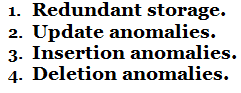
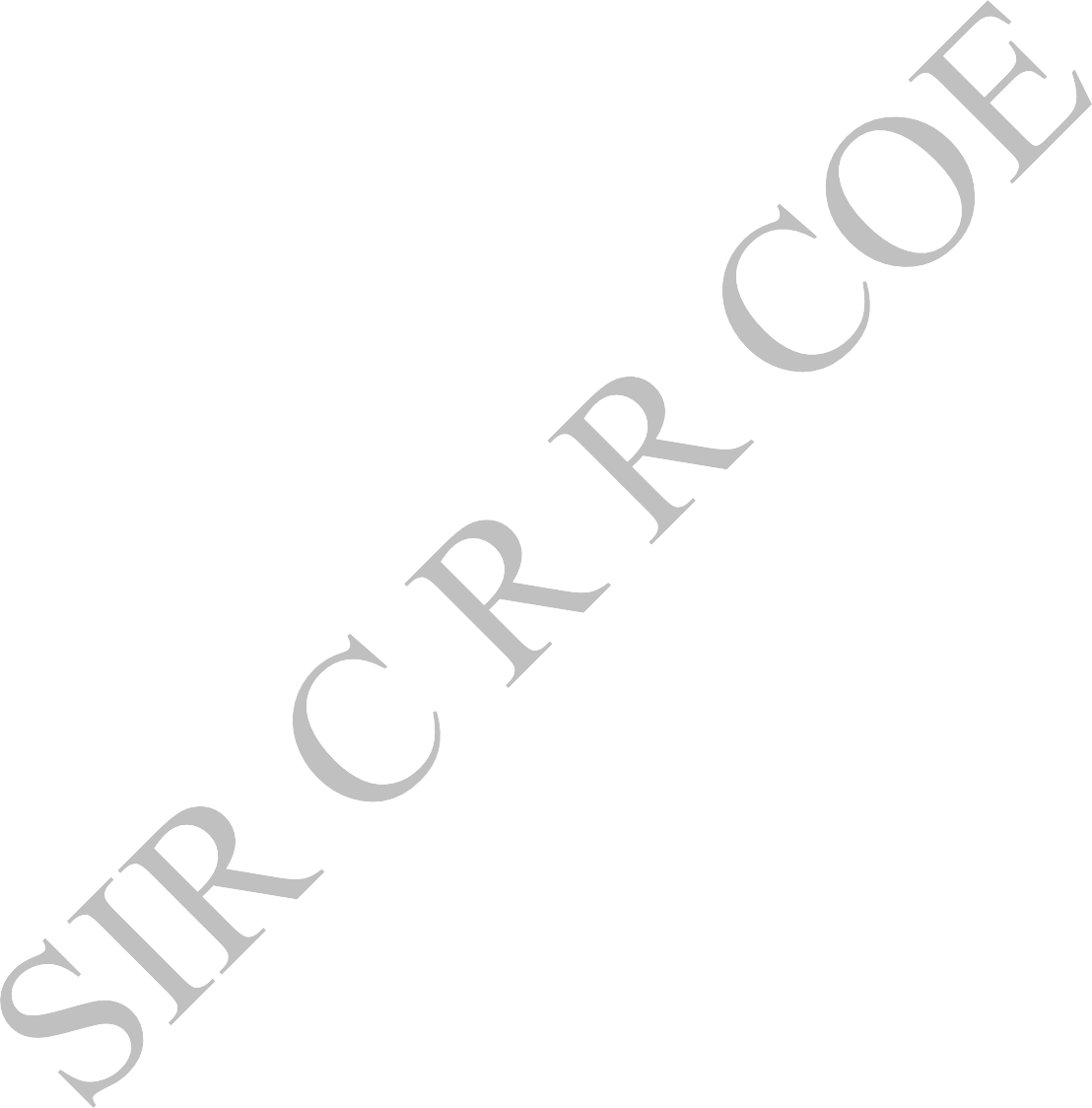
**UNIT IV**

**Syllabus**

**Schema Refinement (Normalization**): Purpose of Normalization or schema refinement, concept of functional dependency, normal forms based on functional dependency(1NF, 2NF and 3 NF), concept of surrogate key, Boyce-codd normal form(BCNF), Lossless join and dependency preserving decomposition, Fourth normal form(4NF), Fifth Normal Form (5NF).

A schema can be defined as a complete description of database. The specifications for database schema are provided during the database design stage and this schema does not change frequently.



**Decomposition** can eliminate the redundancy.

**1. Problems Caused by Redundancy :((\*\*\*\*\*\*\*\*\*\*\*)**

Redundancy is a data organization issue. It allows unnecessary duplication of data to be stored within the database. If modifications are performed to redundant data, then it is necessary to perform the same modification in multiple fields of database.

Storing the same information redundantly, that is, in more than one place within a database, can lead to several problems:

Schema diagram for Employee database is as follows,

**Example:**

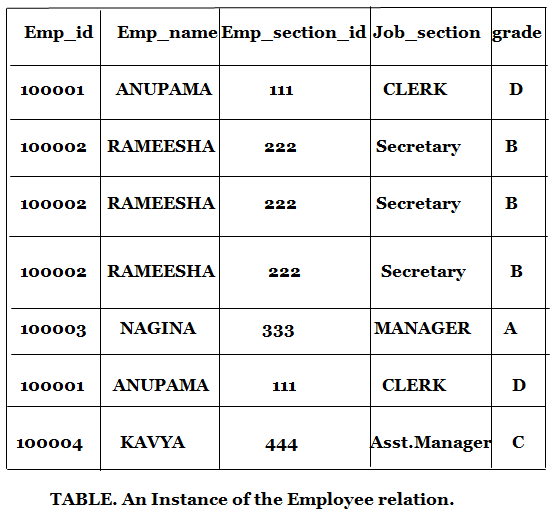
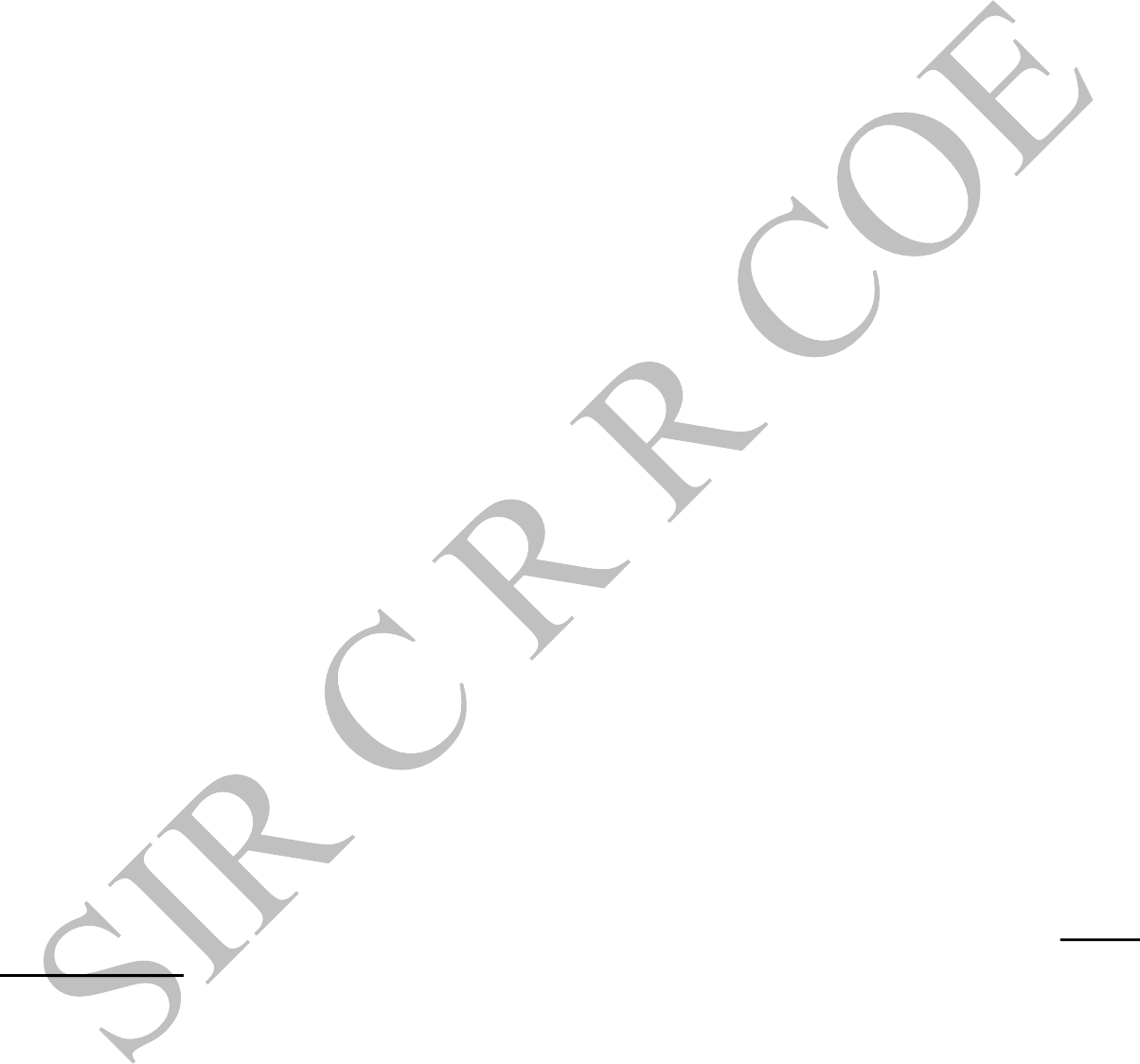
**Schema Refinement** is a technique of organizing the data in the database. It is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies.

**Schema refinement** is the process that re-defines (refining) the schema of a relation so as to solve the problems caused by redundantly storing the information.

**Redundancy** refers to repetition of same data or duplicate copies of same data stored in different locations.

The **Schema Refinement** refers to refine the schema by using some technique. The best technique of schema refinement is **decomposition**.

Consider the above database **table**. The three tuples with **Emp\_id100002** and two tuples with **Emp\_id100001** repeat the same name and same job section information. The repetition wastes space as well as causes data inconsistency i.e., this redundant data may lead to **loss of data integrity.**



For example, some update operation is being carried out, entering new record for an employee with id **100002.** This must be done multiple time i.e., it must be done for each file witch stores the employees details. This leads to redundant storage i.e., the same information is stored multiple times.

1. **Redundant storage:** Some information is stored repeatedly.
2. **Update anomalies:** If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.

If the update operation is performed, for example, the Emp\_section\_id 268 is updated to 520 and this correction is made only to the first record of the database, then this may lead to inconsistent data unless all the copies in the database are updated. This is referred to as update anomalies. The changes must be done to all the copies of data.

1. **Insertion anomalies:** It may not be possible to store some information unless some other information is stored as well.

For example, if a new employee record is being entered, who has not yet assigned an Emp\_id, now if we assume that the null values are not allowed, then it impossible to enter the new record unless the new employee has been assigned an Emp\_id. This is called insertion anomalies.

1. **Deletion anomalies:** It may not be possible to delete some information without losing some other information as well.

For example, if we want to delete the grade entries where grade is equal to ‘A’ then all the

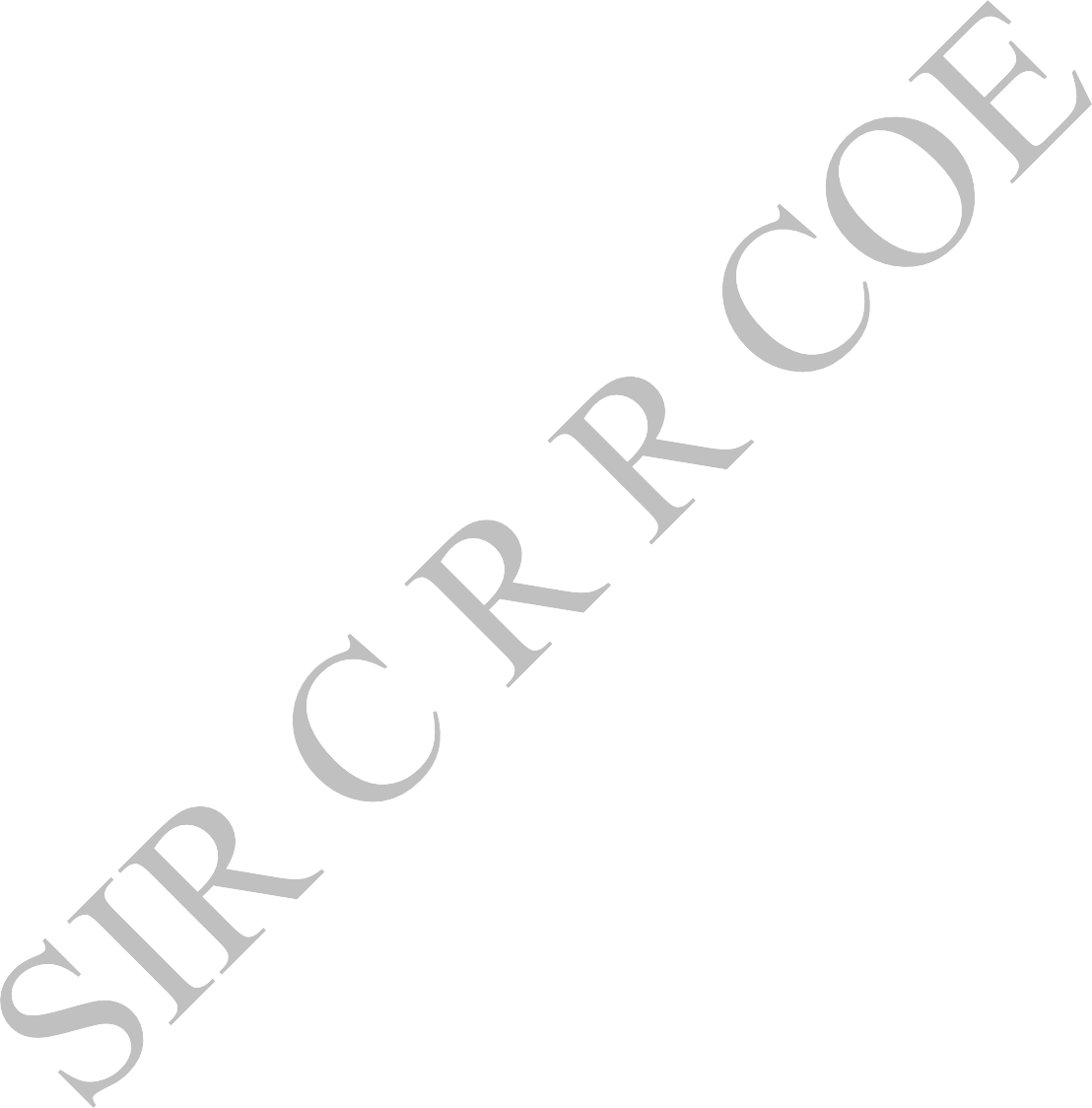
information of Emp\_section\_id 268 will be deleted/loss.

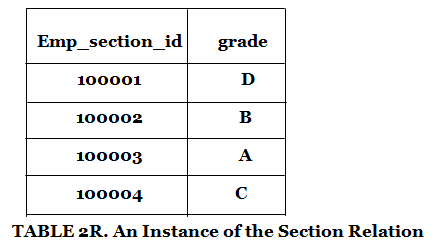
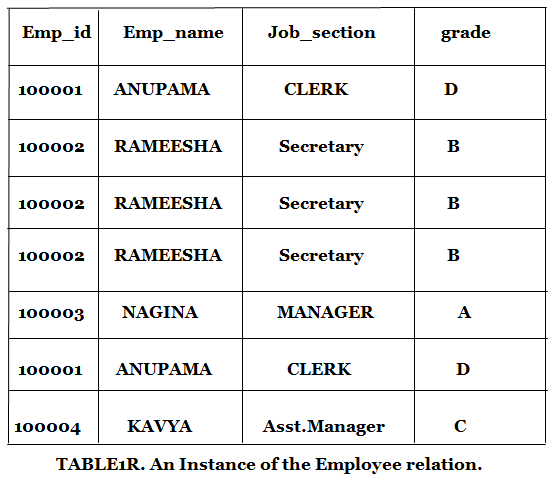
# 2. Use of Decompositions:

**Decomposion** is the solution to the problem caused by data redundancy. **Decomposition** means breaking up the large schema into smaller multiple Schemas. **Decomposition** helps to remove all the anomalies and helps to maintain data integrity.

We can restrict redundancy in **Employee** database by dividing it into two smaller relations/Schemas as in **table1R** and **Table2R.**

Now we can easily update **Emp\_section\_id** in the **Schema Section** without bothering about the updations in the other tuples. To insert a new tuple, we can directly insert the new record in the Schema section (With the help of **Emp\_section-id**) even if the new employee has not yet been assigned the **Emp\_id**. To delete the entry with the **grade** equal to ‘A’, we can do it directly on the **Section schema** which does not lead to loss of other information. Thus, **decomposioneliminates the Problems caused by different anomalies**





Functional Dependency

The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table.

1. X → Y

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

# For example:

Assume we have an employee table with attributes: Emp\_Id, Emp\_Name, Emp\_Address.

Here Emp\_Id attribute can uniquely identify the Emp\_Name attribute of employee table because if we know the Emp\_Id, we can tell that employee name associated with it.

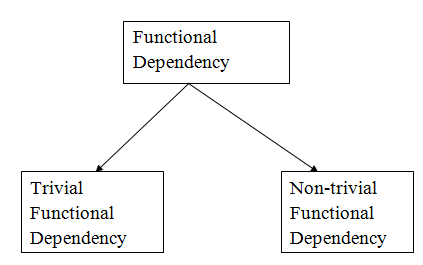
Functional dependency can be written as:

H

1. Emp\_Id → Emp\_Name

We can say that Emp\_Name is functionally dependent on Emp\_Id.

