PREFACE

Adda247 brings you the perfect solution for Specialist Officer (IT): Professional Knowledge preparation. Before you lies the book "Professional Knowledge for IBPS SO IT", the basis of which are the modules that incorporate ***all the important topics of Networking, DBMS, Web Technology, Operating System, Data Structure, Computer Organization and Microprocessor, Software Engineering and much more.*** It has been written to fulfill the requirements of all those aspirants who are going to appear for IBPS SO IT, IBPS RRB SO IT, SBI SO IT or any public or private sector Specialist Officer IT Professional Knowledge Examination. This book provides a framework for learning the professional knowledge that is required to get through the any Specialist Officer IT (Professional Knowledge) Examination against such a toilsome competition.

To describe and explain the concepts of information technology, this book uses eleven modules: each with 60 questions of all difficulty levels (Beginner, Moderate and Difficult), three annexes, and five practice sets (based on questions asked in previous SO IT Professional Knowledge Exam) with detailed explanations. This framework allows the aspirants to conquer new information and skills development into a larger context.

Some features associated with this book are:

- Covers all the important topics for SO IT Professional Knowledge Exam in 11 Modules
- Easy Language and representation for better and quick understanding of the topic
- A Set of 60 Questions at the end of each Module that includes questions of varying difficulty level i.e. Beginner, Moderate and Difficult
- Practice Sets with detailed solution based on updated pattern.

We would like to thank our experienced faculties, subject-matter experts and the expertise of Adda247 Team without whose cooperation it wouldn't have been possible to come up with such a meticulous study material for IBPS SO IT Professional Knowledge. We hope that our readers appreciate our strenuous efforts that we put into this book. Any remarks or suggestions for further refinements are wholeheartedly welcome.

Team Adda247

CONTENTS

1 Software & Hardware

1. Introduction

Computer is an electronic machine which can perform arithmetic and logical operations according to the instructions given by the user.
Modern computer is based on analytical engine developed by Charles Babbage. He is considered as father of modern computer. Basic architecture of electronic computer is given by Johan Von Neumann. According to this architecture computer consists of following components.

1. Memory System which stores input data and result before it is output on the screen
2. Input and Output System like Monitor and keyboard
3. CPU (Central processing Unit) with ALU (Arithmetic Logic Unit) and CU(Control Unit)

Thus, we can say computer is a collection of different sub system. All the processing is done in CPU with the help of ALU and CU. Data in computer is represented by 0 and 1 known as binary form. Single 0 or 1 is known as bit and collection of eight bit is known as byte. Byte is a basic unit to represent data in computer
Computer cannot work by its own, it works according to the instruction given by the user. Collection of instruction in proper sequence to perform some task is known as **Software.** Therefore, we can say that computer is made of basic components -

1) Hardware
2) Software

There are different types of computer which has different speed, cost and application. like Micro Computer, mini Computer, Main Frame Computer and Super Computer.
Micro Computer is slowest, cheapest and used in simple application. e.g. PC, Home Computer laptop. Super Computer is fastest, costliest and use in special application like Weather fore casting, scientific research etc.
Computer can also be classified on the basis of their functions like 1) Analog Computer 2) Digital Computer Analog computer is generally used for measurement and Digital Computer is used for calculation. Most of the modern computers are Digital.

Hardware :- Hardware components of the computer consist of all the electronic components and electromechanical devices that comprise the physical devices of computer. The hardware of computer is divided into four major parts:-

1) Central Processing unit(CPU):- It contain an arithmetic and logic unit for manipulating data, a number of registers for storing data, and control unit for fetching and executing instructions.
2) Memory:- It contains data and instruction before execution and after execution. It is call ed random access memory because CPU can access any location randomly without affecting the other location.
3) Input Output Processor(IOP):- It contain electronic circuit for communication and controlling the transfer of information and the outside devices like input output devices
4) Input Output devices:- They are used to give information to the computer and display output like keyboard, printer, visual display unit(VDU).

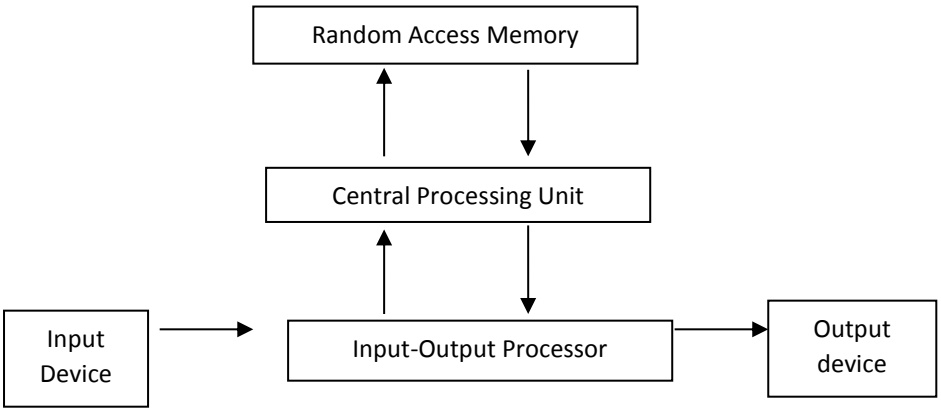


Figure: Block Diagram Of Digital Computer

Logic Gate: All the data in computer is present in binary form i.e. 0 and 1. All the computer components like RAM, ALU, CU and registered are made of Gates. There are different types of gates which perform different function e.g. AND gated, OR gate NOT Gate, etc. Therefore, we can say that logic gate are the main building blocks of digital computer.

Memory System: All the devices which are used to store information in computer is called memory system. Data and instruction is stored in memory before processing i.e. is before going to the CPU. In computer we use different type of memory because some memory are volatile but fast but some are nonvolatile but slow. To maintain the efficiency of computer we use different type of memory. Following is the hierarchy.

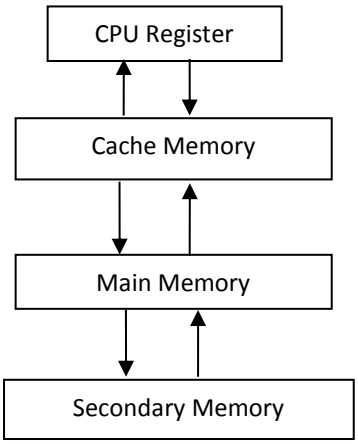


Figure: Memory Hierarchy

CPU Register: - They are small memory location which reside in CPU, they are fast and small. They are generally used to store data and instruction temporarily.
Cache Memory: - It is a fast-used memory between main memory and CPU. It's an electronic memory. It's size is smaller than main memory.
Main Memory: - It is also an electronic memory. All the programs before execution are loaded in main memory. Main memory consists of two parts

a) **RAM (**Random Access Memory**):-** It is read and write memory. It consists of flip-flop. It is volatile memory. There are different type of Ram 1) SRAM 2) DRAM
b) **ROM (**Read only Memory):- It is nonvolatile memory. It stores those programs which are essential to boot the program. It consists of combinational circuit. There are different types of ROM Like PROM,EPROM, EEPROM, Flash memory.

Secondary Memory: - Secondary Memory is a permanent storage device. It is nonvolatile memory, which can store data when there is no power.

The following table highlights the points that differentiate a hardware from a software.

Hardware	Software
It is the physical component of a computer system.	It is the programming language that makes hardware functional.
It has the permanent shape and structure, which cannot be modified.	It can be modified and reused, as it has no permanent shape and structure.
The external agents such as dust, heat etc. can affect the hardware (as it is tangible).	The external agents such as dust, heat etc. cannot affect (as it is not tangible).
It works with binary code (i.e., 1's to 0's) .	It functions with the help of high level language like COBOL, BASIC, JAVA, etc.
It takes in only machine language, i.e., lower level language.	It takes in higher level language, easily readable by a human being.
It is not affected by the computer bug or virus.	It is affected by the computer bug or virus.
It cannot be transferred from one place to other electronically.	It can transfer from one place to other electronically.
Duplicate copy of hardware cannot be created.	A user can create copies of a software as many as he wishes.

Basics of CPU(Central Processing Unit)-Central Processing Unit (CPU) performs all the arithmetic and logical calculations in a computer. The CPU is said to be the brain of the computer system. It reads and executes the program instructions, perform calculations and makes decisions. The CPU is responsible for storing and retrieving information on disks and other media.

The CPU consists of Control Unit, Arithmetic and Logic Unit (ALU) and register set.

- **Control Unit:** The control unit issue control signals to perform specific operation and it directs the entire computer system to carry out stored program instructions
- **Arithmetic and Logic Unit:** The ALU is the 'core' of any processor. It executes all arithmetic operations (addition, subtraction, multiplication and division), logical operations (compare numbers, letters, special characters etc.) and comparison operators (equal to, less than, greater than etc.).
- **Register Set:** Register set is used to store immediate data during the execution of instruction. This area of processor consists of various registers.

