UNIT – IV

- Introduction to Schema Refinement
- Functional Dependencies
- Reasoning about FDs
- Normal Forms
- Properties of Decomposition
- Normalization
- Transaction concept
- Transaction states
- Concurrent Executions
- Serialiazability
- Recoverability
- Testing for Serializability

1) Introduction to Schema Refinement

- The Schema Refinement refers to refining the schema by using techniques like **decomposition.**
- Normalization or Schema Refinement means 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.
- Redundancy refers to repetition of same data or duplicate copies of same data stored in different locations.
- **Data redundancy leads to insertion/updation/deletion anomalies**

Anomalies or problems due to redundancy

- Anomalies refers to the problems occurred after poorly planned and unnormalised databases where all the data is stored in one table which is sometimes called a flat file database.

- Let us consider such type of schema –

Sid	sname	cid	cname	Fee
S1	A	C1	C	5K
S2	B	C1	C	5K
S3	C	C2	DBMS	10k
S4	D	C2	DBMS	10K
S1	A	C3	JAVA	15K

- Here all the data is stored in a single table which causes redundancy of data or say anomalies

  1) **Insertion anomalies** : It may not be possible to store some information unless some other information is stored as well.
  2) **update anomalies**: If one copy of redundant data is updated, then inconsistency is created unless all redundant copies of data are updated.
  3) **deletion anomalies**: It may not be possible to delete some information without losing some other information as well.

Problem in updation / updation anomaly

If there is updation in the fee from 5000 to 7000, then we have to update FEE column in all the rows, else data will become inconsistent

sid	sname	cid	cname	fee
S1	A	C1	C	~~5K~~ 7K
S2	B	C1	C	5K
S3	C	C2	DBMS	10K
S4	D	C2	DBMS	10K
S1	A	C3	JAVA	15K

For the same course c1, A has &k as fees and B has 5K as fees .Inconsistency due to updation

InsertionAnomalies

New course is introduced C4, But no student is there who is having C4 subject.

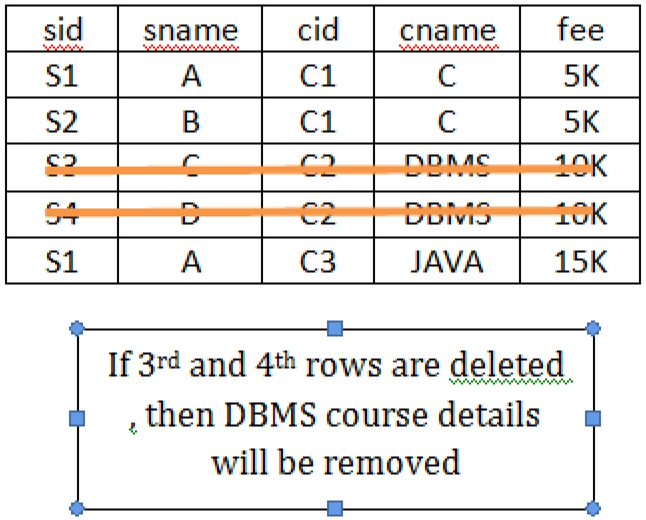
sid	sname	cid	cname	fee
S1	A	C1	C	5K
S2	B	C1	C	5K
S3	C	C2	DBMS	10K
S4	D	C2	DBMS	10K
S1	A	C3	JAVA	15K

XX	XX	C4	.NET	12K

To insert new course, it is necessary to insert dummy values because can't insert null to sid

Deletion Anomaly

Deletion of S3 student cause the deletion of course. Because of deletion of some data forced to delete some other useful data.



sid	sname	cid	cname	fee
S1	A	C1	C	5K
S2	B	C1	C	5K
~~S3~~	~~C~~	~~C2~~	~~DBMS~~	~~10K~~
~~S4~~	~~D~~	~~C2~~	~~DBMS~~	~~10K~~
S1	A	C3	JAVA	15K

If 3rd and 4th rows are deleted , then DBMS course details will be removed

- ❖ **Data Redundancy leads to Anomalies**
- ❖ **Schema Refinement or Normalization helps to remove redundancy .**
- ❖ **The best Technique for Schema Refinement is Decomposition of Tables**

sid	sname	cid	cname	fee
S1	A	C1	C	5K
S2	B	C1	C	5K
S3	C	C2	DBMS	10K
S4	D	C2	DBMS	10K
S1	A	C3	JAVA	15K

sid	sname	cid
S1	A	C1
S2	B	C1
~~S3~~	~~C~~	~~C2~~
~~S4~~	~~D~~	~~C2~~
S1	A	C3

Deletion Anomaly Removed

cid	cname	fee
C1	C	~~5K~~ 7K
C2	DBMS	10K
C3	JAVA	15K
C4	.NET	12K

Updation Anomaly Removed

Insertion Anomaly Removed

2) Functional Dependencies

- **Functional dependency** is a relationship that exist when one attribute uniquely determines another attribute.

  $X \rightarrow Y$

  X Determines Y or Y functionally dependent on X
- Functional dependency is a form of integrity constraint that can identify **schema with redundant storage problems and to suggest refinement**.
- A functional dependency A→B in a relation holds true if two tuples having the same value of attribute A also have the same value of attribute B
- **IF t1.X=t2.X then t1.Y=t2.Y where t1,t2 are tuples and X,Y are attributes.**

Consider relation obtained from Hourly_Emps:

Hourly_Emps (*ssn, name, lot, rating, hrly_wages, hrs_worked)*

Notation: We will denote this relation schema by listing the attributes: SNLRWH

Some FDs on Hourly_Emps:

ssn is the key:

S →SNLRWH

rating determines hrly_wages:

R →W

Problems due to R→ W :

- ❑ ***Update anomaly**: Can* we change W in just the 1st tuple of SNLRWH?
- ❑ ***Insertion anomaly**: What if we* want to insert an employee and don't know the hourly wage for his rating?
- ❑ ***Deletion anomaly**: If we delete* all employees with rating 5,we lose the information about the wage for rating 5!

S	N	L	R	W	H
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

S	N	L	R	H
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

R	W
8	10
5	7

3) Reasoning about FDs

- Given some FDs, we can usually infer additional FDs:

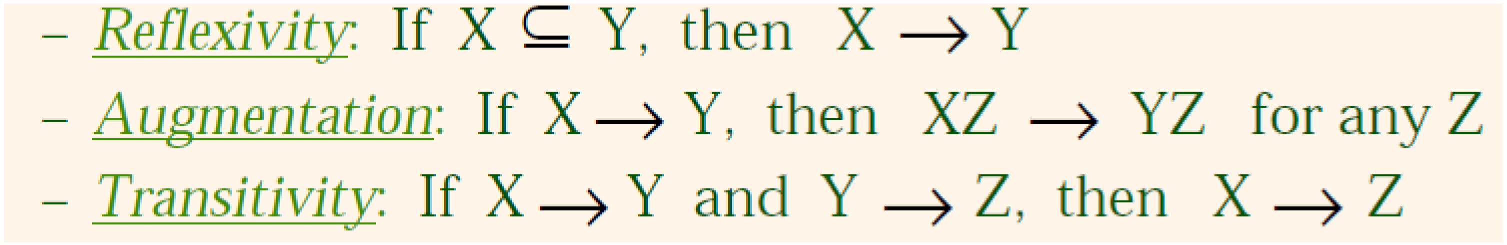
  ssn → did, did → lot implies ssn → lot

  F⁺= closure of F is the set of all FDs that are implied by F.

- Armstrong axioms defines the set of rules to infer all the functional dependencies on a relational database.

- **Various axioms rules or inference rules:**

Primary axioms:



- *Reflexivity*: If $X \subseteq Y$, then $X \rightarrow Y$
- *Augmentation*: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
- *Transitivity*: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

- These are *sound and complete inference rules for FDs!*

R1(SSN,DID,LOT)

F = SSN →DID
DID →LOT

F⁺ = SSN →LOT [TRANSITIVITY]
SSN,LOT →DID,LOT [AUGMENTATION]

secondary or derived axioms:

- *Union:* If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
- *Decomposition:* If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

Consider the relation called **Contracts(*cid,sid,jid,did,pid,qty,value*)** *.let us denote* ***cid, sid, jid, did, pid, qty, value*** *as* ***c, s, j, d, p, q, v*** ***and its FD's are***

$$F = \left\{ \begin{array}{l} C \rightarrow CSJDPQV \\ JP \rightarrow C \\ SD \rightarrow P \end{array} \right\}$$

Then closure of F i.e **F^+ is**

F^+ = JP → C, C → CSJDPQV implies JP → CSJDPQV [From Transitivity]

SD → P implies SDJ → JP [From Augmentation]

SDJ → JP, JP → CSJDPQV implies SDJ → CSJDPQV [From Transtivity]

So for set FD's in F

$$F = \left\{ \begin{array}{l} C \rightarrow CSJDPQV \\ JP \rightarrow C \\ SD \rightarrow P \end{array} \right\}$$

The closure of F i.e F^+ is

$$= \left\{ \begin{array}{l} JP \rightarrow CSJDPQV \\ SDJ \rightarrow JP \\ SDJ \rightarrow CSJDPQV \end{array} \right\}$$

- Computing the closure of a set of FDs can be expensive. (Size of closure is exponential in # attrs!)
- Typically, we just want to check if a given FD *X* →*Y is* in the closure of a set of FDs *F.*
- *An efficient check is*
  - Compute *attribute closure of X (denoted X^+) wrt F*
  - Check if Y is in X^+

Does F = $\{A \longrightarrow B,\ B \longrightarrow C,\ C\,D \longrightarrow E\}$ imply $A \longrightarrow E$?
– i.e, is $A \longrightarrow E$ in the closure F^+? Equivalently, is E in A^+ ?

Algorithm to compute Attribute Closure of X

```
closure = $X$;
repeat until there is no change: {
        if there is an FD $U \rightarrow V$ in $F$ such that $U \subseteq closure$,
            then set $closure = closure \cup V$.
    }
```

Does $F = \{A \rightarrow B,\ B \rightarrow C,\ C\,D \rightarrow E\}$ imply $A \rightarrow E$?

– i.e, is $A \rightarrow E$ in the closure F^+? Equivalently, is E in A^+ ?