Types of Functional dependency



* 1. Trivial functional dependency
     + A → B has trivial functional dependency if B is a subset of A.
     + The following dependencies are also trivial like: A → A, B → B

# Example:

1. Consider a table with two columns Employee\_Id and Employee\_Name.
2. {Employee\_id, Employee\_Name} → Employee\_Id is a trivial functional dependency as
3. Employee\_Id is a subset of {Employee\_Id, Employee\_Name}.
4. Also, Employee\_Id → Employee\_Id and Employee\_Name → Employee\_Name are trivial dependencies too.
   1. Non-trivial functional dependency
      * A → B has a non-trivial functional dependency if B is not a subset of A.
      * When A intersection B is NULL, then A → B is called as complete non-trivial.

# Example:

1. ID → Name,
2. Name → DOB

Inference Rule (IR):

* + The Armstrong's axioms are the basic inference rule.
  + Armstrong's axioms are used to conclude functional dependencies on a relational database.
  + The inference rule is a type of assertion. It can apply to a set of FD(functional dependency) to derive other FD.
  + Using the inference rule, we can derive additional functional dependency from the initial set.

The Functional dependency has 6 types of inference rule:

1. Reflexive Rule (IR1)

In the reflexive rule, if Y is a subset of X, then X determines Y.

1. If X ⊇ Y then X → Y

# Example:

1. X = {a, b, c, d, e}

2. Y = {a, b, c}

1. Augmentation Rule (IR2)

The augmentation is also called as a partial dependency. In augmentation, if X determines Y, then XZ determines YZ for any Z.

1. If X → Y then XZ → YZ

# Example:

1. For R(ABCD), **if** A → B then AC → BC

3. Transitive Rule (IR3)

In the transitive rule, if X determines Y and Y determine Z, then X must also determine Z.

point

1. If X → Y and Y → Z then X → Z

4. Union Rule (IR4)

Union rule says, if X determines Y and X determines Z, then X must also determine Y and Z.

1. If X → Y and X → Z then X → YZ

# Proof:

* 1. X → Y (given)
  2. X → Z (given)
  3. X → XY (using IR2 on 1 by augmentation with X. Where XX = X)
  4. XY → YZ (using IR2 on 2 by augmentation with Y)
  5. X → YZ (using IR3 on 3 and 4)

5. Decomposition Rule (IR5)

Decomposition rule is also known as project rule. It is the reverse of union rule.

This Rule says, if X determines Y and Z, then X determines Y and X determines Z separately.

1. If X → YZ then X → Y and X → Z

# Proof:

* 1. X → YZ (given)
  2. YZ → Y (using IR1 Rule)
  3. X → Y (using IR3 on 1 and 2)

6. Pseudo transitive Rule (IR6)

In Pseudo transitive Rule, if X determines Y and YZ determines W, then XZ determines W.

1. If X → Y and YZ → W then XZ → W

# Proof:

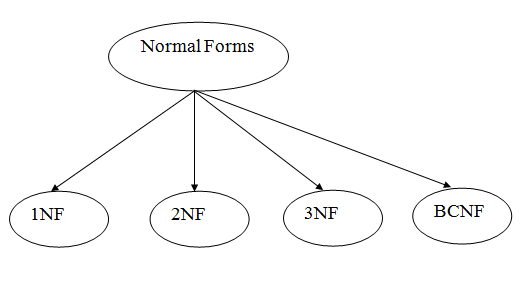
* 1. X → Y (given)
  2. WY → Z (given)
  3. WX → WY (using IR2 on 1 by augmenting with W)
  4. WX → Z (using IR3 on 3 and 2)

Normalization

* + - Normalization is the process of organizing the data in the database.
    - Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.
    - Normalization divides the larger table into the smaller table and links them using relationship.
    - The normal form is used to reduce redundancy from the database table.

Types of Normal Forms

There are the four types of normal forms:



|  |  |
| --- | --- |
| **Normal Description Form** | |
| [1NF](https://www.javatpoint.com/dbms-first-normal-form) | A relation is in 1NF if it contains an atomic value. |
| [2NF](https://www.javatpoint.com/dbms-second-normal-form) | A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key. |
| [3NF](https://www.javatpoint.com/dbms-third-normal-form) | A relation will be in 3NF if it is in 2NF and no transition dependency exists. |
| [4NF](https://www.javatpoint.com/dbms-forth-normal-form) | A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency. |
| [5NF](https://www.javatpoint.com/dbms-fifth-normal-form) | A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless. |

First Normal Form (1NF)

* + - A relation will be 1NF if it contains an atomic value.
    - It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
    - First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

**Example:** Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP\_PHONE.

# EMPLOYEE table:

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_PHONE** | **EMP\_STATE** |
| 14 | John | 7272826385,  9064738238 | UP |
| 20 | Harry | 8574783832 | Bihar |
| 12 | Sam | 7390372389,  8589830302 | Punjab |

The decomposition of the EMPLOYEE table into 1NF has been shown below:

**EMP\_ID EMP\_NAME EMP\_PHONE EMP\_STATE**

|  |  |  |  |
| --- | --- | --- | --- |
| 14 | John | 7272826385 | UP |
| 14 | John | 9064738238 | UP |
| 20 | Harry | 8574783832 | Bihar |
| 12 | Sam | 7390372389 | Punjab |
| 12 | Sam | 8589830302 | Punjab |

Second Normal Form (2NF)

* + - In the 2NF, relational must be in 1NF.
    - In the second normal form, all non-key attributes are fully functional dependent on the primary key

**Example:** Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.

# TEACHER table

|  |  |  |
| --- | --- | --- |
| **TEACHER\_ID** | **SUBJECT** | **TEACHER\_AGE** |
| 25 | Chemistry | 30 |
| 25 | Biology | 30 |
| 47 | English | 35 |
| 83 | Math | 38 |
| 83 | Computer | 38 |

In the given table, non-prime attribute TEACHER\_AGE is dependent on TEACHER\_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:

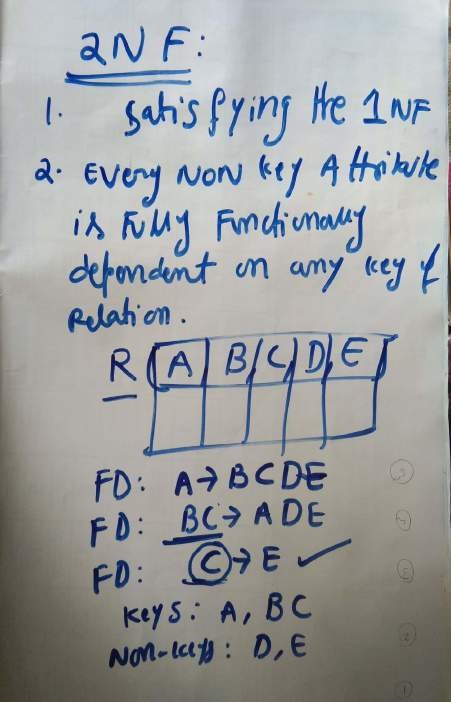
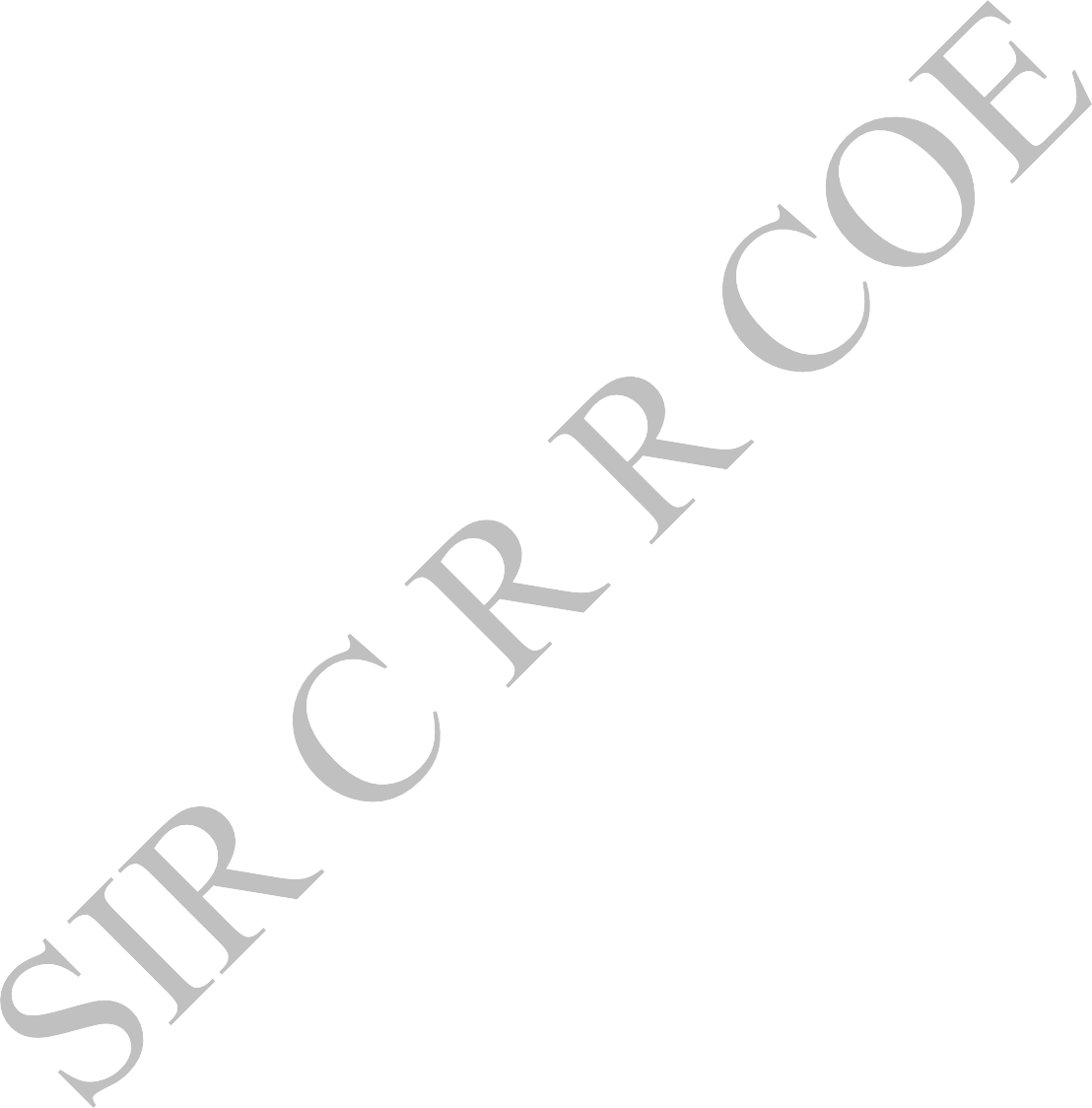
# TEACHER\_DETAIL table:

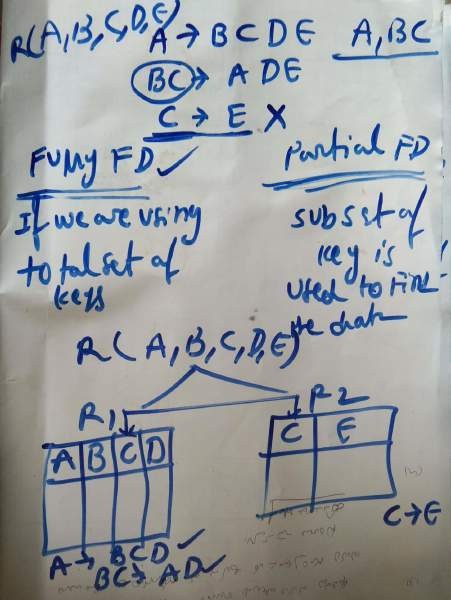
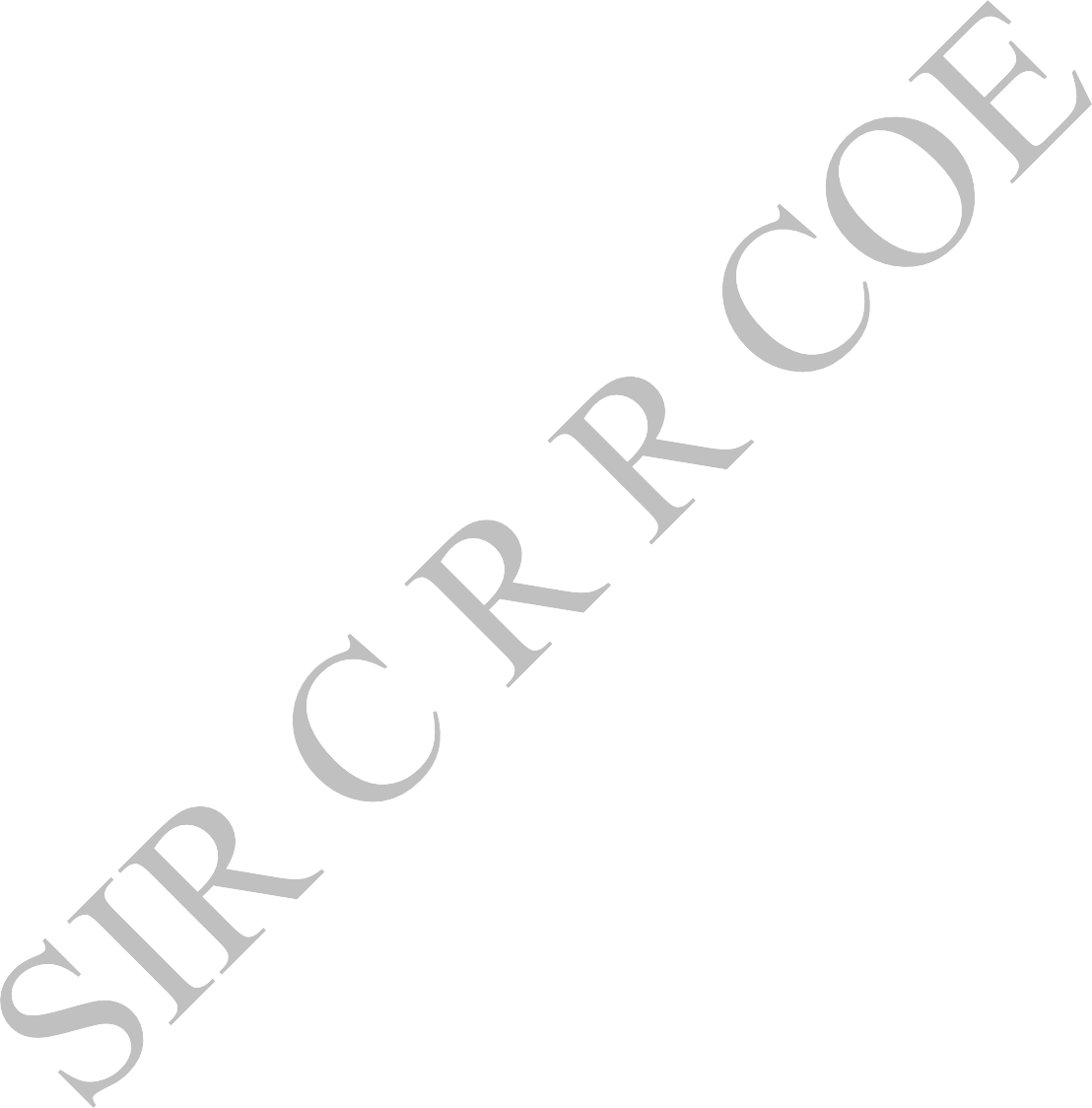
|  |  |
| --- | --- |
| **TEACHER\_ID TEACHER\_AGE** | |
| 25 | 30 |

|  |  |
| --- | --- |
| 47 | 35 |
| 83 | 38 |

**TEACHER\_SUBJECT table:**

|  |  |
| --- | --- |
| **TEACHER\_ID** | **SUBJECT** |
| 25 | Chemistry |
| 25 | Biology |
| 47 | English |
| 83 | Math |
| 83 | Computer |





Third Normal Form (3NF)

* + - A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
    - 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
    - If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency X → Y.

1. X is a super key.
2. Y is a prime attribute, i.e., each element of Y is part of some candidate key.

# Example:

**EMPLOYEE\_DETAIL table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_ZIP** | **EMP\_STATE** | **EMP\_CITY** |
| 222 | Harry | 201010 | UP | Noida |
| 333 | Stephan | 02228 | US | Boston |
| 444 | Lan | 60007 | US | Chicago |
| 555 | Katharine | 06389 | UK | Norwich |
| 666 | John | 462007 | MP | Bhopal |

# Super key in the table above:

1. {EMP\_ID}, {EMP\_ID, EMP\_NAME}, {EMP\_ID, EMP\_NAME, EMP\_ZIP} so on

**Candidate key:** {EMP\_ID}

**Non-prime attributes:** In the given table, all attributes except EMP\_ID are non- prime.

Here, EMP\_STATE & EMP\_CITY dependent on EMP\_ZIP and EMP\_ZIP dependent on EMP\_ID. The non-prime attributes (EMP\_STATE, EMP\_CITY) transitively dependent on super key(EMP\_ID). It violates the rule of third normal form.

