1.**Introduction to DBMS**
DBMS | Introduction, Evolution and Advantages
2.**ER and Relational Models**
DBMS | Architectures
DBMS | ER - Model
DBMS | Relational Model
DBMS | Keys in Relational Model
3.**Database Design(Normal Forms)**
Database Normalization
Database Normalization | Functional Dependencies
DBMS | Normal Forms
4.**File structures**
Indexing in Databases
Introduction of B-Tree
Introduction of B+ Tree
5.**Transactions and concurrency control**

DBMS | Introduction, Evolution and Advantages

Important Terminologies

Database: Database is a collection of inter-related data which helps in efficient retrieval, insertion and deletion of data from database and 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.

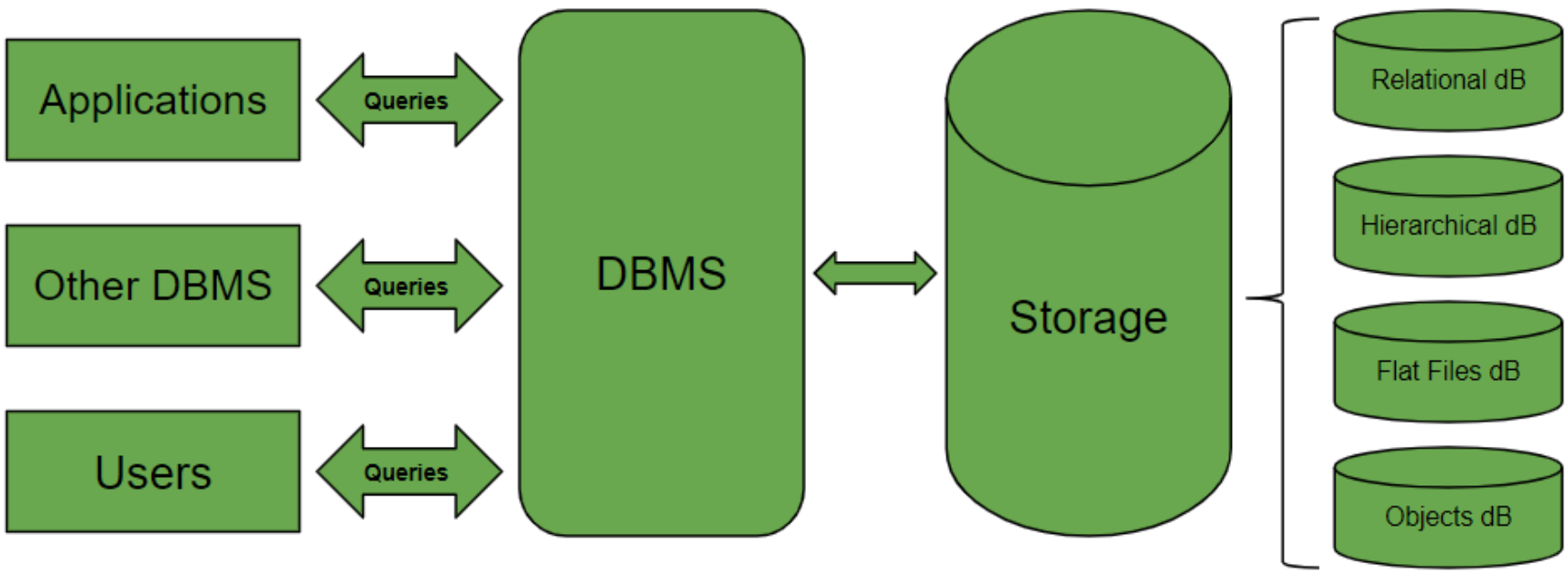
Database Management System: The software which is used to manage the database is called Database Management System (DBMS). For Example, MySQL, Oracle, etc. are popular commercial DBMS used in different applications. DBMS allows users the following tasks:

Data Definition: It helps in creation, modification, and removal of definitions that define the organization of data in the database.

Data Updation: It helps in insertion, modification and deletion of the actual data in the database.

Data Retrieval: It helps in retrieval of data from the database which can be used by applications for various purposes.

User Administration: It helps in registering and monitoring users, enforcing data security, monitoring performance, maintaining data integrity, dealing with concurrency control and recovering information corrupted by unexpected failure.



The history of DBMS evolved in three primary phases:

1. File-based System

2. Relational DBMS

3. NoSQL

Paradigm Shift from File System to DBMS

A File Management system is a DBMS that allows access to single files or tables at a time. In a File System, data is directly stored in a set of files. It contains flat files that have no relation to other files (when only one table is stored in a single file, then this file is known as flat file). File System manages data using files in the hard disk. Users are allowed to create, delete, and update the files according to their requirements. Let us consider the example of a file-based University Management System. Data of students is available to their respective Departments, Academics Section, Result Section, Accounts Section, Hostel Office, etc. Some of the data is common for all sections like Roll No, Name, Father Name, Address and Phone number of students but some data is available to a particular section only like Hostel allotment number which is a part of hostel office. Let us discuss the issues with this system:

- **Redundancy of data:** Data is said to be redundant if same data is copied at many places. If a student wants to change a Phone number, he has to get it updated at various sections. Similarly, old records must be deleted from all sections representing that student.
- **Inconsistency of Data:** Data is said to be inconsistent if multiple copies of the same data do not match with each other. If a Phone number is different in the Accounts Section and Academics Section, it will be inconsistent. Inconsistency may be because of typing errors or not updating all copies of same data.
- **Difficult Data Access:** A user should know the exact location of the file to access data, so the process is very cumbersome and tedious. If a user wants to search student hostel allotment number of a student from 10000 unsorted students' records, how difficult it can be.
- **Unauthorized Access:** File System may lead to unauthorized access to data. If a student gets access to file having his marks, he can change it in unauthorized way.
- **No Concurrent Access:** The access of same data by multiple users at same time is known as concurrency. File system does not allow concurrency as data can be accessed by only one user at a time.
- **No Backup and Recovery:** File system does not incorporate any backup and recovery of data if a file is lost or corrupted.

These are the main reasons which made a shift from file system to Relational DBMS.

Relational DBMS

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

STUDENT

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 relationships that we will learn later. The relational DBMS provides us with Structured Query Language or SQL which is a standard Database language that is used to create, maintain and retrieve the relational database. Following are some interesting facts about SQL.

- SQL is case insensitive. But it is a recommended practice to use keywords (like SELECT, UPDATE, CREATE, etc) in capital letters and use user-defined things (liked table name, column name, etc) in small letters.

- We can write comments in SQL using "–" (double hyphen) at the beginning of any line.

- SQL is the programming language for relational databases (explained below) like MySQL, Oracle, Sybase, SQL Server, Postgre, etc. Other non-relational databases (also called NoSQL) databases like MongoDB, DynamoDB, etc do not use SQL

- Although there is an ISO standard for SQL, most of the implementations slightly vary in syntax. So we may encounter queries that work in SQL Server but do not work in MySQL.

NoSQL

A NoSQL originally referring to non SQL or non-relational is a database that provides a mechanism for storage and retrieval of data. This data is modelled in means other than the tabular relations used in relational databases. NoSQL databases are used in real-time web applications and big data and their use are increasing over time. NoSQL systems are also sometimes called Not only SQL to emphasize the fact that they may support SQL-like query languages. A NoSQL database includes simplicity of design, simpler horizontal scaling to clusters of machines and finer control over availability. The data structures used by NoSQL databases are different from those used by default in relational databases which makes some operations faster in NoSQL. The suitability of a given NoSQL database depends on the problem it should solve. Data structures used by NoSQL databases are sometimes also viewed as more flexible than relational database tables. Types of NoSQL databases and the name of the databases system that falls in that category are:

1. MongoDB falls in the category of NoSQL document-based database.
2. Key value store: Memcached, Redis, Coherence
3. Tabular: Hbase, Big Table, Accumulo
4. Document based: MongoDB, CouchDB, Cloudant

Advantages of DBMS

DBMS helps in efficient organization of data in database which has following advantages over typical file system.

- **Minimized redundancy and data consistency:** Data is normalized in DBMS to minimize the redundancy which helps in keeping data consistent. For Example, student information can be kept at one place in DBMS and accessed by different users.
- **Simplified Data Access:** A user need only name of the relation not exact location to access data, so the process is very simple.
- **Multiple data views:** Different views of same data can be created to cater the needs of different users. For Example, faculty salary information can be hidden from student view of data but shown in admin view.
- **Data Security:** Only authorized users are allowed to access the data in DBMS. Also, data can be encrypted by DBMS which makes it secure.
- **Concurrent access to data:** Data can be accessed concurrently by different users at same time in DBMS.
- **Backup and Recovery mechanism:** DBMS backup and recovery mechanism helps to avoid data loss and data inconsistency in case of catastrophic failures.

DBMS | Architectures

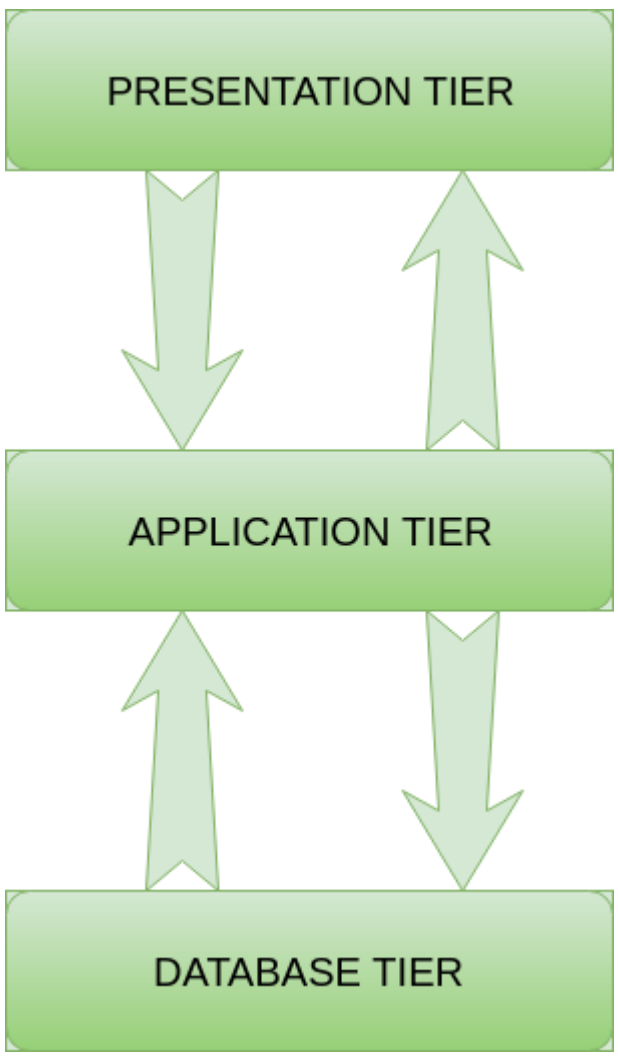
Two Level Architecture

Two level architecture is similar to a basic **client-server** model. The application at the client end directly communicates with the database at the server side. API's like ODBC,JDBC are used for this interaction. The server side is responsible for providing query processing and transaction management functionalities. On the client side, the user interfaces and application programs are run. The application on the client side establishes a connection with the server side in order to communicate with the DBMS.

An **advantage** of this type is that maintenance and understanding are easier, compatible with existing systems. However, this model gives poor performance when there are a large number of users.

DBMS 3-tier Architecture

DBMS 3-tier architecture divides the complete system into three inter-related but independent modules as shown in Figure below.



Presentation or User Tier: This tier is presented before the user of who knows nothing regarding the other existing complex databases underneath. This tier can provide multiple views of the databases, generated by the application residing in the application tier.

Application Tier: This is the middle lined tier between the Database and the Presentation tier and acts as a connection between the end-user and the database. This tier holds the programs and the application server along with the programs that could access the database. This is the last layer that the users are made aware of. It provides a complete abstraction to the database layer.

Database Tier: This tier holds the primary database along with all the data and the query languages for processing. The relations of data with there constraints are also defined in this level.

Advantages:

- **Enhanced scalability** due to distributed deployment of application servers. Now,individual connections need not be made between client and server.
- **Data Integrity** is maintained. Since there is a middle layer between client and server, data corruption can be avoided/removed.
- **Security** is improved. This type of model prevents direct interaction of the client with the server thereby reducing access to unauthorized data.

Disadvantages:
Increased complexity of implementation and communication. It becomes difficult for this sort of interaction to take place due to presence of middle layers.

Data Independence

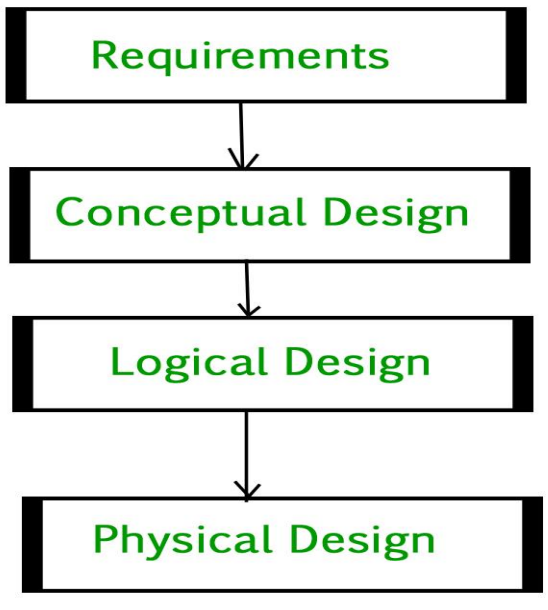
Data independence means a change of data at one level should not affect another level. Two types of data independence are present in this architecture:

Physical Data Independence: Any change in the physical location of tables and indexes should not affect the conceptual level or external view of data. This data independence is easy to achieve and implemented by most of the DBMS.

Conceptual Data Independence: The data at conceptual level schema and external level schema must be independent. This means, change in conceptual schema should not affect external schema. e.g.; Adding or deleting attributes of a table should not affect the user's view of table. But this type of independence is difficult to achieve as compared to physical data independence because the changes in conceptual schema are reflected in user's view.

Phases of database design

Database designing for a real world application starts from capturing the requirements to physical implementation using DBMS software which consists of following steps shown in Figure.



Conceptual Design: The requirements of database are captured using high level conceptual data model. For Example, ER model is used for conceptual design of database.

Logical Design: Logical Design represents data in the form of relational model. ER diagram produced in the conceptual design phase is used to convert the data into the Relational Model.

Physical Design: In physical design, data in relational model is implemented using commercial DBMS like Oracle, DB2.

DBMS | ER - Model

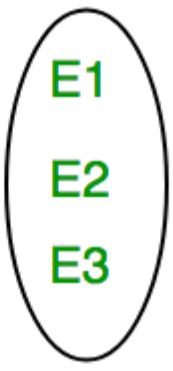
ER Model is used to model the logical view of the system from the data perspective which consists of these components:

Entity, Entity Type, Entity Set

An **Entity** may be an object with a physical existence - a particular person, car, house, or employee - or it may be an object with a conceptual existence - a company, a job, or a university course.

An Entity is an object of **Entity Type** and set of all entities is called as **Entity Set**. e.g.; E1 is an entity having Entity Type Student and set of all students is called Entity Set. In ER diagram, Entity Type is represented as:

Entity Type

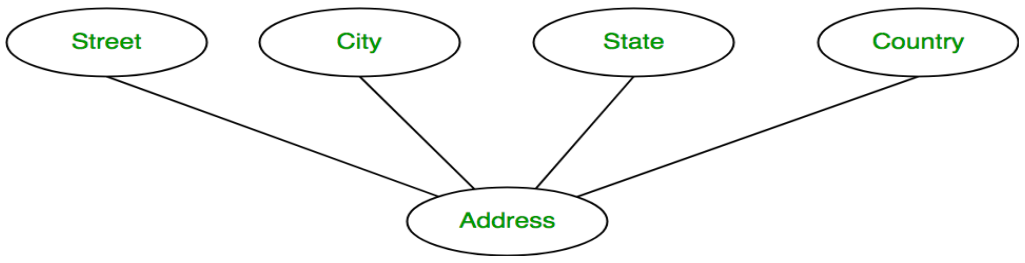


Entity Set

Attribute(s) Attributes are the **properties which define the entity type**. For example, Roll_No, Name, DOB, Age, Address, Mobile_No are the attributes which defines entity type Student. In ER diagram, attribute is represented by an oval.

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

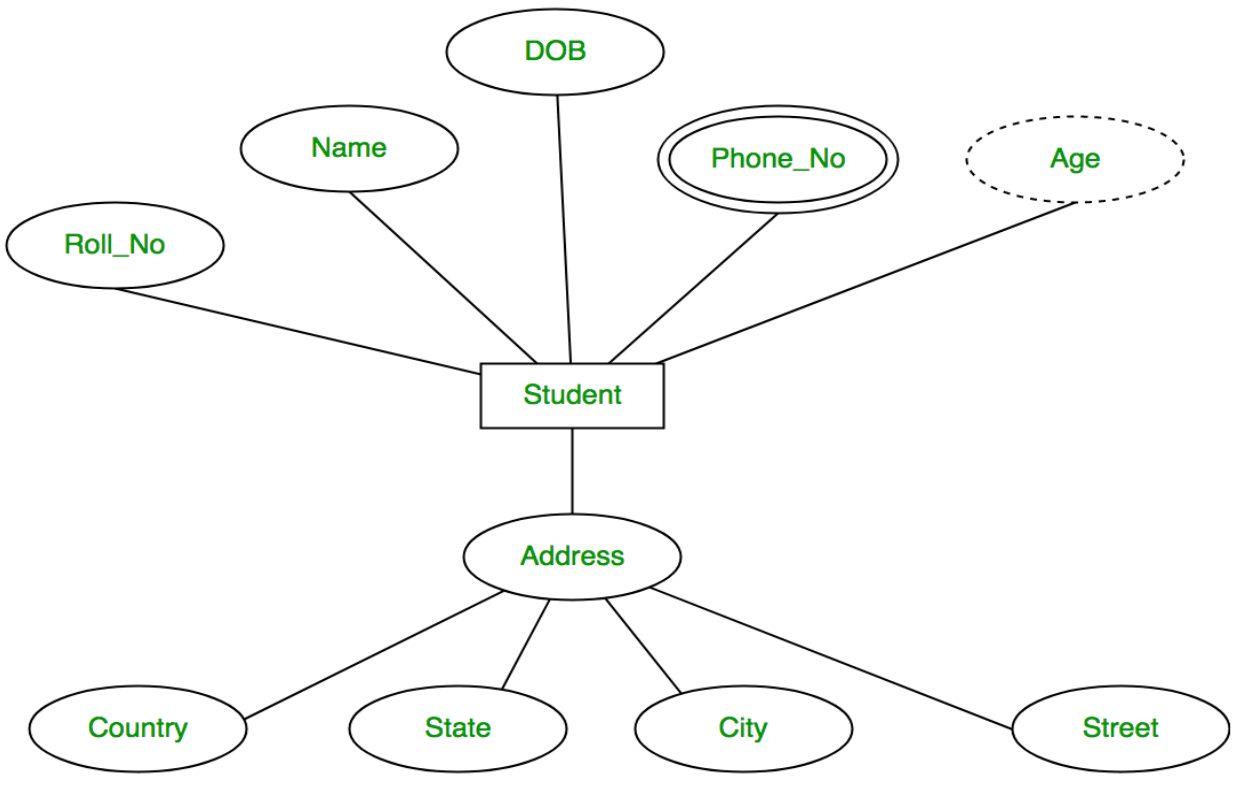
2. **Composite Attribute -** An attribute **composed of many other attribute** is called as composite attribute. For example, Address attribute of student Entity type consists of Street, City, State, and Country. In ER diagram, composite attribute is represented by an oval comprising of ovals.



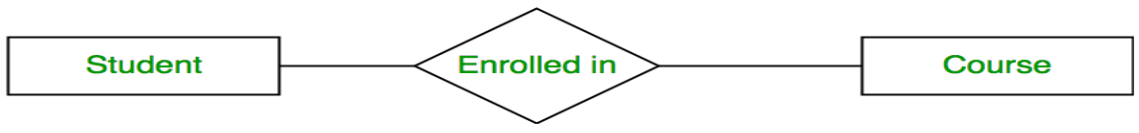
3. **Multivalued Attribute -** An attribute consisting **more than one value** for a given entity. For example, Phone_No (can be more than one for a given student). In ER diagram, multivalued attribute is represented by double oval.

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

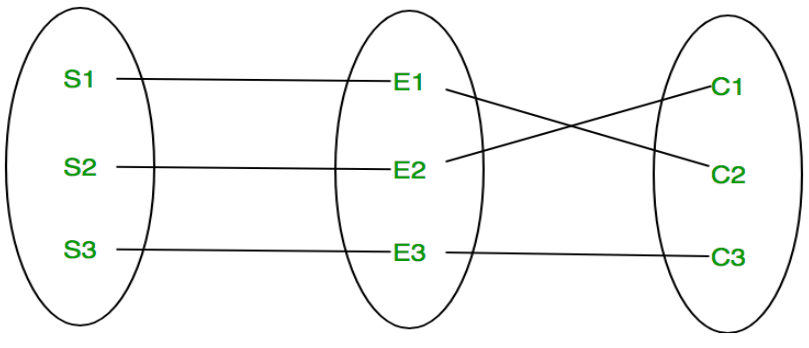
The complete entity type **Student** with its attributes can be represented as:



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

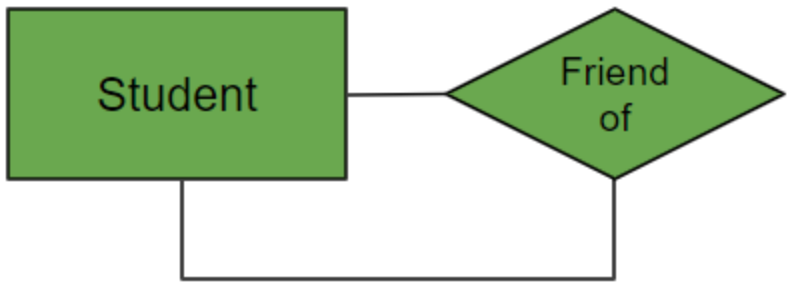


A set of relationships of the same type is known as a relationship set. The following relationship set depicts S1 is enrolled in C2, S2 is enrolled in C1 and S3 is enrolled in C3.

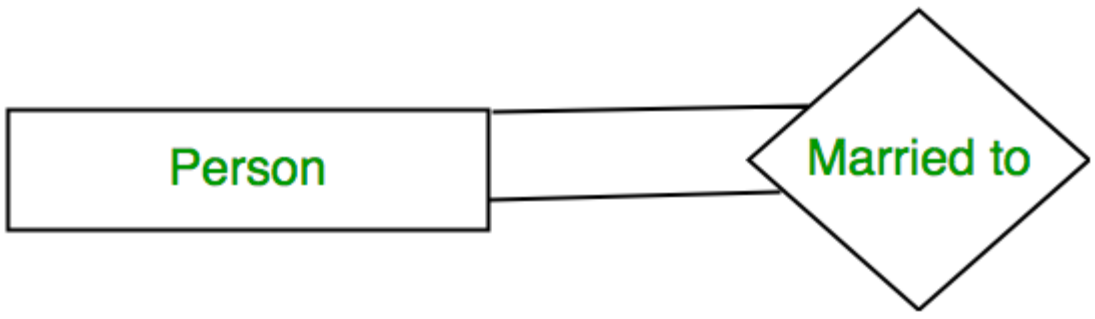


Degree of a relationship set: The number of different entity sets **participating in a relationship** set is called as degree of a relationship set.

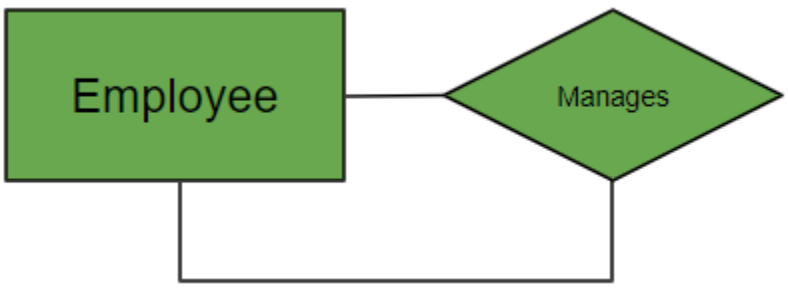
1. **Unary Relationship -** When there is **only ONE entity set participating in a relation**, the relationship is called as unary relationship.
   Example 1: A student is a friend of itself.



   Example 2: A person is married to a person.

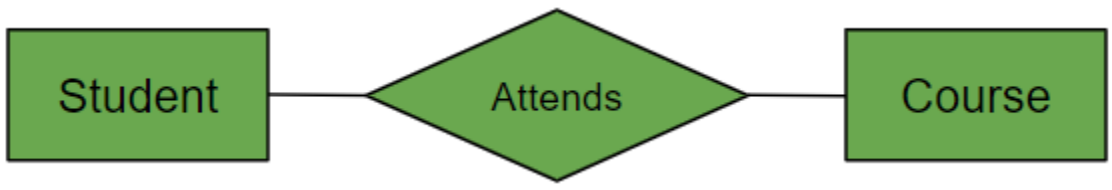


Example 3: An employee manages an employee.

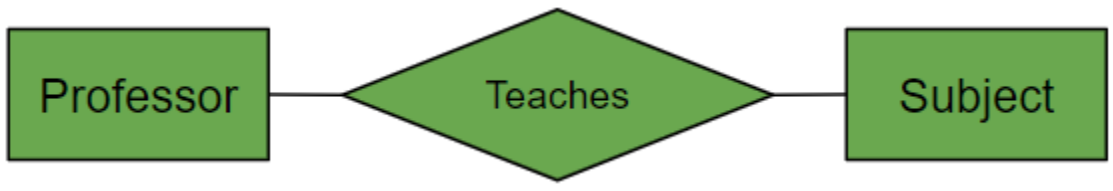


2. **Binary Relationship -** When there are **TWO entities set participating in a relation**, the relationship is called as binary relationship.For example, Student is enrolled in Course.

