transaction *Ti* is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

The modification to the timestamp-ordering protocol, called **Thomas' write rule**, is this: Suppose that transaction *Ti* issues write(*Q*).

- If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*. Hence, this write operation can be ignored.

VALIDATION-BASED PROTOCOL

The **validation protocol** requires that each transaction *Ti* executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order:
1. Read phase. During this phase, the system executes transaction *Ti*. It reads the values of the various data items and stores them in variables local to *Ti*. It

performs all write operations on temporary local variables, without updates of the actual database.
2. Validation phase. The validation test (described below) is applied to transaction *Ti* . This determines whether *Ti* is allowed to proceed to the write phase without causing a violation of serializability. If a transaction fails the validation test, the system aborts the transaction.
3. Write phase. If the validation test succeeds for transaction *Ti*, the temporary local variables that hold the results of any write operations performed by *Ti* are copied to the database. Read-only transactions omit this phase.

To perform the validation test, we need to know when the various phases of transactions took place. We shall, therefore, associate three different timestamps with each transaction *Ti* :
1. Start(*Ti*), the time when *Ti* started its execution.
2. Validation(*Ti*), the time when *Ti* finished its read phase and started its validation phase.
3. Finish(*Ti*), the time when *Ti* finished its write phase.

- The serializability order is determined by the timestamp-ordering technique, using the value of the timestamp Validation(*Ti*).
- Thus, the value TS(*Ti*) = Validation(*Ti*) and, if TS(*Tj*) < TS(*Tk*), then any produced schedule must be equivalent to a serial schedule in which transaction *Tj* appears before transaction *Tk*.

The **validation test** for transaction *Ti* requires that, for all transactions *Tk* with TS(*Tk*) < TS(*Ti*), one of the following two conditions must hold:
1. Finish(*Tk*) < Start(*Ti*). Since *Tk* completes its execution before *Ti* started, the serializability order is indeed maintained.
2. The set of data itemswritten by *Tk* does not intersectwith the set of data items read by *Ti*, and *Tk* completes its write phase before *Ti* starts its validation phase (Start(*Ti*) < Finish(*Tk*) < Validation(*Ti*)). This condition ensures that the writes of *Tk* and *Ti* do not overlap.

This validation scheme is called the **optimistic concurrency-control** scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end.