To Check if $A \rightarrow E$ exists in F^+, it is enough to calculate A^+ and check if E exits in A^+

To Compute A^+

A^+	=	$\{A\}$	
		$\{A,B\}$	[Using $A \rightarrow B$]
		$\{A,B,C\}$	[Using $B \rightarrow C$]

Consider a relation R (A , B , C , D , E , F , G) with the functional dependencies-

$A \rightarrow BC$

$BC \rightarrow DE$

$D \rightarrow F$

$CF \rightarrow G$

Now, let us find the closure of some attributes and attribute sets-

Closure of attribute A-

$$
\begin{aligned}
A^+ &= \{ A \} \\
&= \{ A , B , C \} \text{ (Using } A \rightarrow BC \text{)} \\
&= \{ A , B , C , D , E \} \text{ (Using } BC \rightarrow DE \text{)} \\
&= \{ A , B , C , D , E , F \} \text{ (Using } D \rightarrow F \text{)} \\
&= \{ A , B , C , D , E , F , G \} \text{ (Using } CF \rightarrow G \text{)}
\end{aligned}
$$

Thus,

$\mathbf{A^+ = \{ A , B , C , D , E , F , G \}}$

<u>Closure of attribute D-</u>

D^+ = { D }
= { D , F } (Using $D \rightarrow F$)
We can not determine any other attribute using attributes D and F contained in the result set.

Thus,

D^+ = { D , F }

<u>Closure of attribute set {B, C}-</u>

$\{ B , C \}^+$ = { B , C }
= { B , C , D , E } (Using $BC \rightarrow DE$)
= { B , C , D , E , F } (Using $D \rightarrow F$)
= { B , C , D , E , F , G } (Using $CF \rightarrow G$)

Thus,

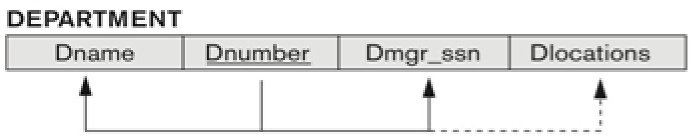
$\{ B , C \}^+$ = { B , C , D , E , F , G }

4.Normal Forms

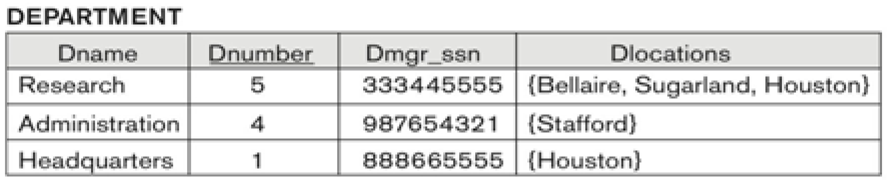
- Normalization is the process of organizing the data in the database.
- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.
- Normalization divides the larger table into the smaller table and links them using relationship.
- Normalization rules are divided into the following normal forms:
  1) First Normal Form
  2) Second Normal Form
  3) Third Normal Form
  4) BCNF
  5) Fourth Normal Form

1) FIRST NORMAL FORM

- ❖ A relation will be 1NF if it contains an atomic value.
- ❖ A Relation will be in 1NF if it does not contain **Multi-valued Attributes**
- ❖ A relation is said to be in 1 NF if the intersection of row and column of a relation contains single value(atomic value)



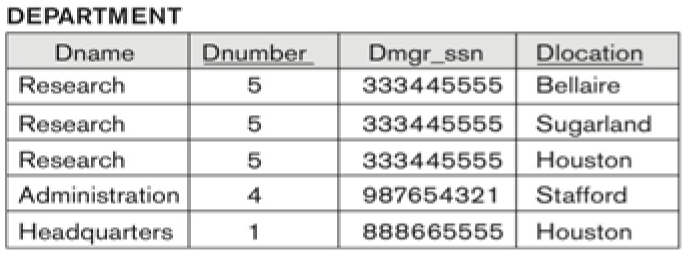
Dnumber → Dname, Dmgr_ssn, Dlocations



DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

→ Relation is not in 1NF



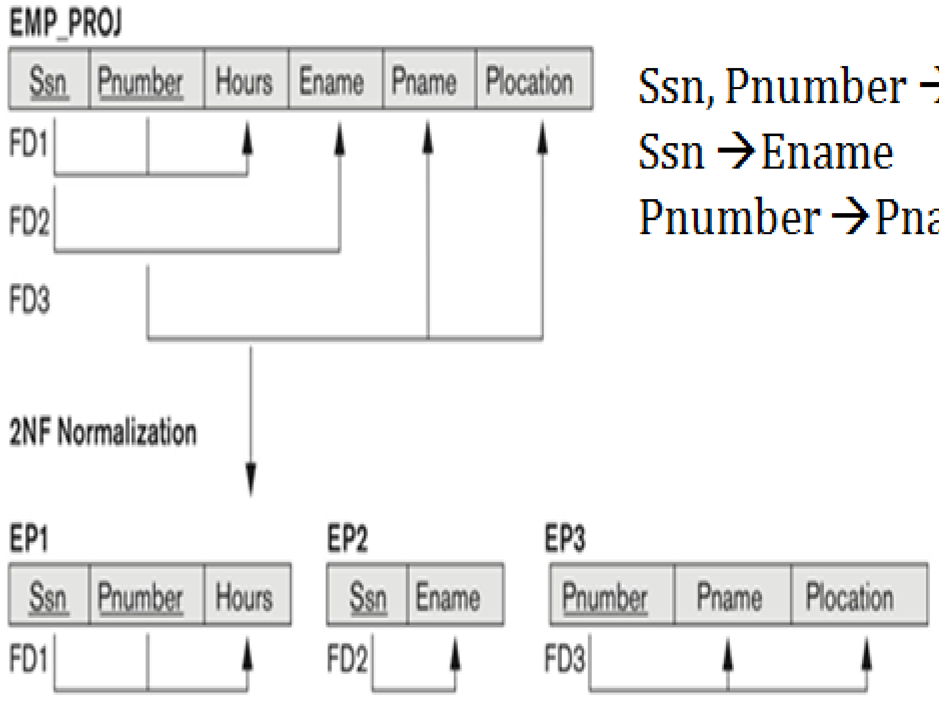
DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

→ Relation is in 1NF

2) SECOND NORMAL FORM

- For a table to be in the Second Normal Form,
  - ✓ It should be in the **First Normal form.**
  - ✓ And, it should not have **Partial Funtional Dependency.**
    - **If a Non prime Attribute depends on a Partial Key , Then such FD is called Partial Functional Dependency . For Eg, Ename is a Non prime attribute and it depends on part of key i.e ssn so , it is a Partial FD**



Ssn, Pnumber → hours → Full Funtional Dependency

Ssn → Ename → Partial Funtional Dependency

Pnumber → Pname,Plocation → Partial Funtional Dependency

Because of Partial FD's, the relation is not in 2NF

The three decomposed relations are in 2 NF

3) THIRD NORMAL FORM

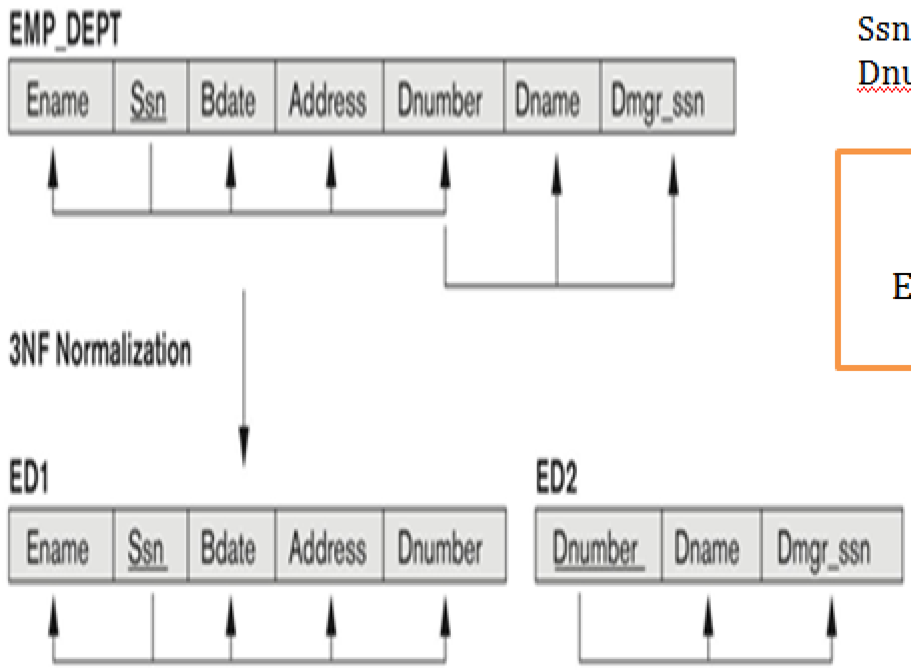
- ❖ For a table to be in the Third Normal Form,
  - ✓ It should be in the **Second Normal form.**
  - ✓ And, it should not have **Transitive Dependency**
    - ❖ **Dependency between two Non prime Attributes is called Transitive Dependency i.e X→Y is a Transtive Dependency if both X and Y are Non Prime Attributes**

(or)

- ❖ For a table to be in the Third Normal Form,
  - ✓ It should not contain **Multi-Valued Attributes**
  - ✓ it should not have **Partial Functional Dependency**
  - ✓ And,it should not have **Transitive Dependency**

(or)

- ❖ A table is in 3NF if it is in 2NF and for each functional dependency X-> Y
  - ✓ Either X should be **super key** of table or
  - ✓ Y should be a prime attribute of table



Ssn → Ename, Bdate, Address, Dnumder
Dnumber → Dname, Dmgr_ssn

Since there exists Transitive Dependence in the relation EMP_DEPT, the relation is not in 3NF

After decomposing the relation EMP_DEPT into ED1 and ED2. The ED1 and ED2 does not contain Transitive Dependency , so ED1 and ED2 are in 3 NF