2 DBMS

1. Introduction

DBMS is one of the most important module for Specialist Officer (IT) Exam. As we've seen that the objective paper of Professional Knowledge (especially for Scale-I Officer) in IBPS Exam has many questions from Database and Networking Modules. Thus, aspirants should prepare DBMS thoroughly. The term DBMS stands for Data Base Management System. Now comes a question that *what is DBMS?*

DBMS is the acronym of Data Base Management System. DBMS is a collection of interrelated data and a set of programs to access this data in a convenient and efficient way. It controls the organization, storage, retrieval, security and integrity of data in a database.

A database management system (DBMS) is a computer software that manages databases, it may use any of a variety of database models, such as the Hierarchical DBMS, Network DBMS and Relational DBMS.

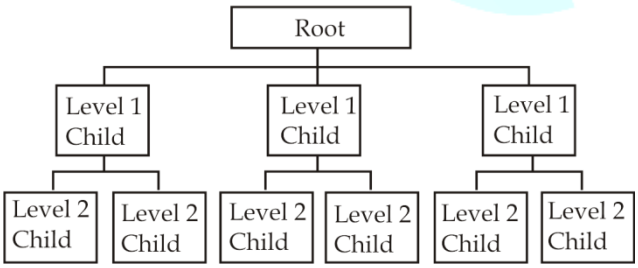
The emergence of the first type of DBMS was between 1960's-70's; that was the Hierarchical DBMS. IBM had the first model, developed on IBM 360 and their (DBMS) was called IMS, originally it was written for the Apollo program. This type of DBMS was based on binary trees, where the shape was like a tree and relations were only limited between parent and child records.

1.1 Database Models

A database model shows the logical structure of a database, including the relationships and constraints that determine how data can be stored and accessed.

Hierarchical Database Model

Hierarchical Database model



Hierarchical Database model is one of the oldest database models. In the hierarchical data model, records are linked with other superior records on which they are dependent and also on the records, which are dependent on them. A tree structure may establish one-to-many relationship. Parents can have many children exhibiting one to many relationships. The grandparents and children are the nodes or dependents of the root. In general, a root may have any number of dependents.

A tree-structure diagram is the schema for a hierarchical database. Such a diagram consists of two basic components:

1. Boxes, which correspond to **record types**
2. Lines, which correspond to **links**

Pros:

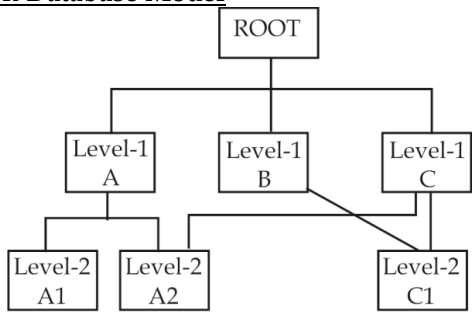
- ✓ The model allows easy addition and deletion of new information.
- ✓ Data at the top of the Hierarchy is very fast to access.
- ✓ It relates well to anything that works through a one to many relationships.

Cons:

- Realtime requirements are of more sophisticated relationships which this model fails to cater.

- The database can be very slow when searching for information on the lower entities.
- Many to many relationships are not supported.

Network Database Model



Network Database Model Diagram

The Network Database model can be viewed as an upside-down tree where each member information is the branch linked to the owner, which is the bottom of the tree. The network database model was a progression from the hierarchical database model and was designed to solve some of that model's problems, specifically the lack of flexibility. It addresses the need to model more complex relationships such as the many-to-many relationship which hierarchical model could not deal with.

The Network model replaces the hierarchical tree with a **graph** thus allowing more general connections among the nodes. The main difference of the network model from the hierarchical model, is its ability to handle many to many (N:N) relations. In other words, **it allows a record to have more than one parent.**

Pros:
- ✓ In the network database terminology, a relationship is a set. Each set comprises of two types of records.- an owner record and a member record,
- ✓ In a network model an application can access an owner record and all the member records within a set.
- ✓ Network Model supports data independence to some level as it draws a clear line of demarcation between programs and the complex physical storage details.

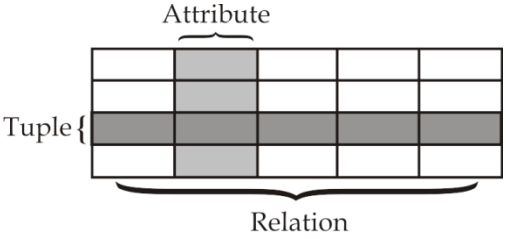
Cons:
- The insertion, deletion and updating operations of any record require large number of pointers adjustments.
- A change in structure demands a change in the application as well, which leads to lack of structural independence.

Relational Database Model

Relational data model is the primary data model, which is used widely around the world for data storage and processing. The relational database model was a huge leap forward from the network database model. Instead of relying on a parent-child or owner-member relationship, the relational model allows any file to be related to any other by means of a common field. Relational databases go hand-in-hand with the development of SQL. Structured Query Language is a standardized language for defining and manipulating data in a relational database.

What are Tables in Relational Model?

Relations are saved in the format of Tables. This format stores the relation among entities. A table has rows and columns, where rows represents records and columns represent the attributes. Following are some terms associated with relations:



1. **Tuple −** A single row of a table, which contains a single record for that relation is called a tuple.

2. **Relation instance −** A finite set of tuples in the relational database system represents relation instance. Relation instances do not have duplicate tuples.
3. **Relation schema −** A relation schema describes the relation name (table name), attributes, and their names.
4. **Relation key −** Each row has one or more attributes, known as relation key, which can identify the row in the relation (table) uniquely.
5. **Attribute domain** − Every attribute has some pre-defined value scope, known as attribute domain.

Object-Oriented Database Model

Object Oriented Database Model (also referred to as object-oriented database management system or OODBMS), is a database management system (DBMS) that supports the modelling and creation of data as objects. This includes some kind of support for classes of objects and the inheritance of class properties and methods by subclasses and their objects. ODBMS were originally thought of to replace RDBMS because of their better fit with object-oriented programming languages. However, high switching cost, the inclusion of object-oriented features in RDBMS to make them ORDBMS, and the emergence of object-relational mappers (ORMs) have made RDBMS successfully defend their dominance in the data center for server-side persistence.

Relational databases store data in tables that are two dimensional. The tables have rows and columns. Relational database tables are "normalized" so data is not repeated more often than necessary. With traditional databases, data manipulated by the application is transient and data in the database is persisted (Stored on a permanent storage device). In object databases, the application can manipulate both transient and persisted data.

Entity-Relationship Database Model

The Entity Relationship Data Model ensure that you get a precise understanding of the nature of the data and how it is used by the enterprise, you need to have a universal model for interaction that is non-technical and free of ambiguities and easy readable to both technical as well as non-technical members. ***This is implemented with use of the ER Diagrams.***

ER model is based on two concepts:

- Entities, defined as tables that hold specific information (data)
- Relationships, defined as the associations or interactions between entities

What is Entity Relationship Diagram (ER-Diagram)?

ER-Diagram is a pictorial representation of data that describes how data is communicated and related to each other. Any object, such as entities, attributes of an entity, sets of relationship and other attributes of relationship can be characterized with the help of the ER diagram.

1.2 Advantages of today's' DBMS over earlier File Management System

These are some important advantages of today's DBMS:

Reduced Data Redundancy and Inconsistency: This means with DBMS the chances of multiple file formats, duplication of information in different files got eliminated. Which means it reduced data duplication and with this the data could stay more consistent.

Data Integrity: data integrity" refers to the accuracy and consistency of data stored in a database
DBMS ensures data integrity by managing transactions through **ACID** test = atomicity, consistency, isolation, durability. While such integrity is absent in file management system.

Sharing of Data: In DBMS, data can be shared by authorized users of the organization. The database administrator manages the data and gives rights to users to access the data.

Control Over Concurrency: In a file-based system, if two users can access data simultaneously, it is possible that they will interfere with each other. For example, if both users attempt to perform update operation on the same record, then one may overwrite the values recorded by the other. Most database management systems have sub-systems to control the concurrency so that transactions are always recorded with accuracy.

Backup and Recovery Procedures: In a computer file-based system, the user creates the backup of data regularly to protect the valuable data from damage due to failures to the computer system or application program. It is very time

consuming method, if amount of data is large. Most of the DBMSs provide the 'backup and recovery' sub-systems that automatically create the backup of data and restore data if required.

Data Independence: The separation of data structure of database from the application program that uses the data is called data independence. In DBMS, you can easily change the structure of database without modifying the application program.

2. Database Architecture

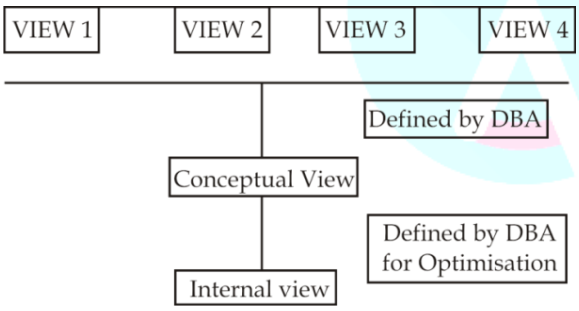
2.1 Data Abstraction – View Levels

The generalized architecture of DBMS is called ANSI/SPARC model. The architecture is divided into three levels:

1. External view or User view/View Level
It is the highest level of data abstraction. This includes only those portions of database of concern to a user or Application program. Each user has a different external view and it is described by means of a scheme called external schema.