That's why we need to move the EMP\_CITY and EMP\_STATE to the new

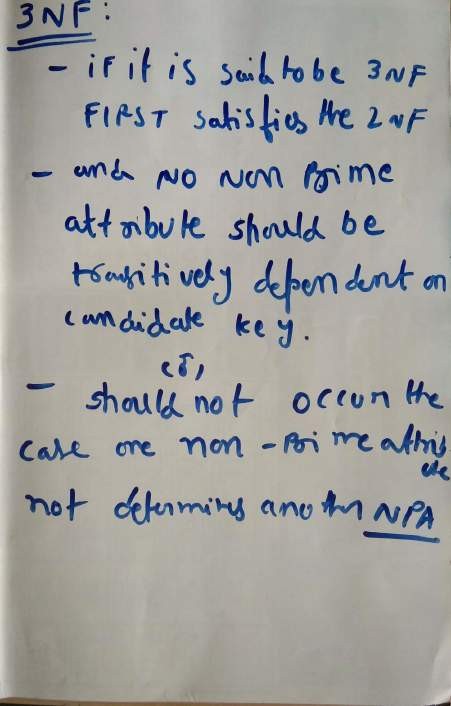
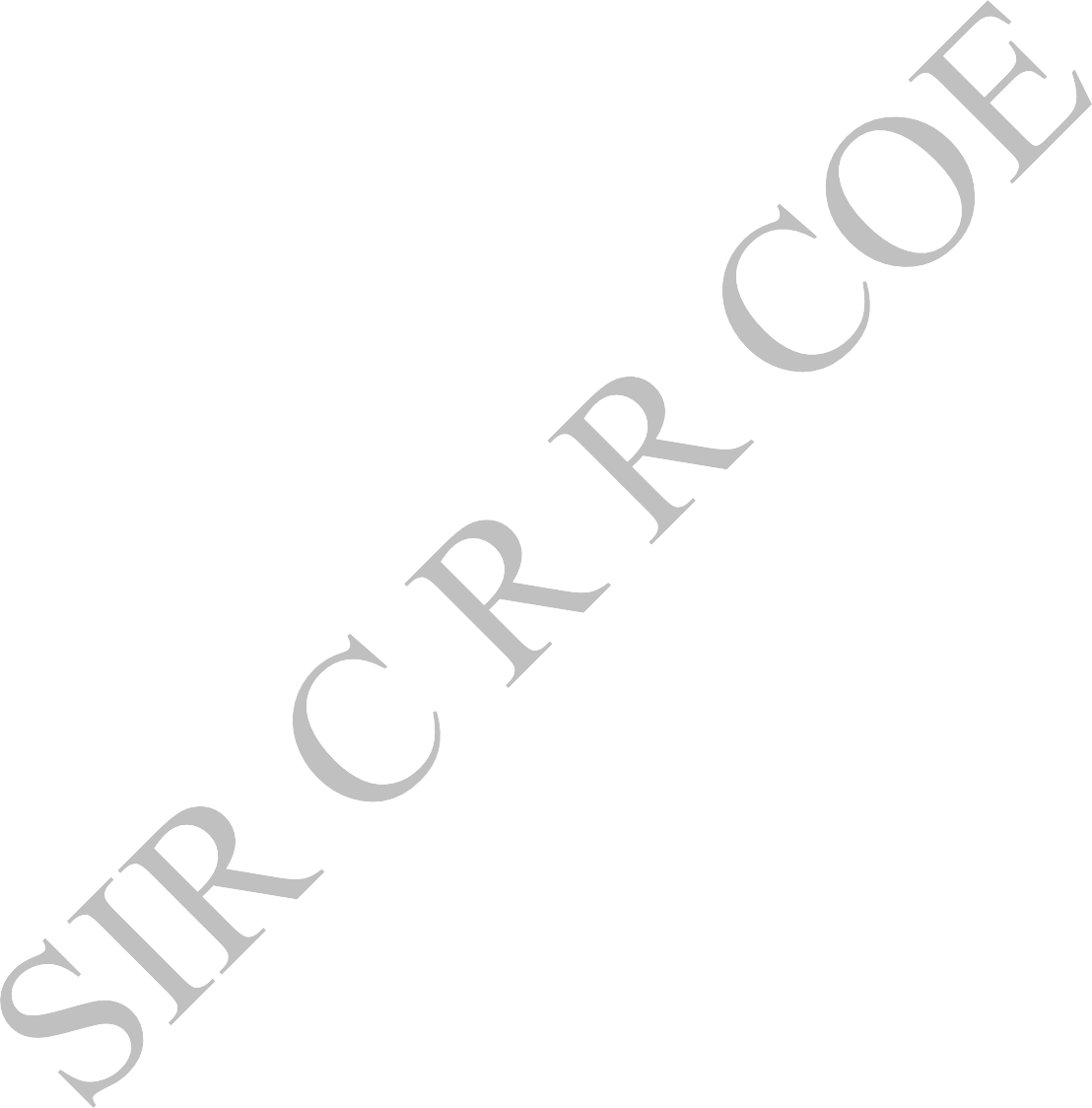
<EMPLOYEE\_ZIP> table, with EMP\_ZIP as a Primary key.

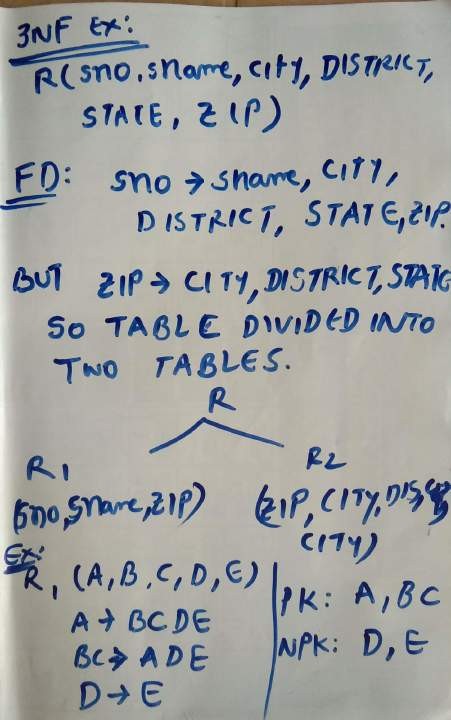
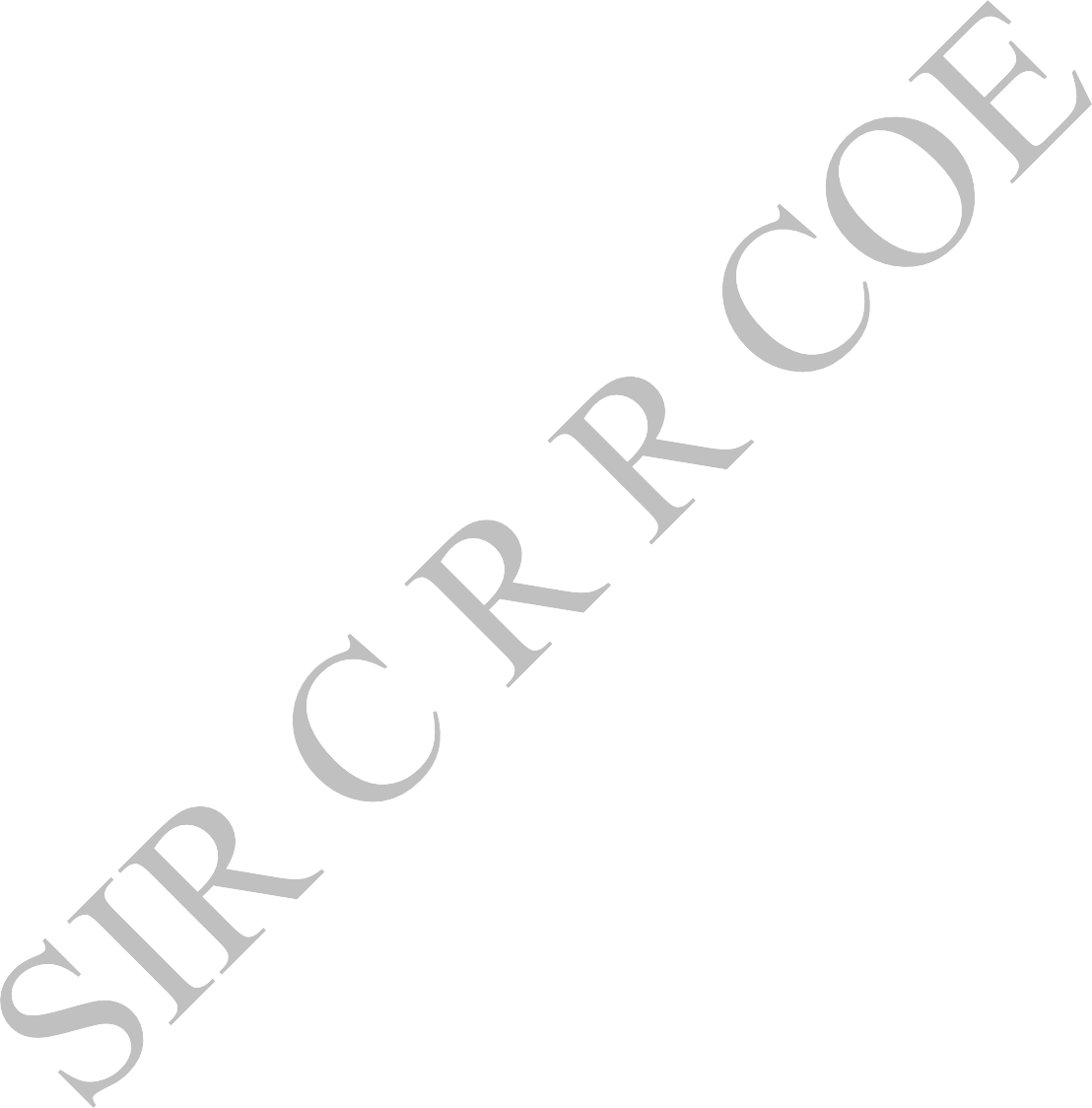
# EMPLOYEE table:

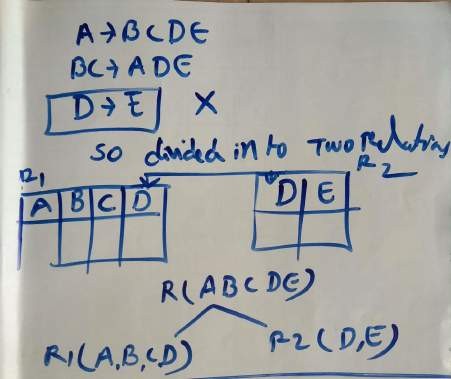
|  |  |  |
| --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_ZIP** |
| 222 | Harry | 201010 |
| 333 | Stephan | 02228 |
| 444 | Lan | 60007 |
| 555 | Katharine | 06389 |
| 666 | John | 462007 |

**EMPLOYEE\_ZIP table:**

|  |  |  |
| --- | --- | --- |
| **EMP\_ZIP** | **EMP\_STATE** | **EMP\_CITY** |
| 201010 | UP | Noida |
| 02228 | US | Boston |
| 60007 | US | Chicago |
| 06389 | UK | Norwich |
| 462007 | MP | Bhopal |







Boyce Codd normal form (BCNF)

* + - BCNF is the advance version of 3NF. It is stricter than 3NF.
    - A table is in BCNF if every functional dependency X → Y, X is the super key of the

table.

* + - For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

**Example:** Let's assume there is a company where employees work in more than one department.

# EMPLOYEE table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_COUNTRY** | **EMP\_DEPT** | **DEPT\_TYPE** | **EMP\_DEPT\_NO** |
| 264 | India | Designing | D394 | 283 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 264 | India | Testing | D394 | 300 |
| 364 | UK | Stores | D283 | 232 |
| 364 | UK | Developing | D283 | 549 |

**In the above table Functional dependencies are as follows:**

1. EMP\_ID → EMP\_COUNTRY
2. EMP\_DEPT → {DEPT\_TYPE, EMP\_DEPT\_NO}

# Candidate key: {EMP-ID, EMP-DEPT}

The table is not in BCNF because neither EMP\_DEPT nor EMP\_ID alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

# EMP\_COUNTRY table:

|  |  |
| --- | --- |
| **EMP\_ID** | **EMP\_COUNTRY** |
| 264 | India |
| 264 | India |

**EMP\_DEPT table:**

|  |  |  |
| --- | --- | --- |
| **EMP\_DEPT** | **DEPT\_TYPE** | **EMP\_DEPT\_NO** |
| Designing | D394 | 283 |
| Testing | D394 | 300 |
| Stores | D283 | 232 |
| Developing | D283 | 549 |

# EMP\_DEPT\_MAPPING table:

|  |  |
| --- | --- |
| **EMP\_ID** | **EMP\_DEPT** |
| D394 | 283 |
| D394 | 300 |

|  |  |
| --- | --- |
| D283 | 232 |
| D283 | 549 |

**Functional dependencies:**

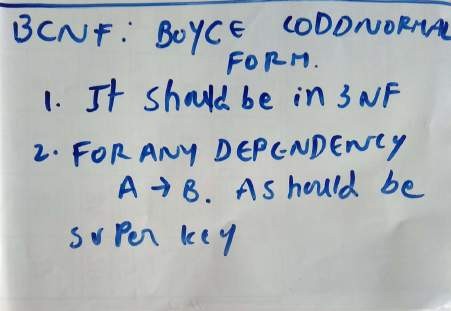
1. EMP\_ID → EMP\_COUNTRY
2. EMP\_DEPT → {DEPT\_TYPE, EMP\_DEPT\_NO}

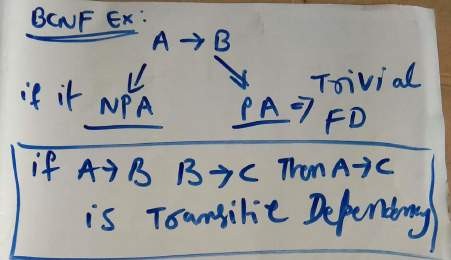
# Candidate keys:

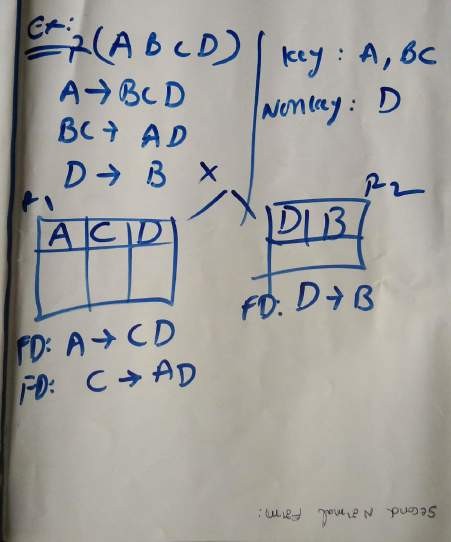
**For the first table:** EMP\_ID

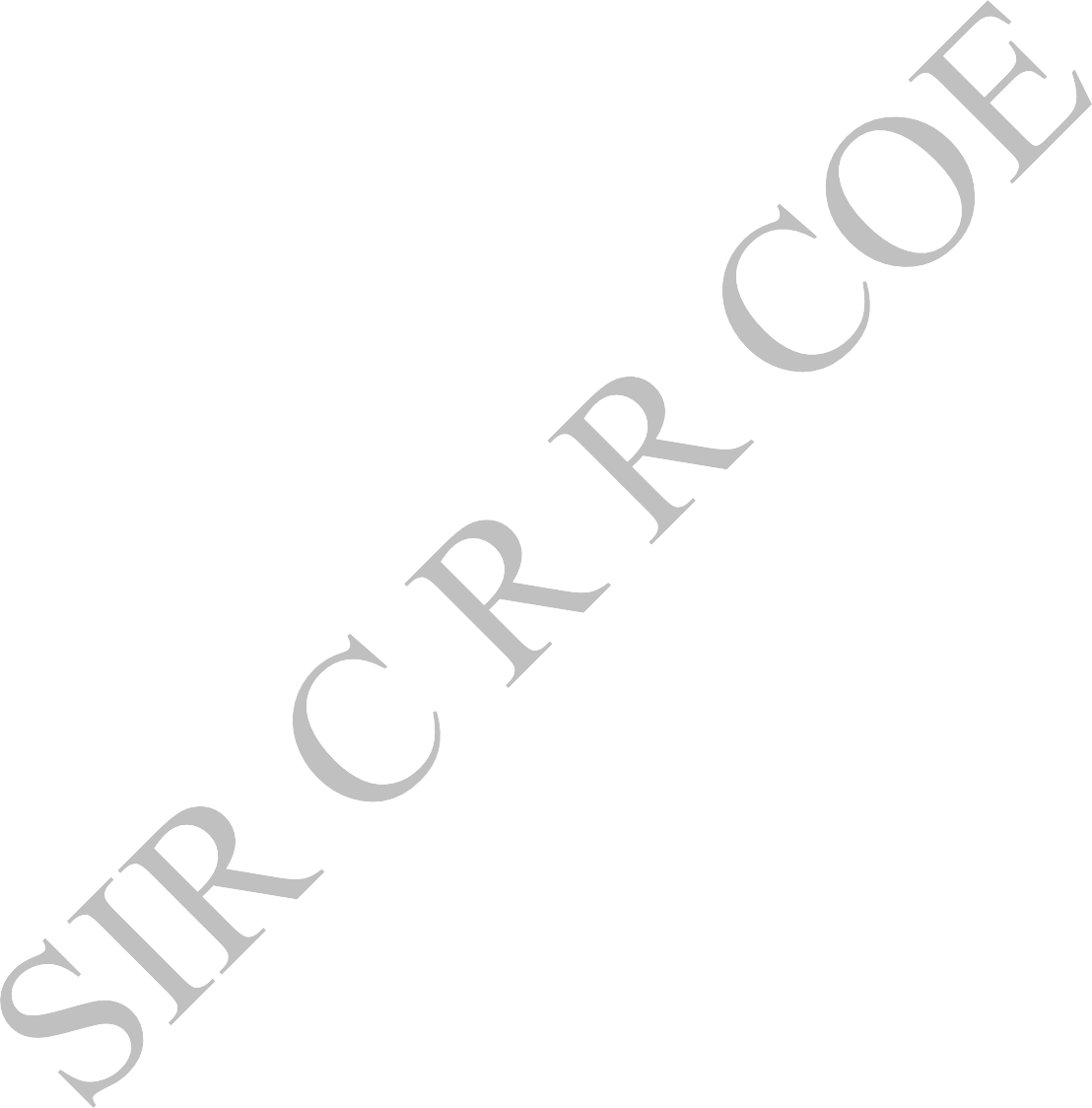
**For the second table:** EMP\_DEPT

**For the third table:** {EMP\_ID, EMP\_DEPT}







Fourth normal form (4NF)

* A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
* For a dependency A → B, if for a single value of A, multiple values of B exists, then

the relation will be a multi-valued dependency.