4) BOYCE CODD NORMAL FORM

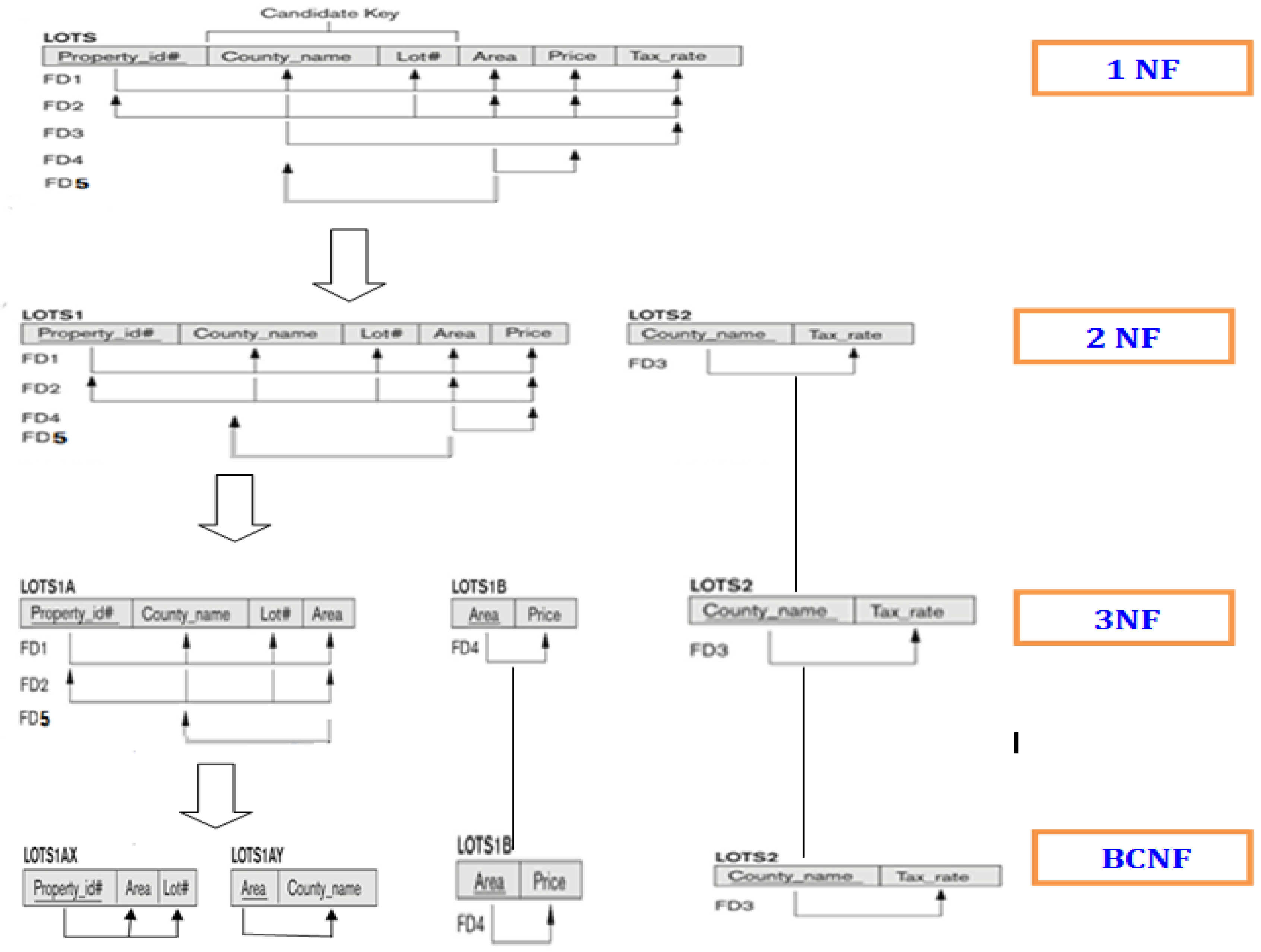
❖ For a table to be in the BCNF

- ✓ It should not contain **Multi-Valued Attributes**
- ✓ it should not have **Partial Functional Dependency**
- ✓ And,it should not have **Transitive Dependency** **and**
- ✓ For each FD X→ Y , **either X should be key attribute or Y may or may not be prime attribute**

(or)

❖ A table is in BCNF if it is in 2NF and for each functional dependency X-> Y

- ✓ X should be **super key** of table and
- ✓ Y may or may not be a prime attribute of table



Candidate Key
LOTS
Property_id#
County_name
Lot#
Area
Price
Tax_rate
FD1
FD2
FD3
FD4
FD5
1 NF
LOTS1
LOTS2
2 NF
LOTS1A
LOTS1B
3NF
LOTS1AX
LOTS1AY
BCNF

5) Properties of Decomposition

- ❖ When a relation in the relational model is not in appropriate normal form then the **decomposition** of a relation is required.
- ❖ In a database, it breaks the table into multiple tables.
- ❖ If the relation has no proper decomposition, then it may lead to problems like loss of information.
- ❖ Two Important Properties of Decomposition are

  1) Loss Less Decomposition

  2) Dependency Preserving Decomposition

1. Lossless decomposition-

Lossless decomposition ensures-

- ❖ No information is lost from the original relation during decomposition.
- ❖ When the sub relations are joined back, the same relation is obtained that was decomposed.
- ❖ Every decomposition must always be lossless.

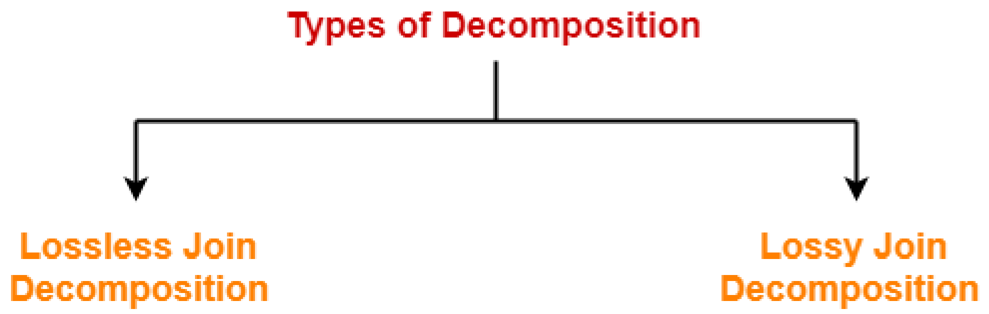
2. Dependency Preservation-

Dependency preservation ensures-

- ❖ None of the functional dependencies that holds on the original relation are lost.
- ❖ The sub relations still hold or satisfy the functional dependencies of the original relation.

Types of Decomposition-

Decomposition of a relation can be completed in the following two ways-



1. Lossless Join Decomposition-

- ❖ Consider there is a relation R which is decomposed into sub relations R_1 , R_2 , ... , R_n.
- ❖ This decomposition is called lossless join decomposition when the join of the sub relations results in the same relation R that was decomposed.
- ❖ For lossless join decomposition, we always have-

  $\mathbf{R_1 \bowtie R_2 \bowtie R_3 \ldots\ldots \bowtie R_n = R}$ where $\bowtie$ is a natural join operator

Lossy Join Decomposition-

- Consider there is a relation R which is decomposed into sub relations R_1 , R_2 , …. , $R_{n.}$
- This decomposition is called lossy join decomposition when the join of the sub relations does not result in the same relation R that was decomposed.
- The natural join of the sub relations is always found to have some extraneous tuples.
- For lossy join decomposition, we always have-

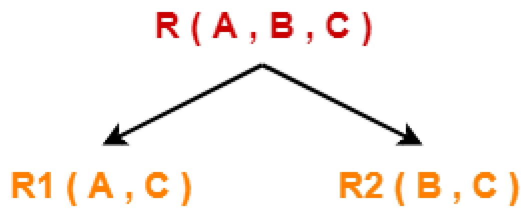
$\mathbf{R_1 \bowtie R_2 \bowtie R_3 \ldots\ldots\ldots \bowtie R_n \supset R}$ where $\bowtie$ is a natural join operator

Consider the following relation R(A , B , C)-

A	B	C
1	2	1
2	5	3
3	3	3

R(A , B , C)

Consider this relation is decomposed into two sub relations as R_1(A , C) and R_2(B , C)-



The two sub relations are-

A	C
1	1
2	3
3	3

R_1(A , B)

B	C
2	1
5	3
3	3

R_2(B , C)

❖ Now, let us check whether this decomposition is **<u>lossy or not</u>**. For lossy decomposition, we must have-

$$\mathbf{R_1 \bowtie R_2 = R}$$

❖ Now, if we perform the natural join (⋈) of the sub relations R_1 and R_2 we get-

A	B	C
1	2	1
2	5	3
3	3	3

❖ **This relation is same as the original relation R.**

❖ **Thus, we conclude that the above decomposition is lossless join decomposition**

EMP_DEPT

EMP_ID	EMP_NAME	EMP_AGE	EMP_CITY	DEPT_ID	DEPT_NAME
22	Denim	28	Mumbai	827	Sales
33	Alina	25	Delhi	438	Marketing
46	Stephan	30	Bangalore	869	Finance
52	Katherine	36	Mumbai	575	Production
60	Jack	40	Noida	678	Testing

EMP

EMP_ID	EMP_NAME	EMP_AGE	EMP_CITY
22	Denim	28	Mumbai
33	Alina	25	Delhi
46	Stephan	30	Bangalore
52	Katherine	36	Mumbai
60	Jack	40	Noida

DEPT

DEPT_ID	EMP_ID	DEPT_NAME
827	22	Sales
438	33	Marketing
869	46	Finance
575	52	Production
678	60	Testing

EMP $\bowtie$ **DEPT**

EMP_ID	EMP_NAME	EMP_AGE	EMP_CITY	DEPT_ID	DEPT_NAME
22	Denim	28	Mumbai	827	Sales
33	Alina	25	Delhi	438	Marketing
46	Stephan	30	Bangalore	869	Finance
52	Katherine	36	Mumbai	575	Production
60	Jack	40	Noida	678	Testing

- Let's take 'E' is the Relational Schema, With instance 'e'; is decomposed into: E1, E2, E3, ... En; With instance: e1, e2, e3, ... en, If e1 ⋈ e2 ⋈ e3 . . . . ⋈ en, then it is called as **'Lossless Join Decomposition'.**
- **In the above example EMP_DEPT is decomposed into EMP and DEPT .we can say that decomposition is Loss Less Decomposition because the natural join of EMP and DEPT is equal to the original relation EMP_DEPT.**
- **The Decomposition is Lossy if**

$$\boxed{\text{EMP} \bowtie \text{DEPT} \ \supset \ \text{EMP\_DEPT}}$$

Natural Join of EMP and DEPT should not be superset of EMP_DEPT

- **The Decomposition is Loss Less if**

$$\boxed{\text{EMP} \bowtie \text{DEPT} \ = \ \text{EMP\_DEPT}}$$

Natural Join of EMP and DEPT is equal to EMP_DEPT

- ❖ If we decompose a relation R into relations R1 and R2,

  1) **Decomposition is lossy if R1 ⋈ R2 ⊃ R**

  2) **Decomposition is lossless if R1 ⋈ R2 = R**

- ❖ **To check for lossless join decomposition using FD set, following conditions must hold:**

  1) **Union of Attributes of R1 and R2 must be equal to attribute of R. Each attribute of R must be either in R1 or in R2.**

     Att(R1) U Att(R2) = Att(R)

  1) **Intersection of Attributes of R1 and R2 must not be NULL.**

     Att(R1) ∩ Att(R2) ≠ Φ

  2) **Common attribute must be a key for at least one relation (R1 or R2)**

     Att(R1) ∩ Att(R2) -> Att(R1) or Att(R1) ∩ Att(R2) -> Att(R2)

Dependency Preserving

❖ A Decomposition D = { R1, R2, R3….Rn } of R is dependency preserving wrt a set F of Functional dependency if

(F1 □ F2 □ … □ Fm)+ = F+.