2. Conceptual view/Logical Level
All the database entities and the relationship among them are included. One conceptual view represents the entire database called conceptual schema.

3. Internal view/Physical Level
It is the lowest level of abstraction, closest to the physical storage method. It describes how the data is stored, what is the structure of data storage and the method of accessing these data. It is represented by internal schema.
View Level …Defined by User



2.2 Instances and Schemas

Schema can be defined as the design of a database. The overall description of the database is called the database schema. You can relate it as something like types and variables in programming languages. Thus, essentially **Schema** is the logical structure of the database. Just like the View Levels in Data Abstraction Schema is of 3 types:

1. Physical Schema:
The design of a database at physical level is called physical schema, how the data stored in blocks of storage is described at this level.

2. Logical schema:
Logical schema can be defined as the design of database at logical level. In this level, the programmers as well as the database administrator (DBA) work. At this level data can be described as certain types of data records which can be stored in the form of data structures. However, the internal details (such as implementation of data structure) will be remaining hidden at this level.

3. View Schema
View schema can be defined as the design of database at view level which generally describes end-user interaction with database systems.

- **Physical Data Independence**–the ability to modify the physical schema without changing the logical schema.
- Applications depend on the logical schema
- In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.

What is an Instance?
Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an instance.

2.3 Database Languages

A database system provides a data-definition language to specify the database schema and a data-manipulation language to express database queries and updates.

1. Data Definition Language: DDL is used for specifying the database schema. It contains commands to create tables, alter the structure, delete tables or rename tables.

Examples of DDL commands in SQL:
To create the database instance – CREATE
To alter the structure of database – ALTER
To drop database instances – DROP
To delete tables in a database instance – TRUNCATE
To rename database instances – RENAME

2. Data Manipulation Language: As the name specifies itself DML is used for accessing and manipulating data in a database.

Examples of DML commands in SQL:
To read records from table(s) – SELECT
To insert records into the tables – INSERT
Update the data in tables– UPDATE
Delete all the records from the table – DELETE

3. Data Control Language: DCL is used for granting and revoking user access on a database –

Examples of DCL commands in SQL:
To grant access to user – GRANT
To revoke access from user – REVOKE

3. Entity-Relationship Model

What is an Entity?
In a database, we would be grouping only related data together and storing them under one group name called Entity / Table. This helps in identifying which data is stored where and under what name. It reduces the time to search for a particular data in a whole database.
Entities can be classified based on their strength. An entity is considered weak if its tables are existence dependent. Following are basic types of entities:

1. **Strong Entity**: Entities having its own attribute as primary keys are called strong entity. For example, EMPLOYE has EMPLOYE_ID as primary key. Hence it is a strong entity.
2. **Weak Entity:** Entities which cannot form their own attribute as primary key are known weak entities. These entities will derive their primary keys from the combination of its attribute and primary key from its mapping entity. The relationship between weak entity and strong entity set is called as ***Identifying Relationship***.
3. **Composite Entity:** Entities participating in the many to many relationships are called composite entity.

The relationship between weak entity and strong entity set is called as Identifying Relationship. The line connecting strong entity set with the relationship is single whereas the line connecting weak entity set with the identifying relationship is double. A member of a strong entity set is called dominant entity and member of weak entity set is called

as subordinate entity. A weak entity set does not have a primary key, but we need a means of distinguishing among all those entries in the entity set that depend on one particular strong entity set. The discriminator of a weak entity set is a set of attributes that allows this distinction be made. A weak entity set is represented by doubly outlined box and corresponding identifying relation by a doubly outlined diamond. It is also called as the Partial key of the entity set.

Weak Entity Sets

Weak Entity Set: An entity set whose members owe their existence to some entity in a *strong entity set.*

- entities are not of independent existence.
- each weak entity is associated with some entity of the *owner* entity set through a special relationship.
- weak entity set may not have a key attribute.

Weak Entity set- Example

We depict a weak entity set by double rectangles. We underline the discriminator of a weak entity set with a dashed line.

For example: payment_number – discriminator of the *payment* entity set

Primary key for *payment* – (*loan_number, payment_number*)

Note: the primary key of the strong entity set is not explicitly stored with the weak entity set, since it is implicit in the identifying relationship. If *loan_number* were explicitly stored, *payment* could be made a strong entity, but then the relationship between *payment* and *loan* would be duplicated by an implicit relationship defined by the attribute *loan_number* common to *payment* and *loan*.

3.1 Attributes

Each entity is described by a set of attributes/properties.

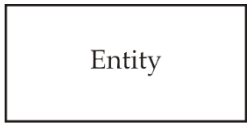
Types of Attributes

- Simple Attributes: having atomic or indivisible values. example: Dept – a string Phone Number - an eight-digit number
- Composite Attributes: having several components in the value. example: Qualification with components (Degree Name, Year, University Name)
- Derived Attributes: Attribute value is dependent on some other attribute. example: Age depends on Date of Birth. So, age is a derived attribute.
- Single-valued: having only one value rather than a set of values. for instance, Place Of Birth - single string value.
- Multi-valued: having a set of values rather than a single value. for instance, Courses Enrolled attribute for student Email Address attribute for student Previous Degree attribute for student.
- Attributes can be: simple single-valued, simple multi-valued, composite single-valued or composite multi-valued.

3.2 E-R Diagram

An ER diagram is a means of visualizing how the information a system produces is related. There are five main components of an ER Diagram:

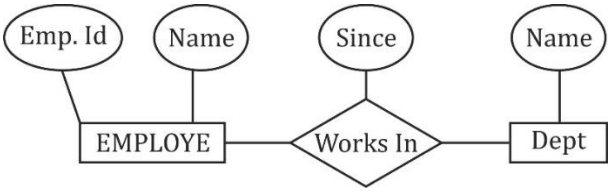
1. **Connecting lines, solid lines that connect attributes to show the relationships of entities in the diagram.**
2. **Entities: Represented by Rectangle**
   - Strong Entity: These shapes are independent from other entities, and are often called parent entities, since they will often have weak entities that depend on them.



   - Weak Entity: A weak entity is an entity that must defined by a foreign key relationship with another entity as it cannot be uniquely identified by its own attributes alone.

3. Relationship: connects two or more entities into an association/relationship - Diamond

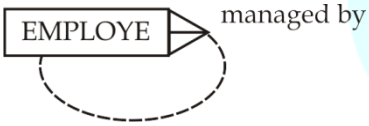
Here you can see: Employee Works in Department. EMPLOYE and Dept are Entity Types and WorksIn is the relationship represented with a diamond figure.



A **recursive relationship** is one in which the same entity participates more than once in the relationship. For Example: Every manager is also an employee. So, manager is not a new entity,
but just a subset of the instances of the entity EMPLOYEE.

Recessive Relationship:

EMPLOYE	**MANAGER**
Nikhil	Anuj
Sumita	Nikhil
Anuj	
Rahul	



Manages of

This also a representation of - many cardinality.

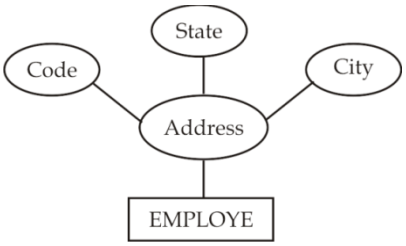
3.3 Attributes-Represented by Ovals

An Attribute describes a property or characteristic of an entity. For example, Name, ID, Age, Address etc can be attributes of an EMPLOYE.

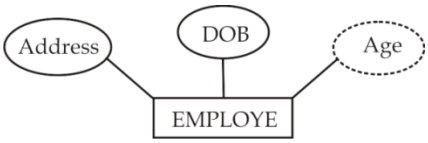
Key attribute represents the main characteristic of an Entity. It is used to represent Primary key. Ellipse with underlying lines represent Key Attribute. Here EmpId is the key attribute that is the primary key which will uniquely identify the EMPLOYE Records.

Double Ellipses is used to represent multivalued attributes.

An attribute can also have their own attributes. These attributes are known as **Composite attribute.**



Derived Attribute is calculated or otherwise derived from another attribute, such as age from a DOB (Date of Birth).

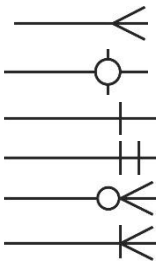


4. Cardinality

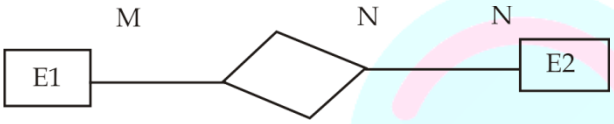
The cardinality of a relationship is the number of instances of entity B that can be associated with entity A. There is a minimum cardinality and a maximum cardinality for each relationship. Cardinality refers to the maximum number of times an instance in one entity can relate to instances of another entity. Ordinality, on the other hand, is the minimum number of times an instance in one entity can be associated with an instance in the related entity.