Example 1: A student attends a course.

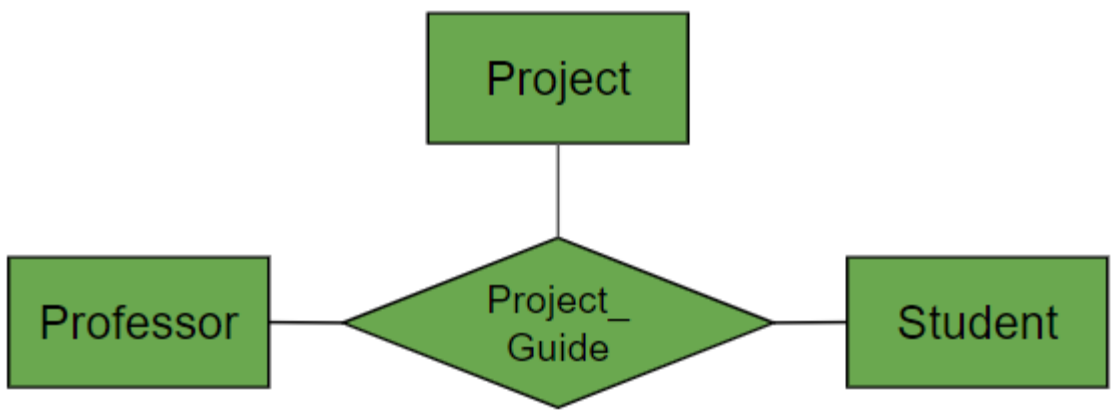


Example 2: A supplier supplies item.



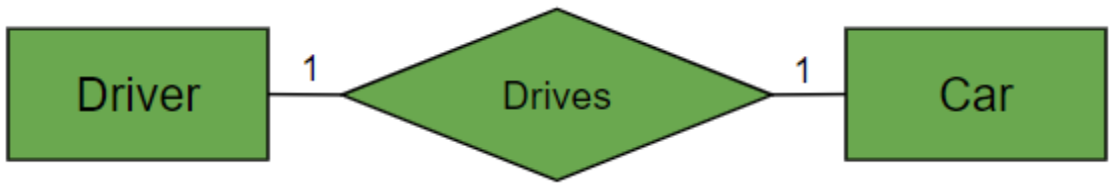
Example 3: A Profssor teaches subject.

3. **n-ary Relationship -** When there are n entities set participating in a relation, the relationship is called as n-ary relationship.
Example: A Professor, student and Project is related to a Project_Guide.

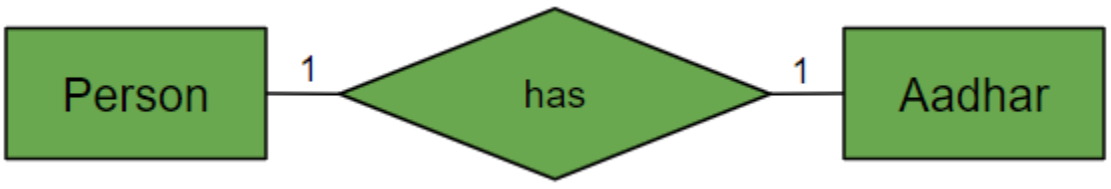


Cardinality The **number of times an entity of an entity set participates in a relationship** set is known as cardinality. Cardinality can be of different types:

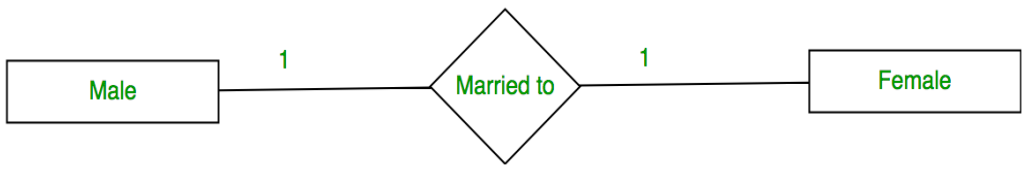
1. **One to One -** When each entity in each entity set can take part **only once in the relationship**, the cardinality is one to one.
Example 1: Let us assume that a driver can drive one carand a cat can be driven by the same driver. So the relationship will be one to one.



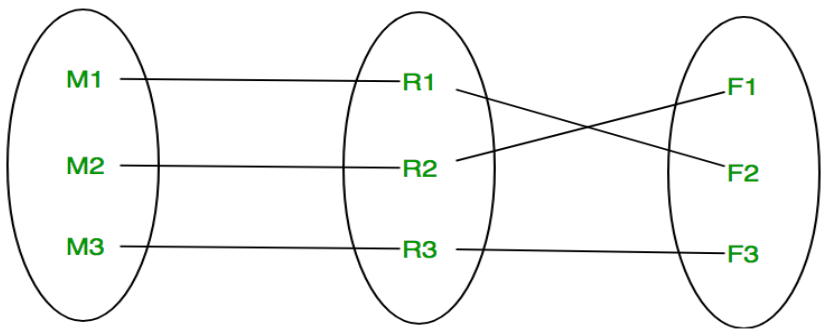
Example 2: A person can have only one Aadhar card and one Aadhar card can belong to only one person.



Example 3: Let us assume that a male can marry to one female and a female can marry to one male. So the relationship will be one to one.

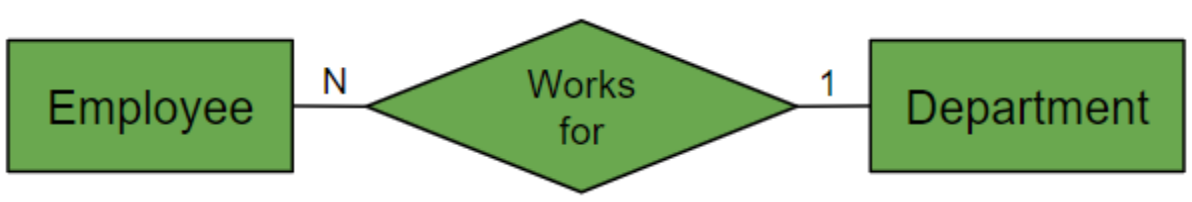


Using Sets, it can be represented as:

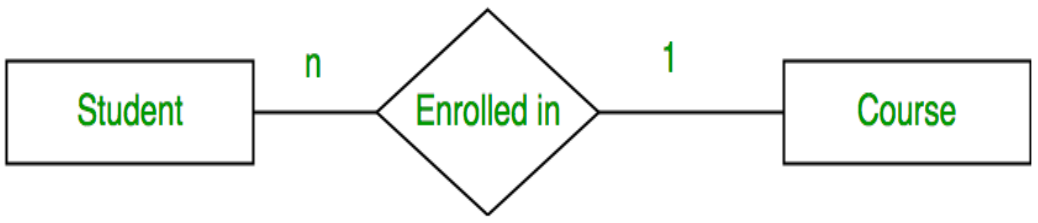


2. **One to Many -** When entities in one entity set **can take part only once in the relationship set and entities in other entity sets can take part more than once in the relationship set,** cardinalities is one to many. Many to one also come under this category.
   Example 1: A professor teaching many courses

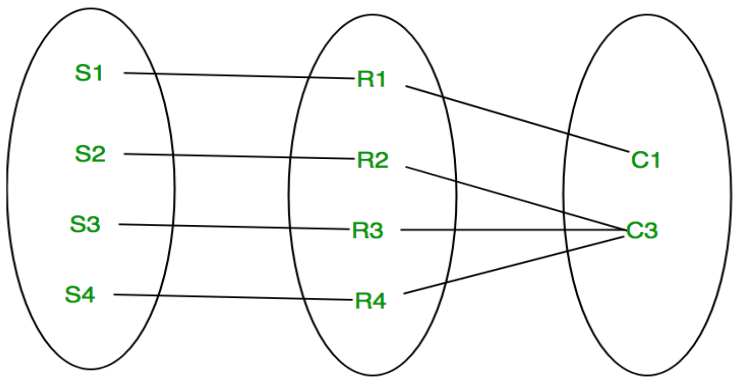
Example 2: Many employees working for one department.



Example 3: Let us assume that a student can take only one course but one course can be taken by many students. So the cardinality will be n to 1. It means that for one course there can be n students but for one student, there will be only one course.



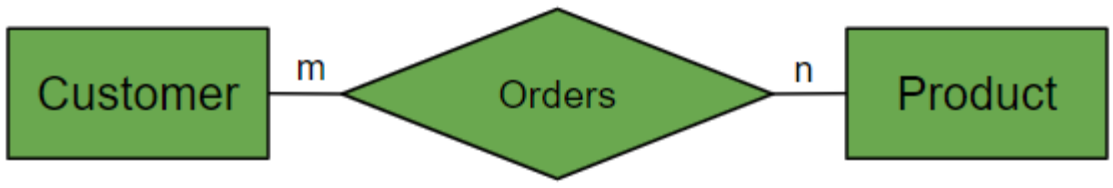
Using Sets, it can be represented as:



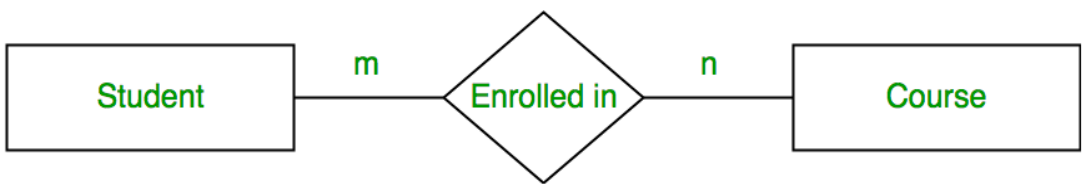
In this case, each student is taking only 1 course but 1 course has been taken by many students.

3. **Many to many -** When entities in all entity sets can **take part more than once in the relationship** cardinality is many to many.
   Example 1: Any number of student can take any number of subjects.

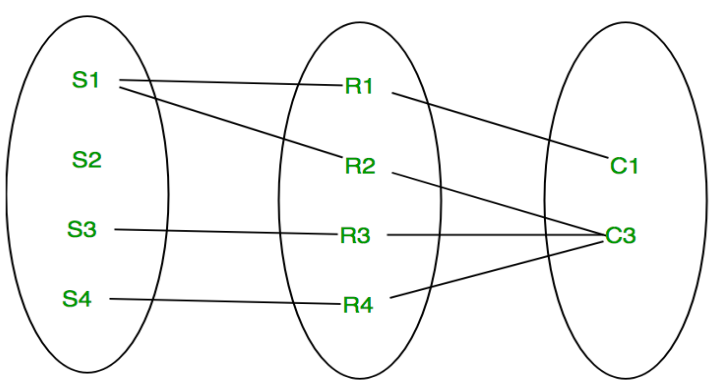
Example 2: Any number of customer can order any number of products.



Example 3: Let us assume that a student can take more than one course and one course can be taken by many students. So the relationship will be many to many.



Using sets, it can be represented as:

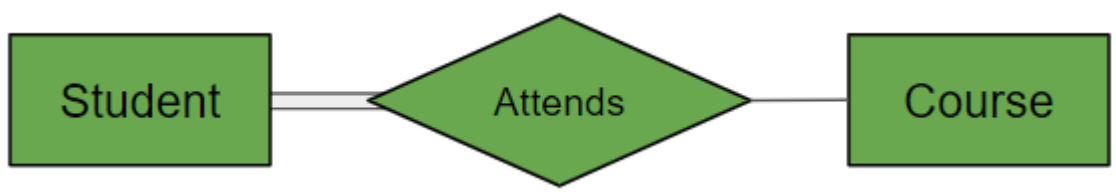


In this example, student S1 is enrolled in C1 and C3 and Course C3 is enrolled by S1, S3, and S4. So it is many to many relationships.

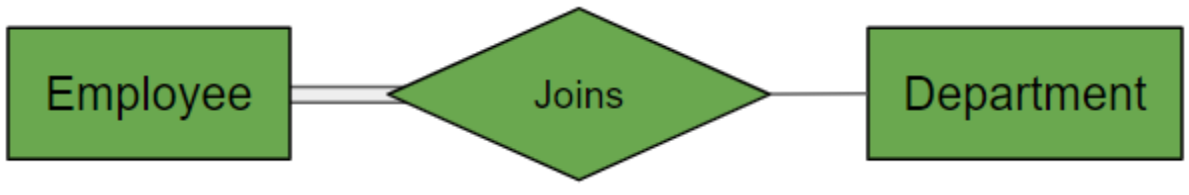
Participation Constraint: Participation Constraint is applied on the entity participating in the relationship set.

1. **Total Participation -** Each entity in the entity set **must participate** in the relationship.

   Example 1: If each student must attend a course, the participation of student will be total. Total participation is shown by double line in ER diagram.

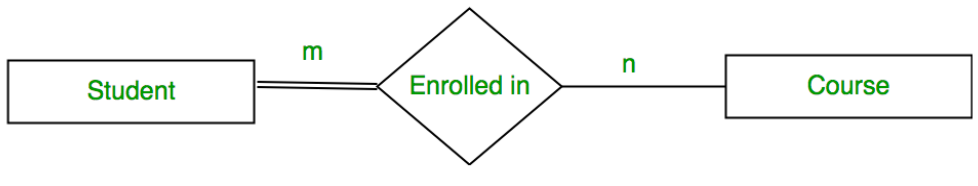


Example 2: Each employee must join a department.

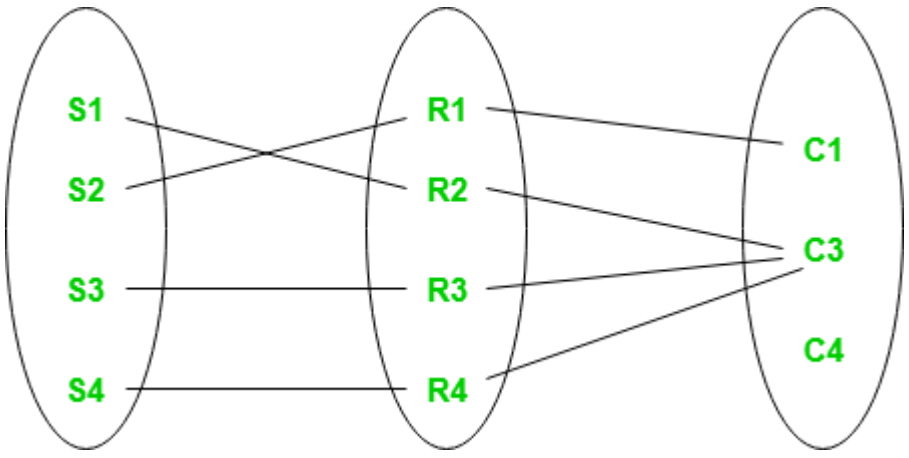


2. **Partial Participation -** The entity in the entity set **may or may NOT participat**e in the relationship. If some courses are not enrolled by any of the student, the participation of course will be partial.

   The diagram depicts the 'Enrolled in' relationship set with Student Entity set having total participation and Course Entity set having partial participation.



   Using set, it can be represented as,

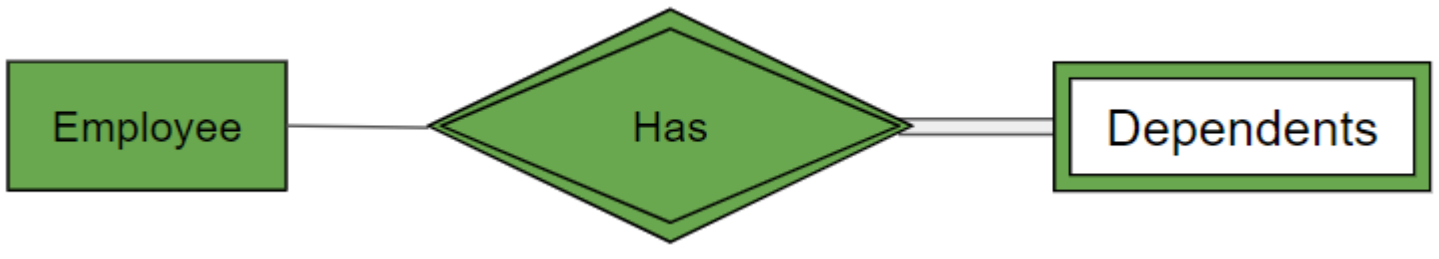


   Every student in Student Entity set is participating in a relationship but there exists a course C4 which is not taking part in the relationship.

Weak Entity Type and Identifying Relationship As discussed before, an entity type has a key attribute that uniquely identifies each entity in the entity set. But there exists **some entity type for which key attribute can't be defined**. These are called the Weak Entity type. A weak entity type is represented by a double rectangle. The participation of a weak entity type is always total. The relationship between a weak entity type and its identifying strong entity type is called an identifying relationship and it is represented by a double diamond.

Example 1: a school might have multiple classes and each class might have multiple sections. The section cannot be identified uniquely and hence they do not have any primary key. Though a class can be identified uniquely and the combination of a class and section is required to identify each section uniquely. Therefore the section is a weak entity and it shares total participation with the class.

Example 2: A company may store the information of dependants (Parents, Children, Spouse) of an Employee. But the dependents don't have existed without the employee. So Dependent will be a weak entity type and the Employee will be Identifying Entity type for Dependant.



Example 3: In case of multiple hosts, the login id cannot become primary key as it is dependent upon the hosts for its

identity.

DBMS | Relational Model

Relational Model was proposed by E.F. Codd to model data in the form of relations or tables. After designing the conceptual model of Database using the ER diagram, we need to convert the conceptual model in the relational model which can be implemented using any RDBMS languages like Oracle SQL, MySQL, etc. So we will see what the Relational Model is.

What is Relational Model?

Relational Model represents how data is stored in Relational Databases. A relational database stores data in the form of relations (tables). Consider a relation STUDENT with attributes ROLL_NO, NAME, ADDRESS, PHONE and AGE shown in Table 1.

STUDENT

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

IMPORTANT TERMINOLOGIES

1. **Attribute:** Attributes are the properties that define a relation. e.g.; **ROLL_NO**, **NAME**
2. **Relation Schema:** A relation schema represents name of the relation with its attributes. e.g.; STUDENT (ROLL_NO, NAME, ADDRESS, PHONE and AGE) is relation schema for STUDENT. If a schema has more than 1 relation, it is called Relational Schema.
3. **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

4. **Relation Instance:** The set of tuples of a relation at a particular instance of time is called as relation instance. Table 1 shows the relation instance of STUDENT at a particular time. It can change whenever there is insertion, deletion or updation in the database.
5. **Degree:** The number of attributes in the relation is known as degree of the relation. The **STUDENT** relation defined above has degree 5.
6. **Cardinality:** The number of tuples in a relation is known as cardinality. The **STUDENT** relation defined above has cardinality 4.
7. **Column:** Column represents the set of values for a particular attribute. The column **ROLL_NO** is extracted from relation STUDENT.

ROLL_NO
1
2
3
4

8. **NULL Values:** The value which is not known or unavailable is called NULL value. It is represented by blank space. e.g.; PHONE of STUDENT having ROLL_NO 4 is NULL.

Constraints in Relational Model

While designing Relational Model, we define some conditions which must hold for data present in database are called Constraints. These constraints are checked before performing any operation (insertion, deletion and updation) in database. If there is a violation in any of constrains, operation will fail.

Domain Constraints: These are attribute level constraints. An attribute can only take values which lie inside the domain range. e.g; If a constrains AGE>0 is applied on STUDENT relation, inserting a negative value of AGE will result in failure.

Key Integrity: Every relation in the database should have at least one set of attributes that defines a tuple uniquely. Those set of attributes is called key. e.g.; ROLL_NO in STUDENT is a key. No two students can have the same roll number. So a key has two properties:

- It should be unique for all tuples.
- It can't have NULL values.

Referential Integrity: When one attribute of a relation can only take values from other attribute of same relation or any other relation, it is called referential integrity. Let us suppose we have 2 relations

STUDENT

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

BRANCH

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

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

ANOMALIES

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

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

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

ON DELETE CASCADE: It will delete the tuples from REFERENCING RELATION if a value used by REFERENCING ATTRIBUTE is deleted from REFERENCED RELATION. e.g; if we delete a row from BRANCH with BRANCH_CODE 'CS', the rows in STUDENT relation with BRANCH_CODE CS (ROLL_NO 1 and 2 in this case) will be deleted.

ON UPDATE CASCADE: It will update the REFERENCING ATTRIBUTE in REFERENCING RELATION if attribute value used by REFERENCING ATTRIBUTE is updated in REFERENCED RELATION. Ex: if we update a row from BRANCH with BRANCH_CODE 'CS' to 'CSE', the rows in STUDENT relation with BRANCH_CODE CS (ROLL_NO 1 and 2 in this case) will be updated with BRANCH_CODE 'CSE'.

SUPER KEYS: Any set of attributes that allows us to identify unique rows (tuples) in a given relationship are known as

superkeys. Out of these super keys, we can always a proper subset that can be used as a primary key. Such keys are known as Candidate keys. If there is a combination of two or more attributes that are being used as the primary key then we call it as a Composite key.

DBMS | Keys in Relational Model
We have considered a Student and Course Relational Model for our Reference Examples.

Example Student Course Relational Model

STUDENT

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY	STUD_AGE
1	RAM	9716271721	Haryana	India	20
2	RAM	9898291281	Punjab	India	19
3	SUJIT	7898291981	Rajsthan	India	18
4	SURESH		Punjab	India	21

Table 1

STUDENT_COURSE

STUD_NO	COURSE_NO	COURSE_NAME
1	C1	DBMS
2	C2	Computer Networks
1	C2	Computer Networks

Table 2

Candidate Key: The minimal set of attribute which can uniquely identify a tuple is known as candidate key. For **Example**, STUD_NO in STUDENT relation.

- The value of Candidate Key is unique and non-null for every tuple.
- There can be more than one candidate key in a relation. For Example, STUD_NO as well as STUD_PHONE both are candidate keys for relation STUDENT.
- The candidate key can be simple (having only one attribute) or composite as well. For **Example**, {STUD_NO, COURSE_NO} is a composite candidate key for relation STUDENT_COURSE.

Note - In Sql Server a unique constraint that has a nullable column, **allows** the value '**null**' in that column **only once**. That's why STUD_PHONE attribute as candidate here, but can not be 'null' values in primary key attribute.

Super Key: The set of attributes that can uniquely identify a tuple is known as Super Key. For Example, STUD_NO, (STUD_NO, STUD_NAME), etc.

- Adding zero or more attributes to candidate key generates super key.
- A candidate key is a superkey but vice versa is not true.

Primary Key: There can be more than one candidate key in relation out of which one can be chosen as the primary key. For **Example**, STUD_NO, as well as STUD_PHONE both, are candidate keys for relation STUDENT but STUD_NO can be chosen as the primary key (only one out of many candidate keys).

Alternate Key: The candidate key other than the primary key is called an alternate key. For **Example**, STUD_NO, as well as STUD_PHONE both, are candidate keys for relation STUDENT but STUD_PHONE will be alternate key (only one out of many candidate keys).

Foreign Key: If an attribute can only take the values which are present as values of some other attribute, it will be a foreign key to the attribute to which it refers. The relation which is being referenced is called referenced relation and the corresponding attribute is called referenced attribute and the relation which refers to the referenced relation is called referencing relation and the corresponding attribute is called referencing attribute. A referenced attribute of the referenced relation should be the primary key for it. For **Example**, STUD_NO in STUDENT_COURSE is a foreign key to STUD_NO in STUDENT relation.

It may be worth noting that unlike, Primary Key of any given relation, Foreign Key can be NULL as well as may contain duplicate tuples i.e. it need not follow uniqueness constraint.

For **Example**, STUD_NO in STUDENT_COURSE relation is not unique. It has been repeated for the first and third tuple. However, the STUD_NO in STUDENT relation is a primary key and it needs to be always unique and it cannot be null.

Database Normalization

Objectives of Good Database Design:

1. No updation, insertion and deletion anomalies

2. Easily Extendible

3. Good Performance for all query sets

4. More Informative

Anomalies: There are different types of anomalies which can occur in referencing and referenced relation:

1. **Updation Anomaly:** If some particular attribute value has to be updated, then it has to be updated in every tuple consisting of that value, which would be too costly operation as all records have to be first transferred to the main memory, and then again transferred to the secondary memory after updation.

Student_Id	Student_Name	Subject_Id	Subject_Name
1001	ABC	CS101	Database Management System
1002	XYZ	CS101	Database Management System
1001	ABC	CS102	Operating System
1003	BCD	CS102	Operating System

   In the above table, if someone tries to rename the Subject_Name Database Management System to DBMS and while doing so, the system crashes in between, then it would lead to inconsistency of the table. Similarly imposing a bad query can also lead to redundancy issue.

2. **Insertion Anomaly:** If a tuple is inserted in referencing relation and referencing attribute value is not present in referenced attribute, it will not allow inserting in referencing relation. Suppose in the above table we need to add a student PQR with Student_ID 1004 and we don't know the Subject_name or the Subject_Id the student intends to take. If the fields are mandatory then, the addition of the data to the table will not be possible.

3. **Deletion Anomaly:** Deletion of some data may lead to loss of some other useful data. For example, Suppose some student details have to be deleted. Now when the required tuple is deleted, the subject name details corresponding to that student record also gets deleted, which might lead to loss of the details of the subject.

Database Normalization | Functional Dependencies

Database normalization is the process of organizing the attributes of the database to reduce or eliminate **data redundancy (having the same data but at different places)** .