Example

# STUDENT

|  |  |  |
| --- | --- | --- |
| **STU\_ID** | **COURSE** | **HOBBY** |
| 21 | Computer | Dancing |
| 21 | Math | Singing |
| 34 | Chemistry | Dancing |
| 74 | Biology | Cricket |
| 59 | Physics | Hockey |

The given STUDENT table is in 3NF, but the COURSE and HOBBY are two independent entity. Hence, there is no relationship between COURSE and HOBBY.

In the STUDENT relation, a student with STU\_ID, **21** contains two courses, **Computer** and **Math** and two hobbies, **Dancing** and **Singing**. So there is a Multi- valued dependency on STU\_ID, which leads to unnecessary repetition of data.

So to make the above table into 4NF, we can decompose it into two tables:

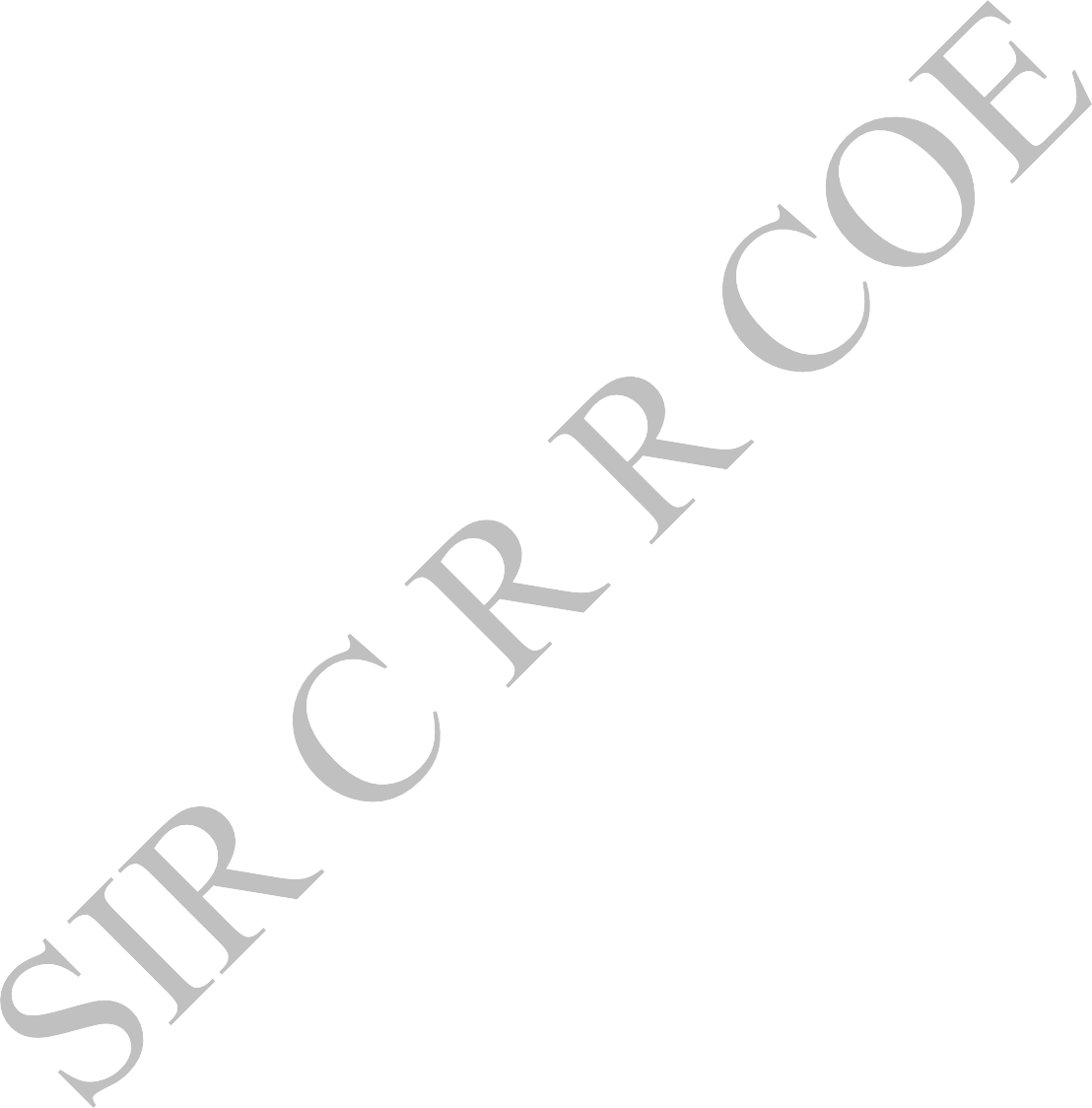
# STUDENT\_COURSE

|  |  |
| --- | --- |
| **STU\_ID** | **COURSE** |
| 21 | Computer |
| 21 | Math |
| 34 | Chemistry |
| 74 | Biology |
| 59 | Physics |

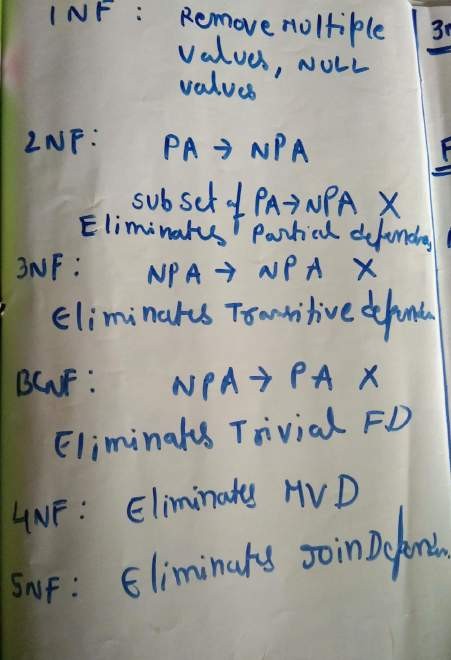
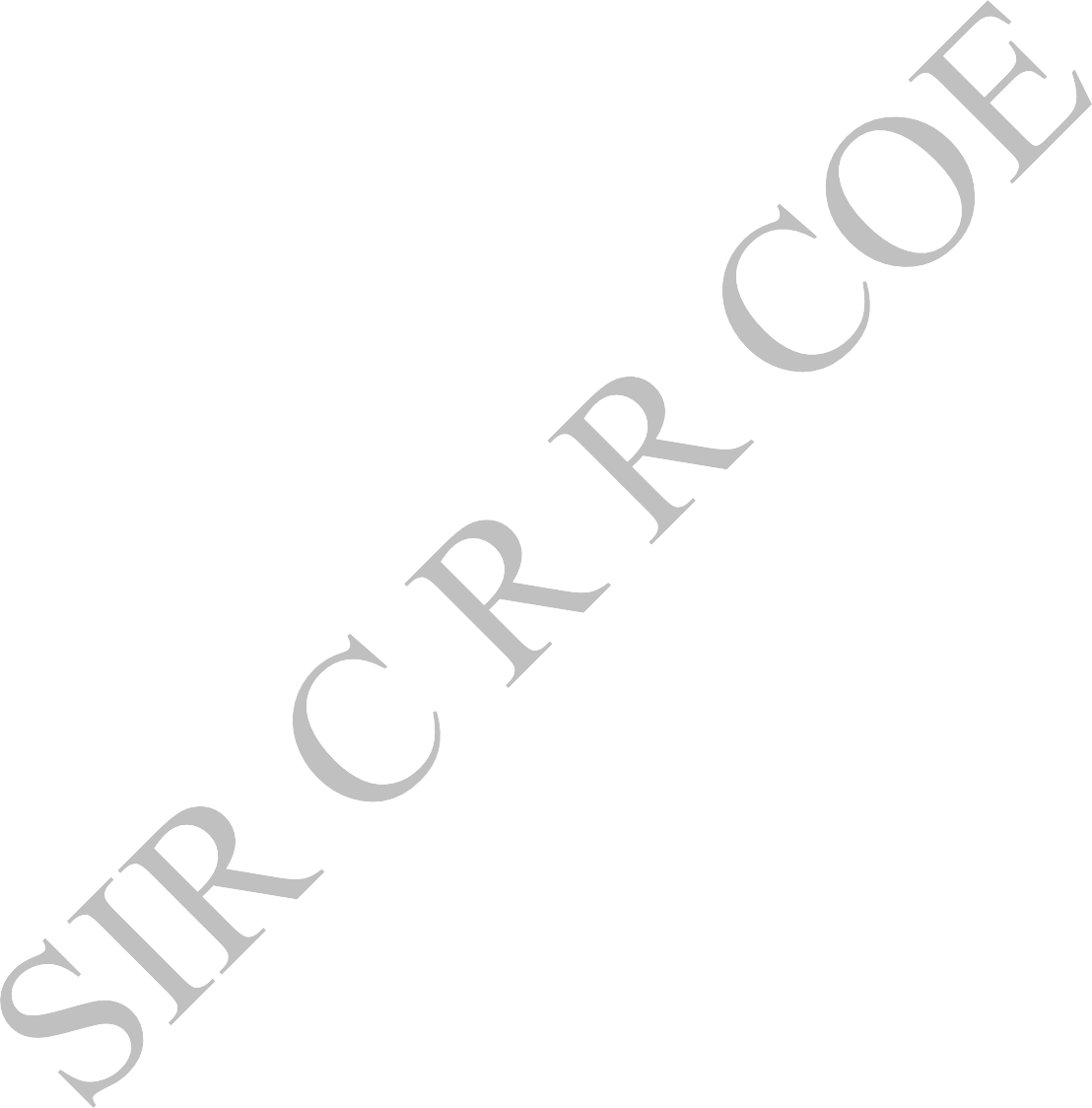
**STUDENT\_HOBBY**

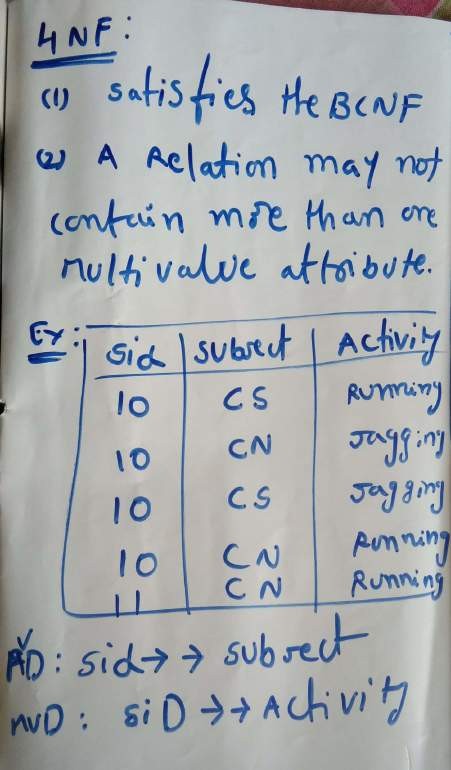
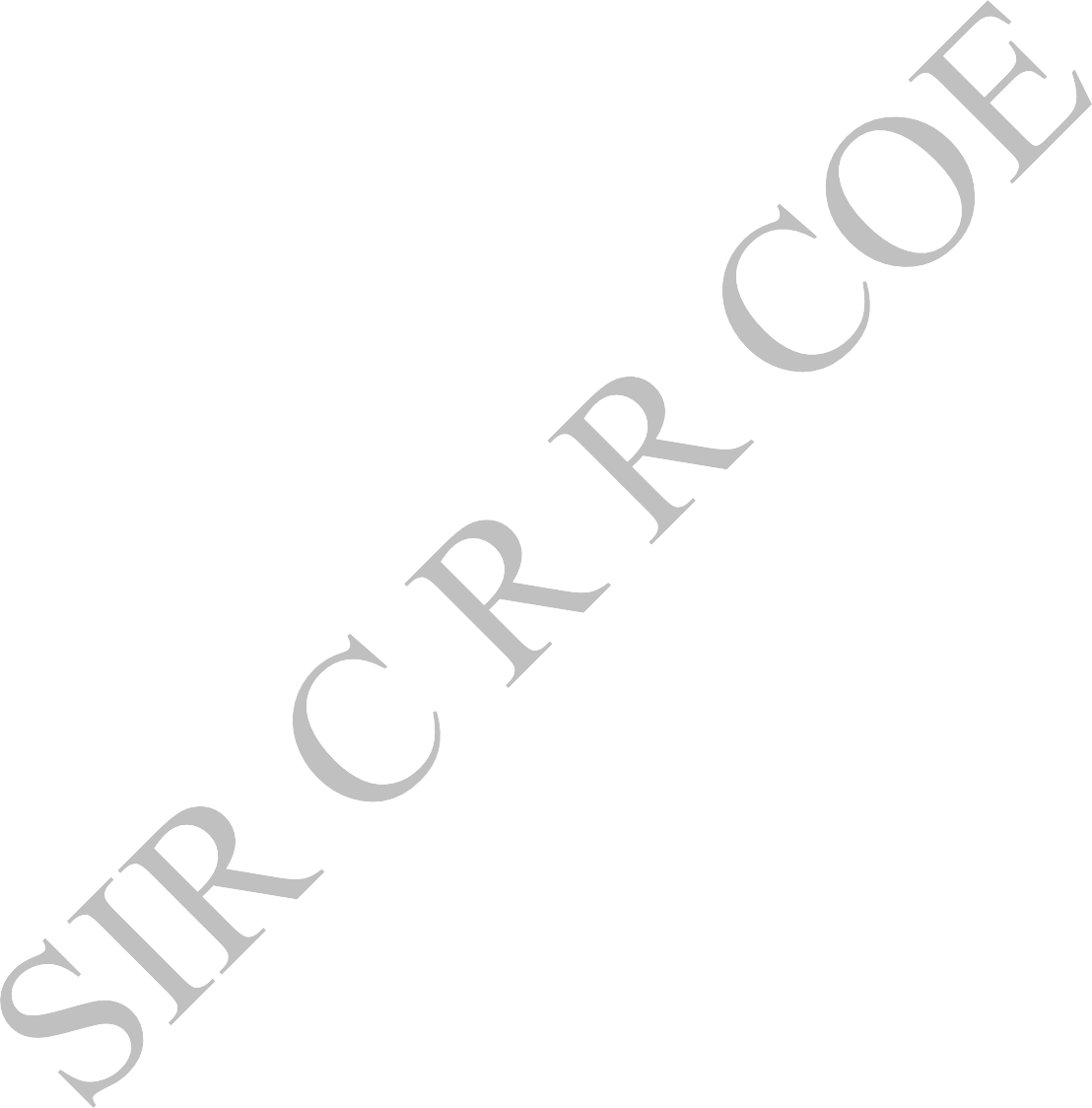
|  |  |
| --- | --- |
| **STU\_ID** | **HOBBY** |
| 21 | Dancing |
| 21 | Singing |
| 34 | Dancing |

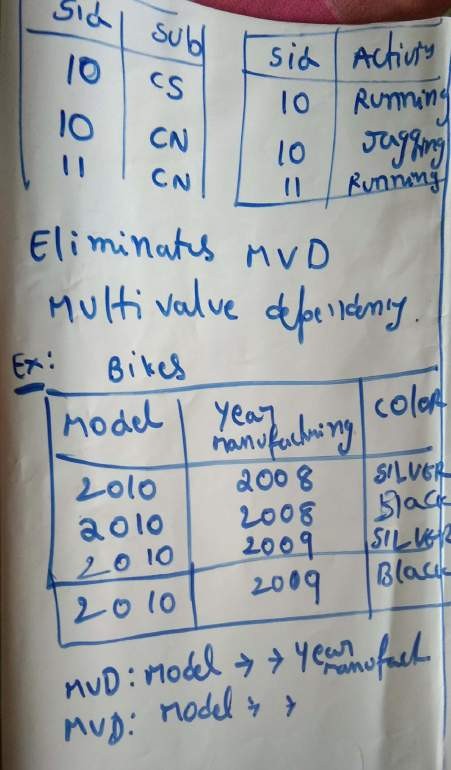
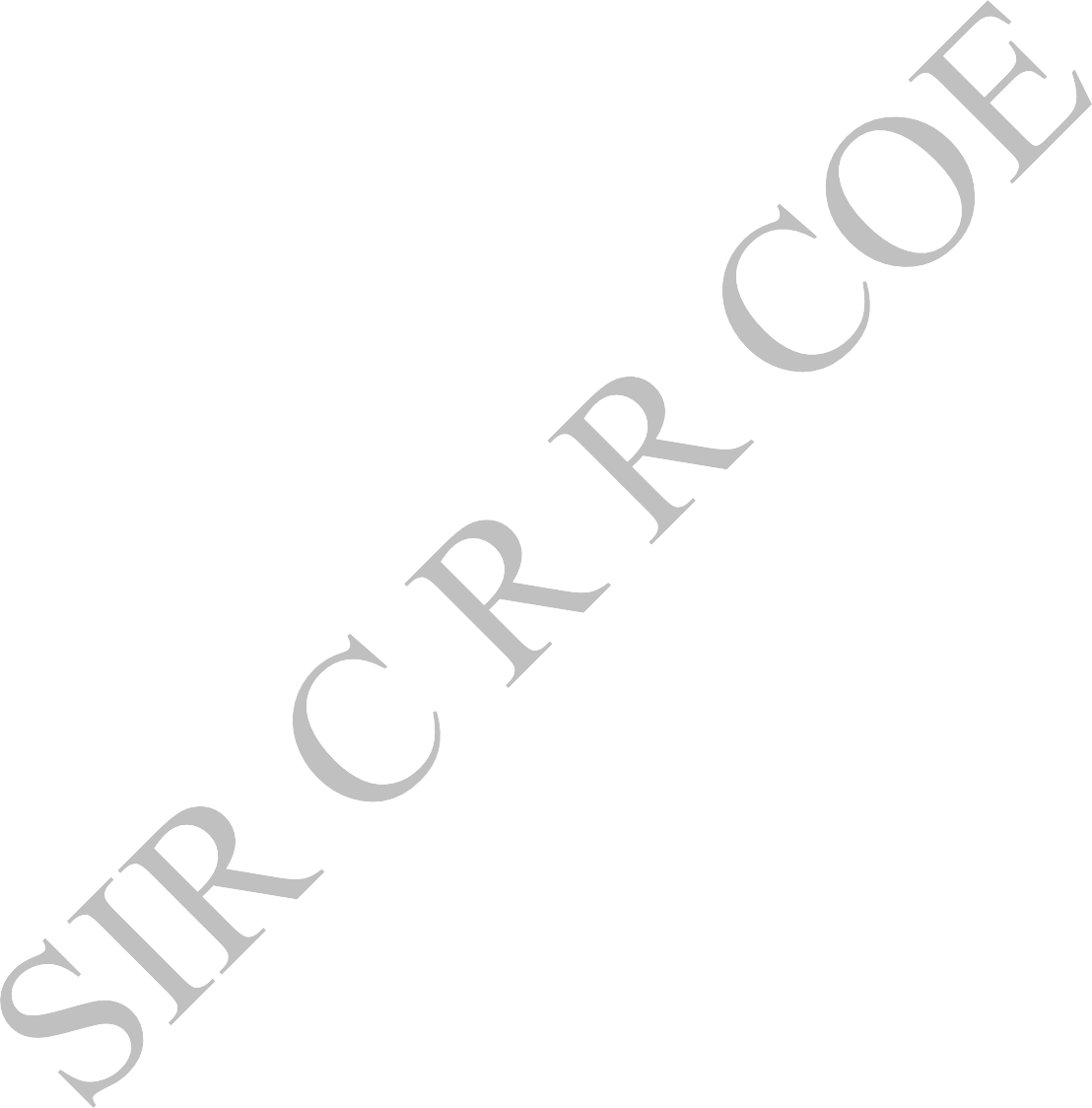
|  |  |
| --- | --- |
| 74 | Cricket |
| 59 | Hockey |











Fifth normal form (5NF)

* A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
* 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
* 5NF is also known as Project-join normal form (PJ/NF).