Cardinalit

Many
Zero or one
One
One (and only one)
Zero or many
One or many



Binary Relationships and Cardinality Ratio



- The number of entities from *E2* that an entity from *E1* can possibly be associated thru *R* (and vice-versa) determines the *cardinality ratio* of *R*.
- Four possibilities are usually specified:
  1. *one-to-one (1:1)*
  2. *one-to-many (1:N)*
  3. *many-to-one (N:1)*
  4. *many-to-many (M:N)*

Cardinality Ratios

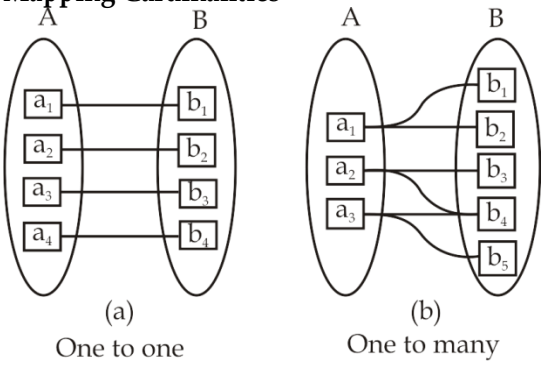
One-to-one: An *E1* entity may be associated with at most one *E2* entity and similarly an *E2* entity may be associated with at most one *E1* entity.

One-to-many: An *E1* entity may be associated with many *E2* entities whereas an *E2* entity may be associated with at most one *E*1 entity.

Many-to-one: … (similar to above)

Many-to-many: Many *E1* entities may be associated with a single *E2* entity and a single *E1* entity may be associated with many *E2* entities.

Mapping Cardinalities

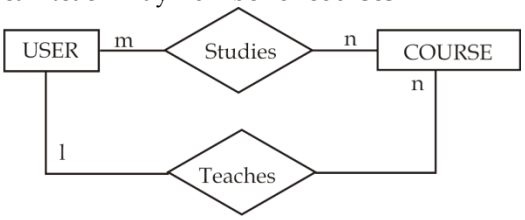


Note: Some elements in A and B may not be mapped to any elements in the other set.

Many to one Many to many

Note: Some elements in A and B may not be mapped to any elements in the other set.

Alternative Representation of Cardinality
Many to Many Relationship b/w User and course meaning any number of users can study or enroll in any number of course and these is one to many relationship b/w a teacher (which is also a user) and course meaning only one instructor can teach may number of courses



In one department we have many employees so the following represents – one to many relationships

And in case any number of employee may work in any number of department (many to many)

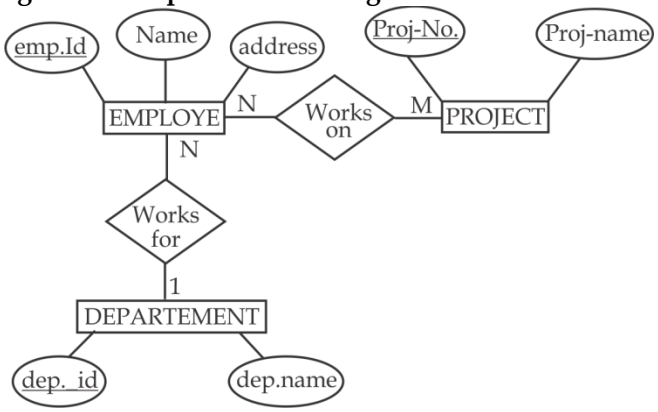
5. Keys
A ***super key*** of an entity set is a set of one or more attributes whose values uniquely determine each entity. A ***candidate key*** of an entity set is a minimal super key

For Example: Customer-id is candidate key of *customer*
account-number is candidate key of *account*
Although several candidate keys may exist, one of the candidate keys is selected to be the ***primary key***.

Keys for Relationship Sets
The combination of primary keys of the participating entity sets forms a super key of a relationship set.
(*customer-id, account-number*) is the super key of *depositor*

- *NOTE: this means a pair of entity sets can have at most one relationship in a particular relationship set.*
- E.g. if we wish to track all access-dates to each account by each customer, we cannot assume a relationship for each access. We can use a multivalued attribute though.
- Must consider the mapping cardinality of the relationship set when deciding the what are the candidate keys.
- Need to consider semantics of relationship set in selecting the *primary key* in case of more than one candidate key.

Following is an example of an ER Diagram:



Entity- Relationship (E-R) Diagram
The overall logical structure of a database can be expressed graphically by an E-R diagram. The diagram consists of the following major components.

- Rectangles: represent entity set.

- Ellipses: represent attributes.
- Diamonds: represents relationship sets.
- Lines: links attribute set to entity set and entity set to relationship set.
- Double ellipses: represent multi-valued attributes.
- Dashed ellipses: denote derived attributes.
- Double lines: represent total participation of an entity in a relationship set.
- Double rectangles: represent weak entity sets.

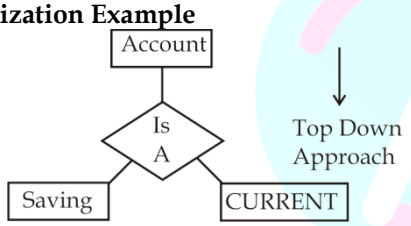
3.3 Specialization, Generalization and Aggregation

Generalization is a bottom-up approach in which two lower level entities combine to form a higher-level entity. In generalization, the higher-level entity can also combine with other lower level entity to make further higher-level entity. Specialization is opposite to Generalization. It is a top-down approach in which one higher level entity can be broken down into two lower level entities.

Top-down design process; we designate subgroupings within an entity set that are distinctive from other entities in the set.

- These subgroupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.
- Depicted by a *triangle* component labelled ISA (E.g. *customer* "is a" *person*).
- **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.

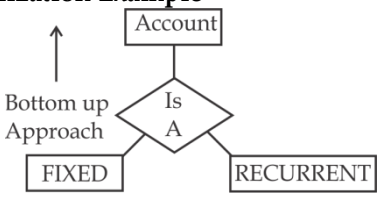
Specialization Example



Generalization

- A bottom-up design process – combine a number of entity sets that share the same features into a higher-level entity set.
- Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.
- The terms specialization and generalization are used interchangeably.

Generalization Example



Specialization and Generalization

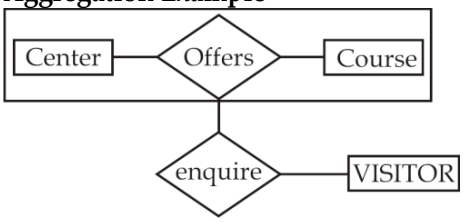
- Can have multiple specializations of an entity set based on different features.
- E.g. *permanent-employee* vs. *temporary-employee*, in addition to *officer* vs. *secretary* vs. *teller*
- Each particular employee would be a member of one of *permanent-employee* or *temporary-employee*, and also a member of one of *officer*, *secretary*, or *teller*
- The IS-A relationship also referred to as **superclass - subclass** relationship.

Aggregation

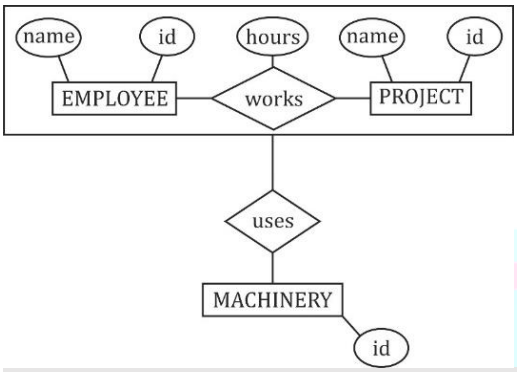
Aggregation is a process when relation between two entities is treated as a single entity. It is an abstraction that treats relationships as entities.

- Eliminate this redundancy via *aggregation*
- Treat relationship as an abstract entity
- Allows relationships between relationships
- Abstraction of relationship into new entity