Problems because of data redundancy: Data redundancy unnecessarily increases the size of the database as the same data is repeated in many places. Inconsistency problems also arise during insert, delete and update operations.

Functional Dependency

Functional Dependency is a constraint between two sets of attributes in a relation from a database. A functional dependency is denoted by an arrow (→). If an attribute A functionally determines B or B is functionally dependent on A, then it is written as A → B.
For example, employee_id → name means employee_id functionally determines the name of the employee. As another example in a time table database, {student_id, time} → {lecture_room}, student ID and time determine the lecture room where the student should be.
Let's look at the tables below:

Enroll_No	Name	Address	Phone_No

In this table we can say that:
Enroll_No → Name
Enroll_No → Address
Enroll_No → Phone No

Subject_Id	Student_Id	Marks	Grade

In this table we can say that:
(Subject_Id, Student_Id) → Marks
(Subject_Id, Student_Id) → Marks, Grades

Examples: For the following table, which of the following functional dependencies hold True.

1. A → B

2. B → A

A	B
x	1
y	1
z	2

Solution: The first condition holds True as for every value of A there is a unique value in B.
The second dependency results in False, as there is no unique identification for every value of B in A.

What does functionally dependent mean?

A function dependency A → B means for all instances of a particular value of A, there is the same value of B.
Let's do an exercise on Functional Dependency:
For the following table define which of the dependency is True or False

Enroll_No	Name	Address
101	Raj	ABC
102	Ravi	ABC
103	Raj	XYZ

1. Enroll_No → Name, Address

2. Name → Address

3. Address → Name

Solution:

1. This holds True as for every Enroll_No, there is a unique representation in Name and Address combined.

2. This is False as there is an anomaly in the Name Raj

3. This is False as there is an anomaly in the Address ABC

For example in the below table A → B is true, but B → A is not true as there are different values of A for B = 3.

```
A   B
------
1   3
```

```
2   3
4   0
1   3
4   0
```

Why do we study Functional Dependency?

We study functional dependency so that we can carve out a few attributes to another table so that the data redundancy can be reduced to a minimum. Let's look at the following table:

Student_ID	Name	Dept_Id	Dept_Name
101	ABC	10	CS
102	BCD	11	ECE
103	ABC	10	CS
104	XYZ	11	ECE
105	CDE	10	CS

In this table let's find out the functional dependency between two attributes.
As we can see that Dept_Id has each unique identification in Dept_Name, so we can say that Dept_Id → Dept_Name
Therefore we can carve out this two table and create another one out of this.

Dept_Id	Dept_Name
10	CS
11	ECE

Trivial Functional Dependency:
X → Y is trivial only when Y is subset of X.
Examples:

```
ABC → AB
ABC → A
ABC → ABC
```

Non Trivial Functional Dependencies X → Y is a non trivial functional dependencies when Y is not a subset of X.

X → Y is called completely non-trivial when X intersect Y is NULL.
Examples:

```
Id → Name,
Name → DOB
```

Semi Non Trivial Functional Dependencies X → Y is called semi non-trivial when X intersect Y is not NULL.
Examples:

```
AB → BC,
AD → DC
```

DBMS | Normal Forms

Normalization is the process of minimizing **redundancy** from a relation or set of relations. Redundancy in relation may cause insertion, deletion and updating anomalies. So, it helps to minimize the redundancy in relations. **Normal forms** are used to eliminate or reduce redundancy in database tables.

1. First Normal Form

If a relation contains a composite or multi-valued attribute, it violates first normal form or relation is in first normal form if it does not contain any composite or multi-valued attribute. A relation is in first normal form if every attribute in that relation is **singled valued attribute**.

Let's understand the concept of 1NF using this exercise:

Customer_ID	Name	Mob No
101	ABC	8860033933, 9899868189
102	BCD	8960133933
103	XYZ	8681899900, 9681888800
104	PQR	8189399888

As we can see that the above table contains multi-valued attributes, which violates the principle of first normal form and hence must be reduced. There are various methods of doing this:

Method 1: This involves the creation of multiple columns, distributing the values alongside the new attributes. We can distribute the multivalued attributes to new columns by naming them as Mob No 1, Mob No 2, Mob No 3 etc, depending on the number of multivalued attributes till the table gets reduced to single-valued attribute.

Customer_ID	Name	Mob No 1	Mobile No 2	Mobile No 3
101	ABC	8860033933	9899868189	
102	BCD	8960133933	-	
103	XYZ	8681899900	9681888800	
104	PQR	8189399888	-	

But this method got few problems like various fields are needed to be left blank, resulting in wastage of space.

Method 2: This method involves the creation of multiple instead of columns, with copying up of the non-repeated attribute values for each repeated attribute values.

Customer_ID	Name	Mob No
101	ABC	8860033933
102	BCD	8960133933
103	XYZ	8681899900
104	PQR	8189399888
101	ABC	9899868189
103	XYZ	9681888800

In this table we have made a copy of the values of the repeated attributes, keeping other attributes the same. This saves our space but introduces another problem of repetition of Customer_ID, which must remain unique.

Method 3: This method involves the creation of another table and shifting the repeated attributes to that table and linking it with the previous table by using any type of ID.

Customer_ID	Name
101	ABC
102	BCD
103	XYZ
104	PQR

ID	Customer_ID	Mob No
1	101	8860033933
2	102	.
.	.	.
.	.	.

- **Example 1 -** Relation STUDENT in table 1 is not in 1NF because of multi-valued attribute STUD_PHONE. Its decomposition into 1NF has been shown in table 2.

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721, 9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 1

Conversion to first normal form

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721	HARYANA	
1	RAM	9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 2

- **Example 2 -**

```
ID   Name   Courses
------------------
1    A      c1, c2
2    E      c3
3    M      C2, c3
```

In the above table, Course is a multi-valued attribute so it is not in 1NF.

Below Table is in 1NF as there is no multi valued attribute

```
ID   Name   Course
------------------
1    A       c1
1    A       c2
2    E       c3
3    M       c1
3    M       c2
```

2. Second Normal Form

To be in second normal form, a relation must be in first normal form and relation must not contain any partial dependency. A relation is in 2NF if it has **No Partial Dependency,** i.e.**,** no non-prime attribute (attributes which are not part of any candidate key) is dependent on any proper subset of any candidate key of the table.

User_ID	Course_ID	Course_Fee
1	CS101	5000
2	CS102	2000
1	CS102	2000
3	CS101	5000
4	CS102	2000
2	CS103	7000

In the above table, the candidate key is the combination of (User_ID, Course_ID) . The non-prime attribute is Course_Fee.
Since this non-prime attribute depends partially on the candidate key, this table is not in 2NF.
To convert this into 2NF, one needs to split the table into two and creating a common link between two. As we have done with the above table after splitting them into two and keeping Course_ID as the common key.

User_ID	Course_ID

Course_ID	Course_Fee

STUD_NO	COURSE_NO	COURSE_NAME
1	C1	DBMS
2	C2	Computers Network
1	C2	Computers Network

Table 3

Example:
Partial Dependency - If the proper subset of candidate key determines non-prime attribute, it is called partial dependency.

- **Example 1 -** In relation STUDENT_COURSE given in Table 3,

```
FD set: {COURSE_NO->COURSE_NAME}
Candidate Key: {STUD_NO, COURSE_NO}
```

In FD COURSE_NO->COURSE_NAME, COURSE_NO (proper subset of candidate key) is determining COURSE_NAME (non-prime attribute). Hence, it is partial dependency and relation is not in second normal form. To convert it to second normal form, we will decompose the relation STUDENT_COURSE (STUD_NO, COURSE_NO, COURSE_NAME) as :

```
STUDENT_COURSE (STUD_NO, COURSE_NO)
COURSE (COURSE_NO, COURSE_NAME)
```

Note - This decomposition will be lossless join decomposition as well as dependency preserving.

- **Example 2 -** Consider following functional dependencies in relation R (A, B , C, D)

- `AB -> C  [A and B together determine C]`

```
BC -> D  [B and C together determine D]
```

In the above relation, AB is the only candidate key and there is no partial dependency, i.e., any proper subset of AB doesn't determine any non-prime attribute.

3. Third Normal Form

Def 1: It follows two rules:

1. The table must be in 2NF.
2. No non-prime attribute must define another non-prime attribute.

Let's understand these rules using the following table:

Stud_No	Stud_Name	Stud_State	Stud_Country
101	Ram	Haryana	India
102	Ramesh	Punjab	India
103	Suresh	Punjab	India

Checking the first condition of 3NF: The candidate key for the table is Stud_No. It has the following dependency on other attributes:
Stud_No → [Stud_Name, Stud_State, State_Country]
Student_State → Stud_Country
In both the above cases, we see that there is only one candidate key and it has no other parts, and also there are no non-prime attributes. Hench the condition of 2NF holds.

Checking the second condition of 3NF: As we can see that Student_State → Stud_Country occurs which means that one non-prime attribute is defining another non-prime attribute, it violates the second rule of 3NF.

This problem can be fixed by splitting the table into two.

T1: Stud_No, Stud_Name, Stud_State
T2: Stud_State, Stud_Country
As in this case, no non-prime attribute is defining another non-prime attribute, hence the condition of 3NF holds.

Let's see another table and try to understand the concept of 3NF:

Exam_Name	Exam_Year	Topper_Name	Topper_DOB
ABC	2016	Ram	15-06-1992
BCD	2017	Ramesh	16-07-1994
EFG	2017	Suresh	10-06-1991
ABC	2017	Kamlesh	11-11-1993

Let's first find out the dependencies:
(Exam_Name, Exam_Year) → (Topper_Name, Topper_DOB)
Topper_Name → Topper_DOB

We can very well see that since there are no repetitions in the columns, 1NF holds. Also, 2NF holds, as no non-prime attribute is partially dependent on the candidate key. But the second dependency violates the 3NF.

To solve this we need to split the table into two as we did in an earlier case:
T1: Exam_Name, Exam_Year, Topper_Name
T2: Topper_Name, Topper_DOB

Let's look at a few other definitions of 3NF:
Def 2: This also defines two sets of rules which are mostly similar to the Def 1.

1. In 2NF and

2. Non-prime attributes are not transitively dependent on prime attributes.

Def 3: It says, for every X → A, one of the following should be true:

1. X is a Superkey

2. A-X is a prime attribute

3. X → is a trivial functional dependency.

Example:

STUD_NO	STUD_NAME	STUD_STATE	STUD_COUNTRY	STUD_AGE
1	RAM	HARYANA	INDIA	20
2	RAM	PUNJAB	INDIA	19
3	SURESH	PUNJAB	INDIA	21

Table 4

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

- **Example 1 -** In relation STUDENT given in Table 4,

  FD set: {STUD_NO -> STUD_NAME, STUD_NO -> STUD_STATE, STUD_STATE -> STUD_COUNTRY, STUD_NO -> STUD_AGE, STUD_STATE -> STUD_COUNTRY}
  Candidate Key: {STUD_NO}

  For this relation in table 4, STUD_NO -> STUD_STATE and STUD_STATE -> STUD_COUNTRY are true. So STUD_COUNTRY is transitively dependent on STUD_NO. It violates the third normal form. To convert it in third normal form, we will decompose the relation STUDENT (STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_COUNTRY_STUD_AGE) as:
  STUDENT (STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_AGE)
  STATE_COUNTRY (STATE, COUNTRY)

- **Example 2 -** Consider relation R(A, B, C, D, E)
  A -> BC,
  CD -> E,
  B -> D,
  E -> A
  All possible candidate keys in above relation are {A, E, CD, BC} All attribute are on right sides of all functional dependencies are prime.

4. Boyce-Codd Normal Form (BCNF)

A relation R is in BCNF if R is in Third Normal Form and for every FD, LHS is super key. A relation is in BCNF if in every non-trivial functional dependency X --> Y, X is a superkey.
Let's understand this in a more basic form. We can say that a table is in BCNF if it satisfies the BCNF condition along with all other normalization condition including 1NF, 2NF and 3NF.
For 1NF, there must not be any repeated values in any attributes.
For 2NF, no non-prime attribute must be defined by any prime attributes.
For 3NF, no non-prime attribute must be defined by any non-prime attributes.
For BCNF, no prime or non-prime attribute must be define any prime attributes
Let's look at the following table and try to understand how this works:

STUD_ID	Subject	Prof_id
1001	DBMS	103
1001	OS	110
1002	DBMS	111

Functional Dependencies:
(Stud_ID, Subject) → Prof_id
Prof_id → Subject

Candidate Keys:
(Stud_ID, Subject), Prof_id, (Prof_id, Stud_ID)

The second condition in Functional Dependencies does not follow the rule of BCNF.
Let's try to analyse whether the relation satisfies all type of Normal Forms.
1NF: Since there are no repeated attributes, this holds true.
2NF: Since there are no non-prime attributes, this also holds true.
3NF: Since there are no non-prime attributes, this also holds true.
BCNF: In the second dependency, Prof_id being a prime attribute defines another prime attribute Subject. Hence the

condition fails to satisfy BCNF.

Now let's modify the table by adding another row to it:

STUD_ID	Subject	Prof_id
1001	DBMS	103
1001	OS	110
1002	DBMS	111
1003	DBMS	103

Then break the table into a BCNF compliance database.

STUD_ID	Prof_id
1001	103
1001	110
1002	111
1003	103

Prof_id	Subject
103	DBMS
110	OS
111	DBMS
103	DBMS

Although this decomposition satisfies with the conditions of BCNF, but the DBMS might not accept this split because while splitting the table the first Functional Dependency i.e., (Subject_ID, Subject) → Prof_ID does not exist anymore.

- **Example 1 -** Find the highest normal form of a relation R(A,B,C,D,E) with FD set as -

```
BC->D,
AC->BE,
B->E
```

Step 1. As we can see, (AC)+ ={A,C,B,E,D} but none of its subset can determine all attribute of relation, So AC will be candidate key. A or C can't be derived from any other attribute of the relation, so there will be only 1 candidate key {AC}.
Step 2. The prime attributes are those attribute which are part of candidate key {A,C} in this example and others will be non-prime {B,D,E} in this example.
Step 3. The relation R is in 1st normal form as a relational DBMS does not allow multi-valued or composite attribute.
Step 4. The relation is in 2nd normal form because BC->D is in 2nd normal form (BC is not a proper subset of candidate key AC) and AC->BE is in 2nd normal form (AC is candidate key) and B->E is in 2nd normal form (B is not a proper subset of candidate key AC).
Step 5.The relation is not in 3rd normal form because in BC->D (neither BC is a superkey nor D is a prime attribute) and in B->E (neither B is a superkey nor E is a prime attribute) but to satisfy 3rd normal for, either LHS of an FD should be super key or RHS should be prime attribute.
So the highest normal form of relation will be 2nd Normal form.

Key Points -

1. BCNF is free from redundancy.
2. If a relation is in BCNF, then 3NF is also also satisfied.

3. If all attributes of relation are prime attribute, then the relation is always in 3NF.
4. A relation in a Relational Database is always and at least in 1NF form.
5. Every Binary Relation (a Relation with only 2 attributes) is always in BCNF.
6. If a Relation has only singleton candidate keys(i.e. every candidate key consists of only 1 attribute), then the Relation is always in 2NF(because no Partial functional dependency possible).
7. Sometimes going for BCNF form may not preserve functional dependency. In that case go for BCNF only if the lost FD(s) is not required, else normalize till 3NF only.
8. There are many more Normal forms that exist after BCNF, like 4NF and more. But in real world database systems it's generally not required to go beyond BCNF.

Exercise 1: Find the highest normal form in R (A, B, C, D, E) under following functional dependencies.

```
ABC --> D
CD --> AE
```

Important Points for solving above type of question.
1) It is always a good idea to start checking from BCNF, then 3 NF and so on.
2) If any functional dependency satisfied a normal form then there is no need to check for lower normal form. For example, ABC --> D is in BCNF (Note that ABC is a super key), so no need to check this dependency for lower normal forms.

Candidate keys in a given relation are {ABC, BCD}

BCNF: ABC -> D is in BCNF. Let us check CD -> AE, the CD is not a superkey so this dependency is not in BCNF. So, R is not in BCNF.

3NF: ABC -> D we don't need to check for this dependency as it already satisfied BCNF. Let us consider a CD -> AE. Since E is not a prime attribute, so the relation is not in 3NF.

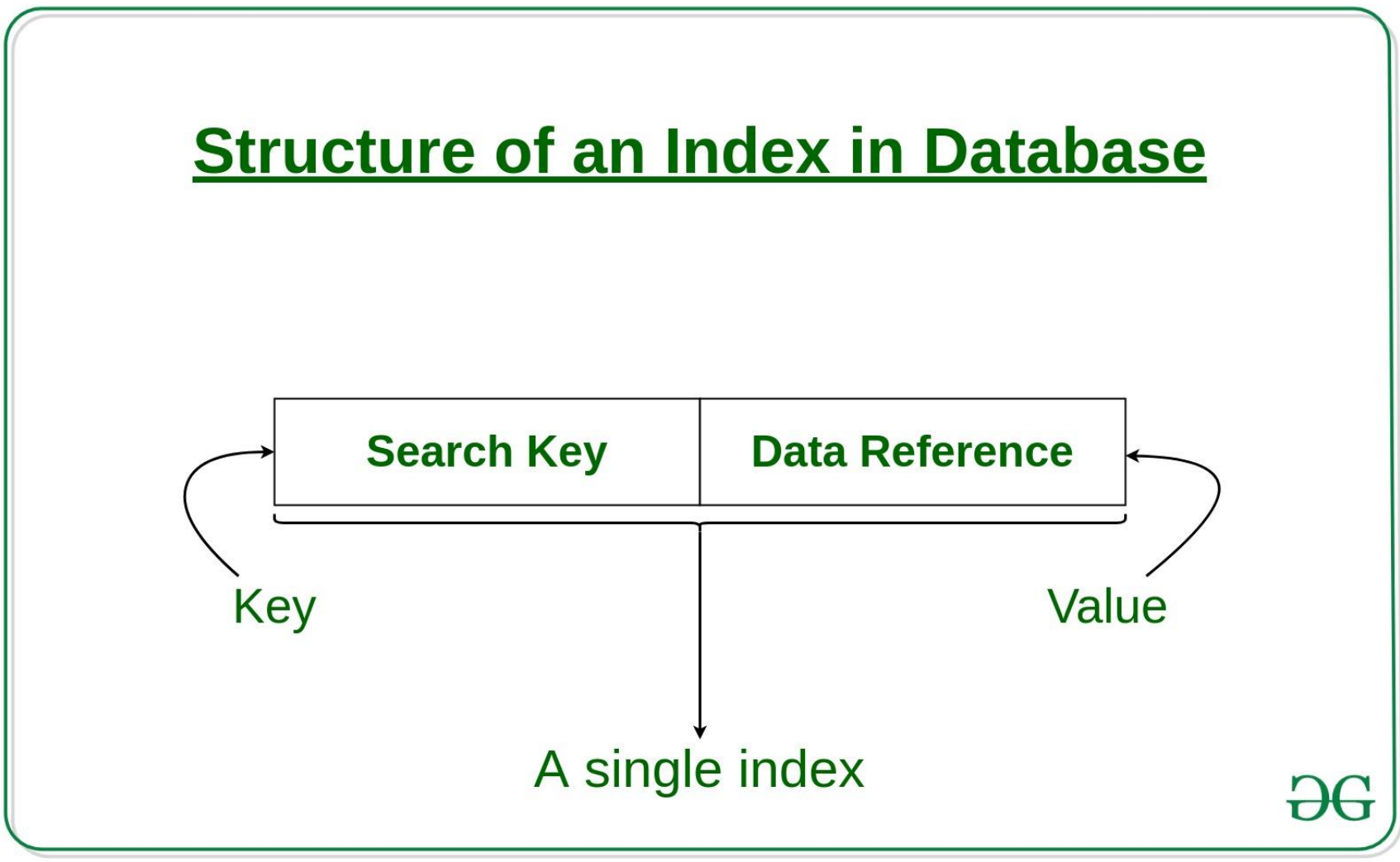
2NF: In 2NF, we need to check for partial dependency. CD which is a proper subset of a candidate key and it determine E, which is a non prime attribute. So, the given relation is also not in 2 NF. So, the highest normal form is 1 NF.

Indexing in Databases

Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed. It is a data structure technique which is used to quickly locate and access the data in a database.

Indexes are created using a few database columns.

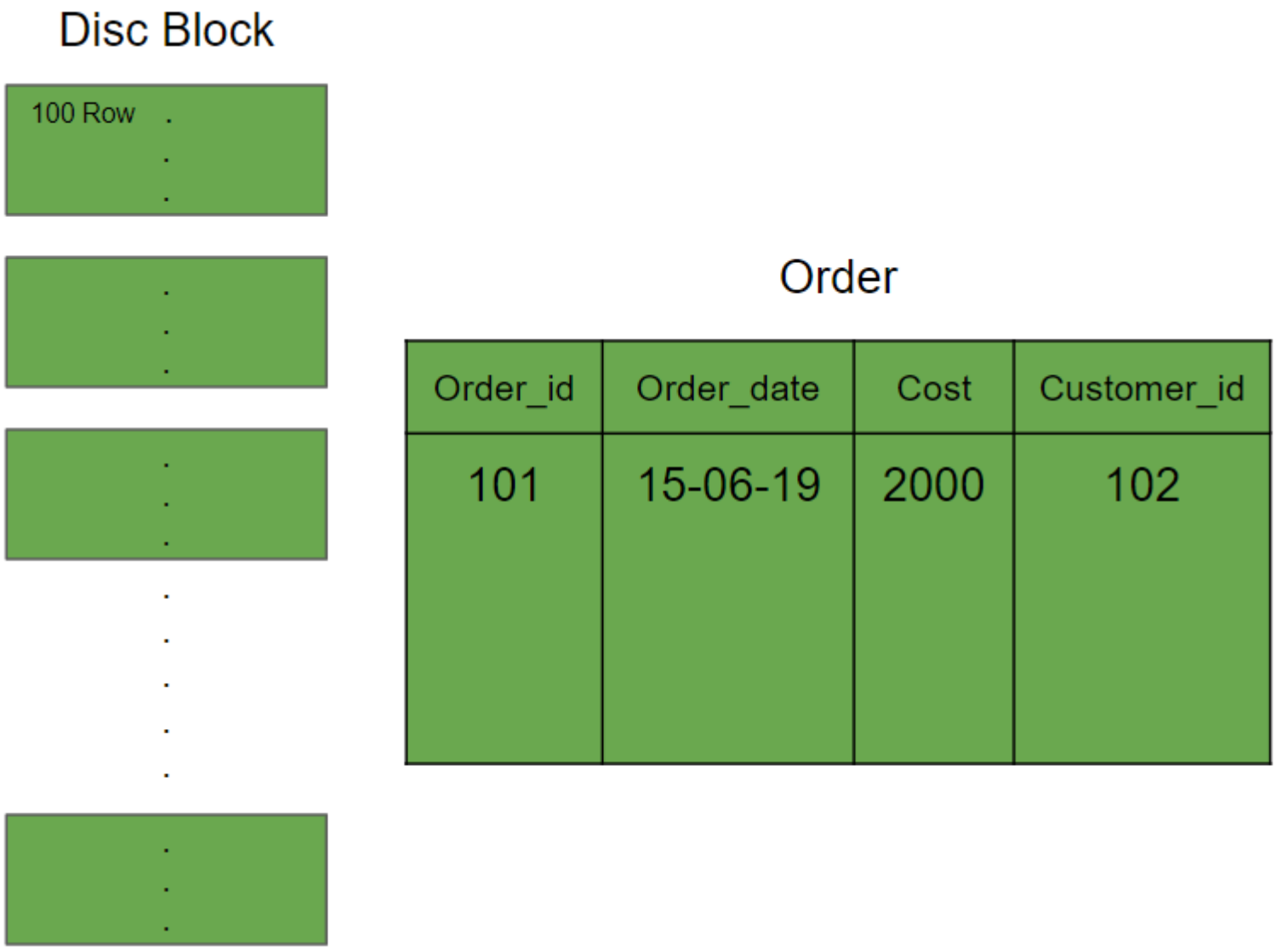
- The first column is the **Search key** that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly.
  Note: The data may or may not be stored in sorted order.
- The second column is the **Data Reference** or **Pointer** which contains a set of pointers holding the address of the disk block where that particular key value can be found.



The indexing has various attributes:

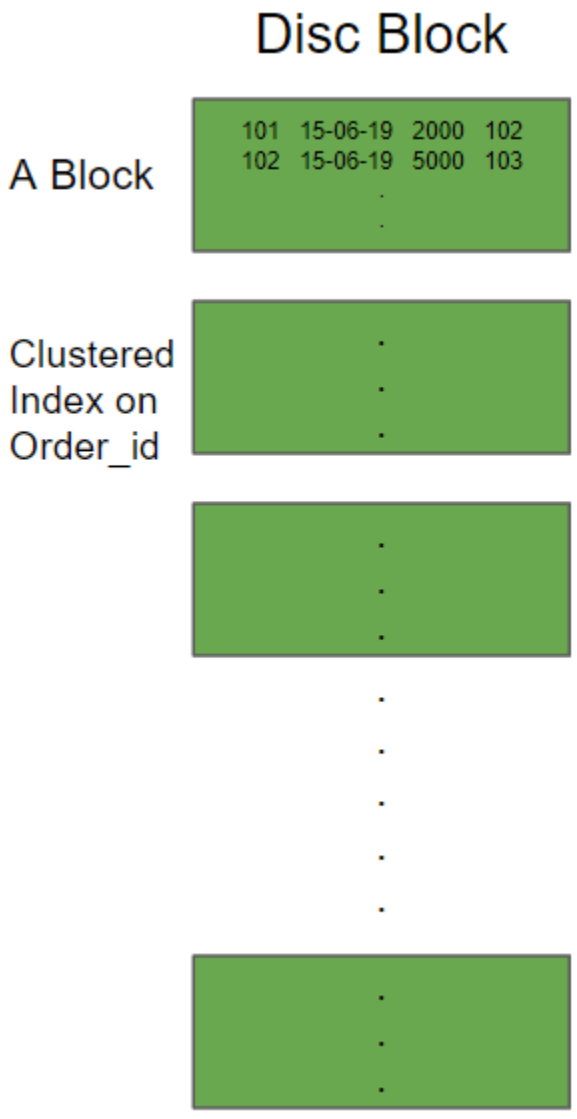
- **Access Types**: This refers to the type of access such as value based search, range access, etc.
- **Access Time**: It refers to the time needed to find particular data element or set of elements.
- **Insertion Time**: It refers to the time taken to find the appropriate space and insert a new data.
- **Deletion Time**: Time taken to find an item and delete it as well as update the index structure.
- **Space Overhead**: It refers to the additional space required by the index.