❖ Consider a relation R

R ---> F{...with some functional dependency(FD)....}

R is decomposed or divided into R1 with FD { f1 } and R2 with { f2 },

then there can be three cases:

1) **f1 U f2 = F** -----> Decomposition is dependency preserving.
2) **f1 U f2** is a subset of F -----> Not Dependency preserving.
3) **f1 U f2** is a super set of F -----> This case is not possible.

- **Consider a schema R(A,B,C,D) and functional dependencies A->B and C->D. Then the decomposition of R into R1(AB) and R2(CD) is A->B can be ensured in R1(AB) and C->D can be ensured in R2(CD). Hence it is dependency preserving decomposition.**

Consider a Relation R(A,B,C,D,E) with FD's
F = {$A \rightarrow B$,$B \rightarrow C$,$C \rightarrow D$,$D \rightarrow A$}
If R(A,B,C,D,E) is decomposed into R1(ABC) and R2(CDE)
To Check the decomposition is Dependency preserving

R1(ABC)
Find A^+,B^+,C^+ w.r.t FD's of R
A^+ = ABCD = $A \rightarrow BC$ [A is trival and D doesn not exists in R1.so remove A and D]
B^+ = BCDA = $B \rightarrow CA$ [B is trival and D doesn not exists in R1.so remove A and D]
C^+ = CDAB = $C \rightarrow AB$ [C is trival and D doesn not exists in R1.so remove A and D]

So, F1 = {$A \rightarrow BC$,$B \rightarrow CA$,$C \rightarrow AB$}

R2 (CDE)
Find C^+,D^+,E^+ w.r.t FD's of R
C^+ = CDAB = $C \rightarrow D$ [C is trival and A,B doesn not exists in R2.so remove A and B]
D^+ = DABC = $D \rightarrow C$ [D is trival and A,B doesn not exists in R2.so remove A and B]
E^+ = E = NO FD

So, F2 = {$C \rightarrow D$,$D \rightarrow C$}

$$\begin{tikzcd}[column sep=tiny]
 & \begin{array}{c}\text{R1(A,B,C,D,E)}\\ F=\{A\rightarrow B, B\rightarrow C, C\rightarrow D, D\rightarrow A\}\end{array} \arrow[dl, dash] \arrow[dr, dash] & \\
\begin{array}{c}\text{R1(A,B,C)}\\ F1=\{A\rightarrow BC, B\rightarrow CA, C\rightarrow AB\}\end{array} & & \begin{array}{c}\text{R2(C,D,E)}\\ F2=\{C\rightarrow D, D\rightarrow C\}\end{array}
\end{tikzcd}$$

To check the decomposition is Dependency Preserving, it is necessary to to check

F = F1 U F2

{A→B,B→C,C→D,D→A} = {A→BC,B→CA,C→AB ,C→D,D→A}

i.e we have to check whether al FD's in F exists in F1 U F2

A→ B exists in F1 U F2 because A→BC means A→B , A→C

B→C exists in F1 U F2 because B→CA means B→C,B→A

C→D exists in F1 U F2

D→A does not exists in F1 U F2 directly but we cant conclude it by seeing so we have to compute D^+ w.r.t F1 U F2

D^+ = DABC. Since D^+ contains A , D→A also exists in F1 U F2

So , all the FD's in F exists in F1 U F2 , the decomposition is a dependency preserving decomposition.

7) Transactions

- ❖ Collection of several operations on the Database appears to be single unit from the point of view of the db user.
- ❖ For example, a transfer of funds from a checking A/C to savings A/C is a single operation from the customer's standpoint, within the database system, however, it consists of several operations.

  For example : Fund Transfer Transaction

  R (CA2090)

  Ca2090 := CA2090 – 10000

  W(CA2090)

  R(SB2091)

  SB2091 := SB2091 + 10000

  W (SB2091)

- ❖ **So, Collection of operations that form a single Logical unit of work are called transaction.**

ACID Properties

To preserve integrity of data, the database system must ensure:

- ❖ **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- ❖ **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- ❖ **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

  That is, for every pair of transactions *Ti and Tj, it appears to Ti* that either *Tj, finished execution before Ti started, or Tj started* execution after *Ti finished.*
- ❖ **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Example of Fund Transfer

Transaction to transfer $50 from account *A to account B:*

1. **read(*A)***
2. *A := A – 50*
3. **write(*A)***
4. **read(*B)***
5. *B := B + 50*
6. **write(*B)***

- ❖ Consistency requirement – the sum of *A and B is unchanged* by the execution of the transaction.
- ❖ Atomicity requirement — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

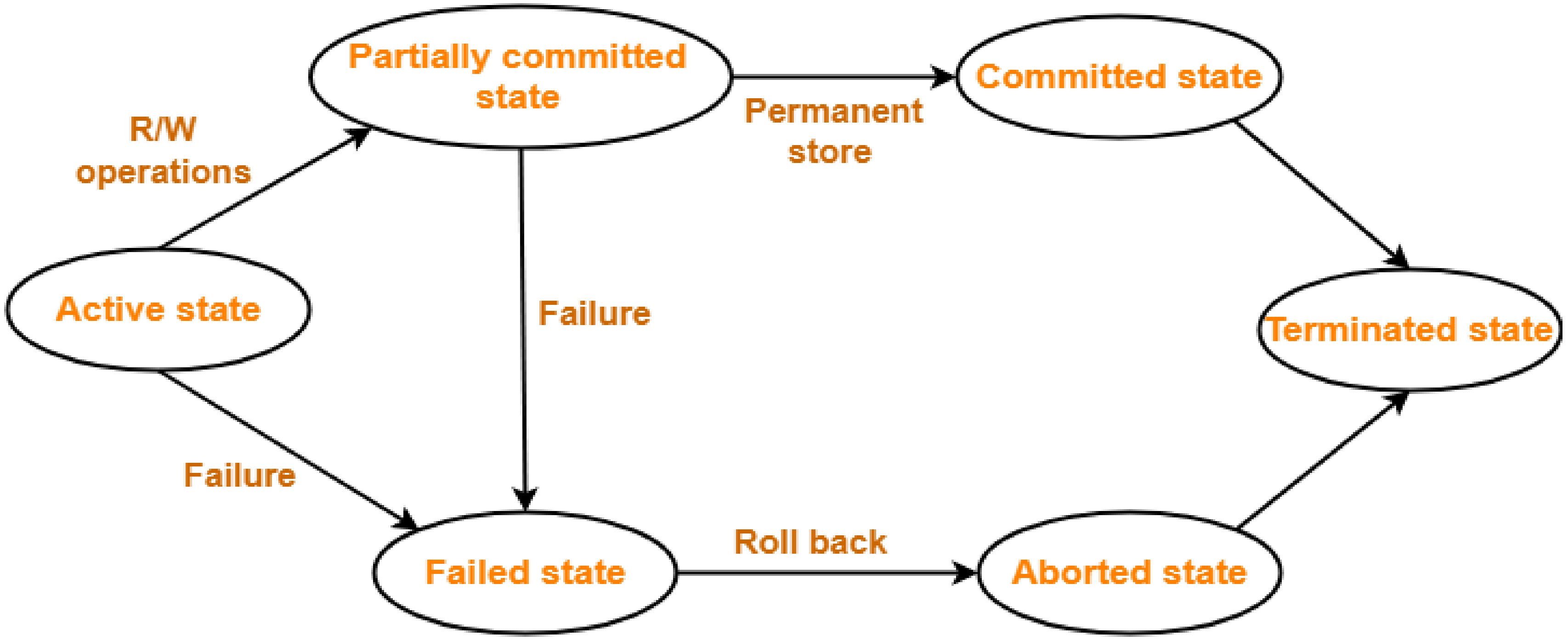
- Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.
- Isolation requirement — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database

  (the sum *A + B will be less than it should be).* Can be ensured trivially by running transactions ***serially,*** that is one after the other. However, executing multiple transactions concurrently has significant benefits, as we

  will see.

8) Transaction States

- A transaction goes through many different states throughout its life cycle. These states are called as **transaction states**.
- Transaction states are as follows-
  1) Active state
  2) Partially committed state
  3) Committed state
  4) Failed state
  5) Aborted state
  6) Terminated state



Transaction States in DBMS

1. Active State-

- This is the first state in the life cycle of a transaction.
- A transaction is called in an **active state** as long as its instructions are getting executed.
- All the changes made by the transaction now are stored in the buffer in main memory.

2. Partially Committed State-

- After the last instruction of transaction has executed, it enters into a **partially committed state**.
- After entering this state, the transaction is considered to be partially committed.
- It is not considered fully committed because all the changes made by the transaction are still stored in the buffer in main memory.

3. Committed State-

- After all the changes made by the transaction have been successfully stored into the database, it enters into a **committed state**.
- Now, the transaction is considered to be fully committed.

NOTE-

- After a transaction has entered the committed state, it is not possible to roll back the transaction.
- In other words, it is not possible to undo the changes that has been made by the transaction.
- This is because the system is updated into a new consistent state.
- The only way to undo the changes is by carrying out another transaction called as **compensating transaction** that performs the reverse operations.

4. Failed State-

- ❖ When a transaction is getting executed in the active state or partially committed state and some failure occurs due to which it becomes impossible to continue the execution, it enters into a **failed state**.

5. Aborted State

- ❖ After the transaction has failed and entered into a failed state, all the changes made by it have to be undone.
- ❖ To undo the changes made by the transaction, it becomes necessary to roll back the transaction.
- ❖ After the transaction has rolled back completely, it enters into an **aborted state**.

4. Failed State-

- When a transaction is getting executed in the active state or partially committed state and some failure occurs due to which it becomes impossible to continue the execution, it enters into a **failed state**.

5. Aborted State

- After the transaction has failed and entered into a failed state, all the changes made by it have to be undone.
- To undo the changes made by the transaction, it becomes necessary to roll back the transaction.
- After the transaction has rolled back completely, it enters into an **aborted state**.

6. Terminated State-

- This is the last state in the life cycle of a transaction.
- After entering the committed state or aborted state, the transaction finally enters into a **terminated state** where its life cycle finally comes to an end.

9) Concurrent Execution

- ❖ Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - ✓ **Increased processor and disk utilization**, leading to better transaction *throughput*

    E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - ✓ **Reduced average response time** for transactions: short transactions need not wait behind long ones.