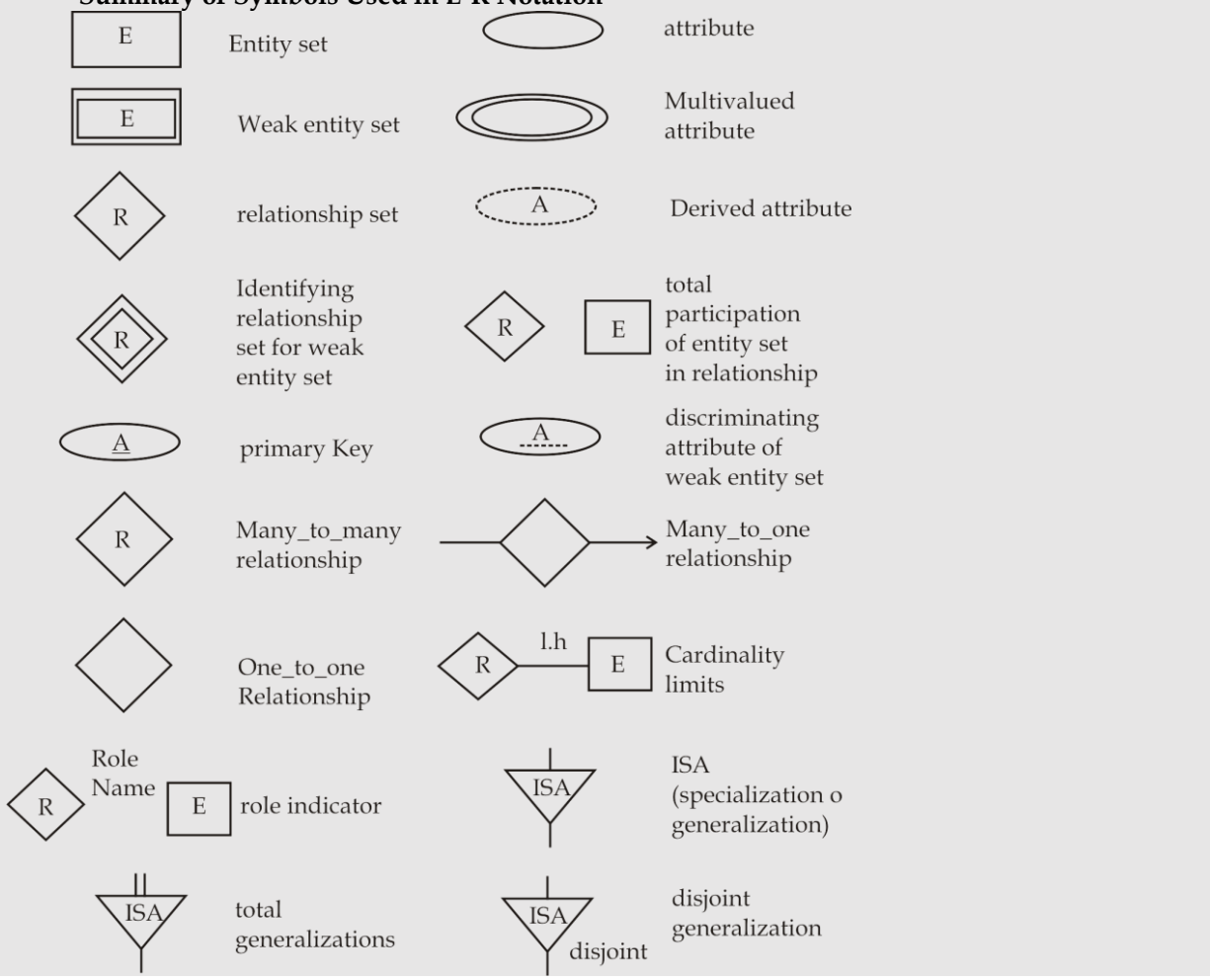
Aggregation Example



ER Diagram with Aggregation



Summary of Symbols Used in E-R Notation



4. Relational Database Management System

The relational model for database management is a data model based on predicate logic and set theory. It was invented by Edgar Codd. The fundamental assumption of the relational model is that all data are represented as mathematical n-ary **relations**, an n-ary relation being a subset of the Cartesian product of n sets.

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

1) Relation - The fundamental organizational structure for data in the relational model is the relation. A relation is a two-dimensional table made up of rows and columns. Each relation also called a table, stores data about entities.
2) Tuples - The rows in a relation are called tuples. They represent specific occurrences (or records) of an entity. Each row consists of a sequence of values, one for each column in the table. In addition, each row (or record) in a table must be unique. A tuple variable is a variable that stand for a tuple.
3) Attributes – The column in a relation is called attribute. The attributes represent characteristics of an entity.
4) Domain – For each attribute there is a set of permitted values called domain of that attribute. For all relations 'r', the domain of all attributes of 'r' should be atomic. A domain is said to be **atomic** if elements of the domain are considered to be indivisible units.

Database Schema – Logical design of the database is termed as database schema.
Database instance – Database instance is a snapshot of the data in a database at a given instant of time.
Relation schema – The concept of relation schema corresponds to the programming notion of type definition. It can be considered as the definition of a domain of values. The database schema is the collection of relation schemas that define a database.
Relation instance – The concept of a relation instance corresponds to the programming language notion of a value of a variable. For relation instance, we actually mean the "relation" itself.

4.1 Database Keys

1. **Primary Key:** Which uniquely identifies a record in a table. Student_ID is the primary key in this STUDENT Table.

2. **Candidate Key:** A candidate key is a single field or the least combination of fields that uniquely identifies each record in the table. Every table must have at least one candidate key but at the same time can have several. For Example in the table STUDENT, Student_ID and Roll_No. Are Candidate keys.

Roll_No.	Student_ID
001	11093100
002	11093101
003	11093126
004	11093127

3. **Foreign Key:** A foreign key is generally a primary key from one table that appears as a field in another. For Example let us consider these two table STUDENT and LIBRARY_RECORD.

STUDENT

Roll_No.	Student_ID	Student_Name	Student_Class
001	11093100	Ravi Kumar	3
002	11093101	Nihal Sharma	4
003	11093126	Astha Mathur	3
004	11093127	Nishi Arora	5

LIBRARY_RECORD

Lib_CardNo	Student_ID	Student_Name	Address
AX120	11093101	Nihal Sharma	12th Avenue Street, Delhi
AX121	11093126	Astha Mathur	XYZ Lane,Delhi

In the table LIBRARY_RECORD Lib_CardNo. Is the Primary key and Student_ID is the foreign key as it is the primary key of the table STUDENT.

4. **Alternate Key:** The candidate key other than primary key is called as alternate key.
5. **Super Key:** The set of attributes which can uniquely identify a tuple is known as Super Key. For Example, Student_Enroll_No, (Student_ID, Student_Name) etc.

Non-key attributes are attributes other than candidate key attributes in a table. And Non-prime Attributes are attributes other than Primary attribute.

4.2 Relational Query Languages

Relational query languages use relational algebra to break the user requests and instruct the DBMS to execute the requests. It is the language by which user communicates with the database. These relational query languages can be procedural or non-procedural.

- In a procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result.

- In a nonprocedural language, the user describes the desired information without giving a specific procedure for obtaining that information.

4.3 Relational Algebra

Relational algebra is a procedural query language. It takes one or more relations / tables and performs the operation and produce the result. This result is also considered as a new table or relation. Specifically, since the result of a relational query is itself a relation, relational operations can be applied to the results of queries as well as to the given set of relations.

An operator can be either unary or binary. Following are some operations of relational algebra:

1. **Selection operator (σ):** Selection operator is used to select tuples from a relation based on some condition.
   Syntax: $\sigma_{(Cond)}$(Relation Name)

   Example

   Extract employees whose age is greater than 30 from EMPLOYEES relation
   $\sigma_{(AGE>30)}$(EMPLOYEES)

2. **Project Operation (∏) :** It projects column(s) that satisfy a given predicate.
   Syntax: $\prod_{(Column\ 1,Column\ 2....Column\ n)}$(Relation Name)

   Example

   Extract EMP_ID and NAME from EMPLOYEE relation.
   $\prod_{(EMP\_ID,NAME)}$(EMPLOYEE)

3. **Union Operation (∪):** It performs binary union between two given relations. Union on two relations R1 and R2 can only be computed if R1 and R2 are union compatible (These two relation should have same number of attributes and corresponding attributes in two relations have same domain) . Duplicate tuples are automatically eliminated in union operation.
   Syntax: Relation1 U Relation2
   $r \cup s = \{ t \mid t \in r \text{ or } t \in s\}$
   Note: r, and s must have the same number of attributes.

Example

Projects the names of th Employees who are Managers in IT_Dept or Managers in FUNC_Dept or Both
∏ Managers (IT_Dept) ∪ ∏ Managers(FUNCT_Dept)

4. **Minus (-):** Minus on two relations R1 and R2 can only be computed if R1 and R2 are union compatible. Minus operator when applied on two relations as R1-R2 will give a relation with tuples which are in R1 but not in R2.
Syntax: Relation1 - Relation2

Example

Find person who are student but not employee, we can use minus operator like:
∏ Name (STUDENTS) − ∏ Name (EMPLOYEE)

5. **Rename(ρ):** Rename operator is used to give another name to a relation.
Syntax: ρ(Relation2, Relation1)

Example

To rename STUDENT relation to STUDENT1, we can use rename operator like:
ρ(STUDENT1, STUDENT)

6. **Cartesian Product (X):** The cartesian product of two tables combines each row in one table with each row in the other table. It combines tuples from two relations, but unlike the join operation, its result contains all pairs of tuples from the two relations, regardless of whether their attribute values match.
Syntax: r X s
Where r and s are relations and their output will be defined as −
r X s = { q t | q ∈ r and t ∈ s}

Tuple Relational Calculus

Relational calculus is a non-procedural query language. It uses mathematical predicate calculus instead of algebra. It provides the description about the query to get the result whereas relational algebra gives the method to get the result.

A query in the tuple relational calculus is expressed as: {t | P(t)}
i.e. the set of tuples for which predicate is true.

{t | EMPLOYEE (t) and t.SALARY>20000} - implies that it selects the tuples from EMPLOYEE relation such that resulting employee tuples will have salary greater than 20000. It is example of selecting a range of values.