Let's look at an example:



The DBMS uses hard disk to store all the records in the above databse. As we know that the access time of the hard disk is very slow, searching for anything in such huge databases could cost performance issue. Moreover, searchin for a repeated item in the database could lead to a greater consumption of time as this will require searching for all the items in every block.
Suppose there are 100 rows in each block, so when a customer id is searched for in the database, will take too much of

time. The hard disk does not store the data in a particular order.
One solution to this problem is to arrange the indexes in a database in sorted order so that any looked up item can be found easily using Binary Search. This creation of orders to store the indexes is called clustered indexing.



Order

Order_id	Order_date	Cost	Customer_id
101	15-06-19	2000	102
102	15-06-19	5000	103
105	15-06-19	6000	101
106	15-06-19	8000	102

This sort of indexing is called as the clustered indexing and this can be applied to any attribute. It is not necessary that the focus of order should be any key. When the primary key is used to order the data in a heap, it is called as Primary Indexing.

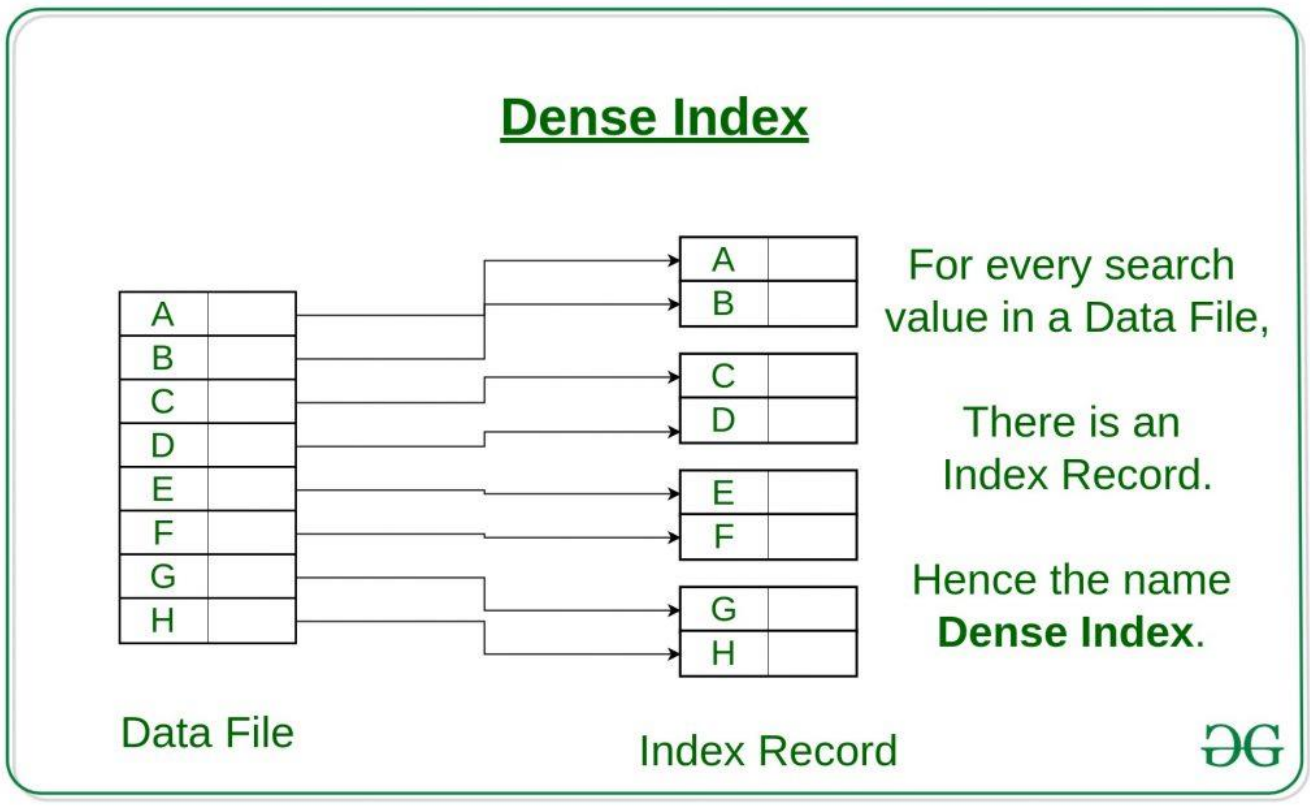
Although we have solved one problem of time cost, another problem arises when there are multiple values that are needed to be fetched using an id. The clustered indexing allows the physical arrangement of the data only on one key, therefore multiple fetching will raise a problem. To solve this we have another method of indexing called the non-clustered indexing or secondary indexing.

To make the understanding clearer, let's take an example of a book, where the index or the content page in which all the chapter numbers are mentioned are organised in a clustered index manner. The dictionary sort of thing mentioned at the end of the book, that provides a reference to the words in various chapters along with the page numbers are arranged in a non-clustered manner.

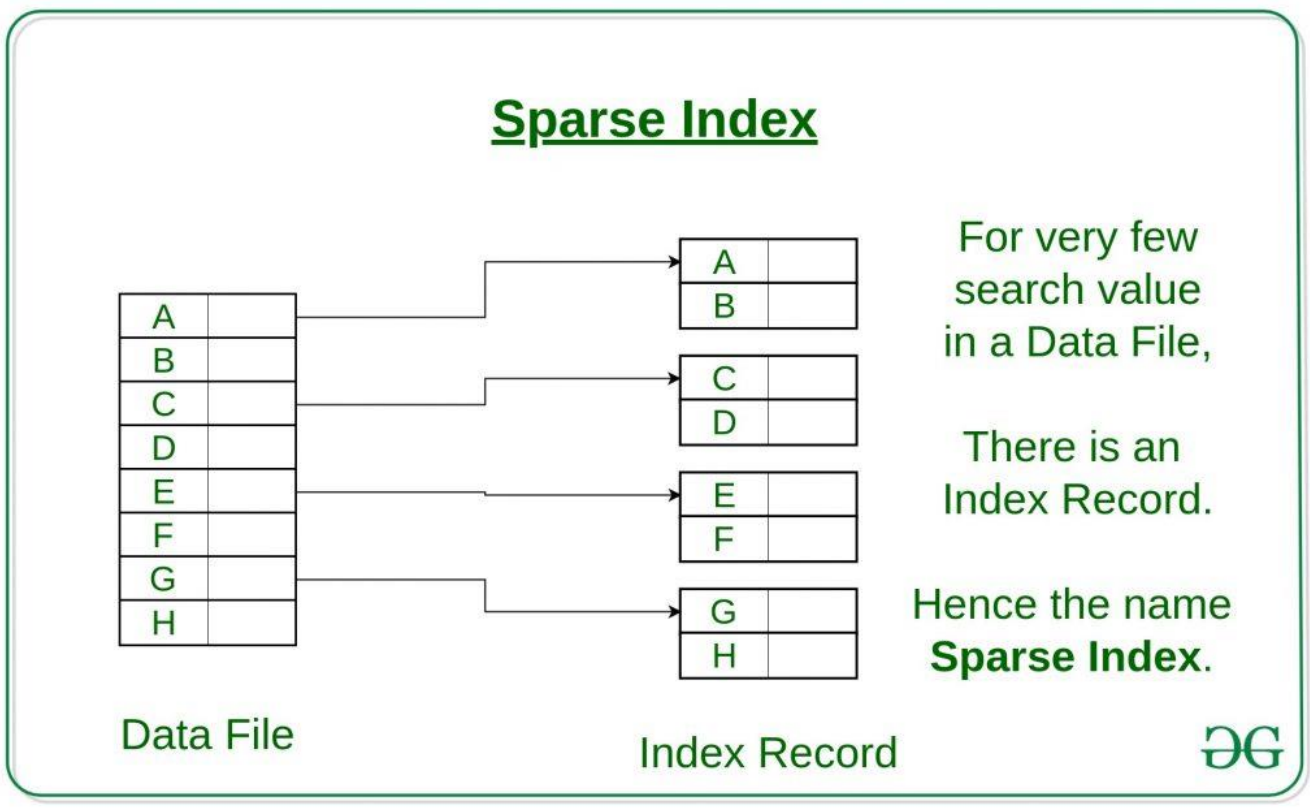
In general, there are two types of file organization mechanism which are followed by the indexing methods to store the data:

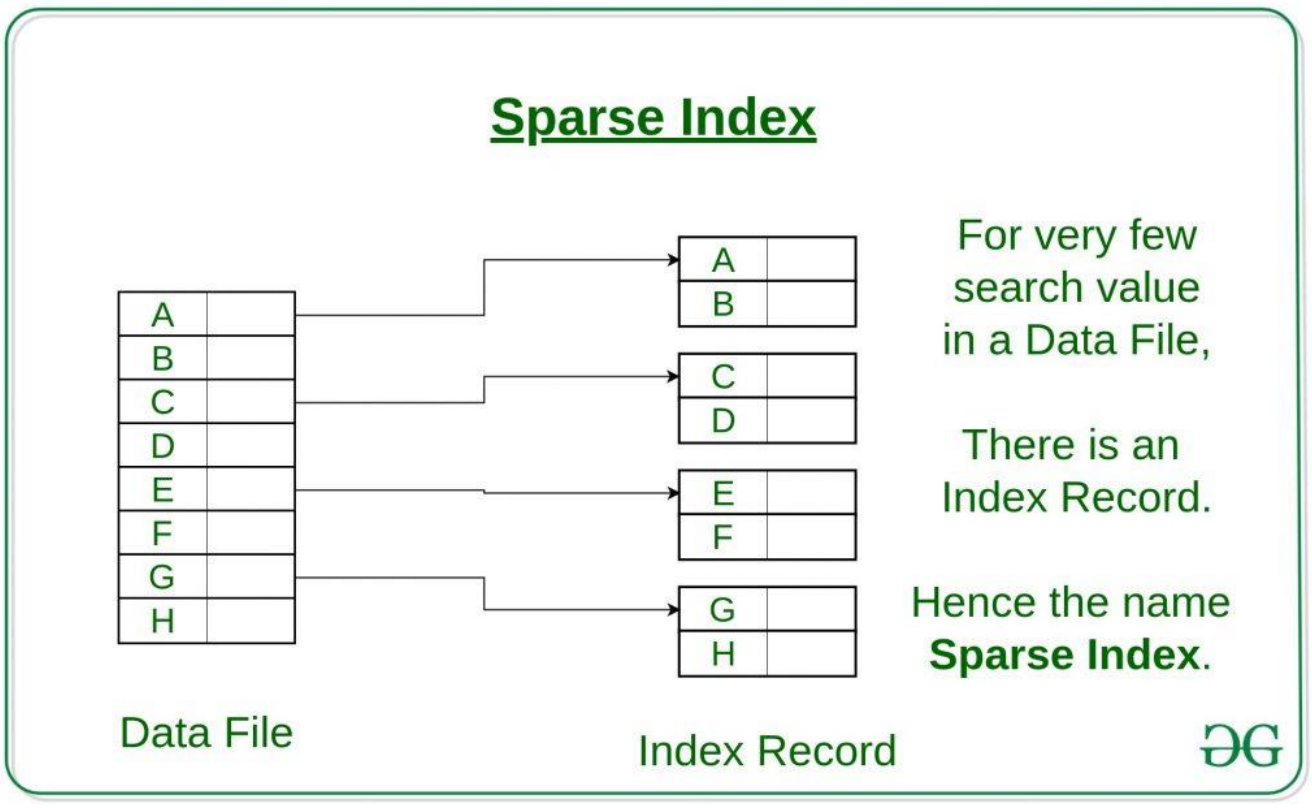
1. **Sequential File Organization or Ordered Index File:** In this, the indices are based on a sorted ordering of the values. These are generally fast and a more traditional type of storing mechanism. These Ordered or Sequential file organization might store the data in a dense or sparse format:

    - *Dense Index:*
        - For every search key value in the data file, there is an index record.
        - This record contains the search key and also a reference to the first data record with that search key value.



- *Sparse Index:*
  - The index record appears only for a few items in the data file. Each item points to a block as shown.
  - To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
  - We start at that record pointed to by the index record, and proceed along with the pointers in the file (that is, sequentially) until we find the desired record.





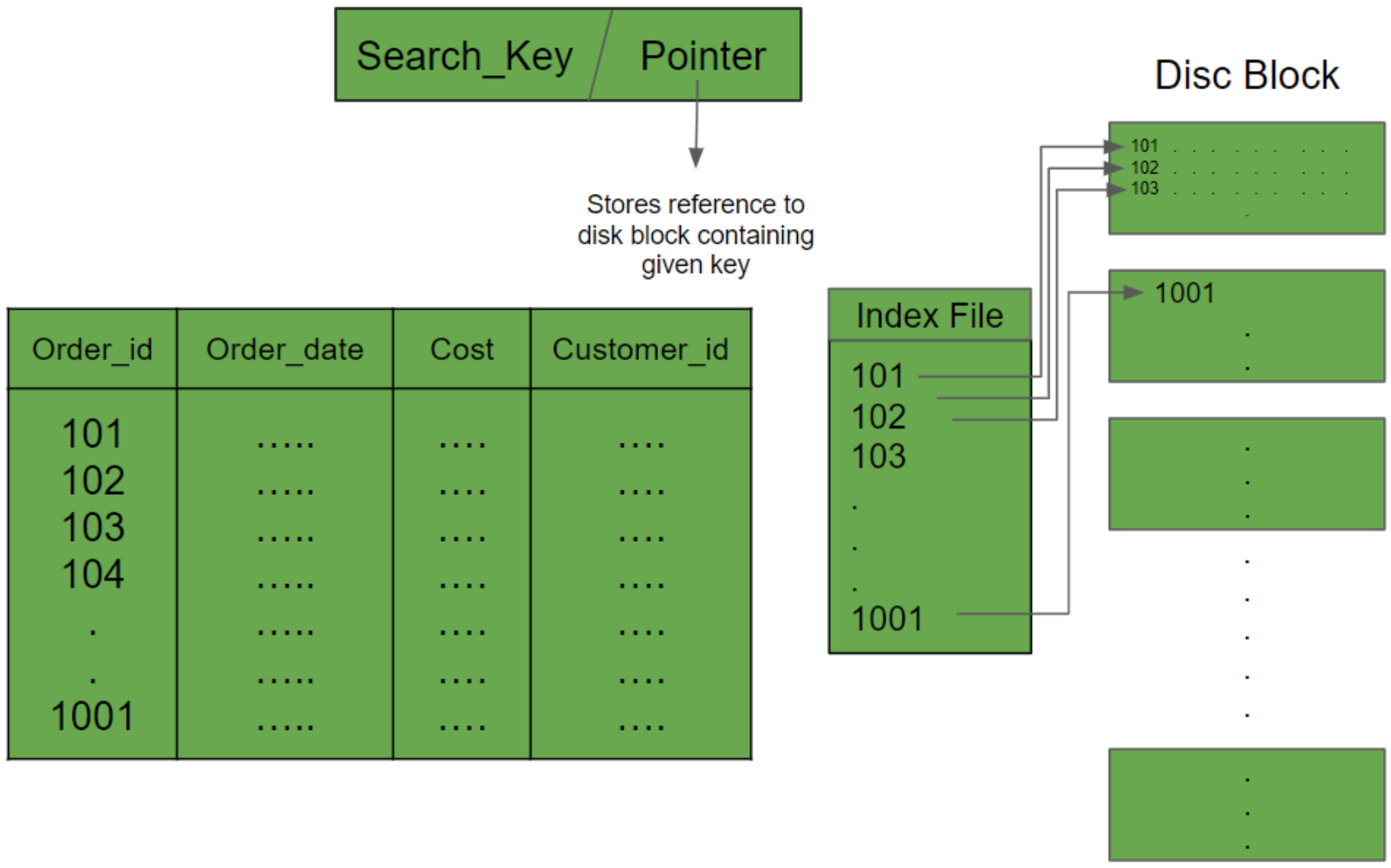
2. **Hash File organization:** Indices are based on the values being distributed uniformly across a range of buckets. The buckets to which a value is assigned is determined by a function called a hash function.

There are primarily two methods of indexing:

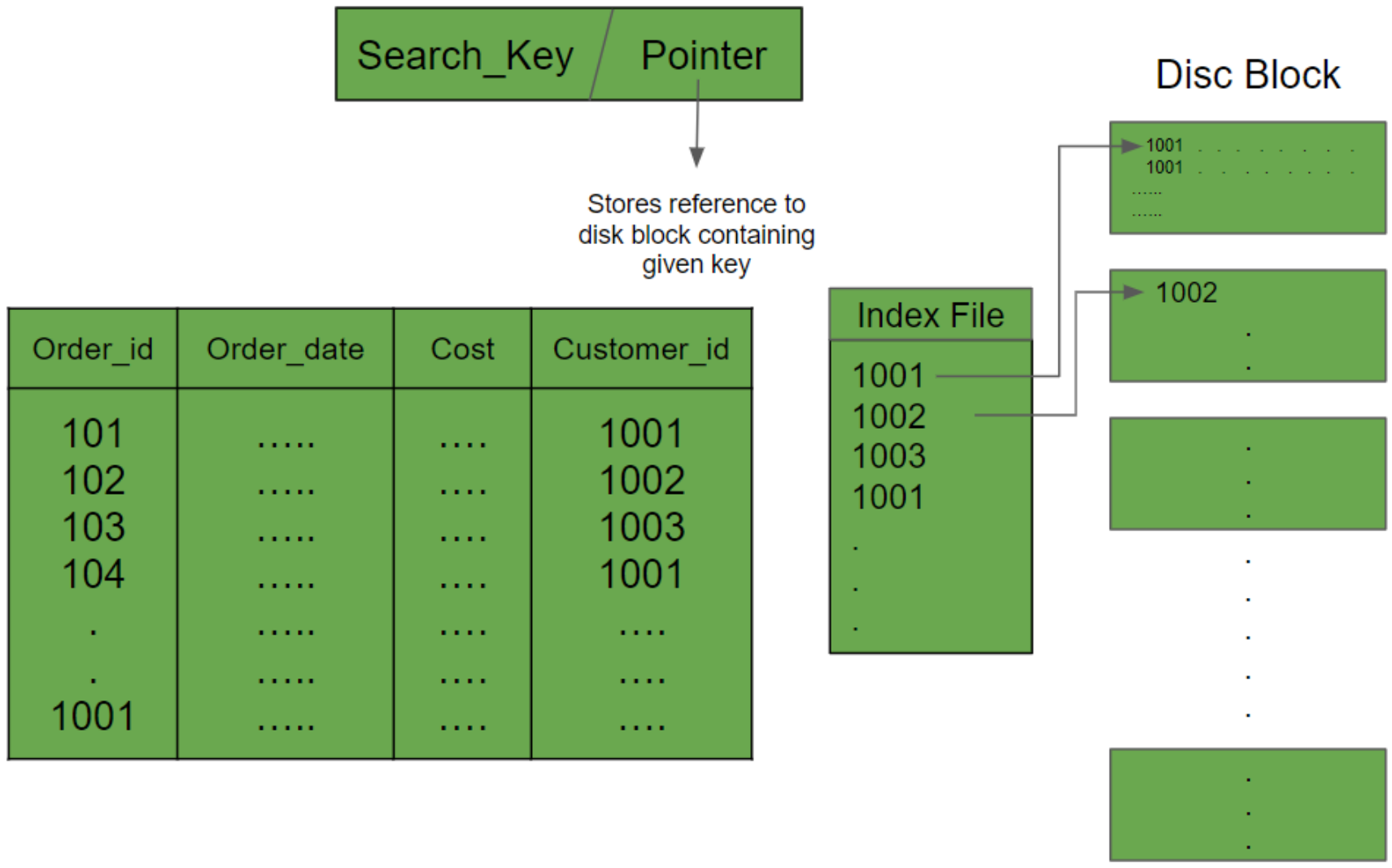
- Clustered Indexing
- Non-Clustered or Secondary Indexing
- Multilevel Indexing

1. **Clustered Indexing** Clustering index is defined on an ordered data file. The data file is ordered on a non-key field. In some cases, the index is created on non-primary key columns which may not be unique for each record. In such cases, in order to identify the records faster, we will group two or more columns together to get the unique values and create index out of them. This method is known as the clustering index. Basically, records with similar characteristics are grouped together and indexes are created for these groups.

   Let's look at this online retailing database, where each Order_id is stored along with Order_date, Cost and Customer_ID.

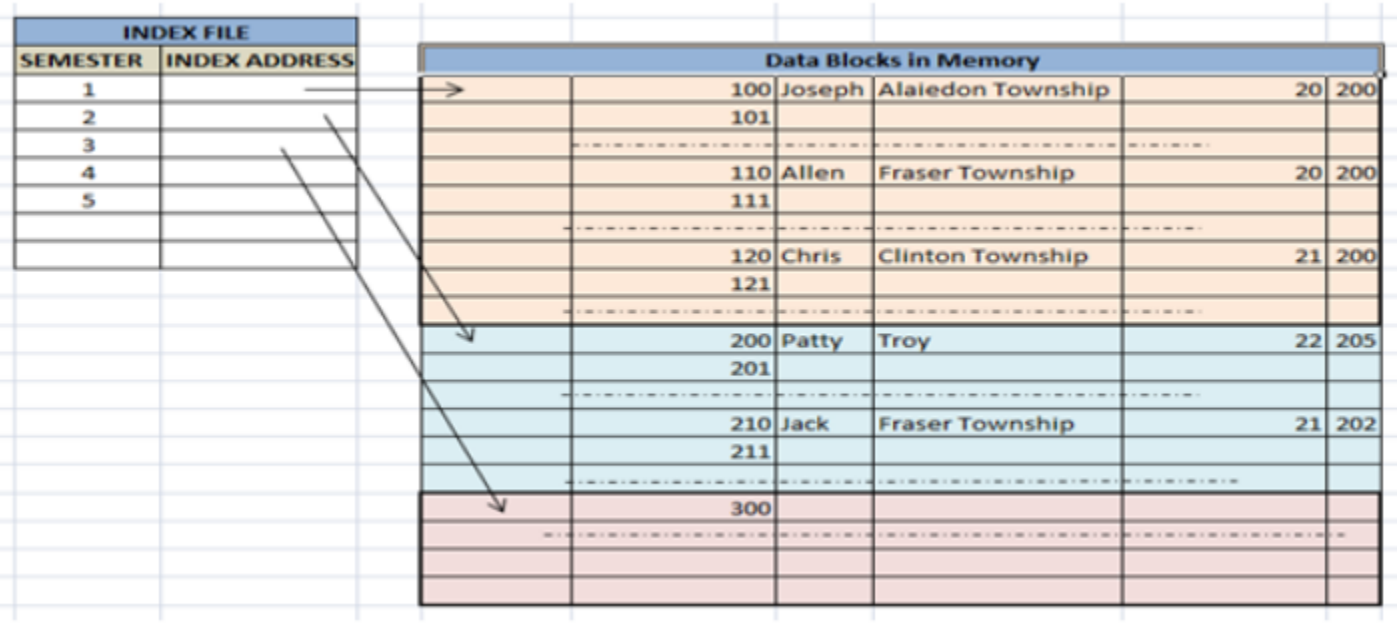


Working is easy to understand. The DBMS maintains an extra index file in clustered indexing. It has entries in the form of Search_key and Pointer, which stores the reference to disk block containing the given key. Here we have assumed that the Order_id is our Primary key on which indexing is done. So the Seach_key consists of the Order_id. The index file can store the data in either the Order Index File or Hashing Index File as shown earlier. The order indexing can be done in either Dense indexing format or sparse indexing format as we have learnt earlier. Dense Indexing saves the time of search and Sparse Index saves index file size.
Let's look at another type where Clustered indexing is done on Customer_ID instead of Order_id.



Although there can be multiple kleys in CUstomer_ID and Disk Blocks, but there is only one entry in the Index File.

For example, students studying in each semester are grouped together. i.e. 1st Semester students, 2nd semester students, 3rd semester students etc are grouped.