Schedules

- ***A schedule thus consists of sequence of operations from a group of transactions subject to the condition that the order of operations for each individual transaction is preserved (or)***
- ***A schedule is the order in which the operations of multiple transactions appear for execution.***

Let T1 and T2 be two transactions

Transaction T1 transfers $50 from account A to account B. It is defined as

```
T1      :      Read (A);
               A: = A – 50;
               Write (A);
               Read (B);
               B: = B + 50;
               Write (B);
```

Transaction T2 transfers 10 percent of the balance from account A to account B . it is defined as

```
T2      :      Read (A);
               Temp: =  A * 0.1;
               A := A – temp;
               Write(A);
               Read(B);
               B := B + temp;
               Write(B);
```

Suppose these two transactions are submitted to the system at the same time, it is the responsibility of the ***Concurrency Control Module*** of DBMS software to schedule these transactions.

The Concurrency Control Module can schedule transactions in two ways

Serial

Non-Serial

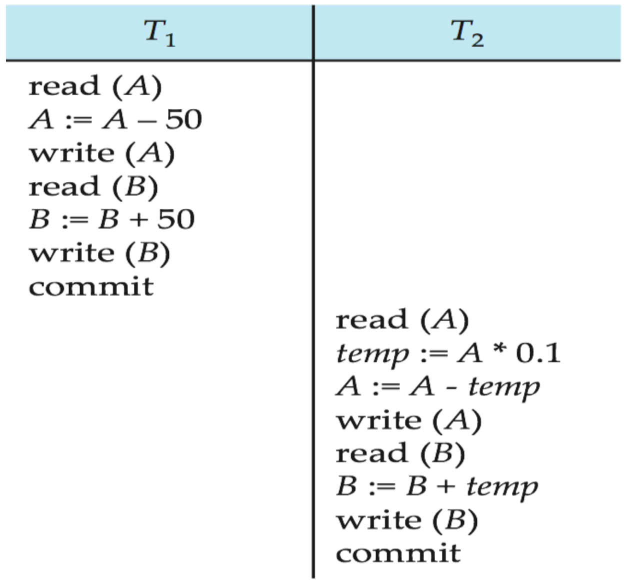
Serial schedule : A Schedule where the operations of each transactions are executed consecutively without any other interference from other transactions is called a Serial Schedule.

Non Serial schedule : A Schedule where the operations from a group of concurrent transactions are interleaved.

SERIAL SCHEDULE :

- In serial schedules,
  - All the transactions execute serially one after the other.
  - When one transaction executes, no other transaction is allowed to execute

Example of Serial Schedule



T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

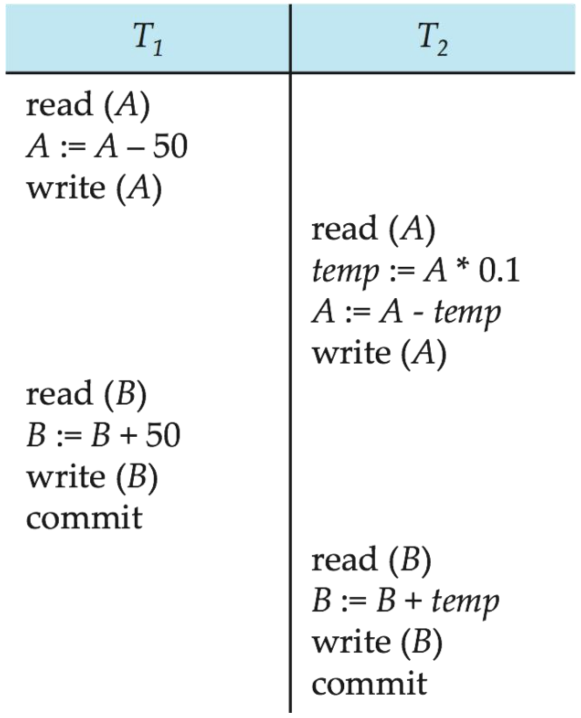
Schedule 1 – a Serial Schedule in which T1 is followed by T2

- In this schedule,
  - ✓ There are two transactions T1 and T2 executing serially one after the other.
  - ✓ Transaction T1 executes first.
  - ✓ After T1 completes its execution, transaction T2 executes.
  - ✓ So, this schedule is an example of a **Serial Schedule**.

NON SERIAL SCHEDULE

A Schedule where the operations from a group of concurrent transactions are executed in an ***<u>interleaving Fashion</u>***.

Example of Non Serial schedule

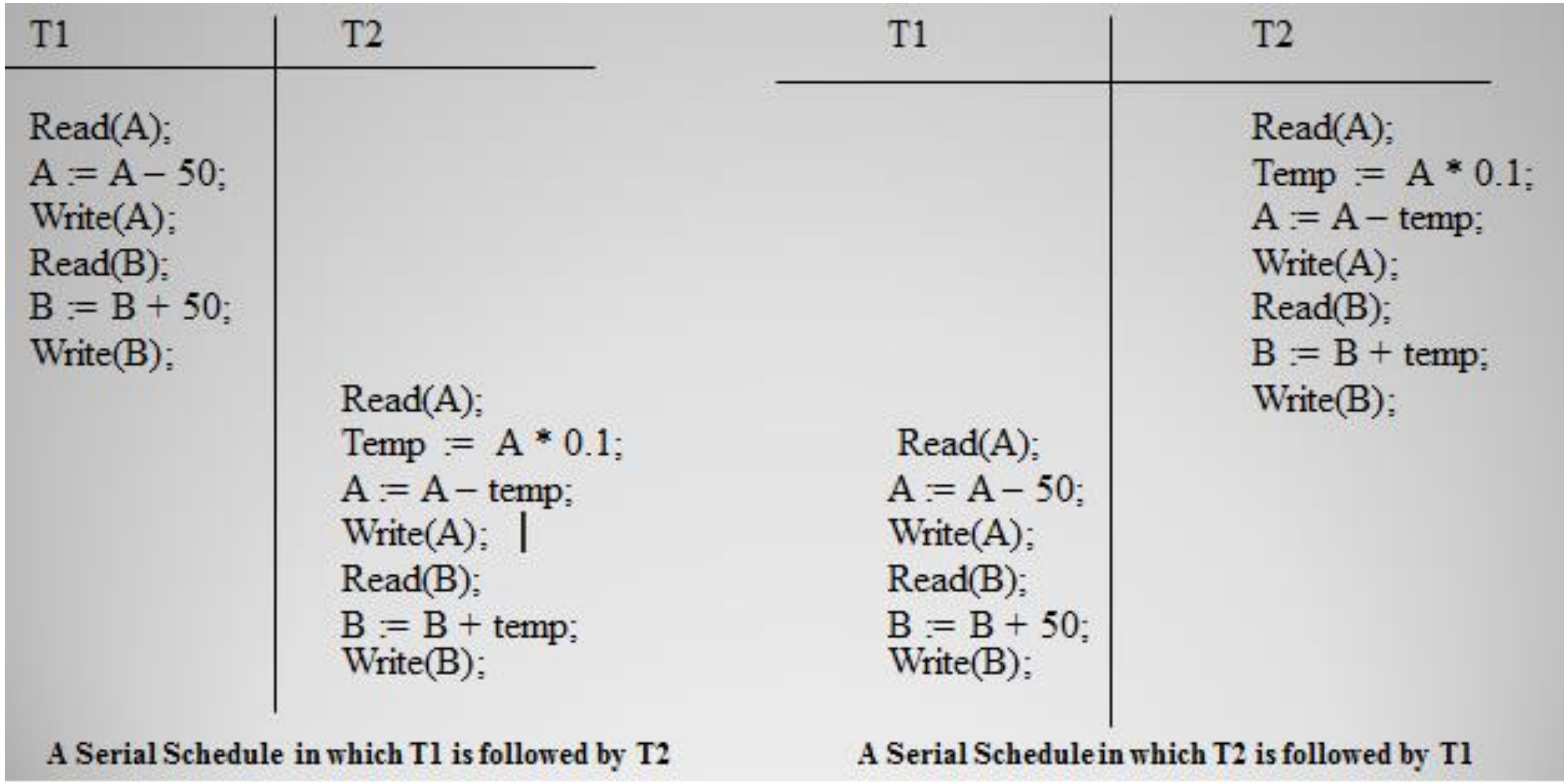


T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 2 – a Non Serial Schedule equivalent to Serial Schedule

Suppose two users submit T1 and T2 at same time, if no interleaving of operations is permitted, there are only 2 possible outcomes i.e.

- T1 after T2
- T2 after T1



T1	T2
Read(A);	
A := A – 50;	
Write(A);	
Read(B);	
B := B + 50;	
Write(B);	
	Read(A);
	Temp := A * 0.1;
	A := A – temp;
	Write(A);
	Read(B);
	B := B + temp;
	Write(B);

A Serial Schedule in which T1 is followed by T2

T1	T2
	Read(A);
	Temp := A * 0.1;
	A := A – temp;
	Write(A);
	Read(B);
	B := B + temp;
	Write(B);
Read(A);	
A := A – 50;	
Write(A);	
Read(B);	
B := B + 50;	
Write(B);	

A Serial Schedule in which T2 is followed by T1

So, for a set of n transactions, there exists n! Different valid serial Schedules.

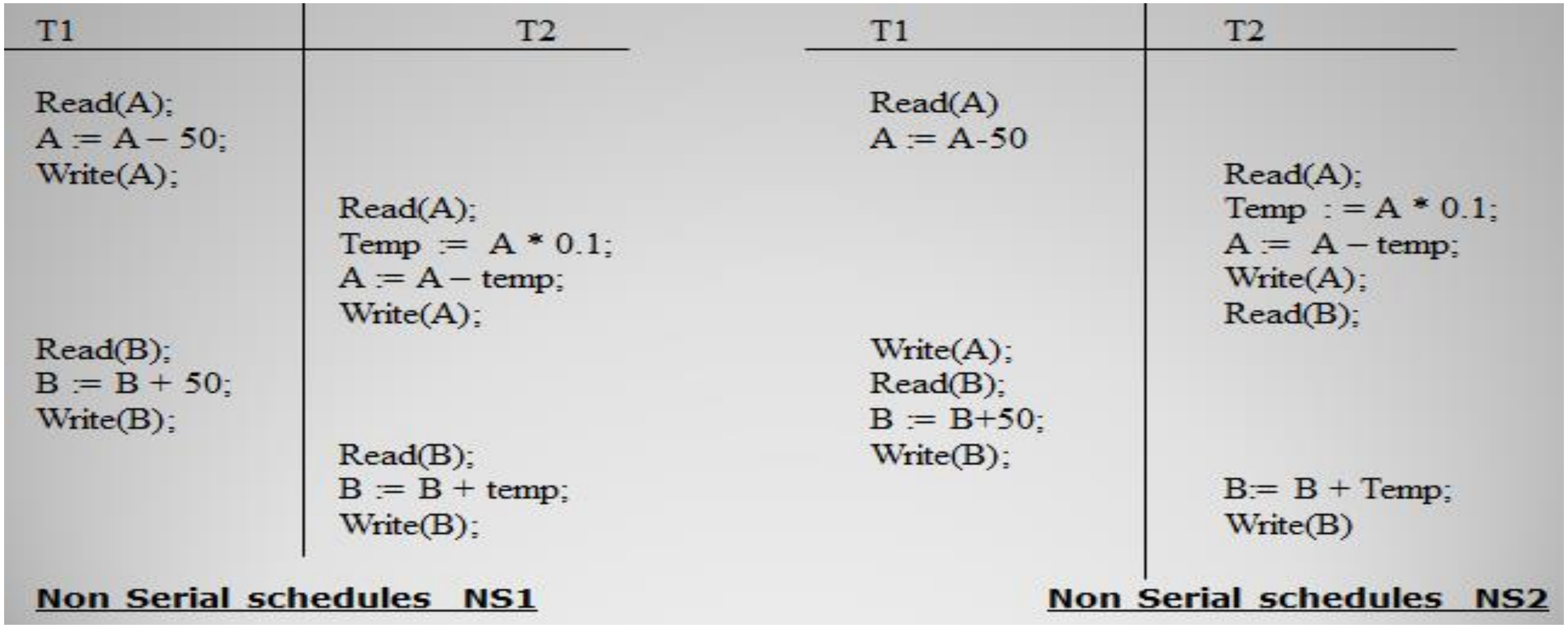
The Problem with Serial schedules is that **they limit concurrency or interleaving of operations.**