Domain Relational Calculus

In the tuple relational calculus, you have use variables that have series of tuples in a relation. In the domain relational calculus, you will also use variables but in this case the variables take their values from domains of attributes rather than tuples of relations. An domain relational calculus expression has the following general format –

{d1, d2, . . . , dn | F(d1, d2, . . . , dm)} m ≥ n
where d1, d2, . . . , dn, . . . , dm stand for domain variables and F(d1, d2, . . . , dm) stands for a formula composed of atoms.

For example, select EMP_ID and EMP_NAME of employees who work for department ID 415

{<EMP_ID, EMP_NAME> | <EMP_ID, EMP_NAME> ? EMPLOYEE Λ DEPT_ID = 415}

5. Normalization

Normalization is a process of organizing the data in database to avoid data redundancy, insertion anomaly, update anomaly & deletion anomaly. Thus, database normalization is a database schema design technique, by which an existing schema is modified to minimize redundancy and dependency of data.

What are Anomalies in Database Management?

Anomalies are inconvenient or error-prone situation arising when we process the tables. There are three types of anomalies:

1. **Update Anomalies:** Incorrect data may have to be changed, which could involve many records having to be changed, leading to the possibility of some changes being made incorrectly.
2. **Delete Anomalies:** A record of data can legitimately be deleted from a database, and the deletion can result in the deletion of the only instance of other, required data, E.g. Deleting a book loan from a library member can remove all details of the particular book from the database such as the author, book title etc
3. **Insert Anomalies:** The nature of a database may be such that it is not possible to add a required piece of data unless another piece of unavailable data is also added. E.g. A library database that cannot store the details of a new member until that member has taken out a book.

5.1 Functional Dependency

We use functional dependencies to: test relations to see if they are legal under a given set of functional dependencies. If a relation *r* is legal under a set *F* of functional dependencies, we say that *r* satisfies *F.*

We say that *F* holds on *R* if all legal relations on *R* satisfy the set of functional dependencies *F.*

Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances. For example, a specific instance of *loan* may, by chance, satisfy *amount →customer_name.*

A functional dependency is trivial if it is satisfied by all instances of a relation

Example*:*

customer_name, loan_number →customer_name

customer_name→customer_name

Inference rules

Armstrong's axioms - sound and complete i.e, enable the computation of any functional dependency. Functional dependencies are:

1. Reflexivity - if the B's are a subset of the A's then A → B
2. Augmentation - If A → B, then A, C → B, C.
3. Transitivity - If A → B and B → C then A → C.

Additional inference rules

4. Decomposition - If A → B, C then A → B
5. Union - If A → B and A → C then A → B, C
6. Pseudo transitive - If A → B and C, B → D then C, A → D

Equivalence of sets of functional dependencies

Two functional dependencies S & T are equivalent iff S→ T and T → S.

The dependency {A_1, ..., A_n} → {B_1, ..., B_m} is trivial if the B's are a subset of the A's is nontrivial if at least one of the B's is not among the A's is completely nontrivial if none of the B's is also one of the A's

Closure (F^+)

All dependencies that include F and that can be inferred from F using the above rules are called closure of F denoted by F^+.

After finding a set of functional dependencies that are hold on a relation, the next step is to find the Super key for that relation (table). The set of identified functional dependencies play a vital role in finding the key for the relation. We can decide whether an attribute (or set of attributes) of any table is a key for that table or not by identifying the attribute or set of attributes' closure. If A is an attribute, (or set of attributes) then its attribute closure is denoted as A+.

Algorithm to compute closure
We have to find out whether $F \models X \rightarrow Y$. This is the case when $X \rightarrow Y \in F^+$
The better method is to generate X^+, closure of X under F and test $F \models X \rightarrow Y$ using the first two axioms augmentation and reflexive rules.

Example:
EMPLOYEE(empid, empname, dept, age, salary, experience)
Let the functional dependencies be as follows:

empid->empname
{age,experience}->salary
empid->{age,dept}
dept->experience

In above example, let us find the closure of the attribute empid, i.e, closure of {empid}

Since we are finding closure of empid. empid is an element of the closure set C^+. Now we go step by step.

- **Step 1:** Select each functional dependency and check whether the left side of functional dependency is a subset of closure. If yes, add the right side of that functional dependency to closure set. if not, check the next functional dependency

- **Step 2:** Keep on checking the functional dependencies until there is no more functional dependencies with its left side as a subset of closure set C^+.

What is a subset? A set M is said to be a subset of another set N only if all elements of set M is present in set N. Set N can have more elements than M.

- So, in our example, empid is an element of the closure set C^+. So, initially, C^+={empid}.
- First functional dependency says empid functionally determines empname. Its left side ({empid}) is subset of C^+. Therefore, its right side is added to C^+. Now C^+={empid, empname}.
- Second fd (functional dependency) says {age, experience}->salary. Here left side ({age, experience}) is not a subset of C^+. So we check the next fd.
- Third fd says, empid->{age, dept}. Here left side ({empid}) is subset of C^+. Therefore, its right side is added to C^+. Now, C becomes, C^+={empid, empname, age, dept}.
- Fourth fd says, dept->experience. Here left side ({dept}) is a subset of C^+. So we are adding its right side ({experience}) to Closure set. Now, C^+={empid, empname, age, dept, experience}.
- We are looking again for a functional dependency with its left side as a subset of closure set. Since the closure set C^+ is getting changed in some steps, there is more possibility to find another functional dependency with its left side as a subset of C^+. Again, we go through every functional dependency.
- Since sets do not allow duplication, we should do nothing if the right side of a functional dependency whose left side is subset of C^+, is already present in closure set C^+.
- Second fd has a left side that is subset of C+. {age, experience}->salary. Therefore, salary is added to C^+. Now, C^+={empid, empname, age, dept, experience, salary}.
- There isn't any more functional dependency whose left side is subset of C+ and give at least one new attribute to closure set. Therefore, we stop now.

Now closure of set C is C+={empid, empname, age, dept, experience, salary}.
Minimal Cover of FD

We say that a set of functional dependencies F covers another set of functional dependencies G, if every functional dependency in G can be inferred from F. More formally, F covers G if $G+ \subseteq F^+$. F is a minimal cover of G if F is the smallest set of functional dependencies that cover G.

We find the minimal cover by iteratively simplifying the set of functional dependencies. To do this, we will use three methods:

Simplifying an FD by the Union Rule: Let X, Y , and Z be sets of attributes. If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow Y Z$

Simplifying an FD by simplifying the left-hand side: Let X and Y be sets of attributes, and B be a single attribute not in X.

Let F be: XB → Y
and H be X → Y
If F ⇒ X → Y , then we can replace F by H. In other words, if Y ⊆ X^+_F, then we can replace F by H.
For example, say we have the following functional dependencies (F):
• AB → C
• A → B
And we want to know if we can simplify to the following (H):
• A → C
• A → B
Then A^+_F = ABC. Since, Y ⊆ X^+_F, we can replace F by H.

Simplifying an FD by simplifying the right-hand side: Let X and Y be sets of attributes, and C be a single attribute not in Y .
Let F be:
X → Y C and H be

X → Y
If H ⇒ X → Y C, then we can replace F by H. In other words, if Y C ⊆ X^+_H, then we can replace F by H.
For example, say we have the following functional dependencies (F):
• A → BC
• B → C
And we want to know if we can simplify to the following (H):
• A → B
• B → C
Then A^+_H = ABC. Since, BC ⊆ X^+_H, we can replace F by H.

Finding the Minimal Cover
Given a set of functional dependencies F :
1. Start with F
2. Remove all trivial functional dependencies
3. Repeatedly apply (in whatever order you like), until no changes are possible
• Union Simplification (it is better to do it as soon as possible, whenever possible)
• RHS Simplification
• LHS Simplification
4. Result is the minimal cover

Example: Applying to algorithm to EGS with
1. E → G
2. G → S
3. E → S
Using the union rule, we combine 1 and 3 and get
1. E → GS
2. G → S
Simplifying RHS of 1 (this is the only attribute we can remove), we get:
1. E → G
2. G → S

Algorithm to find minimal cover for a set of FDs F
Step 1: Let G be the set of FDs obtained from F by decomposing the
right hand sides of each FD to a single attribute.
Step 2: Remove all redundant attributes from the left hand sides of FDs in G.
Step 3: From the resulting set of FDs, remove all redundant FDs.
Output the resulting set of FDs.

Example: Consider R = ABCDEFGH and the following set of FDs, F:
ABH → C

A → D
C → E
BGH → F
F → AD
E → F
BH → E
Converting right hand sides to single attributes, we get:
ABH → C
A → D
C → E
BGH → F
F → A
F → D
E → F
BH → E
Perform steps 2 and 3....

Understanding Normalization

In relational database theory, normalization is the process of restructuring the logical data model of a database to eliminate redundancy, organize data efficiently, and reduce repeating data and to reduce the potential for anomalies during data operations. Data normalization also may improve data consistency and simplify future extension of the logical data model. The formal classifications used for describing a relational database's level of normalization are called normal forms (**NF**).

A non-normalized database can suffer from data anomalies: A non-normalized database may store data representing a particular referent in multiple locations. An update to such data in some but not all of those locations results in an update anomaly, yielding inconsistent data. A normalized database prevents such an anomaly by storing such data (i.e. data other than primary keys) in only one location.