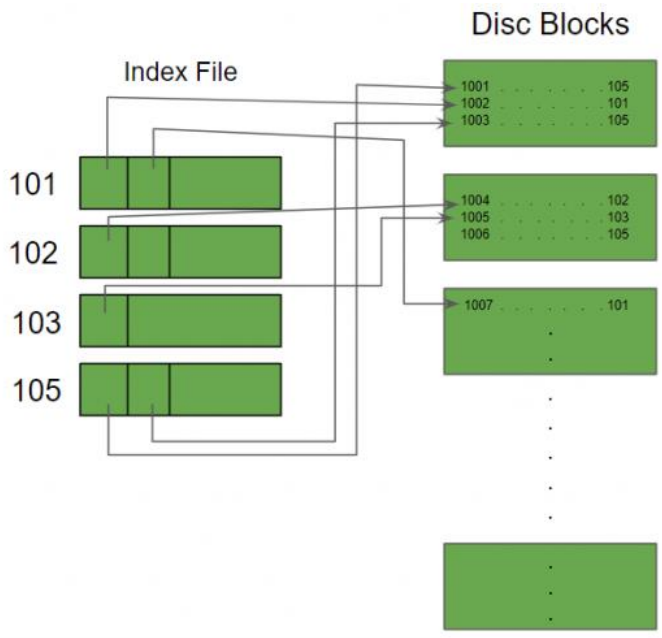


Clustered index sorted according to first name (Search key)

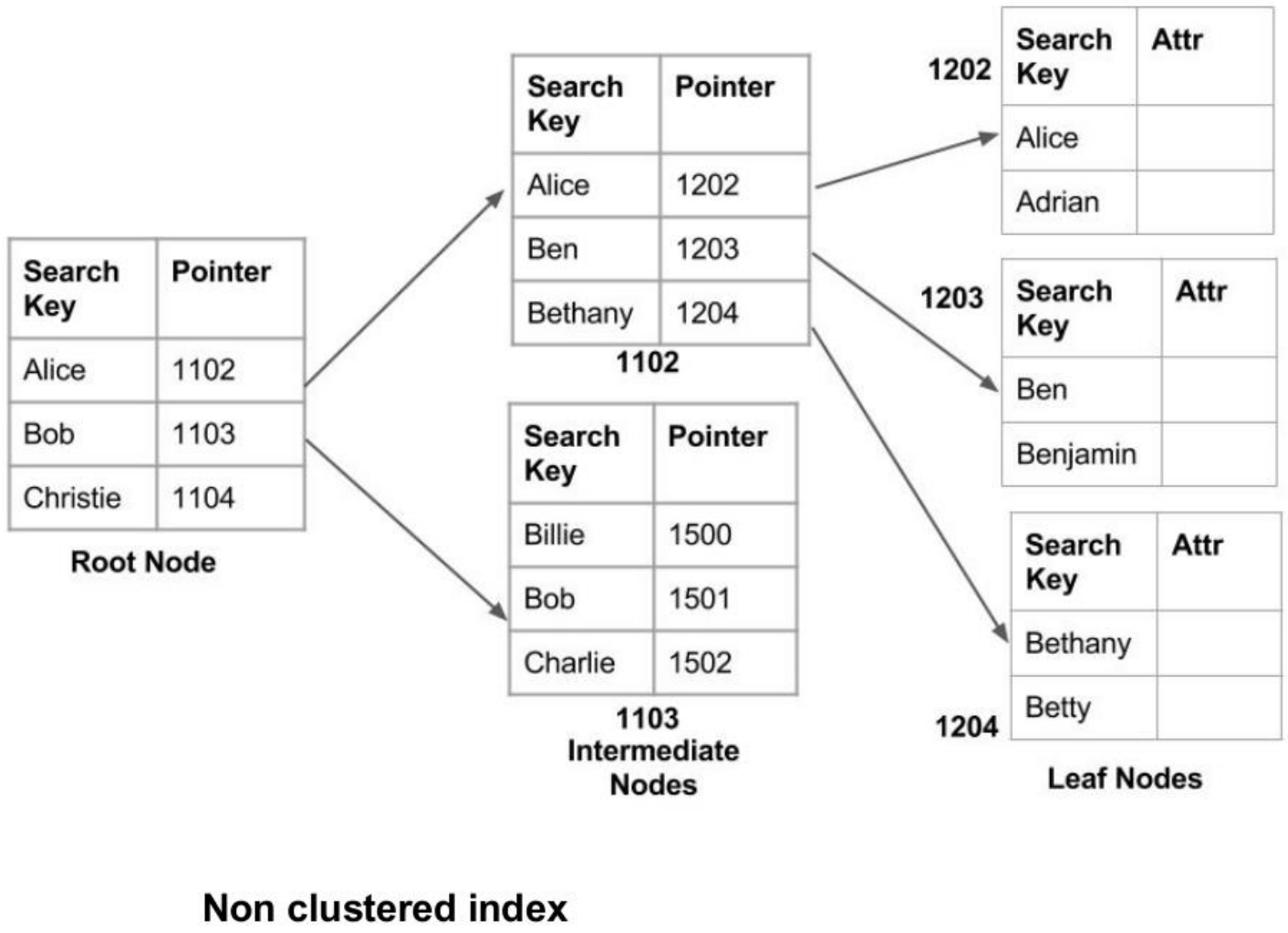
Primary Indexing: This is a type of Clustered Indexing wherein the data is sorted according to the search key and the primary key of the database table is used to create the index. It is a default format of indexing where it induces sequential file organization. As primary keys are unique and are stored in a sorted manner, the performance of the searching operation is quite efficient.

2. **Non-clustered or Secondary Indexing** A non clustered index just tells us where the data lies, i.e. it gives us a list of virtual pointers or references to the location where the data is actually stored. Data is not physically stored in the order of the index. Instead, data is present in leaf nodes. For eg. the contents page of a book. Each entry gives us the page number or location of the information stored. The actual data here(information on each page of the book) is not organized but we have an ordered reference(contents page) to where the data points actually lie. We can have only dense ordering in the non-clustered index as sparse ordering is not possible because data is not physically organized accordingly.
   It requires more time as compared to the clustered index because some amount of extra work is done in order to extract the data by further following the pointer. In the case of a clustered index, data is directly present in front of the index.
   Let's look at the same order table and see how the data is arranged in a non-clustered way in an index file.

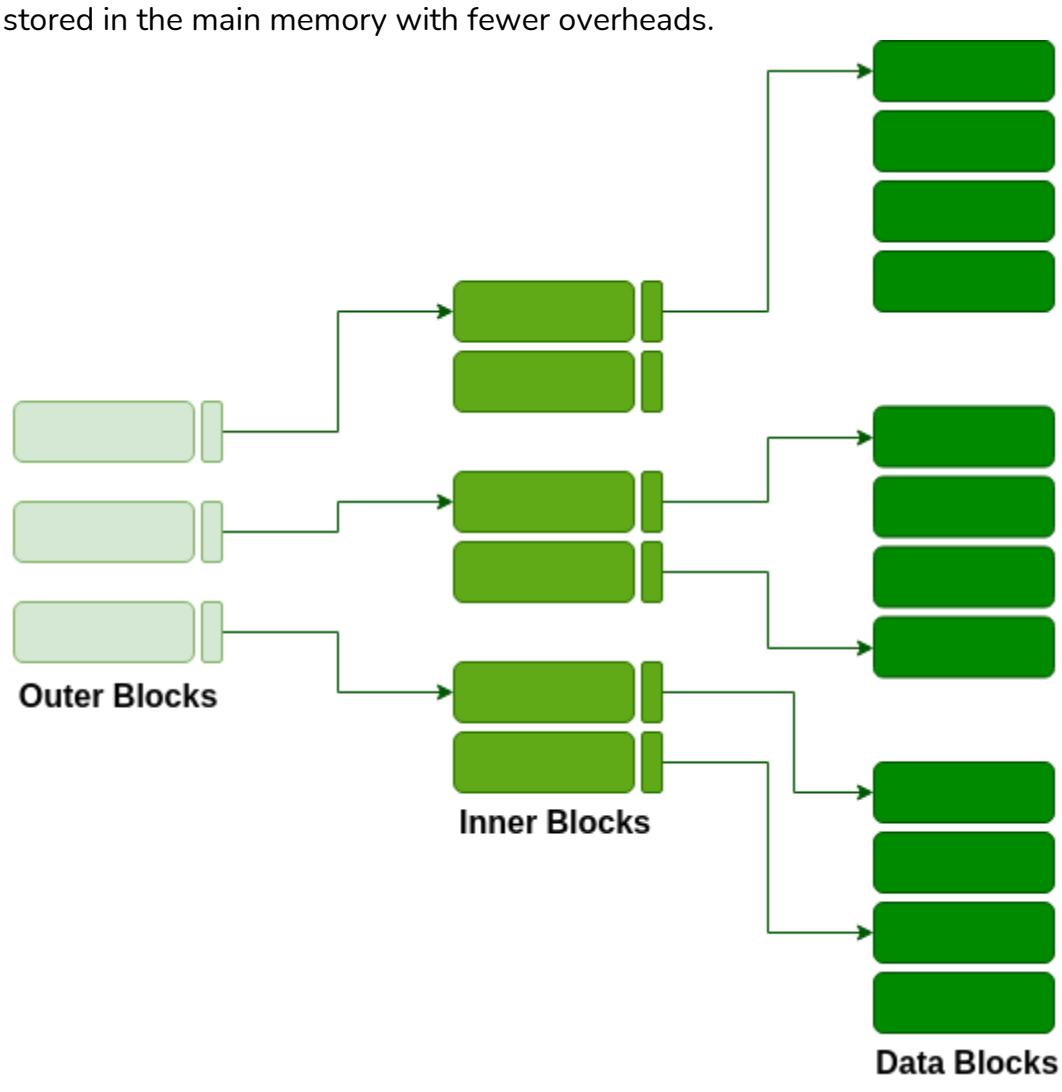
Order_id	Order_date	Cost	Customer_id
1001	15-06-19	2000	105
1002	15-06-19	5000	101
1003	15-06-19	3000	105
1004	15-06-19	500	102
1005	15-06-19	20000	103
1006	15-06-19	5000	105
1007	15-06-19	30000	101
.	.	.	.
.	.	.	.
.	.	.	.



Here we did the indexing according to the Customer_ID. In this, the index file has multiple blocks to point to the multiple Customer_ID. The non-clustered index can only be arranged in a dense manner and not in a sparse manner. We have used an ordered form of indexing to understand this example. A secondary index is created only on the most frequently searched attribute.



3. **Multilevel Indexing** With the growth of the size of the database, indices also grow. As the index is stored in the main memory, a single-level index might become too large a size to store with multiple disk accesses. The multilevel indexing segregates the main block into various smaller blocks so that the same can stored in a single block. The outer blocks are divided into inner blocks which in turn are pointed to the data blocks. This can be easily stored in the main memory with fewer overheads.



Introduction of B-Tree

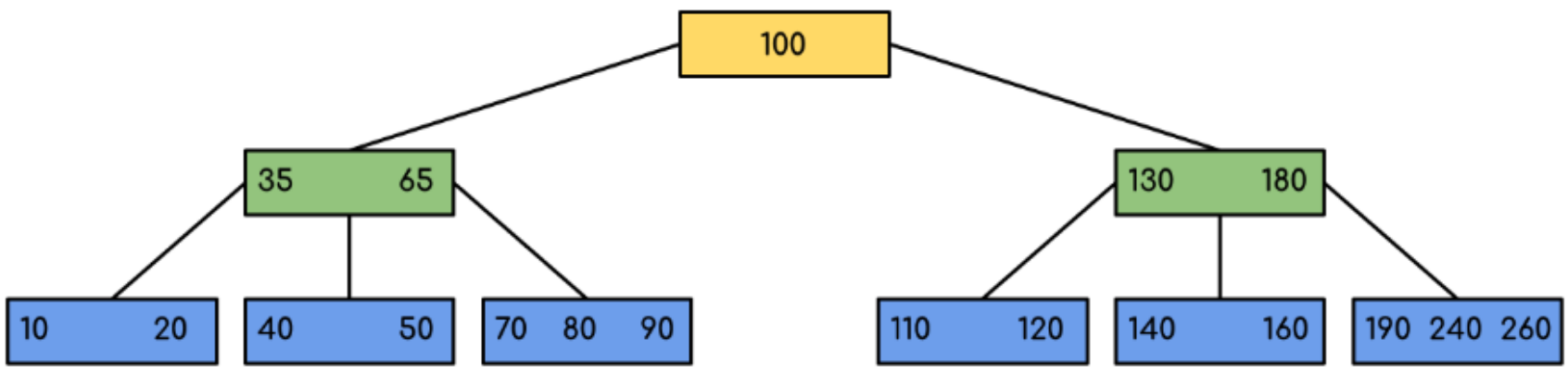
Introduction:
B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size. Since the height of the B-tree is low so total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.
Time Complexity of B-Tree:

Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

"n" is the total number of elements in the B-tree.
Properties of B-Tree:

1. All leaves are at the same level.
2. A B-Tree is defined by the term *minimum degree* 't'. The value of t depends upon disk block size.
3. Every node except root must contain at least t-1 keys. The root may contain minimum 1 key.
4. All nodes (including root) may contain at most 2*t – 1 keys.
5. Number of children of a node is equal to the number of keys in it plus 1.
6. All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
7. B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8. Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(log n).
9. Insertion of a Node in B-Tree happens only at Leaf Node.

Following is an example of B-Tree of minimum order 5. Note that in practical B-Trees, the value of the minimum order is much more than 5.



We can see in the above diagram that all the leaf nodes are at the same level and all non-leaf have no empty sub-tree and have keys one less than the number of their children.
Interesting Facts:

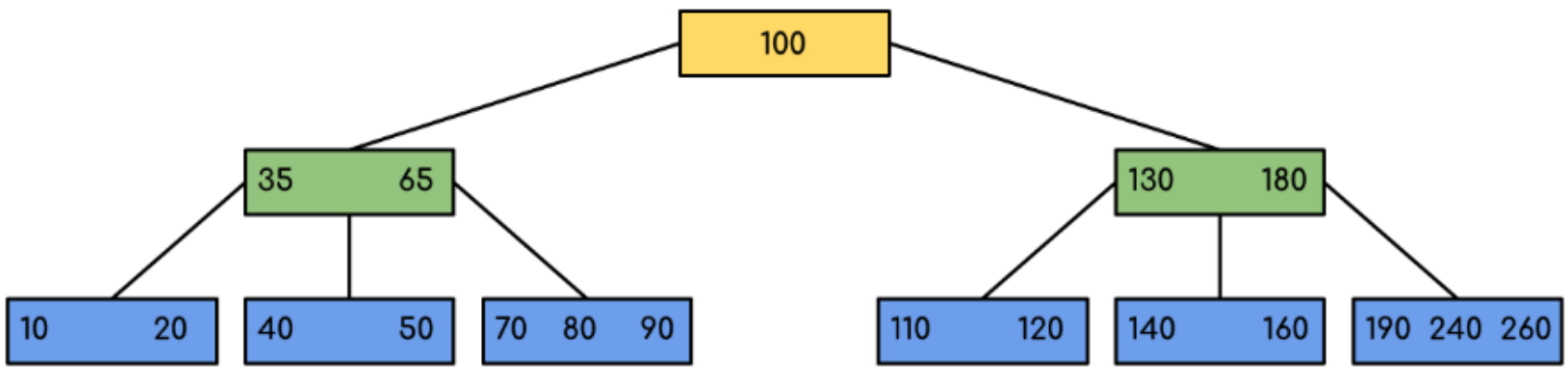
1. The minimum height of the B-Tree that can exist with n number of nodes and m is the maximum number of children of a node can have is: $h_{min}=l\lceil\log\rceil_m(n+1)r\lceil-\rceil 1$hmin=l⌈log⌉m(n+1)r⌈-⌉1
2. The maximum height of the B-Tree that can exist with n number of nodes and t is the minimum number of children that a non-root node can have is: $h_{max}=\lfloor\log_t n+12\rfloor$hmax=⌊logtn+12⌋and $t=l\lceil m2\rceil r\lceil\rceil$t=l⌈m2⌉r⌈⌉

Traversal in B-Tree:
Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

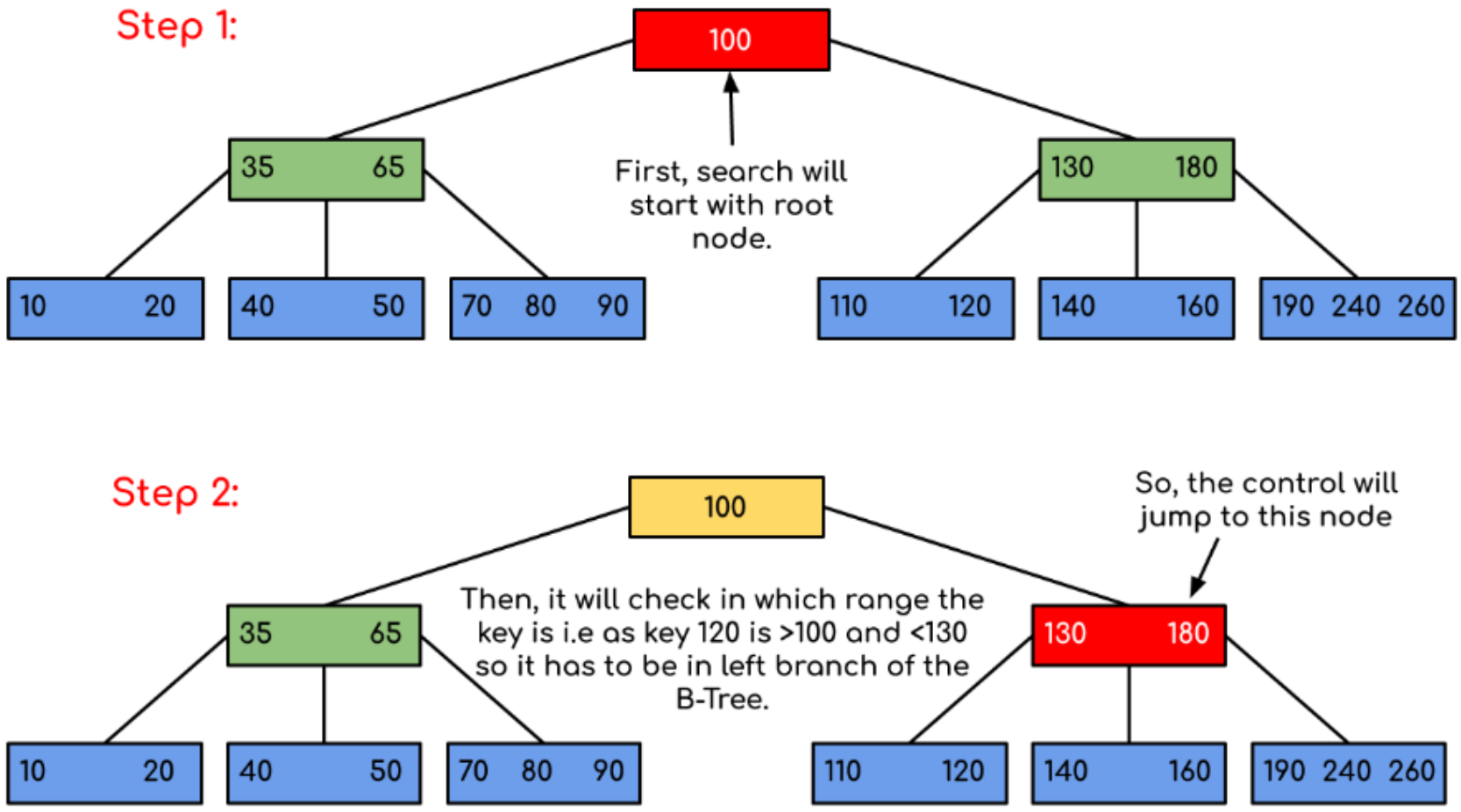
Search Operation in B-Tree:
Search is similar to the search in Binary Search Tree. Let the key to be searched be k. We start from the root and recursively traverse down. For every visited non-leaf node, if the node has the key, we simply return the node. Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL.

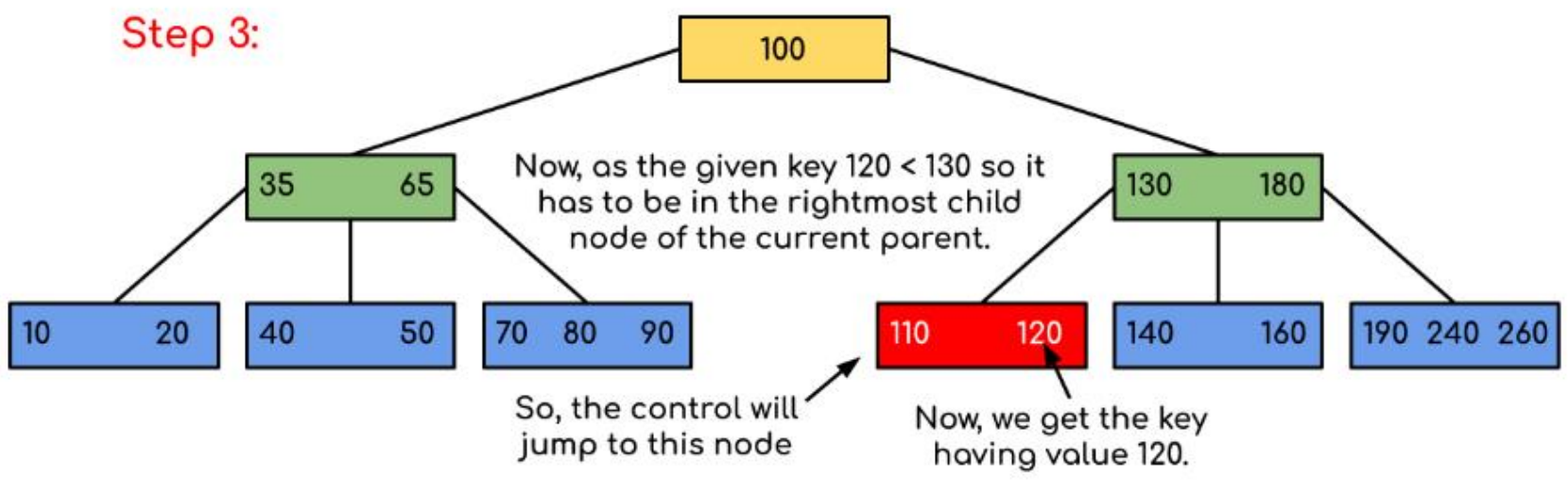
Logic:
Searching a B-Tree is similar to searching a binary tree. The algorithm is similar and goes with recursion. At each level, the search is optimized as if the key value is not present in the range of parent then the key is present in another branch. As these values limit the search they are also known as limiting value or separation value. If we reach a leaf node and don’t find the desired key then it will display NULL.

Example: Searching 120 in the given B-Tree.



Solution:





In this example, we can see that our search was reduced by just limiting the chances where the key containing the value could be present. Similarly if within the above example we've to look for 180, then the control will stop at step 2 because the program will find that the key 180 is present within the current node. And similarly, if it's to seek out 90 then as 90 < 100 so it'll go to the left subtree automatically and therefore the control flow will go similarly as shown within the above example.

C++

```
1
2
3
4
5
6
7
8
9
10
11
```

```
// C++ implementation of search() and traverse() methods
#include<iostream>
using namespace std;
// A BTree node
class BTreeNode
{
    int *keys;  // An array of keys
    int t;      // Minimum degree (defines the range for number of keys)
    BTreeNode **C; // An array of child pointers
    int n;     // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false
public:
    BTreeNode(int _t, bool _leaf);   // Constructor
    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();
    // A function to search a key in the subtree rooted with this node.
    BTreeNode *search(int k);   // returns NULL if k is not present.
// Make the BTree friend of this so that we can access private members of this
// class in BTree functions
friend class BTree;
};
// A BTree
class BTree
{
```

Run

Java C# Javascript

The above code doesn't contain the driver program. We will be covering the complete program in our next post on B-Tree Insertion.

There are two conventions to define a B-Tree, one is to define by minimum degree (followed in Cormen book), second is define by order. We have followed the minimum degree convention and will be following same in coming posts on B-Tree. The variable names used in the above program are also kept same as Cormen book for better readability.

Insertion and Deletion

B-Tree Insertion

B-Tree Deletion

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Introduction of B+ Tree

In order, to implement dynamic multilevel indexing, B-tree and B+ tree are generally employed. The drawback of B-tree used for indexing, however is that it stores the data pointer (a pointer to the disk file block containing the key value), corresponding to a particular key value, along with that key value in the node of a B-tree. This technique, greatly reduces

the number of entries that can be packed into a node of a B-tree, thereby contributing to the increase in the number of levels in the B-tree, hence increasing the search time of a record.

B+ tree eliminates the above drawback by storing data pointers only at the leaf nodes of the tree. Thus, the structure of leaf nodes of a B+ tree is quite different from the structure of internal nodes of the B tree. It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them. Moreover, the leaf nodes are linked to provide ordered access to the records. The leaf nodes, therefore form the first level of index, with the internal nodes forming the other levels of a multilevel index. Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record.

From the above discussion it is apparent that a B+ tree, unlike a B-tree has two orders, 'a' and 'b', one for the internal nodes and the other for the external (or leaf) nodes.

The structure of the internal nodes of a B+ tree of order 'a' is as follows:

1. Each internal node is of the form :
   $<P_1, K_1, P_2, K_2, ..., P_{c-1}, K_{c-1}, P_c>$
   where c <= a and each **P_i is a tree pointer (i.e points to another node of the tree)** and, each **K_i is a key value** (see diagram-I for reference).
2. Every internal node has : $K_1 < K_2 < .... < K_{c-1}$
3. For each search field values 'X' in the sub-tree pointed at by P_i, the following condition holds :
   $K_{i-1} < X <= K_i$, for $1 < i < c$ and,
   $K_{i-1} < X$, for $i = c$
   (See diagram I for reference)
4. Each internal nodes has at most 'a' tree pointers.
5. The root node has, at least two tree pointers, while the other internal nodes have at least \ceil(a/2) tree pointers each.
6. If any internal node has 'c' pointers, c <= a, then it has 'c - 1' key values.