Example

|  |  |  |
| --- | --- | --- |
| **SUBJECT** | **LECTURER** | **SEMESTER** |
| Computer | Anshika | Semester 1 |
| Computer | John | Semester 1 |
| Math | John | Semester 1 |
| Math | Akash | Semester 2 |
| Chemistry | Praveen | Semester 1 |

In the above table, John takes both Computer and Math class for Semester 1 but he doesn't take Math class for Semester 2. In this case, combination of all these fields required to identify a valid data.

Suppose we add a new Semester as Semester 3 but do not know about the subject and who will be taking that subject so we leave Lecturer and Subject as NULL. But all three columns together acts as a primary key, so we can't leave other two columns blank.

So to make the above table into 5NF, we can decompose it into three relations P1, P2 & P3:

# P1

|  |  |
| --- | --- |
| **SEMESTER** | **SUBJECT** |
| Semester 1 | Computer |
| Semester 1 | Math |
| Semester 1 | Chemistry |
| Semester 2 | Math |

**P2**

|  |  |
| --- | --- |
| **SUBJECT** | **LECTURER** |
| Computer | Anshika |
| Computer | John |
| Math | John |
| Math | Akash |
| Chemistry | Praveen |

# P3

|  |  |
| --- | --- |
| **SEMSTER** | **LECTURER** |
| Semester 1 | Anshika |
| Semester 1 | John |
| Semester 1 | John |
| Semester 2 | Akash |
| Semester 1 | Praveen |

**UNIT V**

Transaction Concept: Transaction State, Implementation of Atomicity and Durability, Concurrent Executions, Serializability, Recoverability, Implementation of Isolation, Testing for Serializability, Failure Classification, Storage, Recovery and Atomicity, Recovery algorithm. Indexing Techniques: B+ Trees: Search, Insert, Delete algorithms, File Organization and Indexing, Cluster Indexes, Primary and Secondary Indexes , Index data Structures, Hash Based Indexing: Tree base Indexing ,Comparison of File Organizations, Indexes and Performance Tuning

Transaction

* The transaction is a set of logically related operation. It contains a group of tasks.
* A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing the contents of the database.

**Example:** Suppose an employee of bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low-level tasks:

# X's Account

1. Open\_Account(X)
2. Old\_Balance = X.balance
3. New\_Balance = Old\_Balance - 800
4. X.balance = New\_Balance
5. Close\_Account(X)

# Y's Account

1. Open\_Account(Y)
2. Old\_Balance = Y.balance
3. New\_Balance = Old\_Balance + 800
4. Y.balance = New\_Balance
5. Close\_Account(Y) Operations of Transaction:

Following are the main operations of transaction:

**Read(X):** Read operation is used to read the value of X from the database and stores it in a buffer in main memory.

**Write(X):** Write operation is used to write the value back to the database from the buffer.

Let's take an example to debit transaction from an account which consists of following operations:

1. 1. R(X);

2. 2. X = X - 500;

3. 3. W(X);

Let's assume the value of X before starting of the transaction is 4000.

* The first operation reads X's value from database and stores it in a buffer.
* The second operation will decrease the value of X by 500. So buffer will contain 3500.
* The third operation will write the buffer's value to the database. So X's final value will be 3500.

But it may be possible that because of the failure of hardware, software or power, etc. that transaction may fail before finished all the operations in the set.

**For example:** If in the above transaction, the debit transaction fails after executing operation 2 then X's value will remain 4000 in the database which is not acceptable by the bank.

To solve this problem, we have two important operations:

**Commit:** It is used to save the work done permanently.

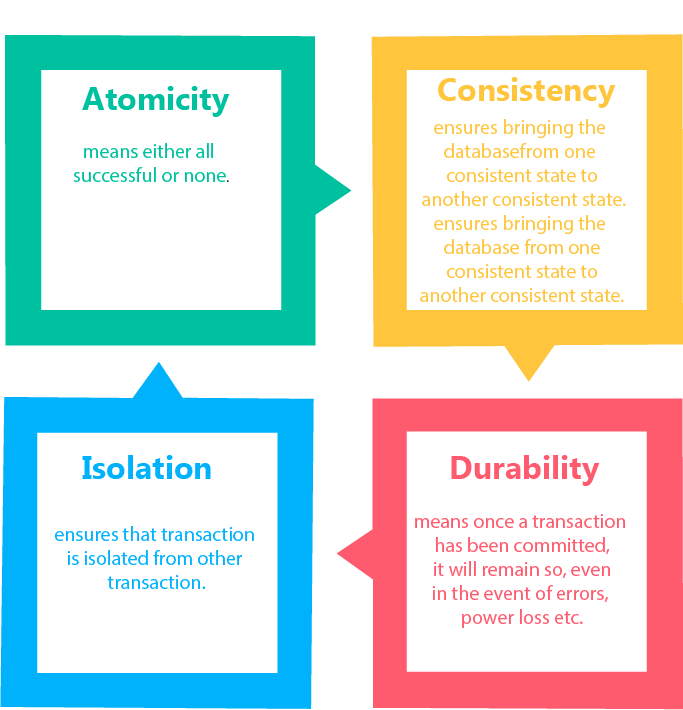
**Rollback:** It is used to undo the work done.

Transaction property

The transaction has the four properties. These are used to maintain consistency in a database, before and after the transaction.

Property of Transaction

* 1. Atomicity
  2. Consistency
  3. Isolation
  4. Durability



Atomicity

* It states that all operations of the transaction take place at once if not, the transaction is aborted.
* There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.

Atomicity involves the following two operations:

**Abort:** If a transaction aborts then all the changes made are not visible.

**Commit:** If a transaction commits then all the changes made are visible.

**Example:** Let's assume that following transaction T consisting of T1 and T2. A consists of Rs 600 and B consists of Rs 300. Transfer Rs 100 from account A to account B.

|  |  |  |  |
| --- | --- | --- | --- |
| **T1** |  | **T2** | |
| Read(A) A:=  Write(A) | A-100 | Read(B) Y:=  Write(B) | Y+100 |

After completion of the transaction, A consists of Rs 500 and B consists of Rs 400.

If the transaction T fails after the completion of transaction T1 but before completion of transaction T2, then the amount will be deducted from A but not added to B. This shows the inconsistent database state. In order to ensure correctness of database state, the transaction must be executed in entirety.

Consistency

* The integrity constraints are maintained so that the database is consistent before and after the transaction.
* The execution of a transaction will leave a database in either its prior stable state or a new stable state.
* The consistent property of database states that every transaction sees a consistent database instance.
* The transaction is used to transform the database from one consistent state to another consistent state.

**For example:** The total amount must be maintained before or after the transaction.

1. Total before T occurs = 600+300=900
2. Total after T occurs= 500+400=900

Therefore, the database is consistent. In the case when T1 is completed but T2 fails, then inconsistency will occur.

Isolation

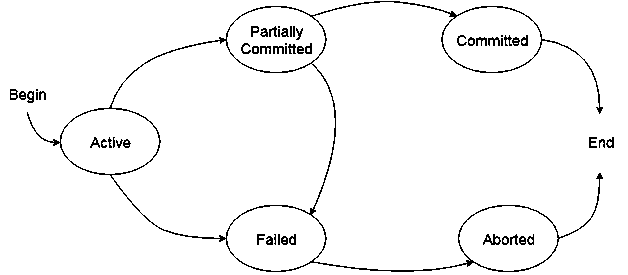
* + It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.
  + In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction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

Durability

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

States of Transaction

In a database, the transaction can be in one of the following states -



Active state

* + The active state is the first state of every transaction. In this state, the transaction is being executed.
  + For example: Insertion or deletion or updating a record is done here. But all the records are still not saved to the database.

Partially committed

* + In the partially committed state, a transaction executes its final operation, but the data is still not saved to the database.
  + In the total mark calculation example, a final display of the total marks step is executed in this state.

Committed

A transaction is said to be in a committed state if it executes all its operations successfully. In this state, all the effects are now permanently saved on the database system.

Failed state

* + If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
  + In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.

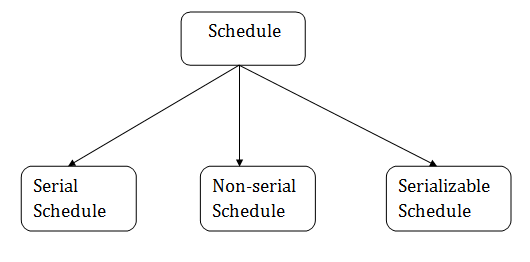
Aborted

* + If any of the checks fail and the transaction has reached a failed state then the database recovery system will make sure that the database is in its previous consistent state. If not then it will abort or roll back the transaction to bring the database into a consistent state.
  + If the transaction fails in the middle of the transaction then before executing the transaction, all the executed transactions are rolled back to its consistent state.
  + After aborting the transaction, the database recovery module will select one of the two operations:

1. Re-start the transaction
2. Kill the transaction

Schedule

A series of operation from one transaction to another transaction is known as schedule. It is used to preserve the order of the operation in each of the individual transaction.



1. Serial Schedule

The serial schedule is a type of schedule where one transaction is executed completely before starting another transaction. In the serial schedule, when the first transaction completes its cycle, then the next transaction is executed.

**For example:** Suppose there are two transactions T1 and T2 which have some operations. If it has no interleaving of operations, then there are the following two possible outcomes:

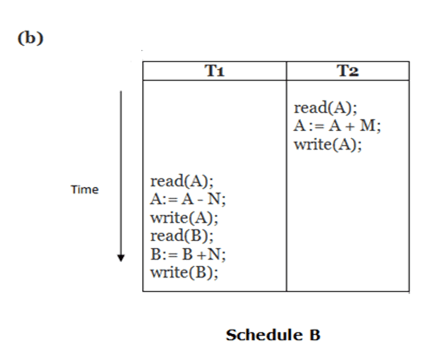
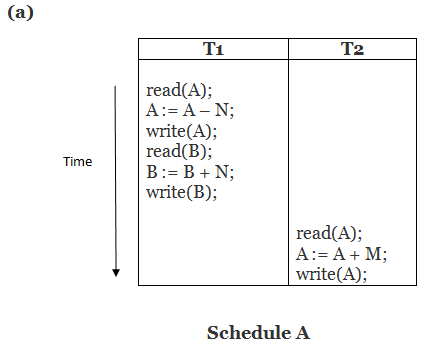
* 1. Execute all the operations of T1 which was followed by all the operations of T2.
  2. Execute all the operations of T1 which was followed by all the operations of T2.
* In the given (a) figure, Schedule A shows the serial schedule where T1 followed by T2.
* In the given (b) figure, Schedule B shows the serial schedule where T2 followed by T1.

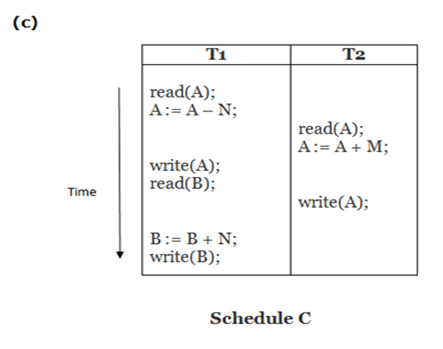
1. Non-serial Schedule

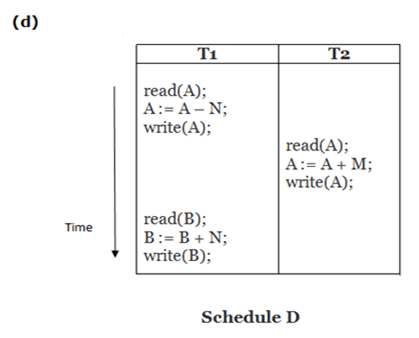
* If interleaving of operations is allowed, then there will be non-serial schedule.
* It contains many possible orders in which the system can execute the individual operations of the transactions.
* In the given figure (c) and (d), Schedule C and Schedule D are the non-serial schedules. It has interleaving of operations.

1. Serializable schedule

* The serializability of schedules is used to find non-serial schedules that allow the transaction to execute concurrently without interfering with one another.
* It identifies which schedules are correct when executions of the transaction have interleaving of their operations.
* A non-serial schedule will be serializable if its result is equal to the result of its transactions executed serially.







# Here,

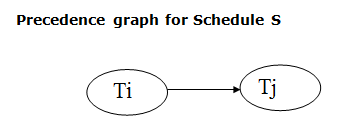
Schedule A and Schedule B are serial schedule. Schedule C and Schedule D are Non-serial schedule.

Testing of Serializability

Serialization Graph is used to test the Serializability of a schedule.

Assume a schedule S. For S, we construct a graph known as precedence graph. This graph has a pair G = (V, E), where V consists a set of vertices, and E consists a set of edges. The set of vertices is used to contain all the transactions participating in the schedule. The set of edges is used to contain all edges Ti ->Tj for which one of the three conditions holds:

* 1. Create a node Ti → Tj if Ti executes write (Q) before Tj executes read (Q).
  2. Create a node Ti → Tj if Ti executes read (Q) before Tj executes write (Q).
  3. Create a node Ti → Tj if Ti executes write (Q) before Tj executes write (Q).

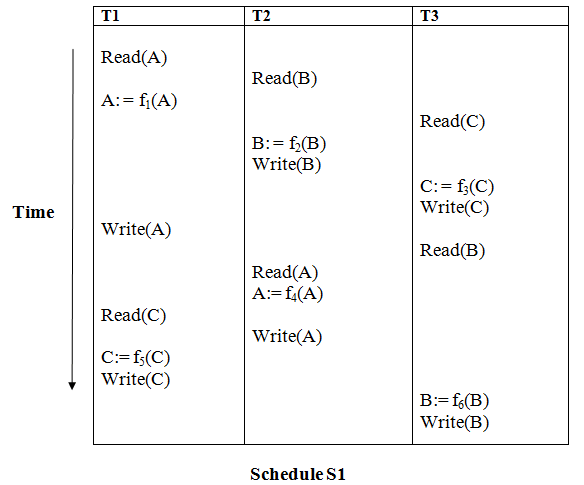


* If a precedence graph contains a single edge Ti → Tj, then all the instructions of Ti

are executed before the first instruction of Tj is executed.

* If a precedence graph for schedule S contains a cycle, then S is non-serializable. If the precedence graph has no cycle, then S is known as serializable.

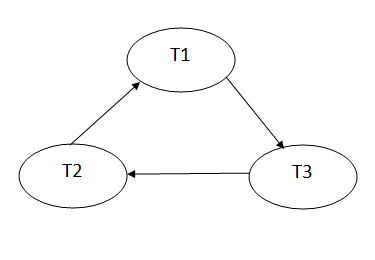
# For example:



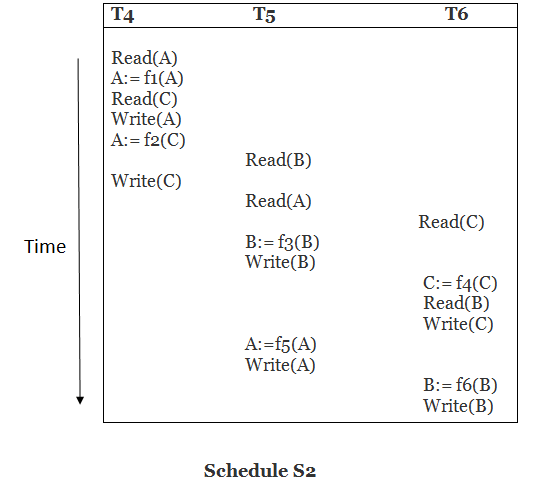
**Explanation:**

**Read(A):** In T1, no subsequent writes to A, so no new edges **Read(B):** In T2, no subsequent writes to B, so no new edges **Read(C):** In T3, no subsequent writes to C, so no new edges **Write(B):** B is subsequently read by T3, so add edge T2 → T3 **Write(C):** C is subsequently read by T1, so add edge T3 → T1 **Write(A):** A is subsequently read by T2, so add edge T1 → T2 **Write(A):** In T2, no subsequent reads to A, so no new edges **Write(C):** In T1, no subsequent reads to C, so no new edges **Write(B):** In T3, no subsequent reads to B, so no new edges

Precedence graph for schedule S1:



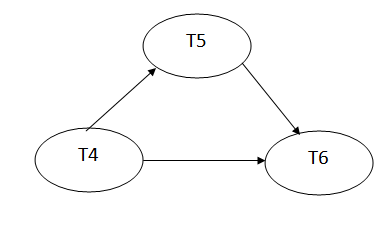
The precedence graph for schedule S1 contains a cycle that's why Schedule S1 is non- serializable.