A non-normalized database may have inappropriate dependencies, i.e. relationships between data with no functional dependencies. Adding data to such a database may require first adding the unrelated dependency. A normalized database prevents such insertion anomalies by ensuring that database relations mirror functional dependencies.
Similarly, such dependencies in non-normalized databases can hinder deletion. That is, deleting data from such databases may require deleting data from the inappropriate dependency. A normalized database prevents such deletion anomalies by ensuring that all records are uniquely identifiable and contain no extraneous information.

Normal forms

Edgar F. Codd originally defined the first three normal forms. The first normal form requires that tables be made up of a primary key and a number of atomic fields, and the second and third deal with the relationship of non-key fields to the primary key. These have been summarized as requiring that all non-key fields be dependent on "the key, the whole key and nothing but the key". In practice, most applications in 3NF are fully normalized. However, research has identified potential update anomalies in 3NF databases. BCNF is a further refinement of 3NF that attempts to eliminate such anomalies. The fourth and fifth normal forms (4NF and 5NF) deal specifically with the representation of many-many and one-many relationships. Sixth normal form (6NF) only applies to temporal databases.

5.2 First Normal Form (1NF)

First normal form (1NF) lays the groundwork for an organized database design: **Ensure that each table has a primary key: minimal set of attributes which can uniquely identify a record.** It states that the domain of an attribute must include only atomic values and the value of any attribute in a tuple must be single value from the domain of that attribute. It doesn't allow nested relation. Data that is redundantly duplicated across multiple rows of a table is moved out to a separate table.

Atomicity: *Each attribute must contain a single value, not a set of values.*
Unnormalized form (UNF): A table that contains one or more repeating groups.
First normal form (1NF): A relation in which the intersection of each row and column contains one and only one value.
UNF → 1NF: remove repeating groups:

- Entering appropriate data in the empty columns of rows.
- Placing repeating data along with a copy of the original
- key attribute in a separate relation. Identifying a primary
- key for each of the new relations.

First normal form (1NF) lays the groundwork for an organized database design: Ensure that each table has a primary key: minimal set of attributes which can uniquely identify a record. It states that the domain of an attribute must include only atomic values and the value of any attribute in a tuple must be single value from the domain of that attribute. It doesn't allow nested relation. Data that is redundantly duplicated across multiple rows of a table is moved out to a separate table.

For Example:
Consider a table STUDENT with fields Roll_No, Name, Course. Here a student may have opted for more than one courses thus the values in Course field will not be atomic:

Roll_No	Name	Course
1	Snehal	Polity, History, Economics
2	Kajal	DBMS, CD
3	Amit	Physics, Chemistry

After converting it to First Normal Form (1NF)

Roll_No	Name	Course
1	Snehal	Polity
1	Snehal	History
1	Snehal	Economics
2	Kajal	DBMS
2	Kajal	CD
3	Amit	Physics
3	Amit	Chemistry

5.3 Second Normal Form (2NF)

General Definition: A relation schema R is in second normal form (2NF) if every nonprime attribute A in R is not partially dependent on any key of R.
Partial Dependency – If proper subset of candidate key determines non-prime attribute, it is called partial dependency.

- Create separate tables for sets of values that apply to multiple records.
- Relate the tables with a foreign key.
- Records should not depend on anything other than a table's primary key (a compound key, if necessary). For example, consider a customer's address in an accounting system. The address is needed by the Customers table, but also by the Orders, Shipping, Invoices, Accounts Receivable, and Collections tables. Instead of storing the customer's address as a separate entry in each of these tables, store it in one place, either in the Customers table or in a separate Addresses table**.**

A table is said to be in 2NF if both the following conditions hold:

- Table is in 1NF (First normal form)
- No non-prime attribute is dependent on the proper subset of any candidate key of table.

1NF → 2NF: remove partial dependencies: the functionally dependent attributes are removed from the relation by placing them in a new relation along with a copy of their determinant.

As per First Normal Form, no two Rows of data must contain repeating group of information i.e each set of column must have a unique value, such that multiple columns cannot be used to fetch the same row. The Primary key is usually a single column, but sometimes more than one column can be combined to create a single primary key which is actually called a Candidate Key. To identify or establish 2NF we must identify Candidate Key and Partial Dependencies.

Example: Let us consider a Relation R with fields A,B,C,D :
R(A B C D)
Here AB-> D and B-> C.

A and B are the essential attributes here as from A and B you can find D and from B you can find C. Thus Candidate Key for this relation R is AB.
$(AB)^+$ = ABCD
AB-> Candidate Key
Hence, AB ∈ Prime Attribute - those who are a part of candidate key
CD ∈ Non-Prime Attribute
Identify Partial Dependency: Here C is only dependent on B and not the complete candidate key set AB, thus in this relation R there is a partial dependency.
Now for 2NF a relation must be in 1NF and there should not be any partial dependency. So we'll elimination partial dependency by creating another Relation R_1 and R_2:
R_1 (A B D)
AB->D
R_2 (B C)
B->C
In this scenario we have eliminated partial dependency as B is the only candidate key and C is dependent on B.

5.4 Third Normal Form (3NF)

For 3NF, first, the table must be in 2NF, plus, we want to make sure that the non-key fields are dependent upon ONLY the PK, and not on any other field in the table. This is very similar to 2NF, except that now you are comparing the non-key fields to OTHER non-key fields.

3NF Rule:
1. The database must meet all the requirements of the second normal form.
2. Any field which is dependent not only on the primary key but also on another field is moved out to a separate table. (No Transitive Dependencies)

Example:
STUDENT(Stu_ID, Stu_Name, City, Zip)

We find that in the above STUDENT 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:

Student_Detail (Stu_ID, Stu_Name, Zip)
ZipCode(Zip, City)

General Definition:
A relation schema R is in 3NF if, whenever a nontrivial functional dependency X→A
holds in R,
Either a) X is a Super key Or b) Y is a prime attribute of R.
i.e. A relation schema R is in 3NF if every nonprime attribute of R meets both of the following terms:
1. It is fully functionally dependent on every key of R.
2. It is non-transitively dependent on every key of R.

5.5 Boyce-Codd Normal Form (BCNF)

A row is in BCNF if and only if every determinant is a candidate key. The second and third normal forms assume that all attributes not part of the candidate keys depend on the candidate keys but does not deal with dependencies within the keys.

BCNF deals with such dependencies.
A relation R is said to be in BCNF if whenever ***X -> A*** *holds in R, and A is not in X, then X is a candidate key for R.*
BCNF covers very specific situations where 3NF miss interdependencies between non-key attributes. It should be noted that most relations that are in 3NF are also in BCNF.

Infrequently, a 3NF relation is not in BCNF and this happens only if

(a) the candidate keys in the relation are composite keys (that is, they are not single attributes),
(b) there is more than one candidate key in the relation, and
(c) the keys are not disjoint, that is, some attributes in the keys are common.
The BCNF differs from the 3NF only when there are more than one candidate keys and the keys are composite and overlapping. Consider for example, the relationship *enrol (sno, sname, cno, cname, date-enrolled)*
Let us assume that the relation has the following candidate keys:
(sno, cno)
(sno, cname)
(sname, cno)
(sname, cname)
(we have assumed *sname* and *cname* are unique identifiers). The relation is in 3NF but not in BCNF because there are dependencies
sno -> sname
cno -> cname
where attributes that are part of a candidate key are dependent on part of another candidate key. Such dependencies indicate that although the relation is about some entity or association that is identified by the candidate keys.
e.g. *(sno, cno)*, there are attributes that are not about the whole thing that the keys identify. For example, the above relation is about an association (enrolment) between students and subjects and therefore the relation needs to include only one identifier to identify students and one identifier to identify subjects. Providing two identifiers about students *(sno, sname)* and two keys about subjects *(cno, cname)* means that some information about students and subjects that is not needed is being provided. This provision of information will result in repetition of information and the anomalies. If we wish to include further information about students and courses in the database, it should not be done by including the information in the present relation but by creating new relations that represent information about entities *student* and *subject*.
These difficulties may be overcome by decomposing the above relation in the following three relations:
(sno, sname)
(cno, cname)
(sno, cno, date-of-enrolment)
We now have a relation that only has information about students, another only about subjects and the third only about enrolments. All the anomalies and repetition of information have been removed.

5.6 Multivalued Dependency and Fourth Normal Form (4NF)

In a relational model, if all of the information about an entity is to be represented in one relation, it will be necessary to repeat all the information other than the multivalue attribute value to represent all the information that we wish to represent. This results in many tuples about the same instance of the entity in the relation and the relation having a composite key (the entity id and the multivalued attribute). Of course, the other option suggested was to represent this multivalued information in a separate relation. The situation of course becomes much worse if an entity has more than one multivalued attributes and these values are represented in one relation by a number of tuples for each entity instance. The multivalued dependency relates to this problem when more than one multivalued attributes exist. Consider the following relation that represents an entity employee that has one multivalued attribute *proj*:
emp (e#, dept, salary, proj)

We have so far considered normalization based on functional dependencies; dependencies that apply only to single-valued information. For example, *e#* -> *dept* implies only one *dept* value for each value of *e#*. Not all information in a database is single-valued, for example, *proj* in an employee relation may be the list of all projects that the employee is currently working on. Although *e#* determines the list of all projects that an employee is working on, *e#* -> *proj* is not a functional dependency.