In a serial schedule, if a Transaction wait for I/O operation to complete, we can't switch the CPU processor to another transaction, thus wasting valuable CPU processing time.

In addition, if some transaction 'T' is quite Long, the other transaction must wait for T to complete all its operations before commencing i.e. (Short transactions will get stuck behind Long transactions.)

Hence, serial schedules are generally considered unacceptable in practice. So, we go for Non-serial Schedules

If the above transactions are submitted to the system, and if interleaving of operations are permitted then there will be many possible outcomes, two of them are



T1	T2
Read(A);	
A := A – 50;	
Write(A);	
	Read(A);
	Temp := A * 0.1;
	A := A – temp;
	Write(A);
Read(B);	
B := B + 50;	
Write(B);	
	Read(B);
	B := B + temp;
	Write(B);

Non Serial schedules NS1

T1	T2
Read(A)	
A := A-50	
	Read(A);
	Temp : = A * 0.1;
	A := A – temp;
	Write(A);
	Read(B);
Write(A);	
Read(B);	
B := B+50;	
Write(B);	
	B:= B + Temp;
	Write(B)

Non Serial schedules NS2

However, some non-serial schedule gives the correct expected result. We would like to determine which of the non-serial schedule always give a correct and which may give erroneous result.

10) Serializability

- A schedule is the order in which the operations of multiple transactions appear for execution.
  - Serial schedules are always consistent.
  - Non-serial schedules are not always consistent.
- Some non-serial schedules may lead to inconsistency of the database.
- Serializability is a concept that helps to identify which non-serial schedules are correct and will maintain the consistency of the database.

If a given non-serial schedule of ‘n’ transactions is equivalent to some serial schedule of ‘n’ transactions, then it is called as a serializable schedule.

(or)

A Non serial schedule which preserves the sequence of operations and produces the same result as that of a serial schedule is called Serializability

Two definitions of Schedule Equivalence are

1) Conflict Equivalence
2) View equivalence

1) <u>CONFLICT EQUIVALENCE</u>

Two operations in a schedule are said to be **conflict**, if they satisfy all three of the following conditions.

1. They belong to different Transactions
2. They access the same data item
3. At least one of the operation is a Write operation

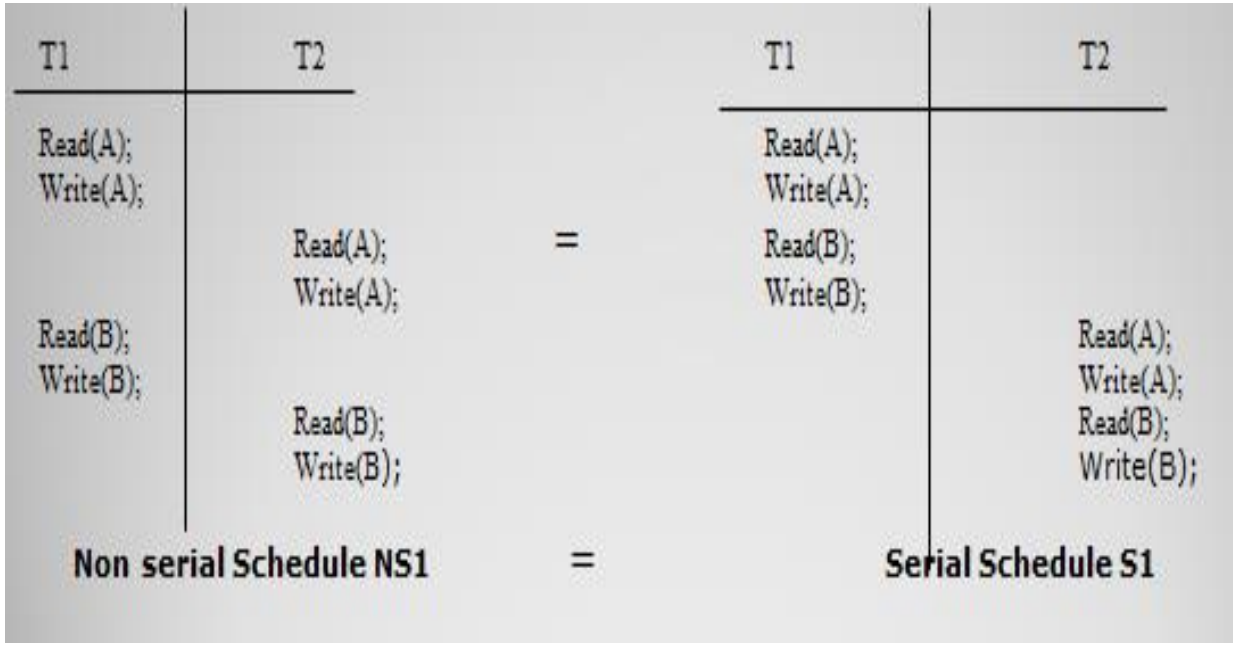
In short we can write above **Schedule NS1** in terms of only Read and write operations

Schedule NS1 = { R1(A) , W1(A), R2(A), W2(A), R1(B), W1(B), R2(B), W2(B)}

Conflict Operations of Schedule NS1	Some of the Non Conflicting Operations of schedule NS1 are
W2(A), R1(B)	R1(A) , W2(A)
R1(B) , R2(A)	W1(A), W2(A)
W1(B) ,W2(A)	R1(B), W2(B)
W1(B), R2(A)	W1(B), W2(B)

<u>If a Schedule 'NS' can be transformed into 'S' by a series of swaps of Non Conflicting operations , we say that NS and S are Conflict Equivalent.</u>

For Example if we consider only Read and Write Operations of a Non serial schedule NS1.



T1	T2
Read(A);	
Write(A);	
	Read(A);
	Write(A);
Read(B);	
Write(B);	
	Read(B);
	Write(B);

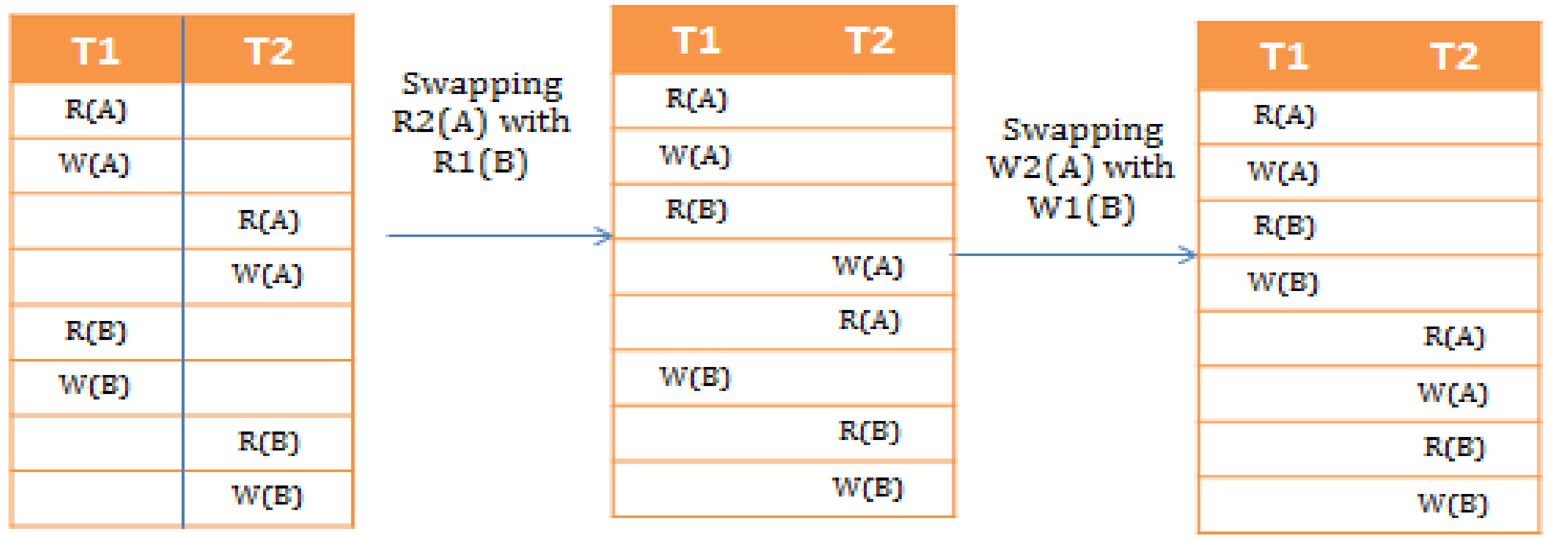
=

T1	T2
Read(A);	
Write(A);	
Read(B);	
Write(B);	
	Read(A);
	Write(A);
	Read(B);
	Write(B);

Non serial Schedule NS1 = Serial Schedule S1

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

Here **W1(A), R2(A)** and **W1(B),R2(B)** are conflicting Operations and



T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

NS1

T1	T2
R(A)	
W(A)	
R(B)	
	W(A)
	R(A)
W(B)	
	R(B)
	W(B)