# Explanation:

**Read(A):** In T4,no subsequent writes to A, so no new edges **Read(C):** In T4, no subsequent writes to C, so no new edges **Write(A):** A is subsequently read by T5, so add edge T4 → T5 **Read(B):** In T5,no subsequent writes to B, so no new edges **Write(C):** C is subsequently read by T6, so add edge T4 → T6 **Write(B):** A is subsequently read by T6, so add edge T5 → T6 **Write(C):** In T6, no subsequent reads to C, so no new edges **Write(A):** In T5, no subsequent reads to A, so no new edges **Write(B):** In T6, no subsequent reads to B, so no new edges

Precedence graph for schedule S2:



The precedence graph for schedule S2 contains no cycle that's why ScheduleS2 is serializable.

Conflict Serializable Schedule

* A schedule is called conflict serializability if after swapping of non-conflicting operations, it can transform into a serial schedule.
* The schedule will be a conflict serializable if it is conflict equivalent to a serial schedule.

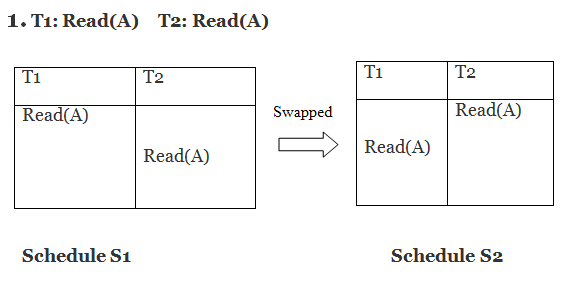
Conflicting Operations

The two operations become conflicting if all conditions satisfy:

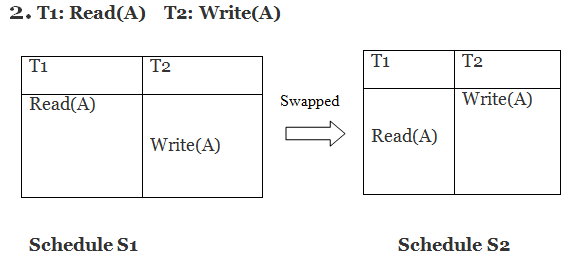
1. Both belong to separate transactions.
2. They have the same data item.
3. They contain at least one write operation.

Example:

Swapping is possible only if S1 and S2 are logically equal.



Here, S1 = S2. That means it is non-conflict.



Here, S1 ≠ S2. That means it is conflict.

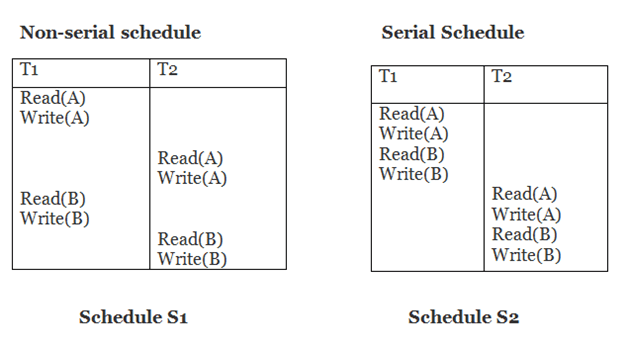
Conflict Equivalent

In the conflict equivalent, one can be transformed to another by swapping non-conflicting operations. In the given example, S2 is conflict equivalent to S1 (S1 can be converted to S2 by swapping non-conflicting operations).

Two schedules are said to be conflict equivalent if and only if:

1. They contain the same set of the transaction.
2. If each pair of conflict operations are ordered in the same way.

Example:



Schedule S2 is a serial schedule because, in this, all operations of T1 are performed before starting any operation of T2. Schedule S1 can be transformed into a serial schedule by swapping non-conflicting operations of S1.

# After swapping of non-conflict operations, the schedule S1 becomes:

|  |  |
| --- | --- |
| **T1 T2** | |
| Read(A) Write(A) Read(B) Write(B) |  |

|  |  |
| --- | --- |
|  | Read(A) Write(A) Read(B) Write(B) |

Since, S1 is conflict serializable.

View Serializability

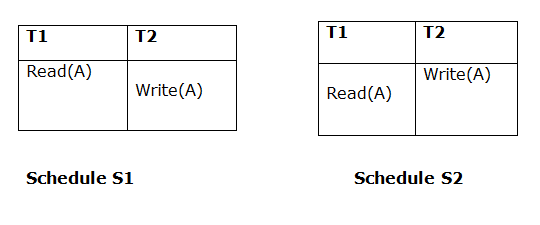
* A schedule will view serializable if it is view equivalent to a serial schedule.
* If a schedule is conflict serializable, then it will be view serializable.
* The view serializable which does not conflict serializable contains blind writes.

View Equivalent

Two schedules S1 and S2 are said to be view equivalent if they satisfy the following conditions:

1. Initial Read

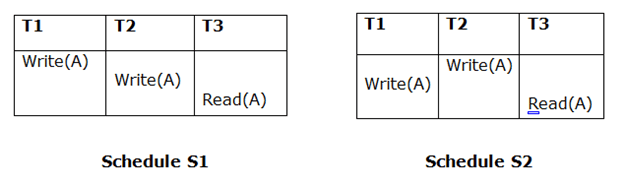
An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.



Above two schedules are view equivalent because Initial read operation in S1 is done by T1 and in S2 it is also done by T1.

1. Updated Read

In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.

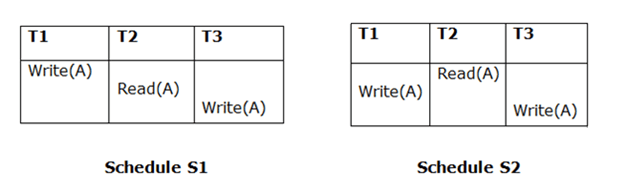


Above two schedules are not view equal because, in S1, T3 is reading A updated by T2 and in S2, T3 is reading A updated by T1.

1. Final Write

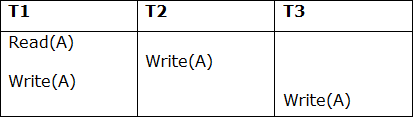
A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.

ginners



Above two schedules is view equal because Final write operation in S1 is done by T3 and in S2, the final write operation is also done by T3.

# Example:



**Schedule S**

With 3 transactions, the total number of possible schedule

1. = 3! = 6

2. S1 = <T1 T2 T3>

3. S2 = <T1 T3 T2>

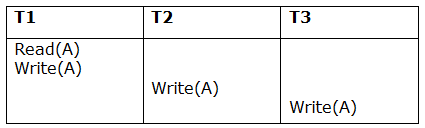
4. S3 = <T2 T3 T1>

5. S4 = <T2 T1 T3>

6. S5 = <T3 T1 T2>

7. S6 = <T3 T2 T1>

# Taking first schedule S1:



**Schedule S1**

**Step 1:** final updation on data items

In both schedules S and S1, there is no read except the initial read that's why we don't need to check that condition.

**Step 2:** Initial Read

The initial read operation in S is done by T1 and in S1, it is also done by T1.

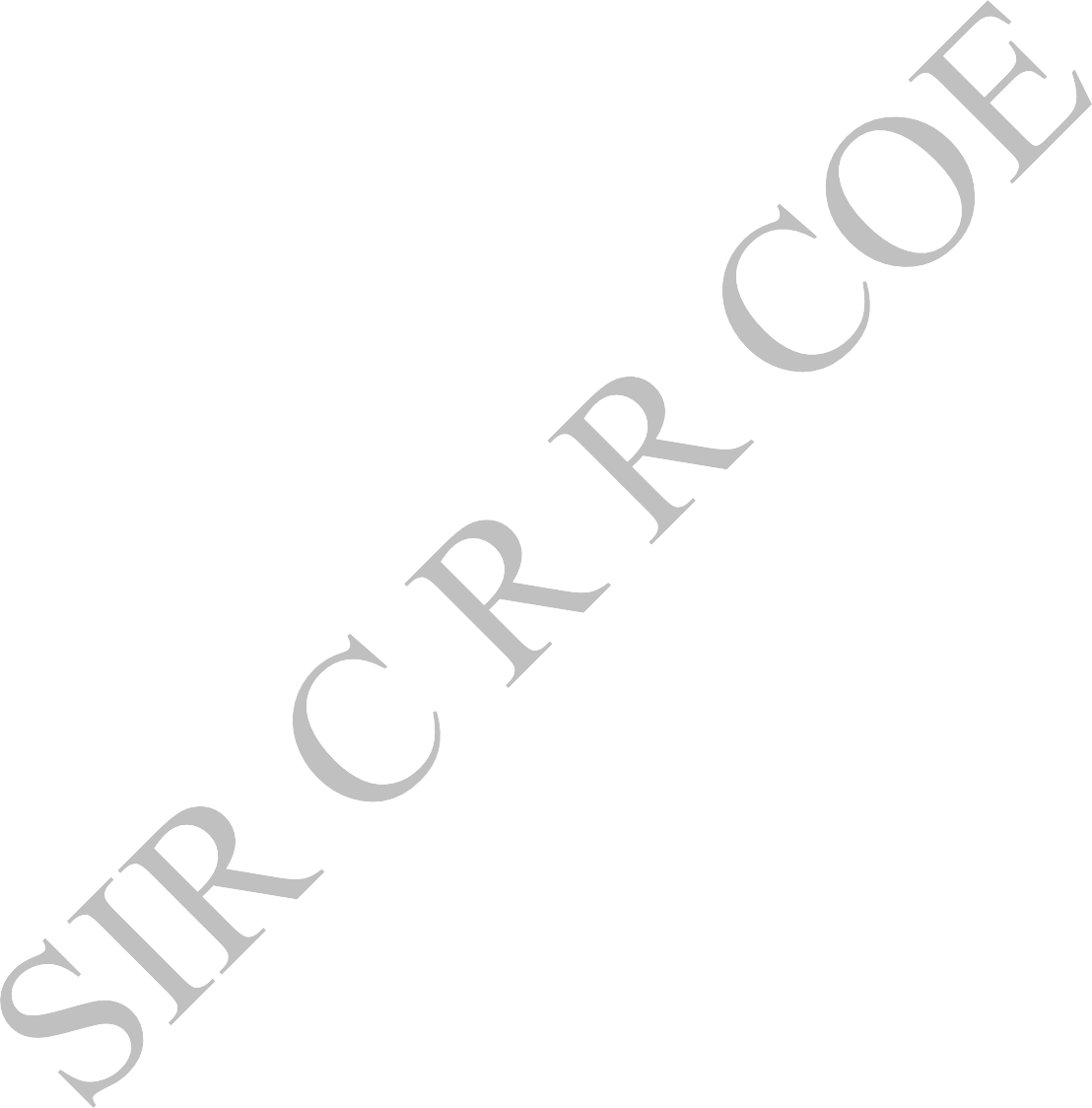
**Step 3:** Final Write

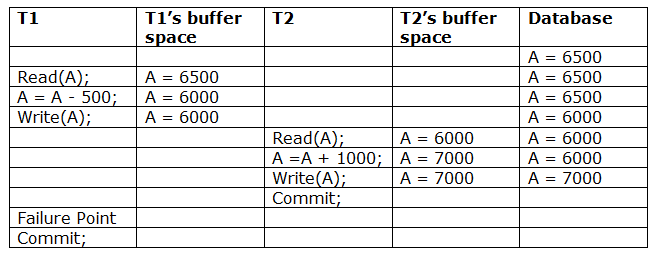
The final write operation in S is done by T3 and in S1, it is also done by T3. So, S and S1 are view Equivalent.

The first schedule S1 satisfies all three conditions, so we don't need to check another schedule.

# Hence, view equivalent serial schedule is:

1. T1 → T2 → T3

Recoverability of Schedule



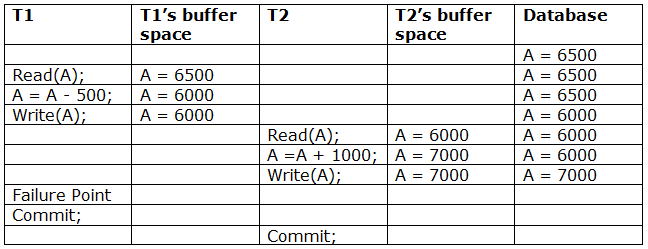
Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those

tr

ansactions.

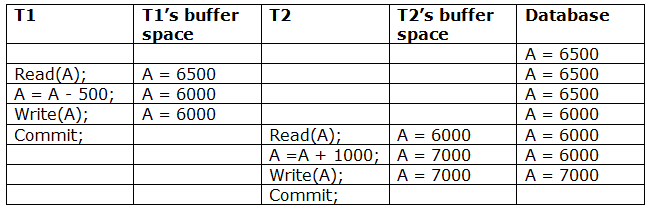
The above table 1 shows a schedule which has two transactions. T1 reads and writes the value of A and that value is read and written by T2. T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1. T2 should also be rollback because it reads the value written by T1, but T2 can't be rollback because it already committed. So this type of schedule is known as irrecoverable schedule.

**Irrecoverable schedule:** The schedule will be irrecoverable if Tj reads the updated value of Ti and Tj committed before Ti commit.



The above table 2 shows a schedule with two transactions. Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails. Due to this, we have to rollback T1. T2 should be rollback because T2 has read the value written by T1. As it has not committed before T1 commits so we can rollback transaction T2 as well. So it is recoverable with cascade rollback.

**Recoverable with cascading rollback:** The schedule will be recoverable with cascading rollback if Tj reads the updated value of Ti. Commit of Tj is delayed till commit of Ti.



The above Table 3 shows a schedule with two transactions. Transaction T1 reads and write A and commits, and that value is read and written by T2. So this is a cascade less recoverable schedule.

Failure Classification

To find that where the problem has occurred, we generalize a failure into the following categories:

* 1. Transaction failure
  2. System crash
  3. Disk failure
     1. Transaction failure

The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further. If a few transaction or process is hurt, then this is called as transaction failure.

Reasons for a transaction failure could be -

* + - 1. **Logical errors:** If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.
      2. **Syntax error:** It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. **For example,** The system aborts an active transaction, in case of deadlock or resource unavailability.
    1. System Crash

o System failure can occur due to power failure or other hardware or software failure. **Example:** Operating system error.

**Fail-stop assumption:** In the system crash, non-volatile storage is assumed not to be corrupted.

* + 1. Disk Failure
* It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.
* Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.

Log-Based Recovery

* The log is a sequence of records. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.
* If any operation is performed on the database, then it will be recorded in the log.
* But the process of storing the logs should be done before the actual transaction is applied in the database.

o

Let's assume there is a transaction to modify the City of a student. The following logs are written for this transaction.

* When the transaction is initiated, then it writes 'start' log.

1. <Tn, Start>

* When the transaction modifies the City from 'Noida' to 'Bangalore', then another log is written to the file.

1. <Tn, City, 'Noida', 'Bangalore' >

* When the transaction is finished, then it writes another log to indicate the end of the transaction.

1. <Tn, Commit>

There are two approaches to modify the database:

1. Deferred database modification:
   * The deferred modification technique occurs if the transaction does not modify the database until it has committed.
   * In this method, all the logs are created and stored in the stable storage, and the database is updated when a transaction commits.
2. Immediate database modification:
   * The Immediate modification technique occurs if database modification occurs while the transaction is still active.
   * In this technique, the database is modified immediately after every operation. It follows an actual database modification.

Recovery using Log records

When the system is crashed, then the system consults the log to find which transactions need to be undone and which need to be redone.

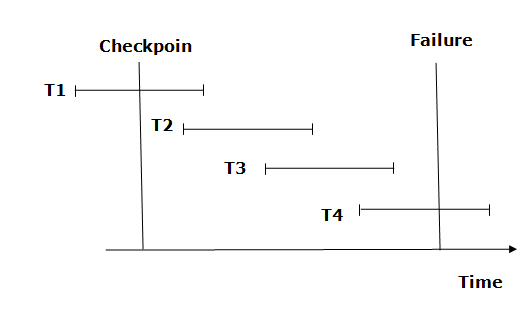
1. If the log contains the record <Ti, Start> and <Ti, Commit> or <Ti, Commit>, then the Transaction Ti needs to be redone.
2. If log contains record<Tn, Start> but does not contain the record either <Ti, commit> or <Ti, abort>, then the Transaction Ti needs to be undone.

Checkpoint

* + The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.
  + The checkpoint is like a bookmark. While the execution of the transaction, such checkpoints are marked, and the transaction is executed then using the steps of the transaction, the log files will be created.
  + When it reaches to the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till next checkpoint and so on.
  + The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.

Recovery using Checkpoint

In the following manner, a recovery system recovers the database from this failure:



* + The recovery system reads log files from the end to start. It reads log files from T4 to T1.
  + Recovery system maintains two lists, a redo-list, and an undo-list.
  + The transaction is put into redo state if the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>. In the redo-list and their previous list, all the transactions are removed and then redone before saving their logs.
  + **For example:** In the log file, transaction T2 and T3 will have <Tn, Start> and <Tn, Commit>. The T1 transaction will have only <Tn, commit> in the log file. That's why the transaction is committed after the checkpoint is crossed. Hence it puts T1, T2 and T3 transaction into redo list.
  + The transaction is put into undo state if the recovery system sees a log with <Tn, Start> but no commit or abort log found. In the undo-list, all the transactions are undone, and their logs are removed.
  + **For example:** Transaction T4 will have <Tn, Start>. So T4 will be put into undo list since this transaction is not yet complete and failed amid.