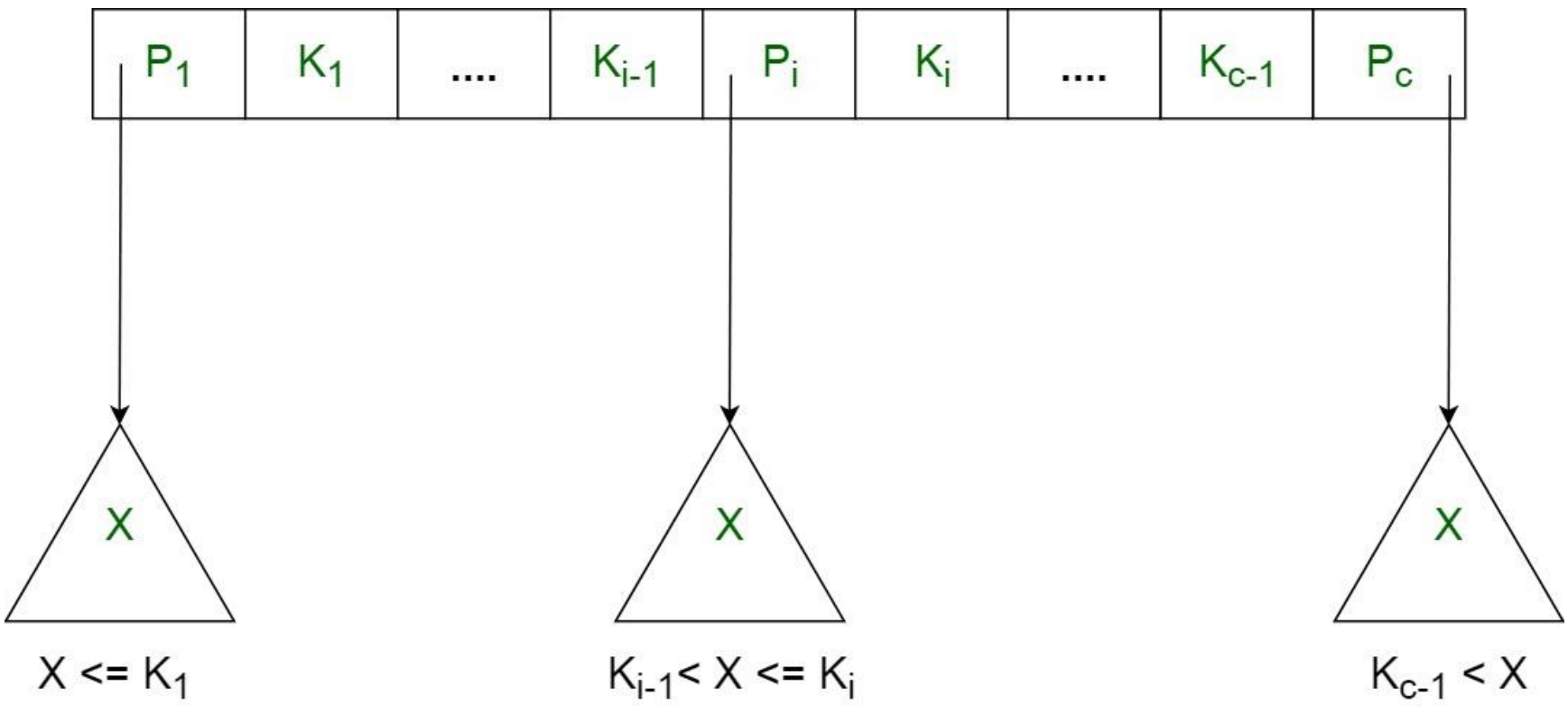


Diagram-I

The structure of the leaf nodes of a B+ tree of order 'b' is as follows:

1. Each leaf node is of the form :
   $<<K_1, D_1>, <K_2, D_2>, ..., <K_{c-1}, D_{c-1}>, P_{next}>$
   where c <= b and each **D_i is a data pointer (i.e points to actual record in the disk whose key value is K_i or to a disk file block containing that record)** and, each **K_i is a key value** and, **P_{next} points to next leaf node in the B+ tree** (see diagram II for reference).
2. Every leaf node has : $K_1 < K_2 < .... < K_{c-1}$, c <= b
3. Each leaf node has at least \ceil(b/2) values.
4. All leaf nodes are at same level.

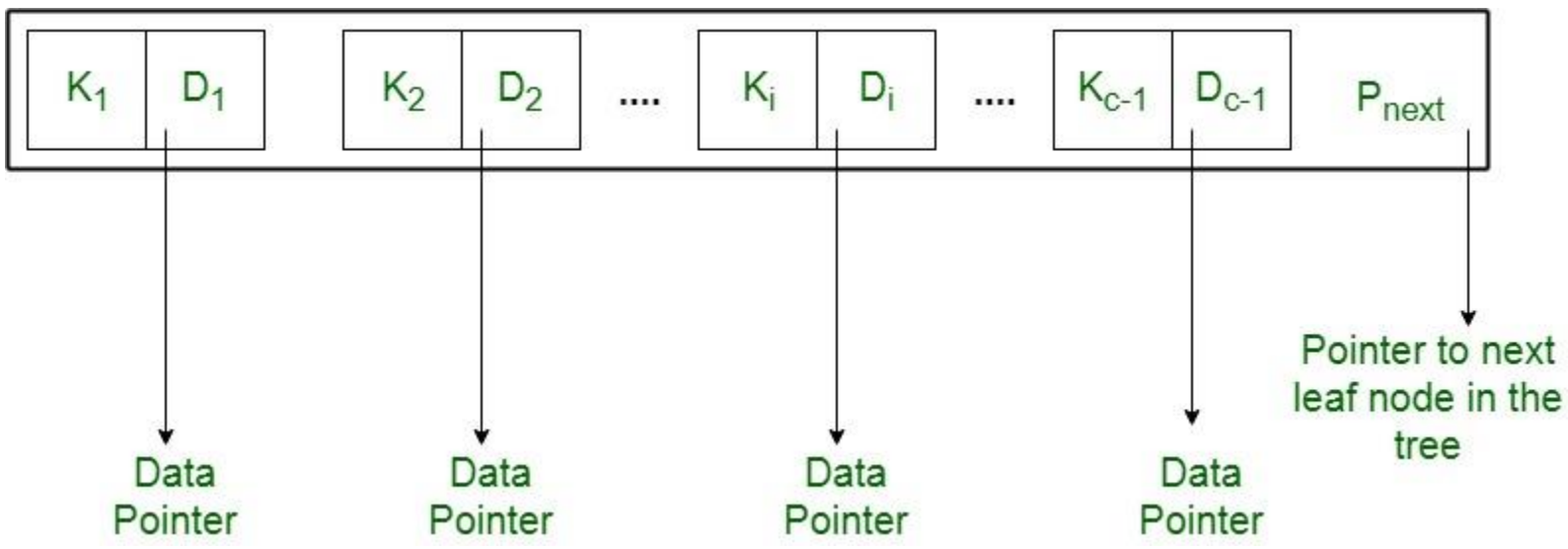
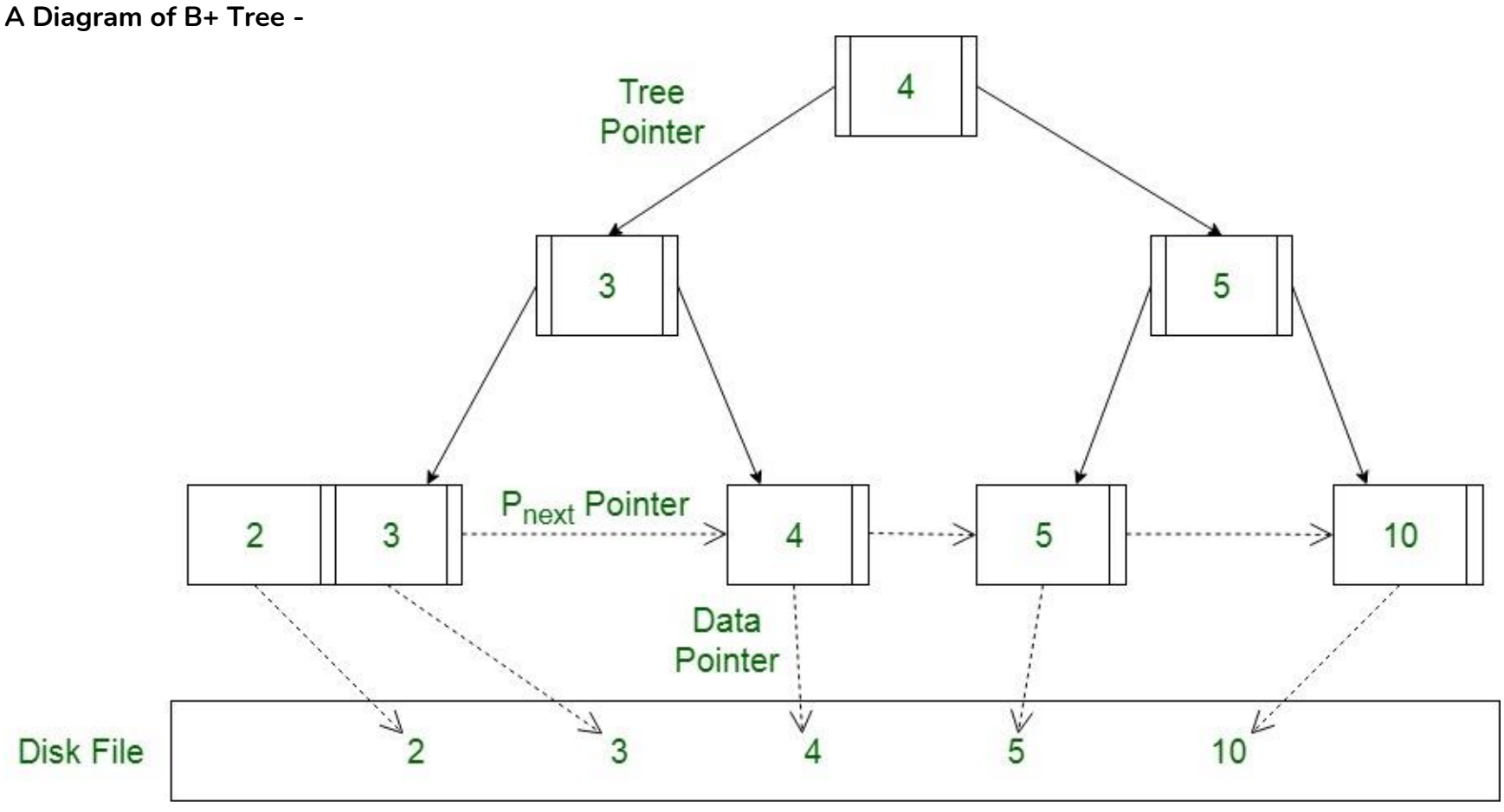


Diagram-II

Using the P_{next} pointer it is viable to traverse all the leaf nodes, just like a linked list, thereby achieving ordered access to the records stored in the disk.

A Diagram of B+ Tree -



Advantage - A B+ tree with 'l' levels can store more entries in its internal nodes compared to a B-tree having the same 'l' levels. This accentuates the significant improvement made to the search time for any given key. Having lesser levels and presence of P_{next} pointers imply that B+ tree are very quick and efficient in accessing records from disks.

DBMS | Concurrency Control - Introduction and ACID Properties

Concurrency Control deals with **interleaved execution** of more than one transaction. In the next article, we will see what is serializability and how to find whether a schedule is serializable or not.

What is Transaction?

A set of logically related operations is known as transaction. The main operations of a transaction are:

Read(A): Read operations Read(A) or R(A) reads the value of A from the database and stores it in a buffer in main memory.

Write (A): Write operation Write(A) or W(A) writes the value back to the database from buffer.

Let us take a debit transaction from an account which consists of following operations:

1. R(A);
2. A=A-1000;
3. W(A);

Assume A's value before starting of transaction is 5000.

- The first operation reads the value of A from database and stores it in a buffer.
- Second operation will decrease its value by 1000. So buffer will contain 4000.
- Third operation will write the value from buffer to database. So A's final value will be 4000.

But it may also be possible that transaction may fail after executing some of its operations. The failure can be because of **hardware, software or power** etc. For example, if debit transaction discussed above fails after executing operation 2, the value of A will remain 5000 in the database which is not acceptable by the bank. To avoid this, Database has two important operations:

Commit: After all instructions of a transaction are successfully executed, the changes made by transaction are made permanent in the database.

Rollback: If a transaction is not able to execute all operations successfully, all the changes made by transaction are undone.

Properties of a transaction

- **Atomicity**: By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.
  --**Abort**: If a transaction aborts, changes made to database are not visible.
  --**Commit**: If a transaction commits, changes made are visible.
  Atomicity is also known as the 'All or nothing rule'.
  Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X − 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

  If the transaction fails after completion of **T1** but before completion of **T2**.(say, after **write(X)** but before **write(Y)**), then amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.
- **Consistency**: This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above,
  The total amount before and after the transaction must be maintained.
  Total **before T** occurs = **500 + 200 = 700**.
  Total **after T occurs** = **400 + 300 = 700**.
  Therefore, database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result, T is incomplete.
- **Isolation**: This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.
  Let **X**= 500, **Y** = 500.
  Consider two transactions **T** and **T''.**

T	T''
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y − 50 Write	Read (X) Read (Y) Z: = X + Y Write (Z)

  Suppose **T** has been executed till **Read (Y)** and then **T''** starts. As a result , interleaving of operations takes place due to which **T''** reads correct value of **X** but incorrect value of **Y** and sum computed by
  T'': (X+Y = 50, 000+500=50, 500)
  is thus not consistent with the sum at end of transaction: **T: (X+Y = 50, 000 + 450 = 50, 450)**.

This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

- **Durability:** This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

The **ACID** properties, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably stored.

DBMS | Concurrency Control | Schedule and Types of Schedules

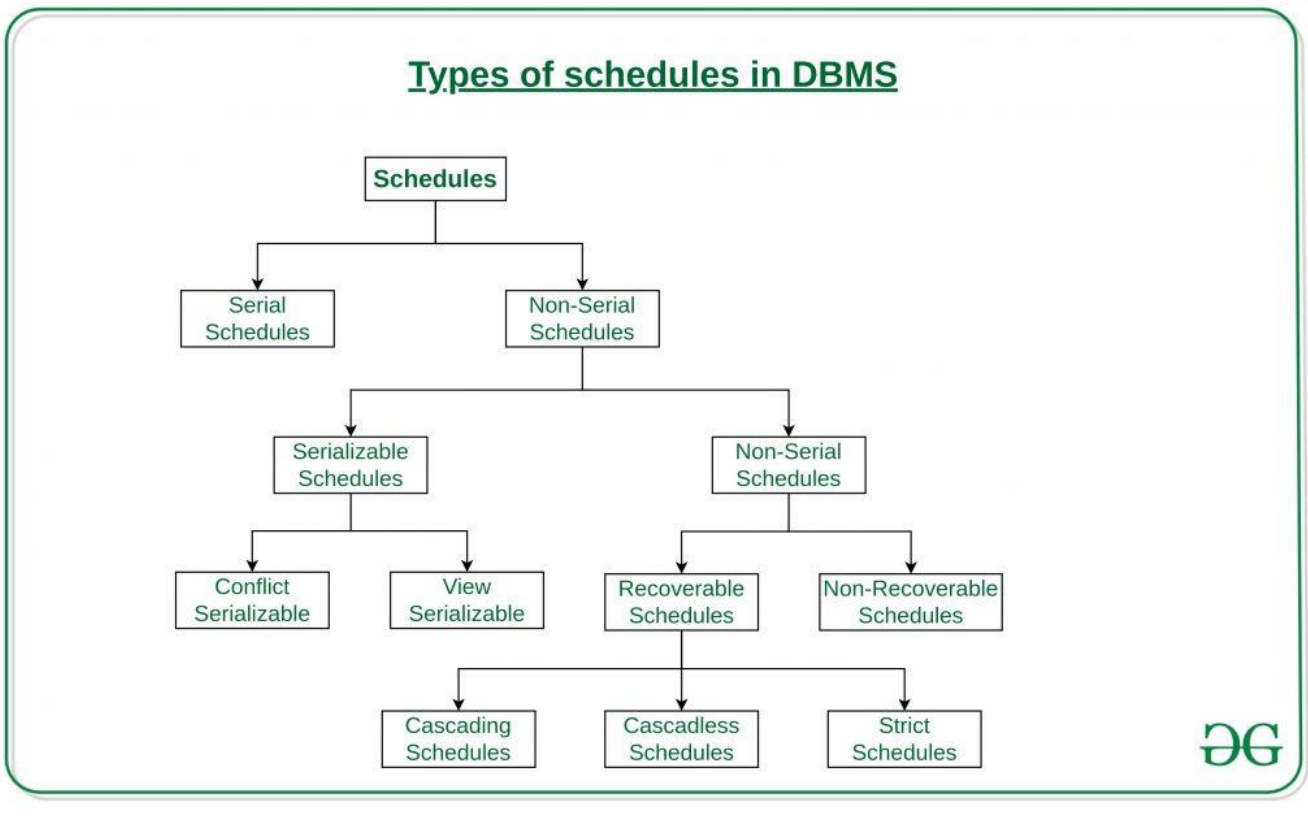
What is a Schedule?

A schedule is a series of operations from one or more transactions. Schedules are used to resolve conflicts between the transactions.

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

Types of Schedules?

Lets discuss various types of schedules.



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

   Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
R(B)	
	W(B)
	R(A)
	R(B)

2.
   where R(A) denotes that a read operation is performed on some data item 'A'
   This is a serial schedule since the transactions perform serially in the order T_1 —> T_2

3. **Non-Serial Schedule:** This is a type of Scheduling where the operations of multiple transactions are interleaved. This might lead to a rise in the concurrency problem. The transactions are executed in a non-serial manner, keeping the end result correct and same as the serial schedule. Unlike the serial schedule where one transaction must wait for another to complete all its operation, in the non-serial schedule, the other transaction proceeds without waiting for the previous transaction to complete. This sort of schedule does not provide any benefit of the concurrent transaction. It can be of two types namely, Serializable and Non-Serializable Schedule.

   The Non-Serial Schedule can be divided further into Serializable and Non-Serializable.

   a. **Serializable:** This is used to maintain the consistency of the database. It is mainly used in the Non-Serial scheduling to verify whether the scheduling will lead to any inconsistency or not. On the other hand, a serial schedule does not need the serializability because it follows a transaction only when the previous transaction is complete. The non-serial schedule is said to be in a serializable schedule only when it is equivalent to the serial schedules, for an n number of transactions. Since concurrency is allowed in this case thus, multiple transactions can execute concurrently. A serializable schedule helps in improving both resource utilization and CPU throughput. These are of two types:

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

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

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

   b. **Non-Serializable:** The non-serializable schedule is divided into two types, Recoverable and Non-recoverable Schedule.

      1. **Recoverable Schedule:** Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction T_j is reading value updated or written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i.
         Example 1:

```
2. S1: R1(x), W1(x), R2(x), R1(y), R2(y),

                W2(x), W1(y), C1, C2;
```

Given schedule follows order of **Ti->Tj => C1->C2**. Transaction T1 is executed before T2 hence there is no chances of conflict occur. R1(x) appears before W1(x) and transaction T1 is committed before T2 i.e. completion of first transaction performed first update on data item x, hence given schedule is recoverable.

Example 2: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
commit	
	commit

This is a recoverable schedule since T_1 commits before T_2, that makes the value read by T_2 correct.

There can be three types of recoverable schedule:

a. **Cascading Schedule:** When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such scheduling is referred to as Cascading rollback or cascading abort. Example:

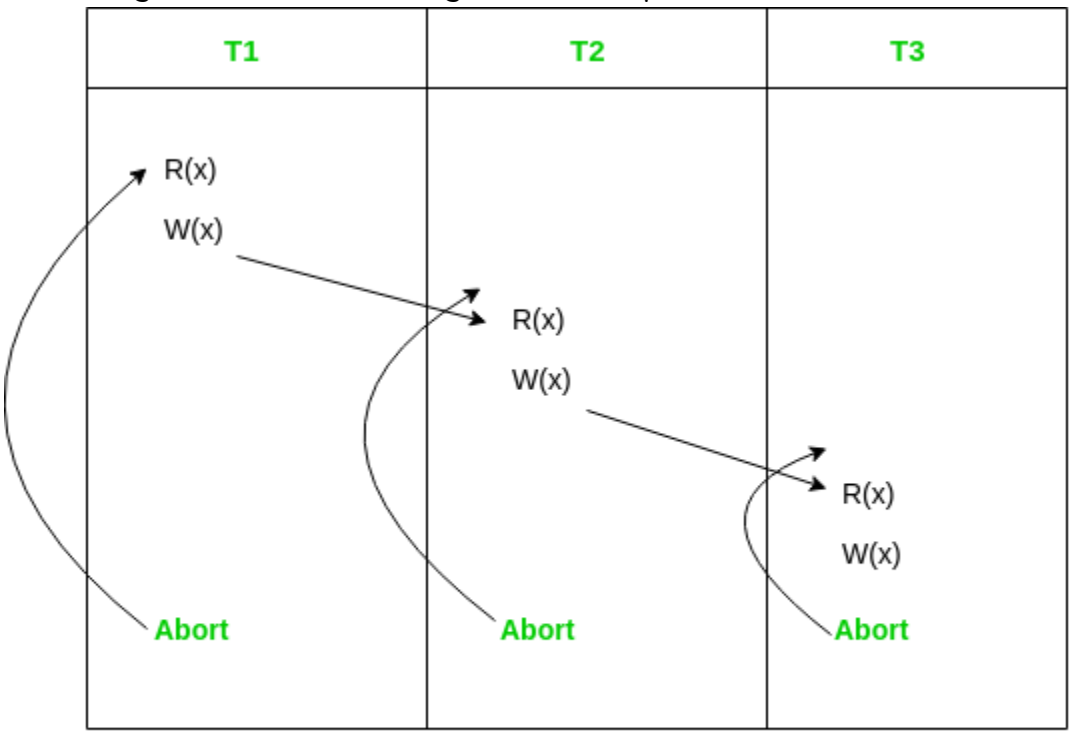


Figure - Cascading Abort

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

   In other words, if some transaction T_j wants to read value updated or written by some other transaction T_i, then the commit of T_j must read it after the commit of T_i.

   Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)

T_1	T_2
commit	
	R(A)
	commit

c.

This schedule is cascadeless. Since the updated value of **A** is read by T_2 only after the updating transaction i.e. T_1 commits.

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
abort	
	abort

d.

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

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

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
	R(A)
W(A)	
commit	
	W(A)
	R(A)
	commit

f.

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

c.

1. **Non-Recoverable Schedule:** The table below shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. T2 commits. But later on, T1 fails. So we have to rollback T1. Since T2 has read the value written by T1, it should also be rollbacked. But we have already committed that. So this schedule is irrecoverable schedule. When Tj is reading the value updated by Ti and Tj is committed before committing of Ti, the schedule will be irrecoverable.

T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
		Commit;		
Failure Point				
Commit;				

Note - It can be seen that:

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

DBMS | Conflicting Serializability

Conflict Serializable: A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

Conflicting operations: Two operations are said to be conflicting if all conditions satisfy:

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

Example: -

- **Conflicting** operations pair ($R_1(A)$, $W_2(A)$) because they belong to two different transactions on same data item A and one of them is write operation.
- Similarly, ($W_1(A)$, $W_2(A)$) and ($W_1(A)$, $R_2(A)$) pairs are also **conflicting**.
- On the other hand, ($R_1(A)$, $W_2(B)$) pair is **non-conflicting** because they operate on different data item.
- Similarly, (($W_1(A)$, $W_2(B)$) pair is **non-conflicting.**

Consider the following schedule:

```
S1: R1(A), W1(A), R2(A), W2(A), R1(B), W1(B), R2(B), W2(B)
```

If O_i and O_j are two operations in a transaction and $O_i < O_j$ (O_i is executed before O_j), same order will follow in schedule as well. Using this property, we can get two transactions of schedule S1 as:

```
T1: R1(A), W1(A), R1(B), W1(B)

T2: R2(A), W2(A), R2(B), W2(B)
```

Possible Serial Schedules are: T1->T2 or T2->T1

-> **Swapping non-conflicting operation**s $R_2(A)$ and $R_1(B)$ in S1, the schedule becomes,

```
S11: R1(A), W1(A), R1(B), W2(A), R2(A), W1(B), R2(B), W2(B)
```

-> Similarly, s**wapping non-conflicting operations** $W_2(A)$ and $W_1(B)$ in S11, the schedule becomes,

```
S12: R1(A), W1(A), R1(B), W1(B), R2(A), W2(A), R2(B), W2(B)
```

S12 is a serial schedule in which all operations of T1 are performed before starting any operation of T2. Since S has been transformed into a serial schedule S12 by swapping non-conflicting operations of S1, S1 is conflict serializable.

Let us take another Schedule:

```
S2: R₂(A), W₂(A), R₁(A), W₁(A), R₁(B), W₁(B), R₂(B), W₂(B)
```

Two transactions will be:

```
T1: R₁(A), W₁(A), R₁(B), W₁(B)

T2: R₂(A), W₂(A), R₂(B), W₂(B)
```

Possible Serial Schedules are: T1->T2 or T2->T1
Original Schedule is:

```
S2: R₂(A), W₂(A), R₁(A), W₁(A), R₁(B), W₁(B), R₂(B), W₂(B)
```

Swapping non-conflicting operations $R_1(A)$ and $R_2(B)$ in S2, the schedule becomes,

```
S21: R₂(A), W₂(A), R₂(B), W₁(A), R₁(B), W₁(B), R₁(A), W₂(B)
```

Similarly, swapping non-conflicting operations $W_1(A)$ and $W_2(B)$ in S21, the schedule becomes,

```
S22: R₂(A), W₂(A), R₂(B), W₂(B), R₁(B), W₁(B), R₁(A), W₁(A)
```

In schedule S22, all operations of T2 are performed first, but operations of T1 are not in order (order should be $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$). So S2 is not conflict serializable.