T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

S1

- Thus , the Non serial schedule NS1 after swapping the non conflicting operations , it is equivalent to a serial schedule S1 .
- So, The non serial schedule NS1 is conflict equivalent to s1 .
- Hence we can say the non serial schedule NS1 is a Conflict Serializability

View Serializability

- ❖ View Serializability is a process to find out that a given Non serial **schedule** is **view serializable or not**.
- ❖ To check whether a given non serial schedule is view serializable, we need to check whether the **given schedule is View Equivalent to its serial schedule.**

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

Non Serial Schedule (NS1)

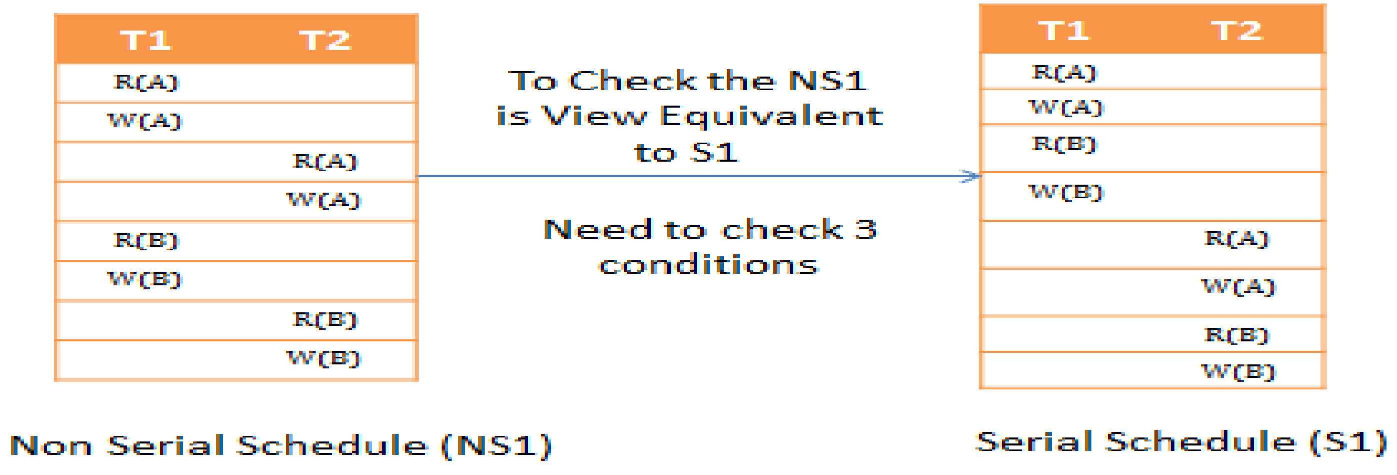
To Check the NS1 is View Equivalent to S1

Need to check 3 conditions

T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

Serial Schedule (S1)

1. **Initial Read:** Initial read of each data item in transactions must match in both schedules.
2. **Final Write:** Final write operations on each data item must match in both the schedules.
3. **Update Read:** If in schedule S1, the transaction T1 is reading a data item updated by T2 then in schedule S2, T1 should read the value after the write operation of T2 on same data item.



T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

Non Serial Schedule (NS1)

To Check the NS1 is View Equivalent to S1

Need to check 3 conditions

T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

Serial Schedule (S1)

Initial Read
In schedule NS1, transaction T1 first reads the data item A. In S1 also transaction T1 first reads the data item A.
In schedule NS1, transaction T1 first reads the data item B .In S1 also transaction T1 first reads the data item B
We checked for both data items A & B and the **initial read** condition is satisfied in NS1 & S1.

Final Write
In schedule NS1, the final write operation on A is done by transaction T2. In S1 also transaction T2 performs the final write on A.
In schedule NS1, the final write operation on B is done by transaction T2. In schedule S2, final write on B is done by T2.
We checked for both data items A & B and the **final write** condition is satisfied in NS1 & S1

Update Read
In NS1, transaction T2 reads the value of A, written by T1. In S1, the same transaction T2 reads the A after it is written by T1.
In NS1, transaction T2 reads the value of B, written by T1. In S1, the same transaction T2 reads the value of B after it is updated by T1.
The update read condition is also satisfied for both the schedules.

Since the three conditions are satisfied among the non serial schedule NS1 and the serial Schedule S1 , so we can say that NS1 is View Equivalent to S1 and NS! Is a View Serializable Schedule.

Testing for conflict Serializable (or) Precedence Graph

- We now present a **Simple and Efficient** method for determining **Conflict Serializabilty of schedule .**
- The **Precedence Graph** for a schedule **S** contains
  - A **Node** for each committed transaction in S.
  - An **Arc** from Ti to Tj **if one the three conditions holds**
    - Ti Executes W(Q) before Tj Executes R(Q)
    - Ti Executes R(Q) before Tj Executes W(Q)
    - Ti Executes W(Q) before Tj Executes W(Q)

A Schedule S is Conflict Serializable if and only if its Precedence Graph is Acyclic.

Lets us consider two Non serial schedules NS1 and Ns2 and draw precedence Graph for each Non serial schedules NS1 and NS2

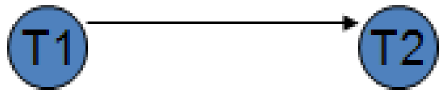
Non Serial schedules NS1

T1	T2
Read(A);	
A := A – 50;	
Write(A);	
	Read(A);
	Temp := A * 0.1;
	A := A – temp;
	Write(A);
Read(B);	
B := B + 50;	
Write(B);	
	Read(B);
	B := B + temp;
	Write(B);

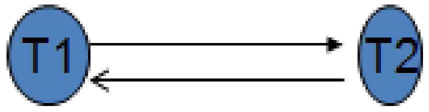
Non Serial schedules NS2

T1	T2
Read(A)	
A := A-50	
	Read(A);
	Temp : = A * 0.1;
	A := A – temp;
	Write(A);
	Read(B);
Write(A);	
Read(B);	
B := B+50;	
Write(B);	
	B:= B + Temp;
	Write(B)

Precedence Graph for NS1


$$\begin{tikzcd} T1 \arrow[r] & T2 \end{tikzcd}$$

Precedence Graph for NS2

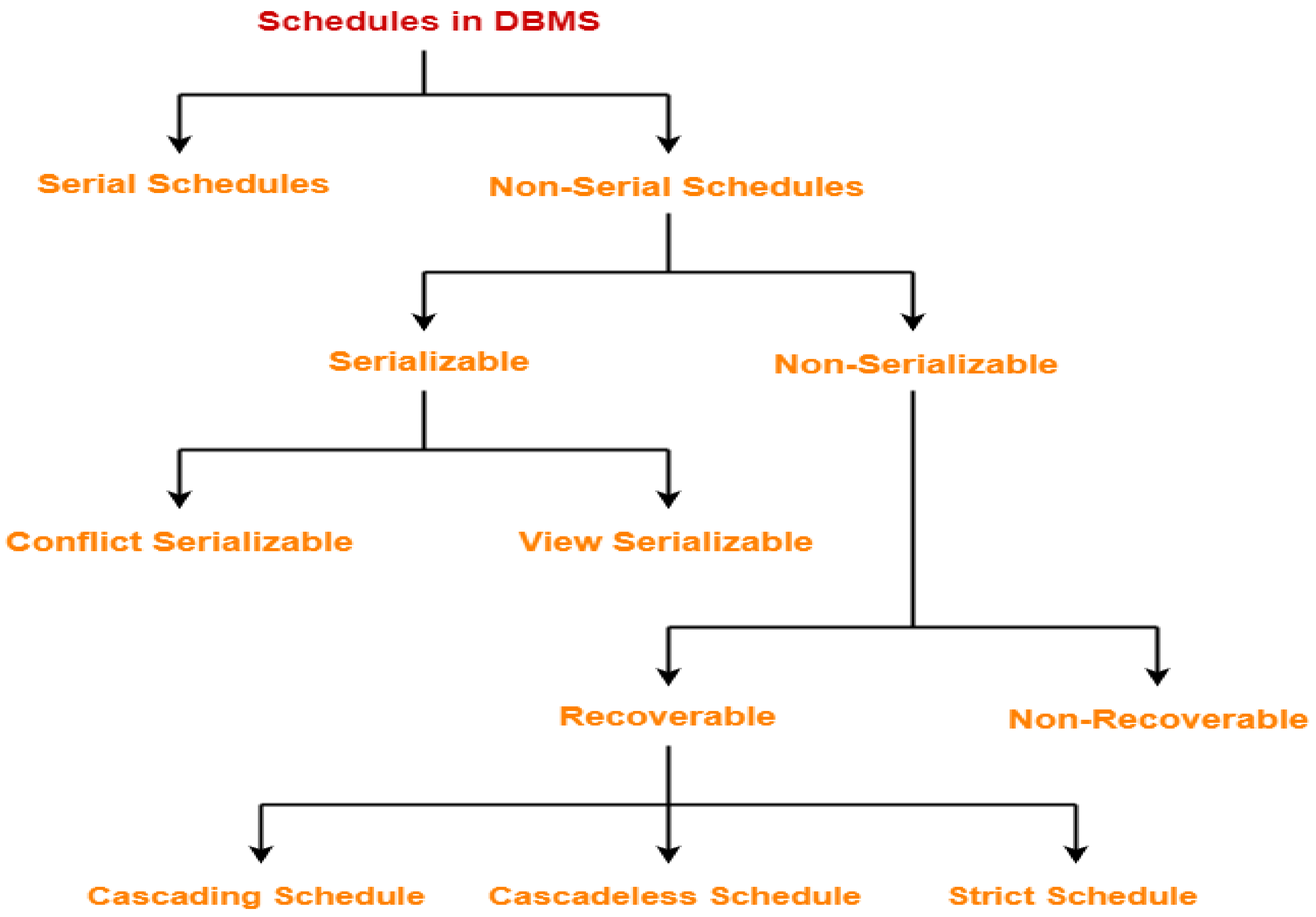

$$\begin{tikzcd} T1 \arrow[r, shift left] & T2 \arrow[l, shift left] \end{tikzcd}$$

➢The Precedence Graph for **Schedule NS1** contains the Edge from T1 to T2 because T1 Executes Read (A) before T2 Executes Write (A).

➢The Precedence Graph for Schedule NS2 Contains the edge from T1 to T2, and it also contains edge from T2 to T1 .

Since the Precedence Graph of Schedule NS1 is acyclic, We say NS1 is a Conflict Serializable Schedule.