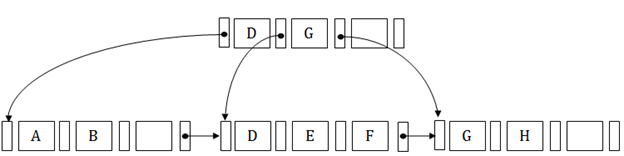
INDEXING TECHNIQUES:

B+ Tree

* + The B+ tree is a balanced binary search tree. It follows a multi-level index format.
  + In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.
  + In the B+ tree, the leaf nodes are linked using a link list. Therefore, a B+ tree can support random access as well as sequential access.

Structure of B+ Tree

* + In the B+ tree, every leaf node is at equal distance from the root node. The B+ tree is of the order n where n is fixed for every B+ tree.
  + It contains an internal node and leaf node.



Internal node

* + An internal node of the B+ tree can contain at least n/2 record pointers except the root node.
  + At most, an internal node of the tree contains n pointers.

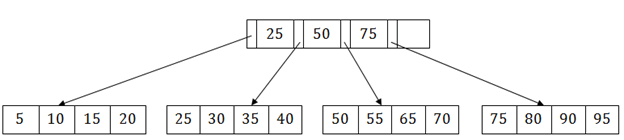
Leaf node

* + The leaf node of the B+ tree can contain at least n/2 record pointers and n/2 key values.
  + At most, a leaf node contains n record pointer and n key values.
  + Every leaf node of the B+ tree contains one block pointer P to point to next leaf node.

Searching a record in B+ Tree

Suppose we have to search 55 in the below B+ tree structure. First, we will fetch for the intermediary node which will direct to the leaf node that can contain a record for 55.

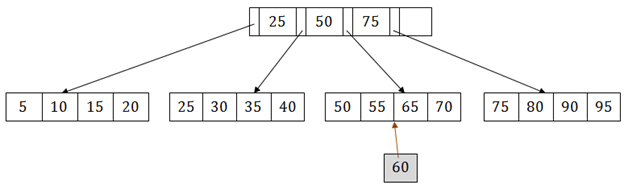
So, in the intermediary node, we will find a branch between 50 and 75 nodes. Then at the end, we will be redirected to the third leaf node. Here DBMS will perform a sequential search to find 55.



B+ Tree Insertion

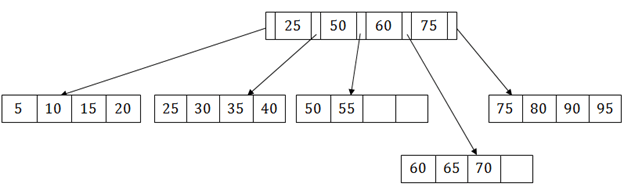
Suppose we want to insert a record 60 in the below structure. It will go to the 3rd leaf node after 55. It is a balanced tree, and a leaf node of this tree is already full, so we cannot insert 60 there.

In this case, we have to split the leaf node, so that it can be inserted into tree without affecting the fill factor, balance and order.



The 3rd leaf node has the values (50, 55, 60, 65, 70) and its current root node is 50. We will split the leaf node of the tree in the middle so that its balance is not altered. So we can group (50, 55) and (60, 65, 70) into 2 leaf nodes.

If these two has to be leaf nodes, the intermediate node cannot branch from 50. It should have 60 added to it, and then we can have pointers to a new leaf node.

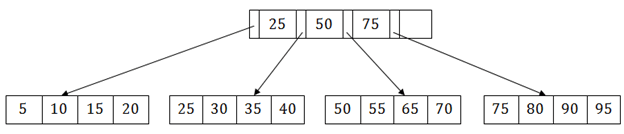


This is how we can insert an entry when there is overflow. In a normal scenario, it is very easy to find the node where it fits and then place it in that leaf node.

B+ Tree Deletion

Suppose we want to delete 60 from the above example. In this case, we have to remove 60 from the intermediate node as well as from the 4th leaf node too. If we remove it from the intermediate node, then the tree will not satisfy the rule of the B+ tree. So we need to modify it to have a balanced tree.

After deleting node 60 from above B+ tree and re-arranging the nodes, it will show as follows:



File Organization

* + The **File** is a collection of records. Using the primary key, we can access the records. The type and frequency of access can be determined by the type of file organization which was used for a given set of records.
  + File organization is a logical relationship among various records. This method defines how file records are mapped onto disk blocks.
  + File organization is used to describe the way in which the records are stored in terms of blocks, and the blocks are placed on the storage medium.
  + The first approach to map the database to the file is to use the several files and store only one fixed length record in any given file. An alternative approach is to structure our files so that we can contain multiple lengths for records.
  + Files of fixed length records are easier to implement than the files of variable length records.

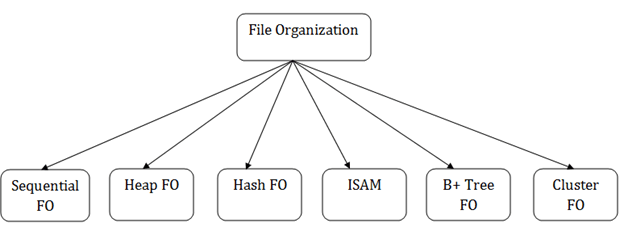
Objective of file organization

* + It contains an optimal selection of records, i.e., records can be selected as fast as possible.
  + To perform insert, delete or update transaction on the records should be quick and easy.
  + The duplicate records cannot be induced as a result of insert, update or delete.
  + For the minimal cost of storage, records should be stored efficiently.

Types of file organization:

File organization contains various methods. These particular methods have pros and cons on the basis of access or selection. In the file organization, the programmer decides the best- suited file organization method according to his requirement.

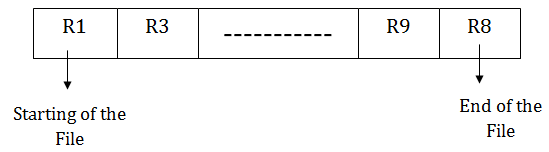
Types of file organization are as follows:



* + [Sequential file organization](https://www.javatpoint.com/dbms-sequential-file-organization)
  + [Heap file organization](https://www.javatpoint.com/dbms-heap-file-organization)
  + [Hash file organization](https://www.javatpoint.com/dbms-hash-file-organization)
  + [B+ file organization](https://www.javatpoint.com/dbms-b-plus-file-organization)
  + [Indexed sequential access method (ISAM)](https://www.javatpoint.com/dbms-indexed-sequential-access-method)
  + [Cluster file organization](https://www.javatpoint.com/dbms-cluster-file-organization) Sequential File Organization

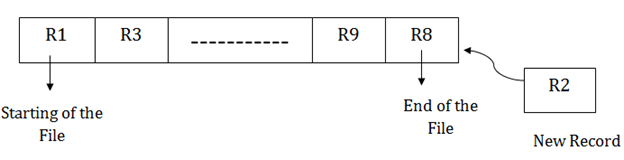
This method is the easiest method for file organization. In this method, files are stored sequentially. This method can be implemented in two ways:

1. Pile File Method:
   * It is a quite simple method. In this method, we store the record in a sequence, i.e., one after another. Here, the record will be inserted in the order in which they are inserted into tables.
   * In case of updating or deleting of any record, the record will be searched in the memory blocks. When it is found, then it will be marked for deleting, and the new record is inserted.

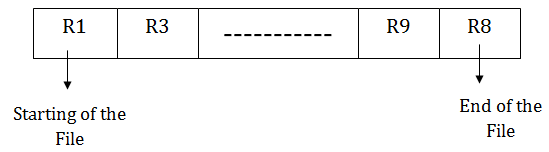


Insertion of the new record:

Suppose we have four records R1, R3 and so on upto R9 and R8 in a sequence. Hence, records are nothing but a row in the table. Suppose we want to insert a new record R2 in the sequence, then it will be placed at the end of the file. Here, records are nothing but a row in any table.

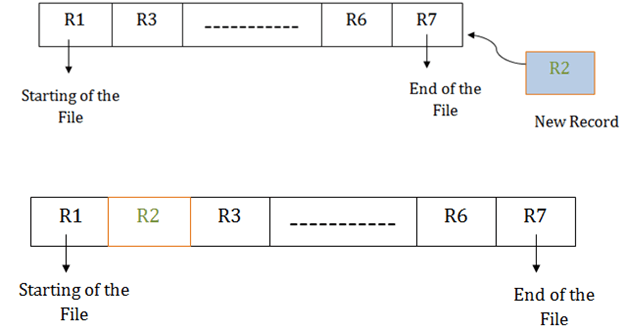


1. Sorted File Method:
   * In this method, the new record is always inserted at the file's end, and then it will sort the sequence in ascending or descending order. Sorting of records is based on any primary key or any other key.
   * In the case of modification of any record, it will update the record and then sort the file, and lastly, the updated record is placed in the right place.



Insertion of the new record:

Suppose there is a preexisting sorted sequence of four records R1, R3 and so on upto R6 and R7. Suppose a new record R2 has to be inserted in the sequence, then it will be inserted at the end of the file, and then it will sort the sequence.



Pros of sequential file organization

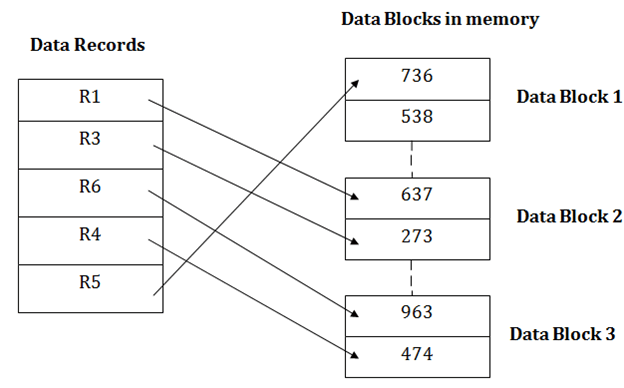
* + It contains a fast and efficient method for the huge amount of data.
  + In this method, files can be easily stored in cheaper storage mechanism like magnetic tapes.
  + It is simple in design. It requires no much effort to store the data.
  + This method is used when most of the records have to be accessed like grade calculation of a student, generating the salary slip, etc.
  + This method is used for report generation or statistical calculations.

Cons of sequential file organization

* + It will waste time as we cannot jump on a particular record that is required but we have to move sequentially which takes our time.
  + Sorted file method takes more time and space for sorting the records.

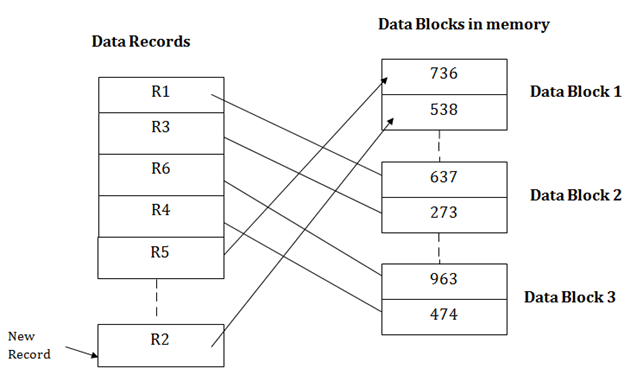
Heap file organization

* + It is the simplest and most basic type of organization. It works with data blocks. In heap file organization, the records are inserted at the file's end. When the records are inserted, it doesn't require the sorting and ordering of records.
  + When the data block is full, the new record is stored in some other block. This new data block need not to be the very next data block, but it can select any data block in the memory to store new records. The heap file is also known as an unordered file.
  + In the file, every record has a unique id, and every page in a file is of the same size. It is the DBMS responsibility to store and manage the new records.



Insertion of a new record

Suppose we have five records R1, R3, R6, R4 and R5 in a heap and suppose we want to insert a new record R2 in a heap. If the data block 3 is full then it will be inserted in any of the database selected by the DBMS, let's say data block 1.



If we want to search, update or delete the data in heap file organization, then we need to traverse the data from staring of the file till we get the requested record.

If the database is very large then searching, updating or deleting of record will be time- consuming because there is no sorting or ordering of records. In the heap file organization, we need to check all the data until we get the requested record.

Pros of Heap file organization

* + It is a very good method of file organization for bulk insertion. If there is a large number of data which needs to load into the database at a time, then this method is best suited.
  + In case of a small database, fetching and retrieving of records is faster than the sequential record.

Cons of Heap file organization

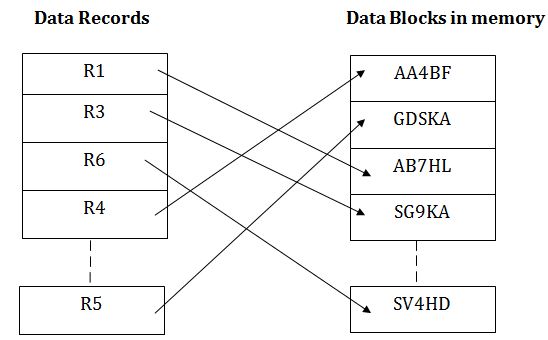
* + This method is inefficient for the large database because it takes time to search or modify the record.

o

* + This method is inefficient for large databases.

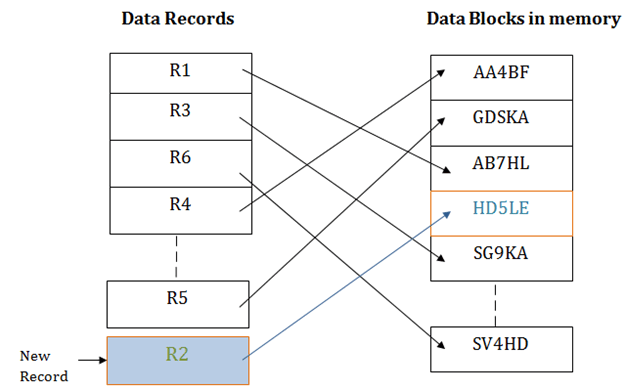
Hash File Organization

Hash File Organization uses the computation of hash function on some fields of the records. The hash function's output determines the location of disk block where the records are to be placed.



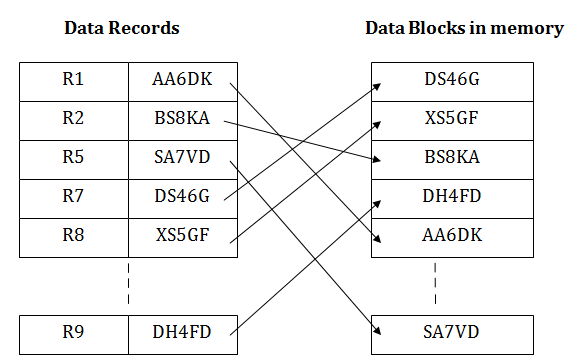
When a record has to be received using the hash key columns, then the address is generated, and the whole record is retrieved using that address. In the same way, when a new record has to be inserted, then the address is generated using the hash key and record is directly inserted. The same process is applied in the case of delete and update.

In this method, there is no effort for searching and sorting the entire file. In this method, each record will be stored randomly in the memory.



Indexed sequential access method (ISAM)

ISAM method is an advanced sequential file organization. In this method, records are stored in the file using the primary key. An index value is generated for each primary key and mapped with the record. This index contains the address of the record in the file.



If any record has to be retrieved based on its index value, then the address of the data block is fetched and the record is retrieved from the memory.

Pros of ISAM:

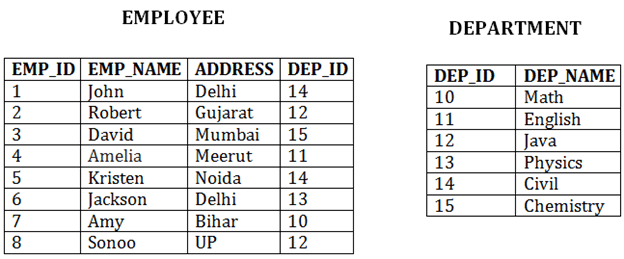
* + In this method, each record has the address of its data block, searching a record in a huge database is quick and easy.
  + This method supports range retrieval and partial retrieval of records. Since the index is based on the primary key values, we can retrieve the data for the given range of value. In the same way, the partial value can also be easily searched, i.e., the student name starting with 'JA' can be easily searched.

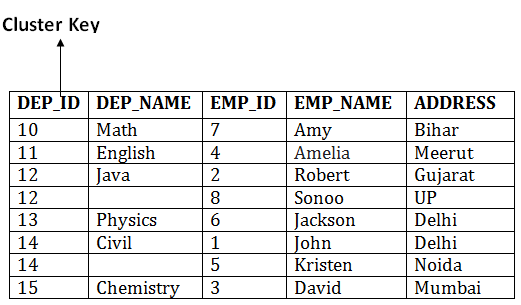
Cons of ISAM

* + This method requires extra space in the disk to store the index value.
  + When the new records are inserted, then these files have to be reconstructed to maintain the sequence.
  + When the record is deleted, then the space used by it needs to be released. Otherwise, the performance of the database will slow down.

Cluster file organization

* + When the two or more records are stored in the same file, it is known as clusters. These files will have two or more tables in the same data block, and key attributes which are used to map these tables together are stored only once.
  + This method reduces the cost of searching for various records in different files.
  + The cluster file organization is used when there is a frequent need for joining the tables with the same condition. These joins will give only a few records from both tables. In the given example, we are retrieving the record for only particular departments. This method can't be used to retrieve the record for the entire department.





In this method, we can directly insert, update or delete any record. Data is sorted based on the key with which searching is done. Cluster key is a type of key with which joining of the table is performed.