We can more clearly analyze the multivalued dependency by the following example.
programmer (emp_name, qualifications, languages)
This relation includes two multivalued attributes of entity *programmer*; *qualifications* and *languages*. There are no functional dependencies.

The attributes qualifications and languages are assumed independent of each other. If we were to consider *qualifications* and *languages* as separate entities, we would have two relationships (one between *employees* and *qualifications* and the other between *employees* and programming *languages*). Both the above relationships are many-to-many i.e. one programmer could have several qualifications and may know several programming languages. Also one qualification may be obtained by several programmers and one programming language may be known to many programmers.

Functional dependency *A* -> *B* relates one value of *A* to one value of *B* while multivalued dependency *A* ->> *B* defines a relationship in which a set of values of attribute *B* are determined by a single value of *A*.

Now, more formally, *X* ->> *Y* is said to hold for *R(X, Y, Z)* if *t1* and *t2* are two tuples in *R* that have the same values for attributes *X* and therefore with *t1[x] = t2[x]* then *R* also contains tuples *t3* and *t4* (not necessarily distinct) such that
t1[x] = t2[x] = t3[x] = t4[x]
t3[Y] = t1[Y] and *t3[Z] = t2[Z]*
t4[Y] = t2[Y] and *t4[Z] = t1[Z]*
In other words if *t1* and *t2* are given by
t1 = [X, Y1, Z1], and
t2 = [X, Y2, Z2]
then there must be tuples *t3* and *t4* such that
t3 = [X, Y1, Z2], and
t4 = [X, Y2, Z1]

We are therefore insisting that every value of *Y* appears with every value of *Z* to keep the relation instances consistent. In other words, the above conditions insist that *X alone determines Y and Z* and there is no relationship between *Y* and *Z* since *Y* and *Z* appear in every possible pair and hence these pairings present no information and are of no significance.

Fourth Normal Form
Fourth normal form (or 4NF) requires that **there must be no non-trivial multivalued dependencies of attribute sets on something other than a superset of a candidate key.** A table is said to be in 4NF if and only if it is in the BCNF and multivalued dependencies are functional dependencies. The 4NF removes unwanted data structures: multivalued dependencies.

Definition: A relation schema R is in 4NF with respect to a set of dependencies F, if, for every non-trivial multivalued dependency X ->>Y in F+, X is a super key for R.

5.7 Properties of Relational Decompositions

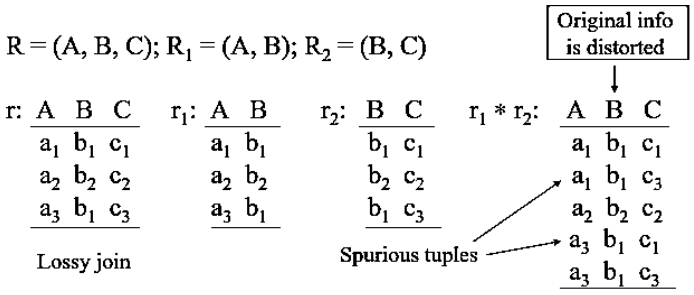
If R doesn't satisfy a particular normal form, we decompose R into smaller schemas
What's a decomposition?
$R = (A_1, A_2, \ldots, A_n)$
$D = (R_1, R_2, \ldots, R_k)$ st $R_i \subseteq R$ and $R = R_1 \cup R_2 \cup \ldots \cup R_k$
(R_i's need not be disjoint)
Replacing R by $R_1, R_2, \ldots, R_k$ – process of decomposing R
Ex: gradeInfo (rollNo, studName, course, grade)
R_1: gradeInfo (rollNo, course, grade)
R_2: studInfo (rollNo, studName)

Decomposition Property: A relation must satisfy the following two properties during decomposition.
i. Lossless: A lossless-join dependency is a property of decomposition, which ensures that spurious rows are generated when relations are united through a natural join operation. i.e. The information in an instance r of R must be preserved in the instances r1, r2, r3, …..rk where ri = ΠRi (r)
Decomposition is lossless with respect to a set of functional dependencies F if, for every relation instance r on R satisfying F, $r = \pi_{R_1}(r) * \pi_{R_2}(r) * \ldots * \pi_{R_k}(r)$

Lossless join property
F – set of FDs that hold on R
R – decomposed into $R_1, R_2, \ldots, R_k$
Decomposition is lossless wrt F if for every relation instance r on R satisfying F,

Lossless joins are also called non-additive joins

$r = \pi_{R_1}(r) * \pi_{R_2}(r) * \ldots * \pi_{R_k}(r)$



ii. Dependency Preserving Property: If a set of functional dependencies hold on R it should be possible to enforce F by enforcing appropriate dependencies on each r_1 .
Decomposition D= (R_1, R_2, R_3, ……, R_k) of schema R preserves a set of dependencies
F if,
(π R1 (F) U π R2 (F) U . . . . . π Rn (F)) +=F+
π Ri(F) is the projection of F onto Ri.
i.e Any FD that logically follows from F must also logically follows from the union of projection of F onto Ri 'S . Then D is called dependency preserving.

Dependency Preserving Decompositions
Decomposition D = (R_1, R_2, …, R_k) of schema R preserves a set of dependencies F if
$\left(\pi_{R_1}(F) \cup \pi_{R_2}(F) \cup ... \cup \pi_{R_k}(F)\right)^+ = F^+$
Here, $\pi_{R_i}(F) = \{(X \rightarrow Y) \in F^+ \mid X \subseteq R_i, Y \subseteq R_i\}$
(called projection of F onto R_i)
Informally, any FD that logically follows from F must also logically follow from the union of projections of F onto R_i's Then, D is called dependency preserving.

Join Dependency
Join dependency is the term used to indicate the property of a relation schema that cannot be decomposed losslesly into two relations schema, but can be decomposed losslesly into three or more simpler relation schema. It means that a table, after it has been decomposed into three or more smaller tables must be capable of being joined again on common keys to form the original table.

Algorithm for BCNF decomposition
R – given schema. F – given set of FDs
D = {R} // initial decomposition
while there is a relation schema R_i in D that is not in BCNF do
{let X → A be the FD in R_i violating BCNF;
Replace R_i by $R_{i1} = R_i - \{A\}$ and $R_{i2} = X \cup \{A\}$ in D;}

Decomposition of R_i is lossless as
$R_{i1} \cap R_{i2} = X$, $R_{i2} - R_{i1} = A$ and X → A
Result: a lossless decomposition of R into BCNF relations

5.8 Fifth Normal Form

Fifth normal form (5NF also called PJ/NF) requires that there are no non-trivial join dependencies that do not follow from the key constraints. A table is said to be in the 5NF if and only if it is in 4NF and the candidate keys imply every join dependency in it.

Goals of Normalization
- Let *R* be a relation scheme with a set *F* of functional dependencies.
- Decide whether a relation scheme *R* is in "good" form.
- In the case that a relation scheme *R* is not in "good" form, decompose it into a set of relation scheme {*R1, R2, ..., R n*} such that each relation scheme is in good form the decomposition is a lossless-join decomposition Preferably, the decomposition should be dependency preserving.

'A relation R is in fifth normal form (5NF) – also called projection-join normal form (PJ/NF) if and only if every join dependency in R is a consequence of the candidate keys of R.'

For every normal form it is assumed that every occurrence of R can be uniquely identified by a primary key using one or more attributes in R.

For R to be in 5th Normal Form it should follow the following rules:

1. It should be in 4NF and
2. If Join Dependency does not exist
   else
   If Join Dependency exits then it should be Trivial in Nature
   else
   If all the Ri are Super Key; R-> (R_1, R_2, R_3 ... R_i)

- **Join dependencies** generalize multivalued dependencies lead to **project-join normal form (**PJNF) (also called **fifth normal form**)
- A class of even more general constraints, leads to a normal form called **domain key normal form**.
- Problem with these generalized constraints: are hard to reason with, and no set of sound and complete set of inference rules exists. Hence rarely used.

In short, Normalization of a Database is achieved by following a set of rules called 'forms' in creating the database.
These rules are 5 in number (with one extra one stuck in-between 3&4) and they are:

- **1st Normal Form or 1NF:** Each Column Type is Unique.
- **2nd Normal Form or 2NF:** The entity under consideration should already be in the 1NF and all attributes within the entity should depend solely on the entity's unique identifier.
- **3rd Normal Form or 3NF:** The entity should already be in the 2NF and no column entry should be dependent on any other entry (value) other than the key for the table. If such an entity exists, move it outside into a new table.
- Now if these 3NF are achieved, the database is considered normalized. But there are three more 'extended' NF for the elitist. These are:
- **BCNF (Boyce & Codd):** The database should be in 3NF and all tables can have only one primary key.
- **4NF:** Tables cannot have multi-valued dependencies on a Primary Key.
- **5NF:** There should be no cyclic dependencies in a composite key.