Recoverability



- A **schedule** is the order in which the operations of multiple transactions appear for execution.
- A Schedule can be either **Serial** or **Non serial Schedule**
- **Non-serial schedules** may be **serializable or non-serializable**.
- A **non-serial schedule** which is not serializable is called as a **non-serializable schedule.**
- **Non-Serializable schedules** may be **recoverable or irrecoverable**
- The **Recoverable schedule** is one where, for each pair of transactions T1 and T2, for T2 to read a data item previously written by T1, the ***Commit operation of T1 should appear before the Commit operation of T2.***
- Consider a situation where the transaction T1 fails before it commits. Since T2 has read the value written by T1, we must abort T2 also to ensure atomicity . But this is not possible as transaction T2 has already been Committed
- Thus we have a situation where it is impossible to recover correctly from the failure of T1.The above situation is an example of a non-**recoverable schedule.**

Non recoverable Schedule

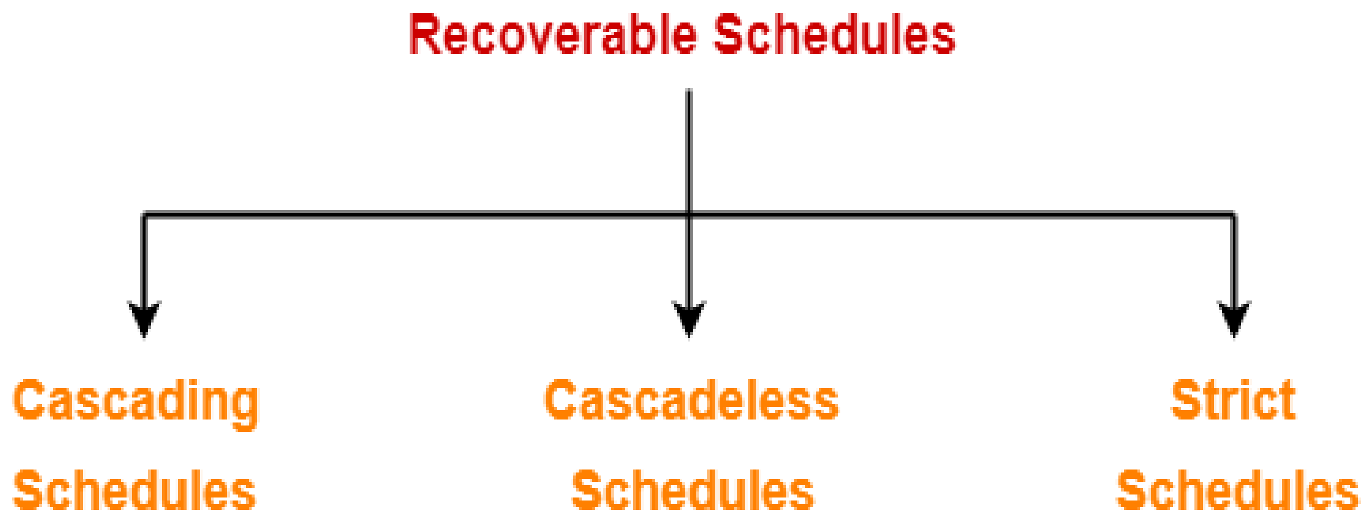
T1	T2
Read(CA2090)	
Write(CA2090)	
	Read(CA2090)
	Commit
abort	

Recoverable Schedule

T1	T2
Read(CA2090)	
Write(CA2090)	
commit	
	Read(CA2090)
	Commit

Types of Recoverable Schedules-

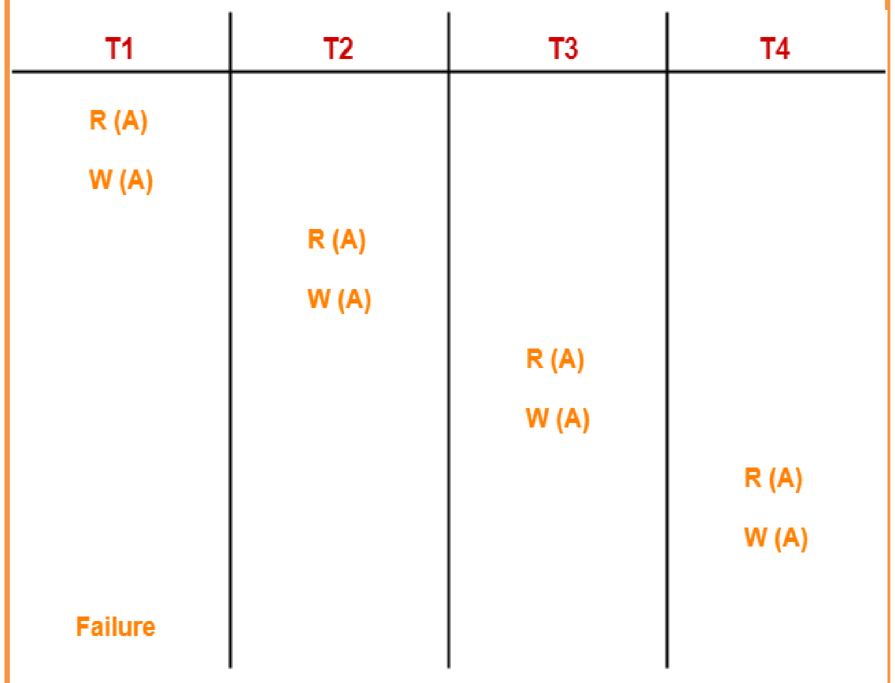
A recoverable schedule may be any one of these kinds-



1) Cascading Schedule
2) Cascadeless Schedule
3) Strict Schedule

1) Cascading Schedule-

❖ 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 Schedule** or **Cascading Rollback** or **Cascading Abort**.



T1	T2	T3	T4
R (A)			
W (A)			
	R (A)		
	W (A)		
		R (A)	
		W (A)	
			R (A)
			W (A)
Failure			

Cascading Recoverable Schedule

In this schedule,

The failure of transaction T1 causes the transaction T2 to rollback.

The rollback of transaction T2 causes the transaction T3 to rollback.

The rollback of transaction T3 causes the transaction T4 to rollback.

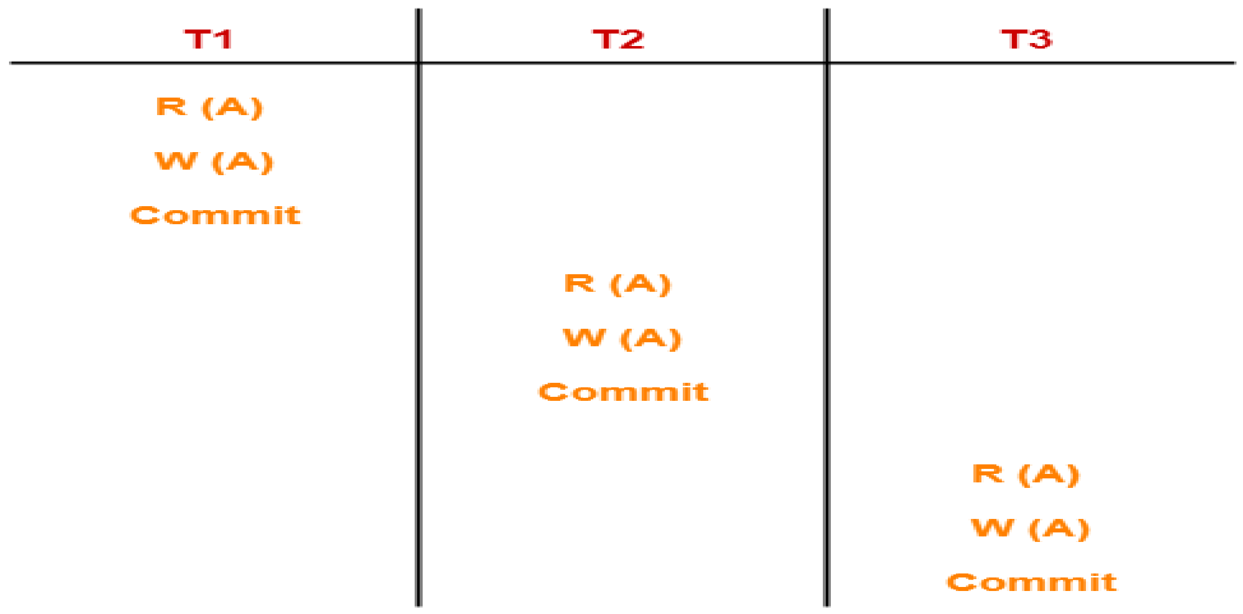
Such a rollback is called as a **Cascading Rollback**.

2) Cascadeless Schedule-

If in a schedule, a transaction is not allowed to read a data item until the last transaction that has written it is committed or aborted, then such a schedule is called as a **Cascadeless Schedule**.

In other words,

Cascadeless schedule allows only committed read operations



T1	T2	T3
R (A)		
W (A)		
Commit		
	R (A)	
	W (A)	
	Commit	
		R (A)
		W (A)
		Commit

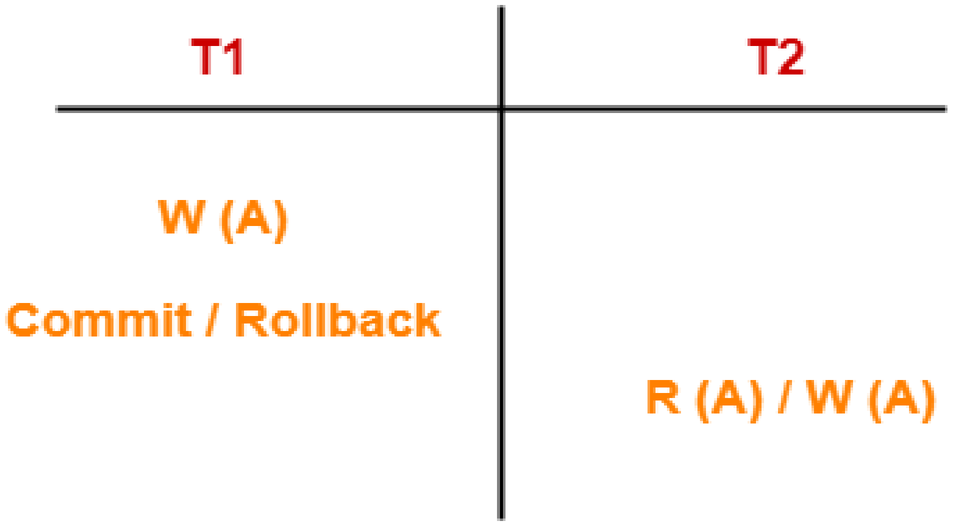
Cascadeless Schedule

3) <u>Strict Schedule-</u>

- ❖ If in a schedule, a transaction is neither allowed to read nor write a data item until the last transaction that has written it is committed or aborted, then such a schedule is called as a **Strict Schedule**.

  In other words,

  - ❖ Strict schedule allows only committed read and write operations.



Strict Schedule