Types of Cluster file organization:

Cluster file organization is of two types:

1. Indexed Clusters:

In indexed cluster, records are grouped based on the cluster key and stored together. The above EMPLOYEE and DEPARTMENT relationship is an example of an indexed cluster. Here, all the records are grouped based on the cluster key- DEP\_ID and all the records are grouped.

1. Hash Clusters:

It is similar to the indexed cluster. In hash cluster, instead of storing the records based on the cluster key, we generate the value of the hash key for the cluster key and store the records with the same hash key value.

Pros of Cluster file organization

* + The cluster file organization is used when there is a frequent request for joining the tables with same joining condition.
  + It provides the efficient result when there is a 1:M mapping between the tables.

Cons of Cluster file organization

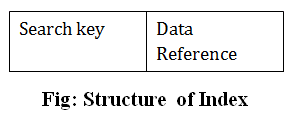
* + This method has the low performance for the very large database.
  + If there is any change in joining condition, then this method cannot use. If we change the condition of joining then traversing the file takes a lot of time.
  + This method is not suitable for a table with a 1:1 condition.

Indexing in DBMS

* + Indexing is used to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.
  + The index is a type of data structure. It is used to locate and access the data in a database table quickly.

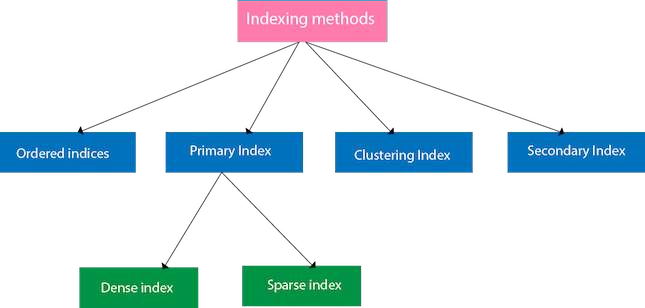
Index structure:

Indexes can be created using some database columns.



* + The first column of the database is the search key that contains a copy of the primary key or candidate key of the table. The values of the primary key are stored in sorted order so that the corresponding data can be accessed easily.
  + The second column of the database is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particular key can be found.

Indexing Methods



Ordered indices

The indices are usually sorted to make searching faster. The indices which are sorted are known as ordered indices.

**Example**: Suppose we have an employee table with thousands of record and each of which is 10 bytes long. If their IDs start with 1, 2, 3. and so on and we have to search student with

ID-543.

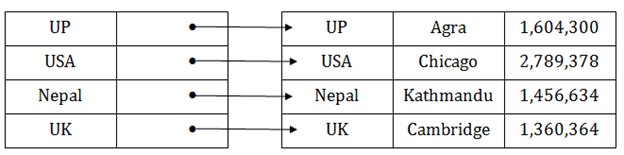
* + In the case of a database with no index, we have to search the disk block from starting till it reaches 543. The DBMS will read the record after reading 543\*10=5430 bytes.
  + In the case of an index, we will search using indexes and the DBMS will read the record after reading 542\*2= 1084 bytes which are very less compared to the previous case.

Primary Index

* + If the index is created on the basis of the primary key of the table, then it is known as primary indexing. These primary keys are unique to each record and contain 1:1 relation between the records.
  + As primary keys are stored in sorted order, the performance of the searching operation is quite efficient.
  + The primary index can be classified into two types: Dense index and Sparse index.

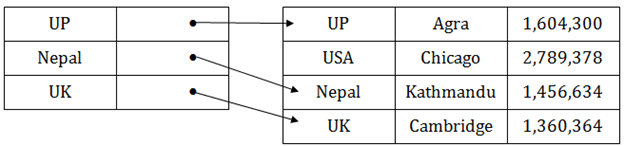
Dense index

* + The dense index contains an index record for every search key value in the data file. It makes searching faster.
  + In this, the number of records in the index table is same as the number of records in the main table.
  + It needs more space to store index record itself. The index records have the search key and a pointer to the actual record on the disk.



Sparse index

* + In the data file, index record appears only for a few items. Each item points to a block.
  + In this, instead of pointing to each record in the main table, the index points to the records in the main table in a gap.

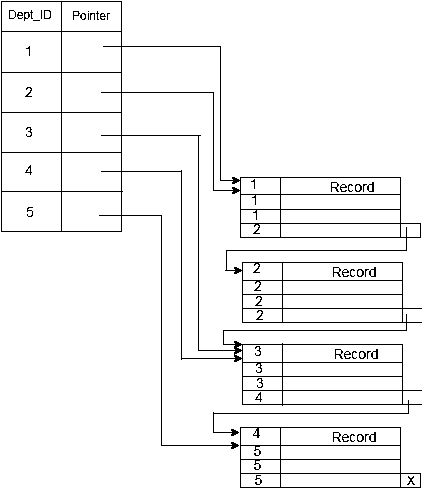


Clustering Index

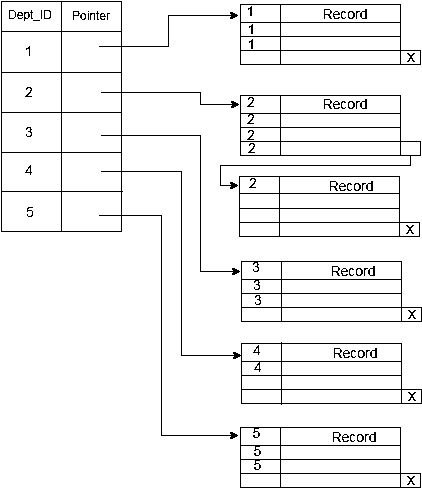
* + A clustered index can be defined as an ordered data file. Sometimes the index is created on non-primary key columns which may not be unique for each record.
  + In this case, to identify the record faster, we will group two or more columns to get the unique value and create index out of them. This method is called a clustering index.
  + The records which have similar characteristics are grouped, and indexes are created for these group.

**Example**: suppose a company contains several employees in each department. Suppose we use a clustering index, where all employees which belong to the same Dept\_ID are

considered within a single cluster, and index pointers point to the cluster as a whole. Here Dept\_Id is a non-unique key.



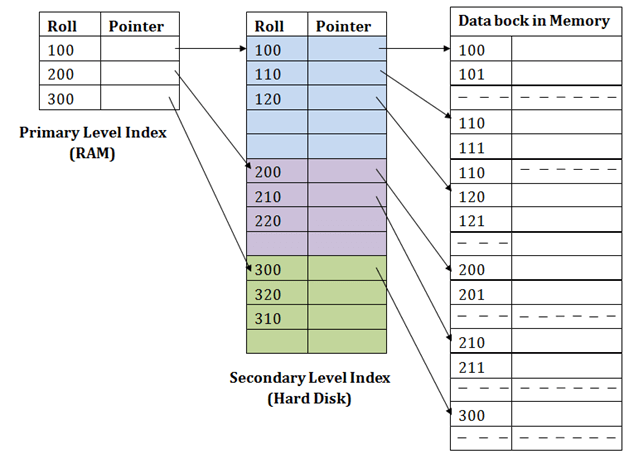
The previous schema is little confusing because one disk block is shared by records which belong to the different cluster. If we use separate disk block for separate clusters, then it is called better technique.



Secondary Index

In the sparse indexing, as the size of the table grows, the size of mapping also grows. These mappings are usually kept in the primary memory so that address fetch should be faster. Then the secondary memory searches the actual data based on the address got from mapping. If the mapping size grows then fetching the address itself becomes slower. In this case, the sparse index will not be efficient. To overcome this problem, secondary indexing is introduced.

In secondary indexing, to reduce the size of mapping, another level of indexing is introduced. In this method, the huge range for the columns is selected initially so that the mapping size of the first level becomes small. Then each range is further divided into smaller ranges. The mapping of the first level is stored in the primary memory, so that address fetch is faster. The mapping of the second level and actual data are stored in the secondary memory (hard disk).



# For example:

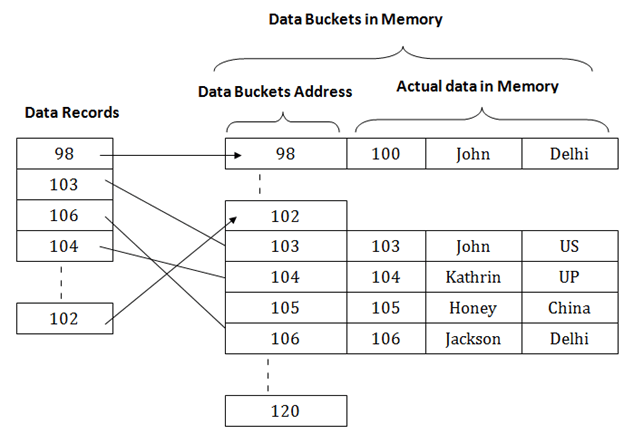
* + If you want to find the record of roll 111 in the diagram, then it will search the highest entry which is smaller than or equal to 111 in the first level index. It will get 100 at this level.
  + Then in the second index level, again it does max (111) <= 111 and gets 110. Now using the address 110, it goes to the data block and starts searching each record till it gets 111.
  + This is how a search is performed in this method. Inserting, updating or deleting is also done in the same manner.

Hashing

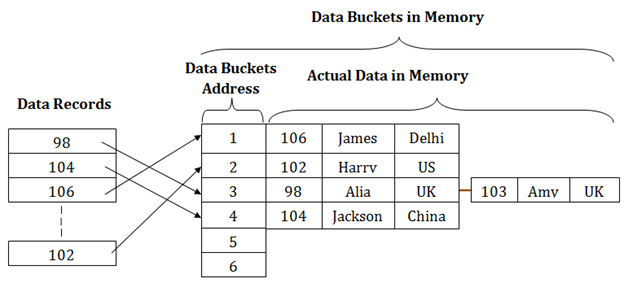
In a huge database structure, it is very inefficient to search all the index values and reach the desired data. Hashing technique is used to calculate the direct location of a data record on the disk without using index structure.

In this technique, data is stored at the data blocks whose address is generated by using the hashing function. The memory location where these records are stored is known as data bucket or data blocks.

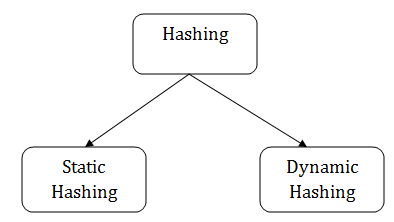
In this, a hash function can choose any of the column value to generate the address. Most of the time, the hash function uses the primary key to generate the address of the data block. A hash function is a simple mathematical function to any complex mathematical function. We can even consider the primary key itself as the address of the data block. That means each row whose address will be the same as a primary key stored in the data block.



The above diagram shows data block addresses same as primary key value. This hash function can also be a simple mathematical function like exponential, mod, cos, sin, etc. Suppose we have mod (5) hash function to determine the address of the data block. In this case, it applies mod (5) hash function on the primary keys and generates 3, 3, 1, 4 and 2 respectively, and records are stored in those data block addresses.



Types of Hashing:

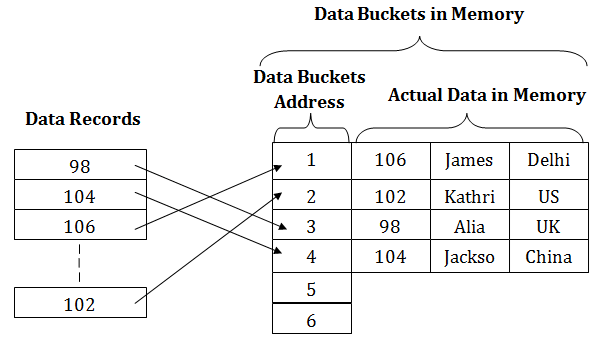


* + [Static Hashing](https://www.javatpoint.com/dbms-static-hashing)
  + [Dynamic Hashing](https://www.javatpoint.com/dbms-dynamic-hashing)

Static Hashing

In static hashing, the resultant data bucket address will always be the same. That means if we generate an address for EMP\_ID =103 using the hash function mod (5) then it will always result in same bucket address 3. Here, there will be no change in the bucket address.

Hence in this static hashing, the number of data buckets in memory remains constant throughout. In this example, we will have five data buckets in the memory used to store the data.



Operations of Static Hashing

# Searching a record

When a record needs to be searched, then the same hash function retrieves the address of the bucket where the data is stored.

# Insert a Record

When a new record is inserted into the table, then we will generate an address for a new record based on the hash key and record is stored in that location.

# Delete a Record

To delete a record, we will first fetch the record which is supposed to be deleted. Then we will delete the records for that address in memory.

# Update a Record

To update a record, we will first search it using a hash function, and then the data record is updated.

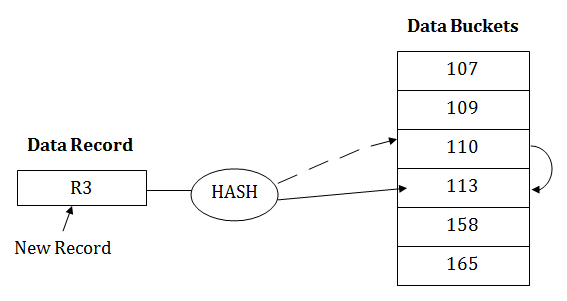
If we want to insert some new record into the file but the address of a data bucket generated by the hash function is not empty, or data already exists in that address. This situation in the static hashing is known as **bucket overflow**. This is a critical situation in this method.

To overcome this situation, there are various methods. Some commonly used methods are as follows:

1. Open Hashing

When a hash function generates an address at which data is already stored, then the next bucket will be allocated to it. This mechanism is called as **Linear Probing**.

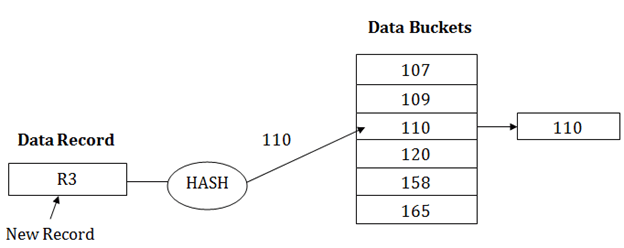
**For example:** suppose R3 is a new address which needs to be inserted, the hash function generates address as 112 for R3. But the generated address is already full. So the system searches next available data bucket, 113 and assigns R3 to it.



1. Close Hashing

When buckets are full, then a new data bucket is allocated for the same hash result and is linked after the previous one. This mechanism is known as **Overflow chaining**.

**For example:** Suppose R3 is a new address which needs to be inserted into the table, the hash function generates address as 110 for it. But this bucket is full to store the new data. In this case, a new bucket is inserted at the end of 110 buckets and is linked to it.



Dynamic Hashing

* The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
* In this method, data buckets grow or shrink as the records increases or decreases. This method is also known as Extendable hashing method.
* This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

How to search a key

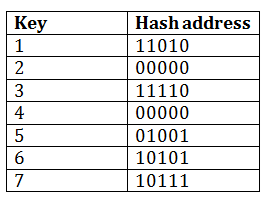
* First, calculate the hash address of the key.
* Check how many bits are used in the directory, and these bits are called as i.
* Take the least significant i bits of the hash address. This gives an index of the directory.
* Now using the index, go to the directory and find bucket address where the record might be.

How to insert a new record

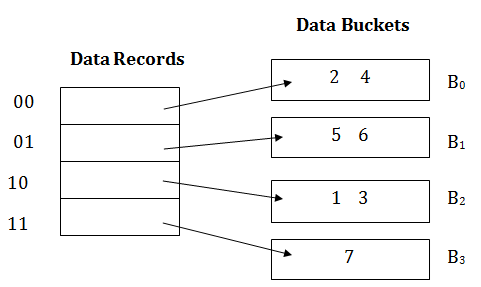
* Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
* If there is still space in that bucket, then place the record in it.
* If the bucket is full, then we will split the bucket and redistribute the records.

For example:

Consider the following grouping of keys into buckets, depending on the prefix of their hash address:

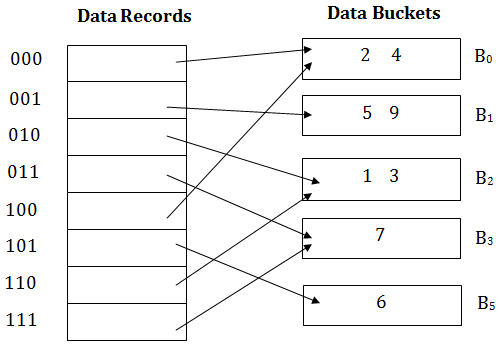


The last two bits of 2 and 4 are 00. So it will go into bucket B0. The last two bits of 5 and 6 are 01, so it will go into bucket B1. The last two bits of 1 and 3 are 10, so it will go into bucket B2. The last two bits of 7 are 11, so it will go into B3.



Insert key 9 with hash address 10001 into the above structure:

* Since key 9 has hash address 10001, it must go into the first bucket. But bucket B1 is full, so it will get split.
* The splitting will separate 5, 9 from 6 since last three bits of 5, 9 are 001, so it will go into bucket B1, and the last three bits of 6 are 101, so it will go into bucket B5.
* Keys 2 and 4 are still in B0. The record in B0 pointed by the 000 and 100 entry because last two bits of both the entry are 00.
* Keys 1 and 3 are still in B2. The record in B2 pointed by the 010 and 110 entry because last two bits of both the entry are 10.
* Key 7 are still in B3. The record in B3 pointed by the 111 and 011 entry because last two bits of both the entry are 11.



Advantages of dynamic hashing

* In this method, the performance does not decrease as the data grows in the system. It simply increases the size of memory to accommodate the data.
* In this method, memory is well utilized as it grows and shrinks with the data. There will not be any unused memory lying.
* This method is good for the dynamic database where data grows and shrinks frequently.

Disadvantages of dynamic hashing

* In this method, if the data size increases then the bucket size is also increased. These addresses of data will be maintained in the bucket address table. This is because the data address will keep changing as buckets grow and shrink. If there is a huge increase in data, maintaining the bucket address table becomes tedious.
* In this case, the bucket overflow situation will also occur. But it might take little time to reach this situation than static hashing.