Conflict Equivalent: Two schedules are said to be conflict equivalent when one can be transformed to another by swapping non-conflicting operations. In the example discussed above, S11 is conflict equivalent to S1 (S1 can be converted to S11 by swapping non-conflicting operations). Similarly, S11 is conflict equivalent to S12 and so on.

Note 1: *Although S2 is not conflict serializable, but still it is conflict equivalent to S21 and S21 because S2 can be converted to S21 and S22 by swapping non-conflicting operations.*

Note 2: *The schedule which is conflict serializable is always conflict equivalent to one of the serial schedule. S1 schedule discussed above (which is conflict serializable) is equivalent to serial schedule (T1->T2).*

Question: Consider the following schedules involving two transactions. Which one of the following statement is true?
S1: $R_1(X)$ $R_1(Y)$ $R_2(X)$ $R_2(Y)$ $W_2(Y)$ $W_1(X)$

S2: $R_1(X)$ $R_2(X)$ $R_2(Y)$ $W_2(Y)$ $R_1(Y)$ $W_1(X)$

- Both S1 and S2 are conflict serializable
- Only S1 is conflict serializable
- Only S2 is conflict serializable
- None

[GATE 2007]

Solution: Two transactions of given schedules are:

```
T1: R₁(X) R₁(Y) W₁(X)

T2: R₂(X) R₂(Y) W₂(Y)
```

Let us first check serializability of S1:

```
S1: R₁(X) R₁(Y) R₂(X) R₂(Y) W₂(Y) W₁(X)
```

To convert it to a serial schedule, we have to swap non-conflicting operations so that S1 becomes equivalent to serial schedule T1->T2 or T2->T1. In this case, to convert it to a serial schedule, we must have to swap $R_2(X)$ and $W_1(X)$ but they are conflicting. So S1 can't be converted to a serial schedule.

Now, let us check serializability of S2:

S2: **$R_1(X)$ $R_2(X)$** $R_2(Y)$ $W_2(Y)$ $R_1(Y)$ $W_1(X)$

Swapping non conflicting operations $R_1(X)$ and $R_2(X)$ of S2, we get

S2': $R_2(X)$ **$R_1(X)$ $R_2(Y)$** $W_2(Y)$ $R_1(Y)$ $W_1(X)$

Again, swapping non conflicting operations $R_1(X)$ and $R_2(Y)$ of S2', we get

S2'': $R_2(X)$ $R_2(Y)$ **$R_1(X)$ $W_2(Y)$** $R_1(Y)$ $W_1(X)$

Again, swapping non conflicting operations $R_1(X)$ and $W_2(Y)$ of S2'', we get

S2''': $R_2(X)$ $R_2(Y)$ $W_2(Y)$ $R_1(X)$ $R_1(Y)$ $W_1(X)$

which is equivalent to a serial schedule T2->T1.

So, **correct option is C**. Only S2 is conflict serializable.

View Serializability

The concurrent execution is allowed into DBMS so that the execution process is not slowed down, and the resources can be shifted when and where necessary. But the concurrency also creates another problem of resource allocation. If any ongoing transaction is under progress and the interleaving occurs in-between, then this may lead to inconsistency of the data. To maintain consistency, the interleaving must be tracked and checked. Isolation is one factor that must be kept in mind. It signifies whether all the individual transactions are run in an isolation manner or not, even when concurrency and interleaving of the process are allowed. Let's look at the below transactions T1 and T2.

Transfering 10 Rs from X to Y	Transfering 10 Rs from Y to Z
T1	T2
Read(X) X = X - 10 Write(X)	
	Read(Y) Y = Y - 10 Write(Y)
Read(Y) Y = Y + 10 Write(Y)	
	Read(Z) Z = Z + 10 Write(Z)

The impact of the interleaving schedule must be such that T1 is executed before T2(T1T2) or T2 is executed before T1(T2T1). The execution must be equivalent to this condition. This is known as serializability. This is used to verify whether an interleaving schedule is valid or not. These are of two types:

1. View Serializability

2. Conflict Serializability

Let's talk about View Serilizabilty:
Considering the above transactions, to check for View Serializability we need to arrange for both the sequences i.e.,

T1T2 and T2T1 and check whether the original sequence is equivalent to the serializable sequence or not.
Let's generate T1T2 and check whether it is view equivalent of the table:

```
Read(X)
X = X - 10
Write(X)
Read(Y)
Y = Y + 10
Write(Y)

Read(Y)
Y = Y - 10
Write(Y)
Read(Z)
Z = Z + 10
Write(Z)
```

To check for view serializability, we need to check these three conditions for every data item as in X, Y and Z:

1. Initial Read
2. Updated Read
3. Final Write

Initial Read: This specifies that the initial read on every data item must be done by the same transaction, in a serialized manner.
Let's check for T1T2:
For X, this is valid as in both the cases X is read by T1 first.
For Y, this condition fails, as in the serial case T1 reads Y first, but in the original table, it is first read by Y.
So, in this case, the view equivalence fails. Hence no more conditions are needed to be checked for T1T2.

Let's check for T2T1:

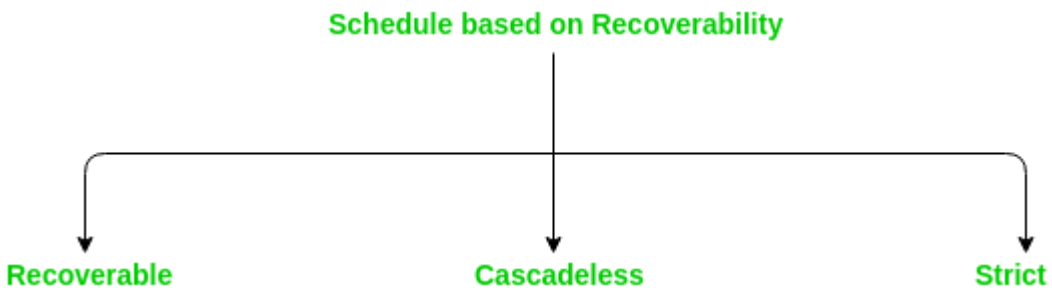
```
Read(Y)
Y = Y - 10
Write(Y)
Read(Z)
Z = Z + 10
Write(Z)

Read(X)
X = X - 10
Write(X)
Read(Y)
Y = Y + 10
Write(Y)
```

For Y, this is valid as in both the cases Y is read by T2 first. We don't need to check for X and Z as they are not shared by both the transactions.

Updated Read: This says that when a transaction is reading a value that is written by another transaction, then the sequence must be the same. We don't need to check for X and Z as they are not sharing. In the case of Y T2T1, Y is written first then T1 reads it. Hence the condition is valid.

Final Write: Let's check for the final write for both the interleaved schedule and the serial schedule. We see that in both the case of T2T1, Y is written in a serial manner.

Therefore, we can conclude that it is viewed serializable in accordance with T2T1.
DBMS | Types of Recoverable Schedules
Generally, there are three types of Recoverable schedule given as follows:



Let's discuss each one of these in detail.

1. **Recoverable Schedule:** A schedule is said to be recoverable if it is recoverable as the name suggest. Only reads are allowed before write operation on the same data. Only reads (Ti->Tj) is permissible.

   Example:

```
S1: R1(x), W1(x), R2(x), R1(y), R2(y),
         W2(x), W1(y), C1, C2;
```

Given schedule follows order of **Ti->Tj => C1->C2**. Transaction T1 is executed before T2 hence there is no chances of conflict occur. R1(x) appears before W1(x) and transaction T1 is committed before T2 i.e. completion of first transaction performed first update on data item x, hence given schedule is recoverable.

Lets see example of **unrecoverable schedule** to clear the concept more:

```
S2: R1(x), R2(x), R1(z), R3(x), R3(y), W1(x),
       W3(y), R2(y), W2(z), W2(y), C1, C2, C3;
```

Ti->Tj => C2->C3 but W3(y) executed before W2(y) which leads to conflicts thus it must be committed before T2 transaction. So given schedule is unrecoverable. if **Ti->Tj => C3->C2** is given in schedule then it will become recoverable schedule.

Note: A committed transaction should never be rollback. It means that reading a value from the uncommitted transaction and commits it will enter the current transaction into the inconsistent or unrecoverable state this is called *Dirty Read problem*.

Example:

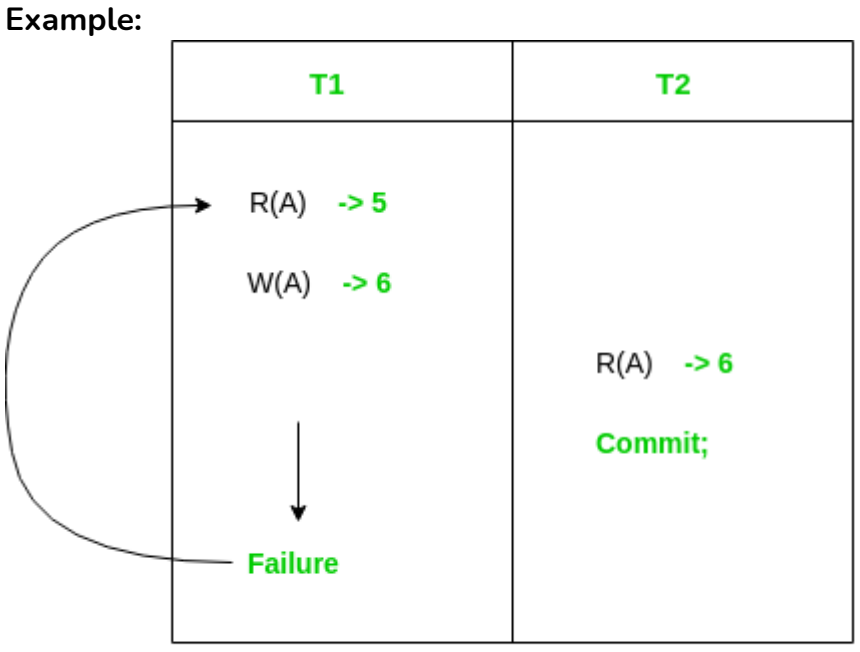


Figure - Dirty Read Problem

3. **Cascadeless Schedule:** When no **read** or **write-write** occurs before execution of transaction then corresponding schedule is called cascadeless schedule.

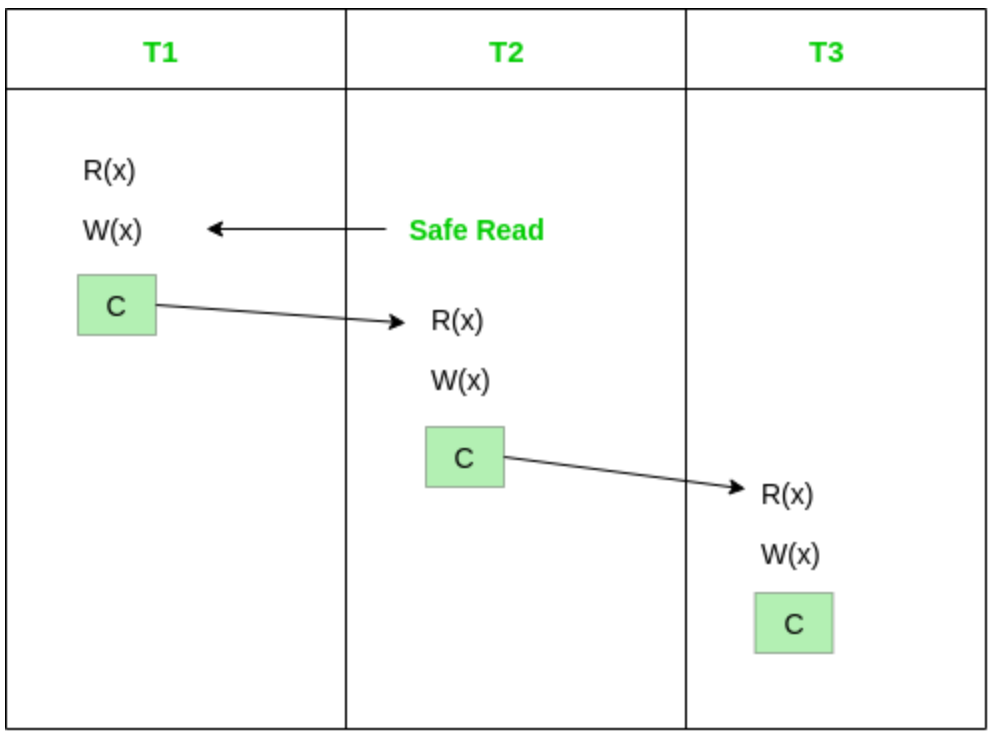
   Example:

```
S3: R1(x), R2(z), R3(x), R1(z), R2(y), R3(y), W1(x), C1,
```

```
        W2(z), W3(y), W2(y), C3, C2;
```

In this schedule **W3(y)** and **W2(y)** overwrite conflicts and there is no read, therefore given schedule is cascadeless schedule.

Special Case: A committed transaction desired to abort. As given below all the transactions are reading committed data hence it's cascade less schedule.



Cascading Abort: Cascading Abort can also be rollback. If transaction T1 abort as T2 read data that written by T1 which is not committed. Hence it's cascading rollback.

Example:

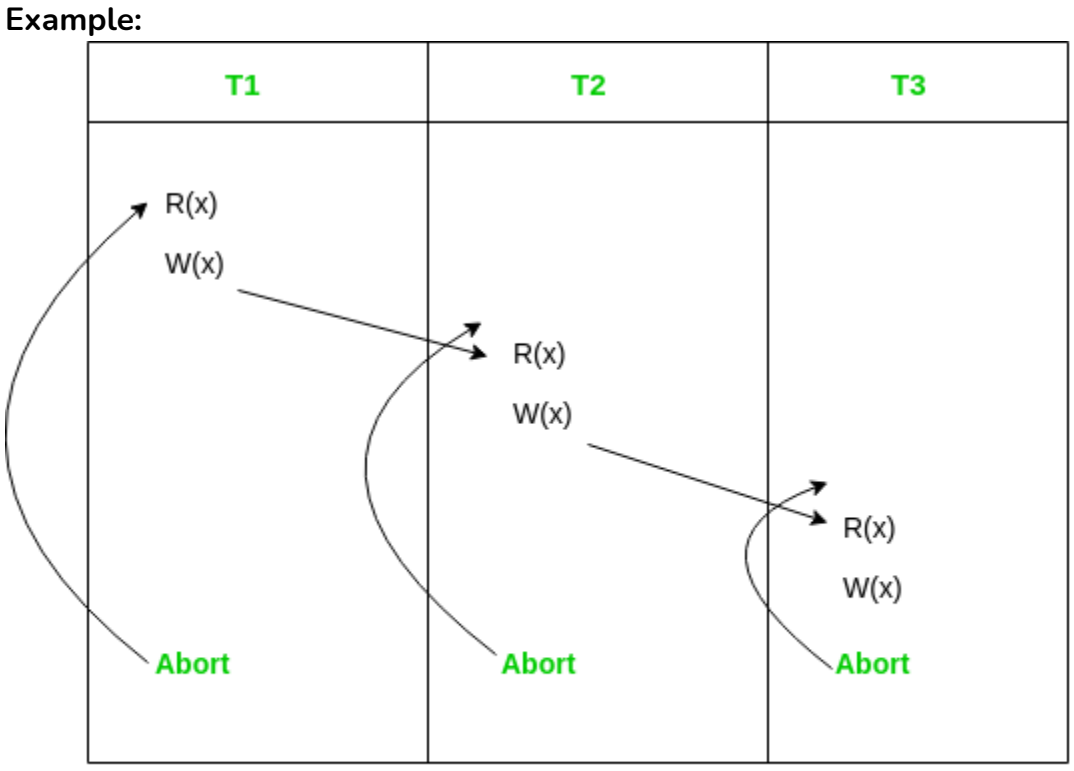


Figure - Cascading Abort

5. **Strict Schedule:** If schedule contains no **read** or **write** before commit then it is known as strict schedule. Strict schedule is strict in nature.

   Example:

```
6. S4: R1(x), R2(x), R1(z), R3(x), R3(y),
```

```
        W1(x), C1, W3(y), C3, R2(y), W2(z), W2(y), C2;
```

In this schedule no read-write or write-write conflict arises before commit hence its strict schedule:

T1	T2	T3
R1(x)		
	R2(x)	
R1(z)		
		R3(x)
		R3(y)
W1(x)		
C1;		
		W3(y)
		C3;
	R2(y)	
	W2(z)	
	W2(y)	
	C2;	

Figure - Strict Schedule

Corelation between Strict, Cascadeless and Recoverable schedule:

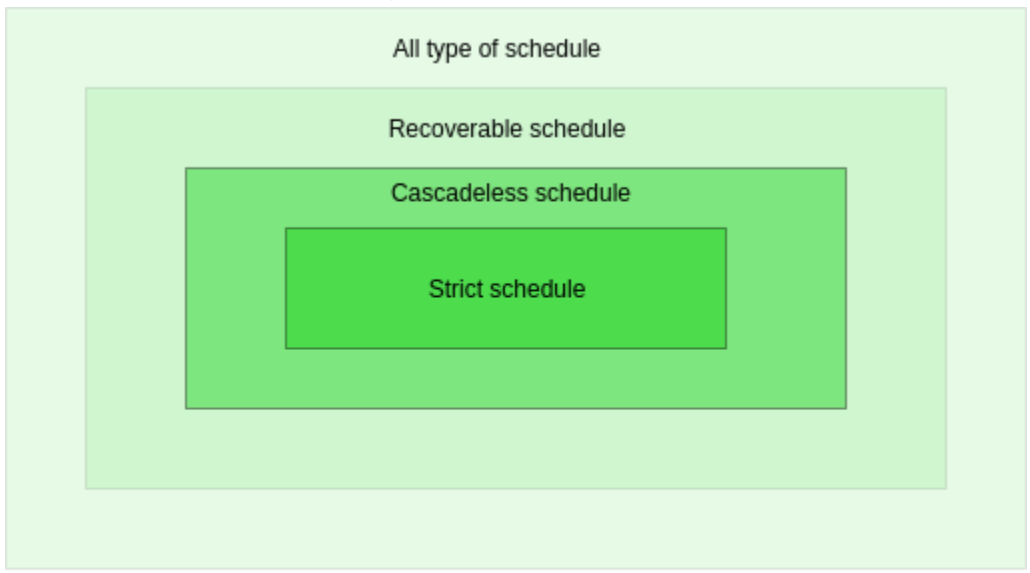


Figure - Venn diagram of these schedules

From above figure:

1. Strict schedules are all recoverable and cascadeless schedules
2. All cascadeless schedules are recoverable

Rollback and its types in DBMS

What is a Rollback?

A Rollback refers to an operation in which the database is brought back to some previous state, inorder to recover the loss of data or to overcome an unexpected situation. A rollback not only recovers the database to a stable position but also helps to maintain the integrity of the database.

A transaction may not execute completely due to hardware failure, system crash or software issues. In that case, we have to roll back the failed transaction. But some other transaction may also have used values produced by the failed transaction. So we have to roll back those transactions as well.

Types of Rollback?

1. **Cascading Recoverable Rollback:** The table below shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. But later on, T1 fails. So we have to rollback T1. Since T2 has read the value written by T1, it should also be rollbacked. As it has not committed, we can rollback T2 as well. So it is recoverable with cascading rollback. Therefore, if Tj is reading value updated by Ti and commit of Tj is delayed till commit of Ti, the schedule is called recoverable with cascading rollback.

T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
Failure Point				
Commit;				
		Commit;		

2. **Cascadeless Recoverable Rollback:** The table below shows a schedule with two transactions, T1 reads and writes A and commits and that value is read by T2. But if T1 fails before commit, no other transaction has read its value, so there is no need to rollback other transaction. So this is a Cascadeless recoverable schedule. So, if Tj reads value updated by Ti only after Ti is committed, the schedule will be cascadeless recoverable.

T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
Commit;				
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
		Commit;		

Dirty Read Problem and How to avoid it in DBMS

What is a Dirty Read Problem?

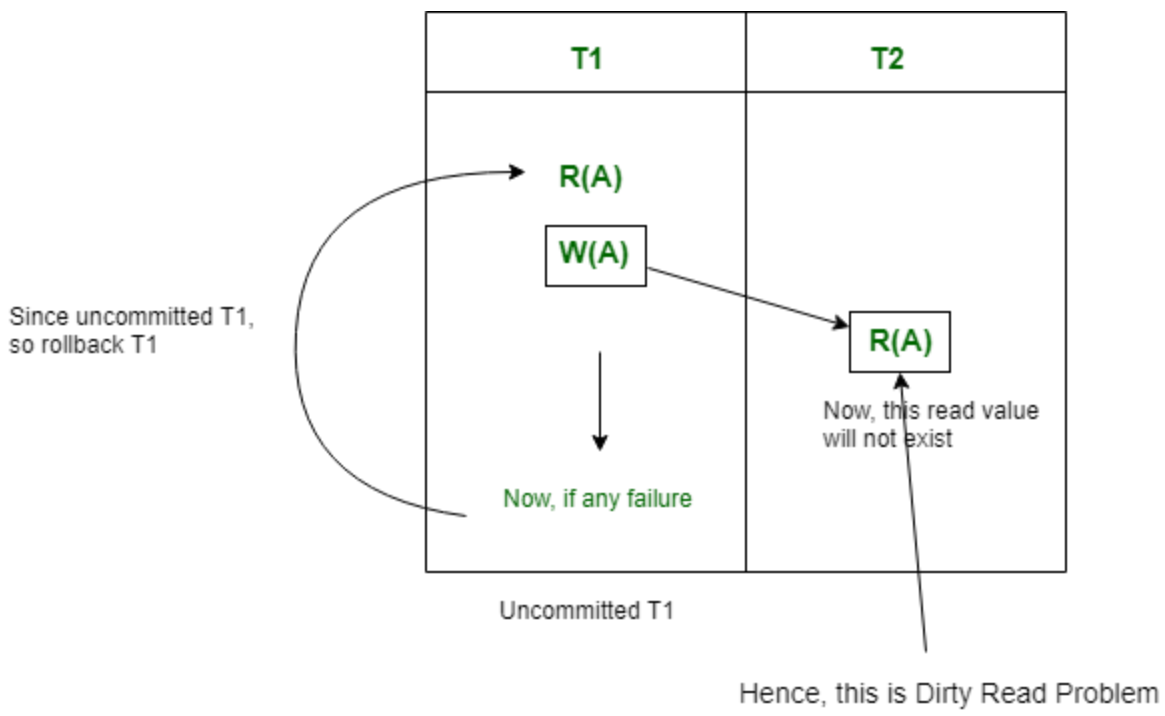
When a Transaction reads data from uncommitted write in another transaction, then it is known as *Dirty Read*. If that writing transaction failed, and that written data may be updated again. Therefore, this causes *Dirty Read Problem*.

In other words,

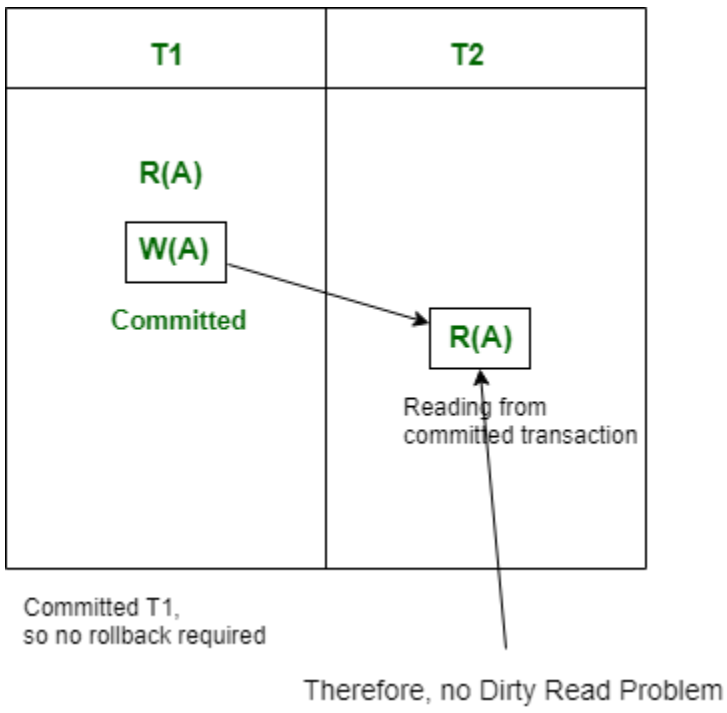
Reading the data written by an uncommitted transaction is called as dirty read.

It is called dirty read because there is always a chance that the uncommitted transaction might roll back later. Thus, the uncommitted transaction might make other transactions read a value that does not even exist. This leads to inconsistency of the database.

For example, let's say transaction 1 updates a row and leaves it uncommitted, meanwhile, Transaction 2 reads the updated row. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.



Note that there is no Dirty Read problem, is a transaction is reading from another committed transaction. So, no rollback required.

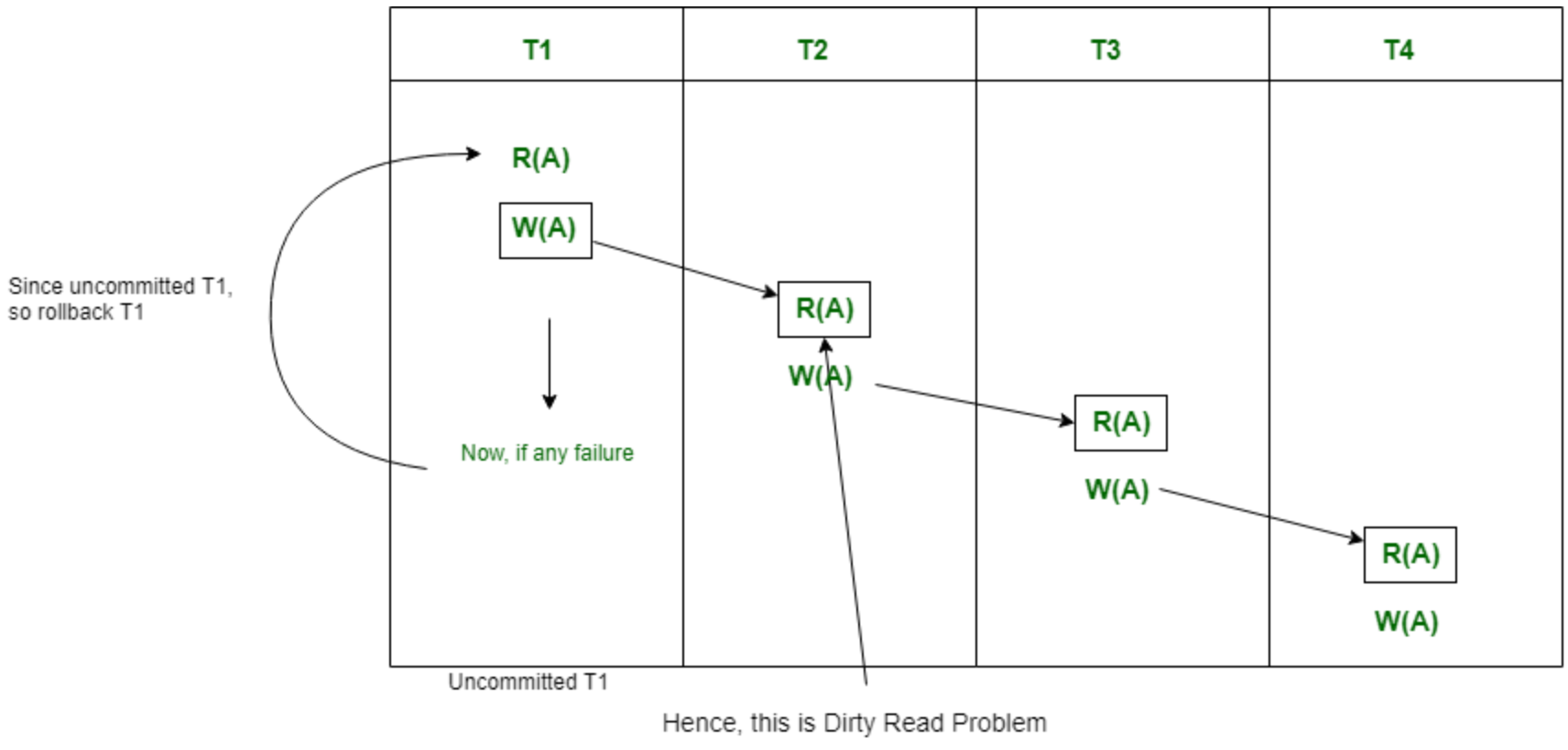


What does Dirty Read Problem result into?

Dirty Read Problem, in DBMS, results in the occurrence of Cascading Rollbacks.

Cascading Rollback: If in a schedule, failure of one transaction causes several other dependent transactions to rollback or abort, then such a schedule is called as a Cascading Rollback or Cascading Abort or Cascading Schedule. It simply leads to the wastage of CPU time.
These Cascading Rollbacks occur because of **Dirty Read problems**.

For example, transaction T1 writes uncommitted x that is read by Transaction T2. Transaction T2 writes uncommitted x that is read by Transaction T3.
Suppose at this point T1 fails.
T1 must be rolled back since T2 is dependent on T1, T2 must be rolled back, and since T3 is dependent on T2, T3 must be rolled back.



Because of T1 rollback, all T2, T3, and T4 should also be rollback (Cascading dirty read problem).

This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks is called **Cascading rollback**.

How to avoid Dirty Read Problem?

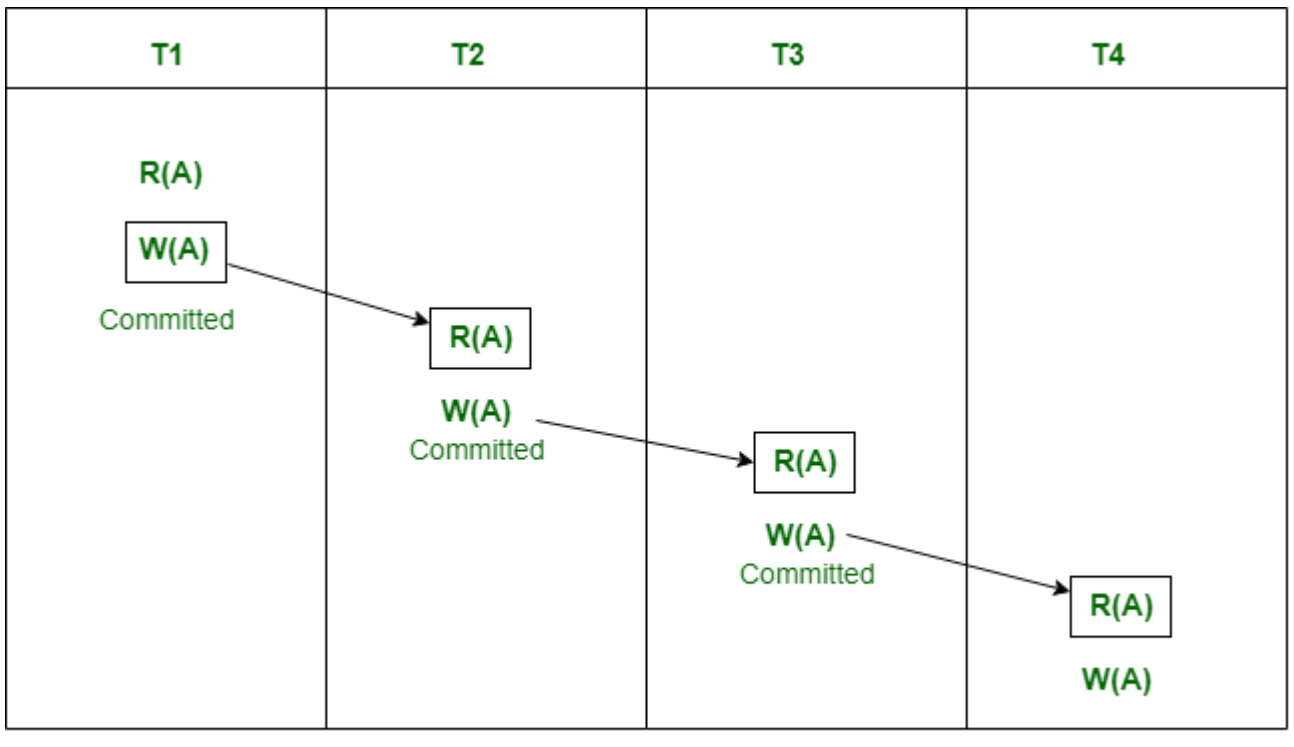
Dirty Read Problem can be avoided with the help of Cascadeless Schedule.

Cascadeless Schedule: This schedule avoids all possible *Dirty Read Problem*.

In Cascadeless Schedule, if a transaction is going to perform read operation on a value, it has to wait until the transaction who is performing write on that value commits. That means there must not be **Dirty Read**. Because Dirty Read Problem can cause *Cascading Rollback*, which is inefficient.

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

In other words, if some transaction Tj wants to read value updated or written by some other transaction Ti, then the commit of Tj must read it after the commit of Ti.



No Dirty Read problem

Note that Cascadeless schedule allows only committed read operations. However, it allows uncommitted to write operations.

Also note that Cascadeless Schedules are always recoverable, but all recoverable transactions may not be Cascadeless Schedule.

Two Phase Locking Protocol

Now, recalling where we last left off, there are two types of Locks available **Shared S(a)** and **Exclusive X(a)**. Implementing this lock system without any restrictions gives us the Simple Lock based protocol (or *Binary Locking*), but it has its own disadvantages, they **does not guarantee Serializability**. Schedules may follow the preceding rules but a non-serializable schedule may result.

To guarantee serializability, we must follow some additional protocol *concerning the positioning of locking and unlocking operations* in every transaction. This is where the concept of Two-Phase Locking(2-PL) comes in the picture, 2-PL ensures serializability. Now, let's dig deep!

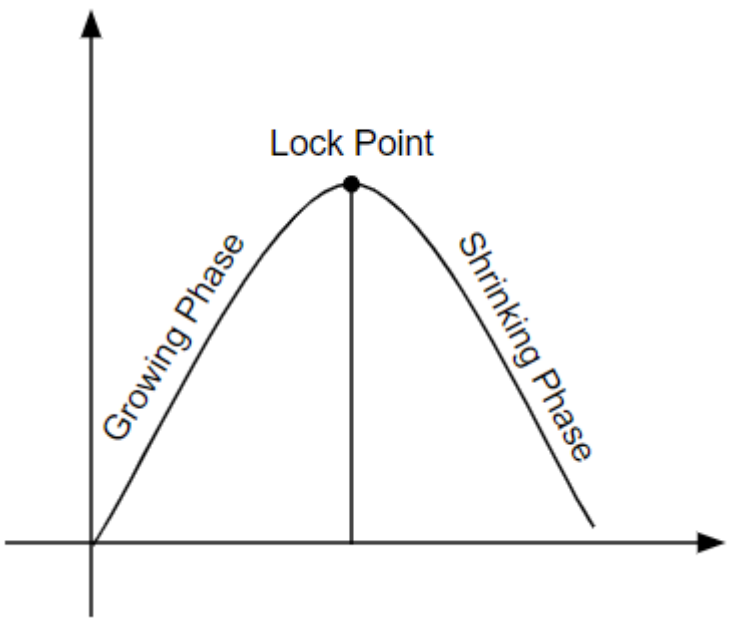
Let's get familiar with the two common locks which are used and some terminology followed in this protocol.

- **Shared Lock (S):** It is also known as Read-only lock. As the name suggests it can be shared between transactions because while holding this lock the transaction does not have the permission to update data on the data item. S-lock is requested using lock-S instruction.
- **Exclusive Lock (X):** In this lock, the data items can be both read and written. This is Exclusive and cannot be held simultaneously on the same data item. X-lock is requested using lock-X instruction.

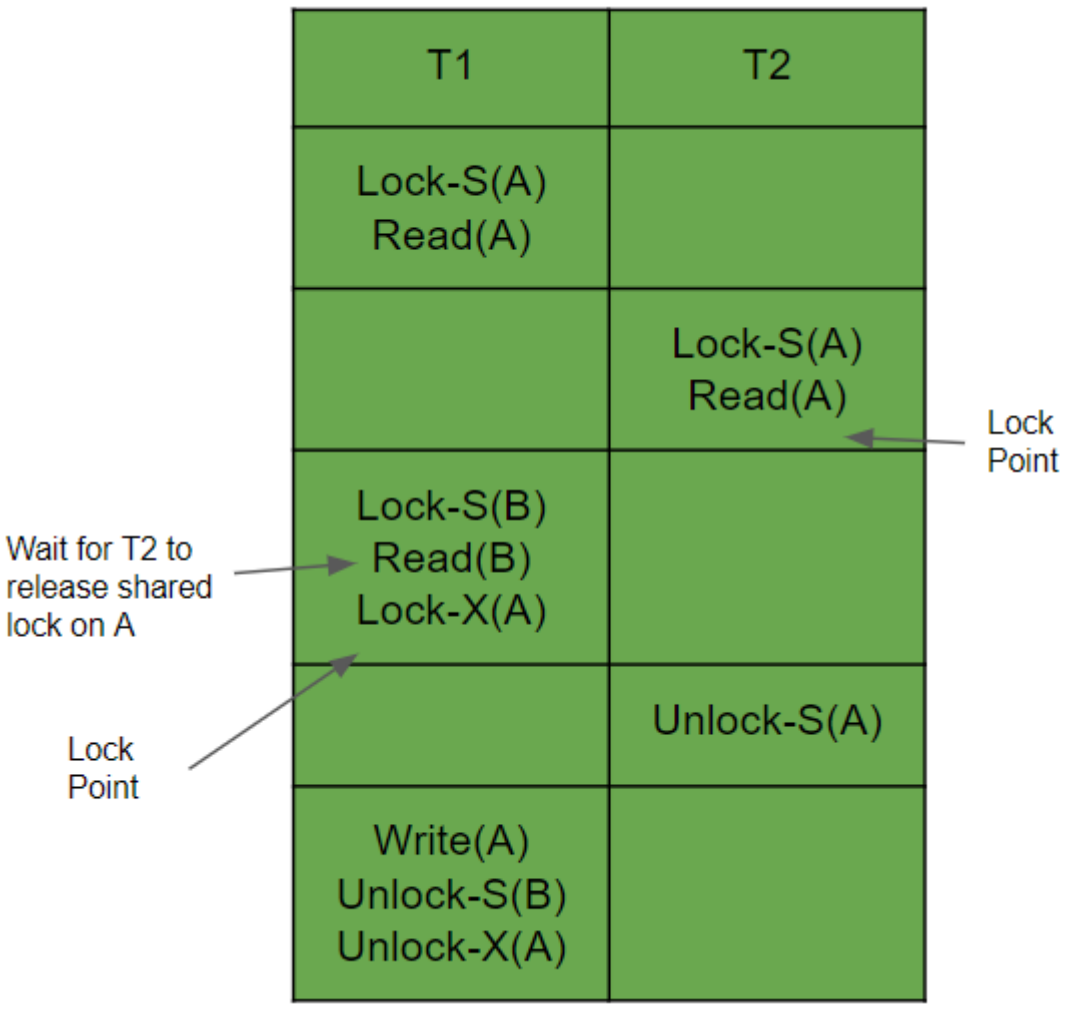
Two Phase Locking -

A transaction is said to follow Two Phase Locking protocol if Locking and Unlocking can be done in two phases.

1. **Growing Phase:** New locks on data items may be acquired but none can be released.
2. **Shrinking Phase:** Existing locks may be released but no new locks can be acquired.



LOCK POINT: The Point at which the growing phase ends, i.e., when the transaction takes the final lock it needs to carry on its work. Now look at the schedule, you'll surely understand.
Note - If lock conversion is allowed, then upgrading of lock(from S(a) to X(a)) is allowed in Growing Phase and downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.
Let's see a transaction implementing 2-PL.



This is just a skeleton transaction which shows how unlocking and locking work with 2-PL. Note for:
For Transaction T1, the growing phase is marked before the Lock point of T1, followed by its shrinking phase.
For Transaction T2, the growing phase is marked before the Lock point of T2, followed by its shrinking phase.

This transaction also takes care of Conflict Serializability as the transactions do not conflict with each other. The use of lock mechanism and lock point avoids the conflict. Since the transaction is conflict serializable, it automatically ensures view serializability. In the above example, T2T1 is conflict equivalent to this schedule. To check this let's draw the precedence graph.

Since there is no cycle involved, it ensures the serializability.

Problems with Basic 2PL: Deadlock in 2-PL - Consider this simple example, it will be easy to understand.Say we have two transactions T_1 and T_2.

Schedule: T1: Lock-X_1(A) T2: Lock-X_2(B) T1: Lock-X_1(B) T2: Lock-X_2(A)

In this T1: Lock-X_1(B) has to wait for T2 and T2: Lock-X_2(A) has to wait for T1.

Recoverability in 2-PL - Looking at the below example we can see that Dirty read occurs between Write(A) of T1 and Read(A) of T2, leading to irrecoverability.

T1	T2
Lock-X(A)	
Read(A)	
Write(A)	
Unlock-X(A)	
	Lock-S(A)
	Read(A)
	Unlock-S(A)
	Commit
Rollback	

Hence this must be avoided, and thus basic 2PL fails to resolve these issues.

Conservative, Strict and Rigorous 2PL

Now that we are familiar with what is Two Phase Locking (2-PL) and the basic rules which should be followed which ensures serializability. Moreover, we came across the problems with 2-PL, Cascading Aborts and Deadlocks. Now, we turn towards the enhancements made on 2-PL which tries to make the protocol nearly error-free. Briefly, we allow some modifications to 2-PL to improve it. There are three categories:

1. Conservative 2-PL
2. Strict 2-PL
3. Rigorous 2-PL

Now recall the rules followed in Basic 2-PL, over that we make some extra modifications. Let's now see what are the modifications and what drawbacks they solve.

Conservative 2-PL -

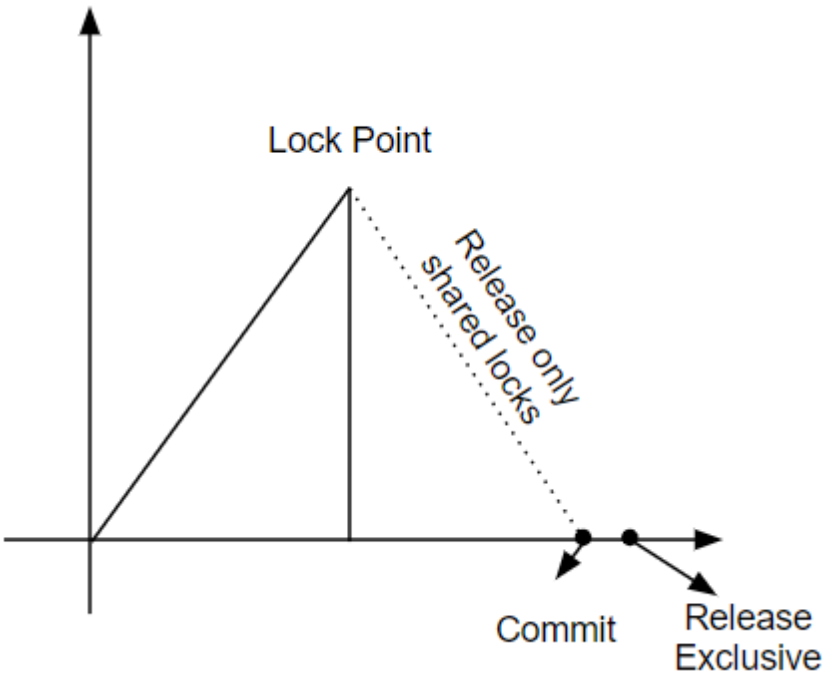
Also known as **Static 2-PL**, this protocol requires the transaction to lock all the items it access before the Transaction begins execution by predeclaring its read-set and write-set. If any of the predeclared items needed cannot be locked, the transaction does not lock any of the items, instead, it waits until all the items are available for locking.
Conservative 2-PL is *Deadlock free* and but it does not ensure Strict schedule(More about this here!). However, it is difficult to use in practice because of the need to predeclare the read-set and the write-set which is not possible in many situations. In practice, the most popular variation of 2-PL is Strict 2-PL.

Strict 2-PL -

This requires that in addition to the lock being 2-Phase **all Exclusive(X) Locks** held by the transaction be released until *after* the Transaction Commits.
Following Strict 2-PL ensures that our schedule is:

- Recoverable
- Cascadeless

Hence it gives us freedom from Cascading Abort which was still there in Basic 2-PL and moreover guarantee Strict Schedules but still *Deadlocks are possible*!



Rigorous 2-PL -
This requires that in addition to the lock being 2-Phase **all Exclusive(X) and Shared(S) Locks** held by the transaction be released until *after* the Transaction Commits.
Following Rigorous 2-PL ensures that our schedule is:

- Recoverable
- Cascadeless

Hence it gives us freedom from Cascading Abort which was still there in Basic 2-PL and moreover guarantee Strict Schedules but still *Deadlocks are possible*!

Note the difference between Strict 2-PL and Rigorous 2-PL is that Rigorous is more restrictive, it requires both Exclusive and Shared locks to be held until after the Transaction commits and this is what makes the implementation of Rigorous 2-PL easier.

The Venn Diagram below shows the classification of schedules which are rigorous and strict. The universe represents the schedules which can be serialized as 2-PL. Now as the diagram suggests, and it can also be logically concluded, if a schedule is Rigorous then it is Strict. We can also think in another way, say we put a restriction on a schedule which makes it strict, adding another to the list of restrictions make it Rigorous. Take a moment to again analyze the diagram and you'll definitely get it.

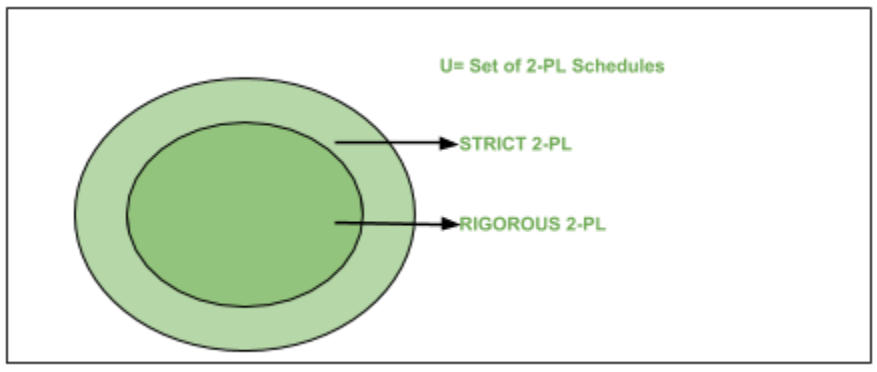


Image - Venn Diagram showing categories of languages under 2-PL

Comparisons:

Conservative 2PL	Strict and Rigorous 2PL
Deadlock Free	Deadlock Possible
Recoverability and Cascadeless-ness not guaranteed	Recoverability and Cascadeless-ness guaranteed

Rigorous Vs Strict:

1. Strict allows more concurrency

2. Rigorous is simple to implement and mostly used protocol

3. Rigorous is suitable in a distributed environment.

Timestamp based Concurrency Control

Concurrency Control can be implemented in different ways. One way to implement it is by using Locks. Now, let us discuss Time Stamp Ordering Protocol.

As earlier introduced, **Timestamp** is a unique identifier created by the DBMS to identify a transaction. They are usually assigned in the order in which they are submitted to the system. Refer to the timestamp of a transaction *T* as ***TS(T)***. For the basics of Timestamp, you may refer here.

Timestamp Ordering Protocol -
The main idea for this protocol is to order the transactions based on their Timestamps. A schedule in which the transactions participate is then serializable and the only *equivalent serial schedule permitted* has the transactions in the order of their Timestamp Values. Stating simply, the schedule is equivalent to the particular *Serial Order* corresponding to the *order of the Transaction timestamps*. An algorithm must ensure that, for each item accessed by *Conflicting Operations* in the schedule, the order in which the item is accessed does not violate the ordering. To ensure this, use two Timestamp Values relating to each database item **X**.

- **W_TS(X)** is the largest timestamp of any transaction that executed **write(X)** successfully.
- **R_TS(X)** is the largest timestamp of any transaction that executed **read(X)** successfully.

Basic Timestamp Ordering -
Every transaction is issued a timestamp based on when it enters the system. Suppose, if an old transaction T_i has timestamp TS(T_i), a new transaction T_j is assigned timestamp TS(T_j) such that **TS(T_i) < TS(T_j)**. The protocol manages concurrent execution such that the timestamps determine the serializability order. The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. Whenever some Transaction *T* tries to issue a R_item(X) or a W_item(X), the Basic TO algorithm compares the timestamp of *T* with **R_TS(X) & W_TS(X)** to ensure that the Timestamp order is not violated. This describes the Basic TO protocol in the following two cases.

1. Whenever a Transaction *T* issues a **W_item(X)** operation, check the following conditions:
    - If ***R_TS(X) > TS(T)*** or if ***W_TS(X) > TS(T)***, then abort and rollback T and reject the operation. else,
    - Execute W_item(X) operation of T and set W_TS(X) to TS(T).
2. Whenever a Transaction *T* issues a **R_item(X)** operation, check the following conditions:
    - If ***W_TS(X) > TS(T)***, then abort and reject T and reject the operation, else
    - If W_TS(X) <= TS(T), then execute the R_item(X) operation of T and set R_TS(X) to the larger of TS(T) and current R_TS(X).

Whenever the Basic TO algorithm detects two conflicting operations that occur in an incorrect order, it rejects the latter of the two operations by aborting the Transaction that issued it. Schedules produced by Basic TO are guaranteed to be *conflict serializable*. Already discussed that using Timestamp can ensure that our schedule will be *deadlock free*.

One drawback of the Basic TO protocol is that **Cascading Rollback** is still possible. Suppose we have a Transaction T_1 and T_2 has used a value written by T_1. If T_1 is aborted and resubmitted to the system then, T must also be aborted and rolled back. So the problem of Cascading aborts still prevails.

Let's gist the Advantages and Disadvantages of Basic TO protocol:

- Timestamp Ordering protocol ensures serializability since the precedence graph will be of the form:

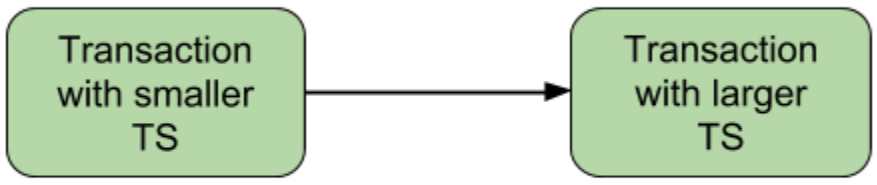


Image - Precedence Graph for TS ordering

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may *not be cascade free*, and may not even be recoverable.

Strict Timestamp Ordering -
A variation of Basic TO is called **Strict TO** ensures that the schedules are both Strict and Conflict Serializable. In this variation, a Transaction T that issues a R_item(X) or W_item(X) such that TS(T) > W_TS(X) has its read or write operation delayed until the Transaction **T'** that wrote the values of X has committed or aborted.